INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

Technique and Automation for Testing of Commercial-off-the-Shelf Components

By

Michał Sówka

A thesis submitted to

The Faculty of Graduate Studies and Research

In partial fulfillment of the requirements of the degree of

Master of Applied Science

Ottawa-Carleton Institute of Electrical and Computer Engineering

Department of Systems and Computer Engineering

Carleton University

Ottawa, Ontario, K1S 5B6

Canada

September 2005

The undersigned hereby recommend to
The Faculty of Graduate Studies and Research
Acceptance of the thesis

Technique and Automation for Testing of Commercial-off-the-Shelf Components

Submitted by
Michal Sówka

In partial fulfillment of the requirements for the degree of
Master of Applied Science

Professor L. C. Briand (Co-Supervisor)

Professor Y. Labiche (Co-Supervisor)

Professor R. A. Goubran (Department Chair)

Carleton University

September 2005

Abstract

Commercial-off-the-Shelf (COTS) components provide a means to construct software systems (component-based system) in reduced time and cost to the system developer. In a COTS component software market there exist component vendors (original developers of the component) and component users (developers of the component-based systems). The component vendors provide the component to the user without source code or design documentation, and as a result the component users find it difficult to adequately test the component when deployed in their particular system. In this thesis we propose a framework that would facilitate an exchange of information (between the vendor and the user) required by the user in adequate testing of the COTS component. Then, based on this framework we adapt an existing specification-based testing technique and describe (and implement) a method for automated test generation. An evaluation of our approach demonstrates that it is possible to automatically generate cost effective test sequences and that these test sequences are effective in detecting complex errors.

Acknowledgements

First and foremost, I would like to thank Dr. Briand and Dr. Labiche for their guidance and financial support throughout the course of my research. Their exemplary practices have opened my eyes to how stimulating research can be.

Warmest thanks go out to my family, close by and overseas, and to dear friends for their support on many levels and encouragement.

Also, I would like to make a special mention of Professor Sylvia Boyd of Ottawa University for her fun lectures on an otherwise dry topic of Combinatorial Optimization.

Table of Contents

List of Tables

List of Figures

Glossary

component user: Organization developing large software by customizing and integrating commercial-off-the-shelf components.

component vendor: Organization that develops and markets commercial-off-the-shelf software component technology.

provides interface: The interface that the component provides, used to exercise the functionality of the component.

requires interface: The interface that the component requires, used by the component to exercise the functionality that it requires (from another component or the deployment platform).

deployment platform: Platform on which components are deployed. Consists of hardware, network, operating system, system services, run-time and component container.

CSPE: Constraints on Succeeding and Preceding Events

CSPE constraint: A constraint on succeeding and preceding event; in the context of COTS component testing this refers to sequencing constraints on component interface method invocations.

CSPE constraint predicate: The predicate associated with a Possibly Valid CSPE constraint. These are required for constraints where the succeeding method can only execute after the preceding method under specific condition (the predicate).

test sequence: Sequences of interface method invocations used to implement the test drivers.

test inputs: Inputs in the form of command line options, file inputs, or interface method arguments used by the test drivers to execute the test sequences.

test driver: The test driver executes the test sequence using test inputs.

test oracle: The test oracle asserts the correctness of the tests executed by the test driver based on the test inputs. The test oracle may be part of the test driver in the form of assertion statements.

test suite: The combination of the four: test sequences, test inputs, test driver, test oracles; forms a complete unit used in testing of a component.

Always Valid constraint: The CSPE constraint that specifies the invocation of two consecutive method calls that represent a correct execution of the component.

Never Valid constraint: The CSPE constraint that specifies the invocation of two consecutive method calls that represent an erroneous execution of the component.

Possibly Valid constraint: The CSPE constraint that specifies the invocation of two consecutive method call that represent a possibly correct execution of the component. This depends on the CSPE constraint predicate.

A criterion: The CSPE constraint criterion requiring that all Always Valid constraints are tested.

AP criterion: The CSPE constraint criterion requiring that all Always Valid and all Possibly Valid constraints are tested. Possibly Valid constraints are tested such that the constraint predicate evaluates to *true*.

N criterion: The CSPE constraint criterion requiring that all Never Valid constraints are tested.

NP criterion: The CSPE constraint criterion requiring that all Never Valid and all Possibly Valid constraints are tested. Possibly Valid constraints are tested such that the constraint predicate evaluates to *false*.

ANP criterion: The exhaustive CSPE constraint criterion requiring that all (Always Valid, Never Valid, Possibly Valid) constraints are tested. Possibly Valid constraints are tested such that the constraint predicate evaluates to both *true* and *false*.

Disjunctive Normal Form (DNF): A form of a Boolean expression composed of a disjunction of disjuncts or fundamental conjunctions only. A Boolean expression may be in DNF or it can be transformed into DNF through Boolean transformations.

PC criterion: The CSPE constraint predicate criterion requiring that each predicate of a Possibly Valid constraint be covered once such that it evaluates to *true*, and once such that it evaluates to *false*.

IC criterion: The CSPE constraint predicate criterion requiring that each implicant of the predicate be covered such that it is *true* and *false*.

PIC criterion: The CSPE constraint predicate criterion requiring that each implicant of the predicate be covered such that it is *true* and *false* while all other implicants are set such that they do not affect the predicate.

CoC criterion: The exhaustive CSPE constraint predicate criterion requiring that all possible clause combinations of the predicate are covered.

DRPP: Directed Rural Postman Problem: a graph routing problem where a directed graph contains a subset of required arcs, and the objective is to traverse the graph such that each required arc is covered at least once.

SSA: Shortest Spanning Arborescence: a graph routing problem where it is required that a subset of arcs in a directed graph be chosen such that there is a path from the root node to all other nodes of the graph with a minimum cost of arc subset.

MCMF: Minimum Cost Maximum Flow: A variant of a flow problem adaptable in solving demand and supply problems on a directed graph. Demand and supply problems entail that supply nodes satisfy demand nodes by routing flow (a sequence of arcs) between the two nodes.

Chapter 1
Introduction

Modern software components are highly reusable pieces of software and play an important role in the next generation of software engineering practices. The concept of software components, software modules, or more generally software reuse, is not new and dates back to the late 1970s. The structure-oriented software development methodology of the time often limited the reuse of the software modules to within the scope of the project. More recently, object-oriented development methodologies have facilitated packaging of sub-systems into software modules for reuse across development teams and projects.

Modern software components promote a greater level of software reuse by supporting well established interface specifications through which they are deployed and exercised. These interface specifications allow for the component to be reused across organizations and not to be limited to the scope of one development team or project. The ability to reuse software components across organizations creates a component market which consists of *component vendors* and *component users*. Component users purchase the vendors' components for use in their component-based software systems. Component-based software engineering practices have many advantages but also pose new problems for developers. One of these problems is in the evaluation of the Commercial-off-the-Shelf (COTS) components by the component user.

1.1 Thesis Motivations

The general problem, discussed in greater detail in Chapter 2, is that the COTS component vendor, the original developer of the component, most often does not include component source code or design documentation when delivering the finished product to the component user. Indeed, the component source code and design documentation would reveal the internals of the component such as private data, design decisions, and algorithms which represent the component vendor's intellectual property. Validation and verification are important steps of the COTS component evaluation process. While the component vendor uses the source code and design documentation to conduct these steps of the evaluation process during the component's development, the component user is at a great disadvantage when evaluating the component without them.

One final and important step of evaluation of the COTS component by the component user is deployment testing of the component in its deployment platform. Deployment testing of the component by the component user is difficult because the user has limited information without the source code or design documentation. Information found in the source code and design documentation may be used to derive test suites as well as evaluate the test suites for adequacy.

Additionally, COTS component deployment testing technique should facilitate the automation of the testing process. As model-based approaches to software development gain popularity and development tools start supporting automated component generation (e.g., Enterprise JavaBeans), manual testing of the used components will prove too inefficient. A testing automation framework should alleviate as much of the testing effort

from the component user as possible, and not require any information beyond the limited set available.

1.2 Thesis Contributions

The main contributions of this thesis include: 1) identification of a set of information that would be useful to the component user during component testing, and could be provided by the component vendor without revealing proprietary details; 2) definition of a testing technique that uses the vendor provided information and enables the component user to adequately test the component; 3) an automation framework, including an algorithm and prototype tool, used to generate test suites based on this testing technique. This thesis begins with a definition of the problem in Chapter 2, and a summary of related works addressing the problem of component testing in Chapter 3. Chapter 4 of the thesis presents a technique to address the problem of COTS component testing by the component user. Chapter 5, along with Chapter 4, is the main contribution of this thesis and proposes a framework for automation of the component testing technique described in Chapter 4. The thesis finishes with a description of the implemented prototype tool, a case studies demonstrating the testing technique and automation framework in Chapters 6 and 7, and a conclusion contained in Chapter 8.

Chapter 2
Problem Definition

The problem of COTS component testing addressed by this thesis is defined in three subsections: definition of component-based software engineering (Section 2.1), brief description of challenges faced by COTS component users (Section 2.2), and a set of requirements for a technique developed to address these challenges (Section 2.3).

2.1 Component-based Software Engineering

A differentiating factor between component-based software development and predecessor software practices is the integral role commercial markets play in software components. By enabling software to be reused across organizations, software components give rise to two relatively new concepts in software practice: *commercial-off-the-shelf* (COTS) components, and *component-based software engineering* (CBSE). Commercial-off-the-shelf components are software components that typically have been designed and implemented from the ground up with intention to be sold outside of the organization. Component vendors, ideally specialists in the application domain addressed by the component, dedicate themselves solely to the development of components based on the component user's requirements resulting in well supported, flexible, high quality COTS components. COTS component vendors, the term used in this thesis, are the original developers of the component. The components users are the organizations that purchase COTS components from the component vendors. Component-based software engineering is an approach taken by component users in developing large, complicated software

systems through composition of COTS components; if applied properly the approach has the potential to lower project costs and reduce development time. This section of the thesis provides a detailed definition of software components and component deployment platforms (Sections 2.1.1 and 2.1.2), introduces the UML 2.0 [21] software component graphical notation used in this thesis (Section 2.1.3), and explains some of the strongest arguments in favor of their use (Section 2.1.4).

2.1.1 Software Components

Modern component models such as ActiveX, CORBA, and Enterprise JavaBeans take a further step in promoting code reuse by specifying strict requirements for exercising software components through contractually specified interfaces[1]. A well known definition was presented by Clement Szyperski at the 1999 European Conference on Object-Oriented Programming [43].

> *A software component is a unit of composition with contractually specified interfaces and context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*

This thesis will, from this point on, refer to software components as defined above; the term should not be used when referring to object-oriented class packages or procedural

[1] ActiveX is Microsoft's proprietary component model targeting desktop system applications. CORBA and Enterprise JavaBeans, owned by OMG and Sun Microsystems respectively, are more open standards that are designed for distributed applications.

code libraries.[2] Composition is the assembly of one or more software components into composite software systems. Contractually specified interfaces are either of the two types: *provides*, representing the functionality that the component provides; or *requires*, representing the functionality that the component requires. Functionality that the component requires is satisfied by the deployment platform and other deployed components, and may include services such as a database server. Specifying contractually what the component requires is sufficient to allow the software component to be reused not only across projects, but also across different organizations and deployment platforms. The interfaces are contractually specified through, for instance, an interface definition language (IDL) in the case of CORBA, or standard Java interfaces in the case of Enterprise Java Beans. The requires and provides interfaces are then packaged with the compiled software component as this is what allows the component to be reused without source code or design documentation.

2.1.2 Component Deployment Platforms

Another key characteristic of software components is that before they can be instantiated they must first be deployed by the component user in a deployment platform. Figure 2-1 is a representation of the deployment platform, components, and component-based software.

---

[2] Although class packages and procedural libraries are not considered software components according to the definition presented here, either could be used as a component by including agreed-upon artifacts specifying provides and requires interfaces. These agreed-upon (standards-based) artifacts are in fact one of the key characteristics of components.

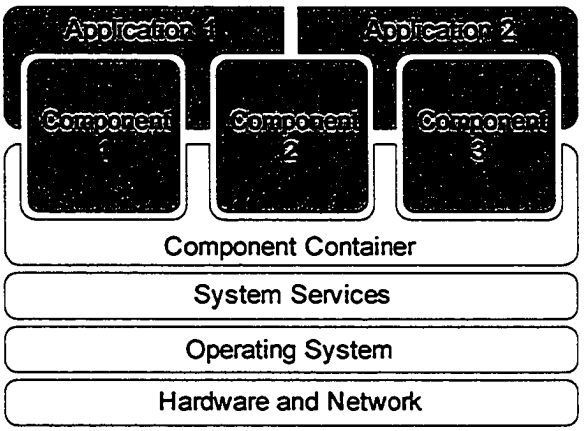


Figure 2-1: Component Deployment Platform (in White); Components, and Component-based Software (in Gray)

The deployment platform, shown in white, typically consists of computer hardware and network, operating system, system services, and a component container.

- *Hardware and network* are distinguished by different computer architectures (Intel x86, IBM PowerPC, Sun UltraSPARC), network hardware (Ethernet, Token Ring, Wireless), and network protocols (TCP/IP, UDP).
- O*perating system* runs on the hardware and network, and provides low level resource management. Traditional applications may run directly on top of the operating system.
- *System services* may include network directory services (used for user authentication, email address lookup), database servers, and e-mail servers. These either reside locally on the deployment platform or across the network.
- *Component container* provides all the interfaces required for component deployment, connects all of the deployed components together, and exposes all

the interfaces provided by the component to the component user. Component containers are implemented strictly according to the specification of a particular component model.

The components and component-based software of Figure 2-1 are shown in gray. Figure 2-1 depicts three components, component 1 through 3, each connected to the deployment platform and potentially to the other components through the component container. A component is connected to the deployment platform and other components using its requires interfaces. The component-based software, applications 1 and 2, then instantiate and exercise the components through the component container. Applications connect to the components using the components' provides interfaces. As shown in Figure 2-1, a component may be used by more than one application.

This definition of a deployment platform is general and oriented at distributed computing component models, also known as server-side component models, such as Enterprise JavaBeans or CORBA. Considering simpler component models such as Eclipse IDE and its plug-in components [18], deployment platforms vary in complexity, but the principles are the same and the challenges faced in testing unchanged.

2.1.3 UML 2.0 Software Component Notation

Figure 2-2 shows the UML 2.0 notation [21] for software components. The notation is similar to the UML class notation, with a stereotype icon in the upper right hand corner of the rectangle. The stereotype icon was used in previous UML specifications, UML 1.5 [19], as a notation for a component.

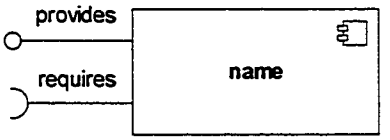


Figure 2-2: UML 2.0 Component Notation

The two extremities of the notation are the requires and provides interfaces. The end point of the requires interface is denoted as an open crescent, while the provides is a full circle; both are labeled according to their roles. The ends points of the requires and provides interfaces illustrates well wiring between component interfaces: the provides interfaces, full circle, plugs into the requires interface of another component, open crescent, as if it where a receptacle.

2.1.4 Advantages of Component-based Software Engineering

The case for using software components and component-based software engineering is based on a need to reduce the cost of developing software. An organization should have the choice of purchasing components developed outside of the organization. In earlier software reuse, object-oriented class packages and procedural libraries represented units of reusable programming. Class packages and procedural libraries are not suitable for commercial-off-the-shelf use for the following reasons: 1) they may require that the vendor reveal implementation details in order for the packages and libraries to be reused; 2) they are closely coupled with specific hardware architectures, operating systems, and programming languages.

The above mentioned characteristics of earlier software reuse limit the use of class libraries and procedural modules to within a project or development team, and at best

made their reuse across organizations difficult. Software components address the need for more general software reuse through contractually specified interfaces and deployment-platform-independent implementations.

Software components benefit both the component vendor and the component user. The component vendor is able to market their expertise in a particular field by packaging it in a software component, and reach a large clientèle. The component user purchases the software component for only a dividend of what it cost the vendor to develop. However, the use of a COTS components may be associated with a loss of flexibility because they are developed for wide variety of applications in component-based systems. In a lot of applications this is outweighed by the cost savings. In a mature software component market the component user also has a choice between offerings from multiple component vendors. This allows the component user to test a component vendor's component and, if unsatisfied by the components functionality and performance, exchange it for another from a different vendor.

2.2 Challenges in Testing of COTS Components

Along with all the benefits of COTS component use, component users are faced with challenges [22] that are non-issues in traditional software development practices. Some of the challenges that users are faced with in adequate testing of COTS components are:

- heterogeneity of deployment platforms
- source code and design documentation availability
- test adequacy

2.2.1 Heterogeneity of Deployment Platforms

Component users deploy COTS components across heterogeneous platforms where the hardware architecture, operating system, and deployment environment are not necessarily the same as those used by the component vendor during the component's development and testing. The goal of component models and their implementations is to provide true cross platform compatibility. They accomplish this to a great degree through well defined component model specification that is adhered to during component implementation and packaging. Component containers are implemented according to the same specification, targeting different computer and operating system environments and may be available from a number of different vendors; components developed according to the same component model may be deployed on any of these containers. Despite the specifications, components and component containers are not 100% interchangeable. Some of the reasons for this are: differences in operating systems, availability of specific services (e.g., database), and discrepancies in container implementation (resulting from parts of the component model specification being subject to interpretation by container developer). This is one of the main reasons why deployment testing of COTS components is important.

2.2.2 Source Availability

Component source includes the component's source code and design documentation. As already mentioned in previous sections, the source code and design documentation for COTS components represents the component vendor's intellectual property and disclosing them would make it difficult for the vendor to license their technology. The

lack of source code limits the testing techniques available to the component user, prohibits maintenance, and could prove disastrous in the event that the component vendor goes out of business. The exception to this may be the component vendors that release their product under an open source license and operate under a business model that allows them to remain profitable.

The lack of component source code and design documentation limits the types of testing techniques that can be used as well as limits controllability and observability of the testing process. In the case of controllability and observability, for instance, to test a specific functionality of the component-based system under test, it is required that the component be set in a specific state and this may turn out to be difficult or even impossible without controllability. Another example, in the case of testing techniques, *source code coverage* [6] cannot be used to guide the generation of test suites since, as the name suggests, it requires the availability of source code. Source code coverage, among other coverage-based testing techniques, is primarily used to asses test adequacy, another challenge faced by component users described next.

2.2.3 Test Adequacy

Test adequacy is a measure of sufficiency of a test suite. It is often defined in a set of criteria, from which one is chosen when generating the tests. Based on the measure of test adequacy it is possible to determine when to stop testing. Test adequacy is the second of the challenges of focus in this thesis. Though any testing technique needs to define test adequacy criteria according to which test suites are generated and validated, none of the

white-box techniques that require availability of source code or design documentation are applicable, e.g., control flow coverage, data flow coverage, mutation testing. The contractually specified interfaces and the user documentation of the component make it suitable for black-box and functional testing techniques.

Black-box and functional testing techniques can be used based on the component's contractual interfaces and the component vendor-supplied documentation. Black-box testing techniques such as all-method coverage and all-exception coverage are based on components' contractual interfaces but are not sufficient, as shown in [23]. Close assessment for adequacy of black-box and functional testing techniques is difficult since there is no standard representation for testing information that can be used by the component user to measure the test suite.

Another reason why test adequacy is particularly challenging in CBSE is due to a software component's design for broad applicability. Since COTS components are not designed specifically for the software system in which they are deployed, they often include functionality that is not required by that system. Testing the entire component would then be a waste of the user's resources.

2.3 Requirements for Testing of COTS Components

This section investigates how the challenges presented in the previous section can be addressed, that is, it clearly identifies what are the requirements for a COTS component testing strategy.

2.3.1 Testing Components without Source

A black-box testing strategy for COTS component testing requires additional testing metadata according to which tests are generated and validated. Testing metadata is a standard representation for testing information (need for which is mentioned in Section 2.2.3) developed as part of the testing strategy. Component testing metadata should be derived from available resources, including component interfaces and documentation, or additionally provided by the component vendor in form of additional artifacts delivered with the component, e.g., component use case scenarios or high-level UML diagrams.

A technique for testing COTS components needs to define: 1) a systematic method by which testing metadata are derived, 2) a set of criteria according to which tests suites are assessed for adequacy, 3) a systematic method for using the derived metadata and criteria in generating test suites. The data from which metadata is derived should not be tied to any one source or limited to a stage of component development or use, i.e., users and vendors should be able to derive the metadata from various pieces of information: source code, UML diagrams, API documentation. The defined criteria should be based on the component interfaces, documentation, and the testing metadata.

2.3.2 Variability of Testing Effort

The COTS component testing strategy should allow the component user to vary the amount of testing effort according to their budget and required component reliability. The variability of the testing effort should be provided by way of the set of criteria available

based on the testing technique. The criteria should therefore vary in terms of the testing effort they entail to suit a wide range of testing budgets and requirements.

Component users that are developing critical system and have the budget for very thorough testing may require that the deployed component be tested with respect to the most thorough testing criterion. Component users that are developing non-critical, lower dependability software systems and have a limited budget may only want to test the deployed component with respect to the minimum testing criterion.

2.3.3 Targeting Functionalities

The component user should be able to test only a specified subset of the component functionalities. This would enable testing of only the component functionalities that are used by the composed system.

In order to facilitate targeting functionalities, the COTS component testing strategy should define schemes for selecting units of the component. The units used for targeting specific functionalities may include: interface method calls, sequences of interface method calls, and interface method calls with specific method argument values. The sources from which the units are identified should also be wide ranging, such that the component user can use any artifact available to them, e.g., UML diagrams that model the component usage in the context of the developed system such as collaboration diagrams and sequence diagrams.

2.3.4 Test Automation Framework

The automation of the COTS component testing strategy should allow the component user to generate test suites based chosen test criterion and targeted functionalities. With this in mind, the strategy and its automation should be implemented in a widely applicable component testing framework. The component testing framework should: 1) model the work flow of the component testing strategy, 2) specify how component vendors can facilitate testing of the component, 3) provide a high level of automation and relieve the component user of as much testing effort as possible, and 4) provide interfaces through which the component user will control and observe the testing process.

The framework should model the work flow of the testing strategy by providing facilities for each step of the testing process. Some of these steps may include: deployment of the component, collection of testing data (test criterion, test inputs, and test oracle), validation of the test data, generation of the test suite, and execution of the test suite. The framework should be extensible to make it possible to use different testing data as explained in Sections 2.3.1 and 2.3.2, and should be general enough to be used in testing of any component model type (ActiveX, Enterprise JavaBeans, CORBA).

The framework should specify component development guidelines that could be adopted to facilitate the testing strategy. Adhering to such guidelines should be optional as the component vendor may not always be able to change their development practices, and the framework should make it possible for the component user to employ the testing strategy without any component vendor involvement. An advantage of the involvement by the component vendor is that they could provide additional built-in methods to

provide the component user with controllability and observability. For example, given a stateful component, in addition to the original set of methods the component vendor may implement a method `getState` that returns the current state of the component. With some documentation on the expected behavior for each of its states, the component user is then able to better test the component. The built-in methods should be useful to the component user testing the component, but not indispensable such that the testing strategy is still possible (but potentially less effective).

The testing strategy should provide as much of automation as possible, and as such, require little involvement from the component user. Admittedly, some of tasks part of the test automation are difficult, are not addressed in this thesis, and will require some involvement form the user. For instance, generation of test scaffolding (which includes test drivers, test inputs, and test oracles) may not be part of the automation, but, based on the part of the strategy that is automated would require that the component user manually generate it. Further extension of the testing framework to allow complete automation should be considered, i.e., the framework should be extensible.

Chapter 3
Related Works

This section of the thesis presents works related to testing of COTS components and to the testing techniques defined in this thesis. The first of the reviewed works addresses the issue of targeted testing of COTS components (Section 3.1). The next two reviewed works present techniques for specification based testing and logical testing (Sections 3.2 and 3.3), both adapted in this thesis. Next, two techniques addressing component testing are presented in Sections 3.4 and 3.5. The last section, Section 3.6, reviews an example use of graph-based techniques applied to software testing.

3.1 Formal Model of Component-based Software

Rosenblum defined a formal model of component-based software [36] that is useful in understanding concepts related to this thesis. The formal model of component-based software defines why targeting component functionalities (Section 2.3.3) is required.

Rosenblum's formal model of component-based software is defined around a component-based software system *P*, and a component *M* composed into the system *P*. Based on the invoked system *P* functionality and invoked component *M* interfaces, there are four possible scenarios: M-bypass, M-traverse, P-relevant, P-irrelevant.

M-bypass: The invoked system functionality does not exercise the component, i.e., does not invoke any of its interface methods of *M*.

M-traverse: The invoked system functionality does exercise the component, i.e., invokes the interface methods of *M* since *P* functionality depends on, in part or entirely, the functionality of *M*.

P-relevant: The invocation of the component interface represents a possible use case of component *M* by program *P*, i.e., this represents the functionality of the component that is used in *P*.

P-irrelevant: The invocation of the component interface is not part of any of the possible use cases of component *M* by program *P*, i.e., this represents the functionality of the component that is not used in *P*.

As defined in Section 2.3.3, the COTS components testing strategy is required to be able to allow the component user to test only the component functionality that is part of the software system. Rosenblum's formal model of component-based software is useful in formalizing targeting of test suites. *M-traverse* and *P-relevant* type invocations are relevant to the users' component and component-based system, and therefore represent the target functionalities of the test suites. *M-bypass* type invocations are relevant to the users' component-based system but not the component, and therefore do not represent target functionalities of the component. And finally, *P-irrelevant* type invocations represent functionality of the component that are not relevant to the users' component-based system, and therefore are not considered for testing at all.

3.2 Constraints-based Testing Technique

Constraints on Succeeding and Preceding Events (CSPE) is a testing technique first used in specification-based testing of concurrent programs [9]. The notion of event in the context of testing concurrent programs is the synchronization events that the concurrent program under test can receive. The CSPE technique was adopted for the purpose of class unit testing by Daniels and Tai [12] (there it was referred to as intra-class testing). Events in this context are invocation of the class member methods. This section presents a review of the CSPE technique based on the work by Daniels and Tai in [12].

The CSPE technique is considered in this thesis because: it is based only on the interface of the system under test, and it is applicable at different levels of granularity. CSPE constraints are derived from documented system interfaces, synchronization events of the concurrent system interface or methods of the class interface in the context of concurrent systems and class testing, respectively. The technique was equally effective in the context of concurrent system testing (testing of a system of modules or classes) and class unit testing (testing of a single class). These observations suggest that the technique could be used for COTS component testing.

The CSPE technique consists of a two steps. Note that the terms used in the remainder of this section are specific to class unit testing and events in this context are referred to as method invocations, or methods for short. First, CSPE constraints, defining constraints on sequences of invocation of pairs of methods, are derived from the class specification (Section 3.2.1). Second, the CSPE constraints are used to generate method test sequences according to CSPE constraint coverage criteria (Section 3.2.2). Last we

discuss the work by Karçali and Tai [25] which addresses automation of the CSPE-based testing technique (Section 3.2.3).

3.2.1 CSPE Constraints

CSPE constraints are sequencing constraints that specify whether two methods can be invoked one after the other, and if so, under which conditions. In other words, a CSPE constraint is a 3-tuple (*preceding method*, *succeeding method*, *predicate*) indicating that an invocation to the *succeeding method* can be performed after an invocation to the *preceding method* when the *predicate* is true. CSPE constraints thus specify valid method executions sequences (of two methods), or not valid execution sequences that would result in failures.

CSPE constraints, and more specifically predicates, can be derived from the postcondition of the preceding method and the precondition of the succeeding method. The postcondition of the preceding method either implies, contradicts, or partially implies the precondition of the succeeding method, and as such CSPE constraints (as interpreted from [12]) fall into three types: *Always Valid*, *Never Valid*, *Possibly Valid*, and *Possibly Invalid*. Table 3-1 contains the four constraint types as interpreted from [12], including the exact notation used and interpreted definition.

Table 3-1: CSPE Constraints in the Context of Class Unit Testing, as Interpreted from [12]

Constraint Type (Notation)	Definition
Always Valid a[M1; → M2]	Invoking the succeeding method M2 immediately after the preceding method M1 is always valid. This happens when the postcondition of the preceding method implies the precondition of the succeeding method.
Never Valid ~[M1; → M2]	Invoking the succeeding method M2 immediately after the preceding method M1 is never valid. This happens when the postcondition of the preceding method contradicts the precondition of the succeeding method.
Possibly Valid pT[M1; → M2]K	Invoking the succeeding method M2 immediately after the preceding method M1 is valid only when the predicate K is satisfied, and not valid otherwise. This happens when the postcondition of the preceding method M1 does not imply the precondition of the succeeding method M2, but the conjunction of the preceding method's postcondition and the predicate K implies the precondition of the succeeding method.
Possibly Invalid pF[M1; → M2](*not* K)	Invoking the succeeding method M2 immediately after the preceding method M1 is not valid only when the predicate is not satisfied, and valid otherwise. This is similar as in the case of the *Possibly Valid* constraint type, but in this case the negation of the predicate K is considered.

Two things should be noted of the CSPE constraint definition interpreted from [12]. One is that the notation used may be considered inconsistent, i.e., arbitrary symbols are used to denote the constraint type (a, ~), and also pT and pF type constraints include a second

operator after the square brackets that the other two do not. Secondly, two constraint types, *Possibly Valid* and *Possibly Invalid*, conceptually represent only one constraint; the *Possibly Invalid* type constraint is redundant since a *Possibly Valid* constraint may be transformed into a *Possibly Invalid* constraint by replacing the predicate with its complement.

3.2.2 CSPE-based Test Adequacy Criteria

The adequacy of a test set, the set of test cases that make up a test suite, can be evaluated according to CSPE constraints. For instance one may first want to ensure that all the valid method pair sequences are exercised and verify functionalities of the class under test are correct. One may then ensure that all the not valid method pair sequences are exercised and verify whether the class under test is robust. These basic criteria are further developed into a complete set of CSPE-based test adequacy criteria reviewed in Table 3-2.

Table 3-2: CSPE Testing Criteria in the Context of Class Unit Testing, as Interpreted from [12]

Criterion (shorthand)	Definition
Always Valid Coverage (a)	Requires that all *Always Valid* constraints be covered at least once with a minimum number of test cases.
Always Valid Several Coverage (a-s)	Requires that all *Always Valid* constraints be covered at least once with more than the minimum number of test sequences.

Always/Possibly Valid Coverage (a/pT)	Requires that all *Always Valid* and *Possibly Valid* constraints be covered at least once with a minimum number of test sequences.
Always/Possibly Valid Several Coverage (a/pT-s)	Requires that all *Always Valid* and *Possibly Valid* constraints be covered at least once with more than the minimum number of test sequences.
Possibly Invalid/Never Valid Coverage (pF/~)	Requires that all *Possibly Invalid* and *Never Valid* constraints be covered at least once with a minimum number of test sequences.
Always/Possibly Valid Several/Possibly Invalid/Never Valid Coverage (a/pT-s/pF/~)	Requires that all *Always Valid*, *Possibly Valid*, *Possibly Invalid*, and *Never Valid* constraints be covered at lease once with more than the minimum number of test sequences.

It is worth noting that there is no criterion that requires only the *Never Valid* constraints to be tested, yet they are equally as important in testing. Secondly, test criterion definitions that ask for more than the minimum number of test sequences, typed as *Several*, result in suboptimal test suites (in terms of number of method executions), but do not specify which constraints should be tested more then once or how many times they should be tested. Additionally, only some criterion types have *Several* variants and not others (*pF/~* criterion does not have a *Several* type variant), and the motivation behind this is also not found in [12]. Possibly adequate in the context of class unit testing, these criteria are too ambiguous for use in component testing and will have to be redefined.

3.2.3 Test Generation

CSPE constraints can be used to generate test sequences. Each CSPE constraint forms a subsequence of two methods calls, and by concatenating CSPE constraints, complete test cases involving all the methods of the system under test can be generated. Test cases are defined as a sequence of methods invoked on an object. Test cases begin with class instantiation and end normally (for a test case involving valid constraints) or in a failure (for a test case involving not valid constraints). For test cases that involve not valid constraints, one test case is required to cover each not valid constraint since coverage of the constraint results in failure.

CSPE constraints are concatenated by appending one constraint to another, such that the preceding method of the first constraint matches the succeeding method of the second constraint. For example, given a constraint `C1` with a preceding method `methodOne()` and succeeding method `methodTwo()`, and a constraint `C2` with a preceding method `methodTwo()` and succeeding method `methodThree()`; constraint `C2` would be appended to constraint `C1` since the preceding method of `C2` matches succeeding method of `C1`. The concatenation of the two would result in part of the test sequence `methodOne()→methodTwo()→methodThree()`.

Consecutive constraints with a preceding methods matching the succeeding methods of the previous constraint are concatenated, and the succeeding method of the concatenated constraint is appended to the test sequence. More generally, given a set of CSPE constraint S_C, entire CSPE-based test sequences are generated by concatenating the constraints of S_C until all are included in the test sequence. The set of CSPE constraints

S_C is determined by the selected test adequacy criterion, e.g., *Always Valid* criterion which requires S_C to contain all valid constraints. Test sequences were generated manually in [9, 12]. A method for automated generation of test sequences from CSPE constraints was described in [25], and although it was done in the context of concurrent program testing, the method is adaptable to class unit and COTS component testing, as suggested by Daniels and Tai [12].

The generated test sequences in [25] are represented by tree graphs such as the example in Figure 3-1. The nodes of the tree represent the covered CSPE constraints and the arcs represent the concatenation of the CSPE constraints into test sequences. The root node is the first CSPE constraint covered in each of the test sequences and the leaf nodes are the terminal CSPE constraints, such that every tree path represents a unique test sequence. Figure 3-1 represents a set of three constraint sequences: `c1→c2→c3`, `c1→c3→c4`, and `c1→c3→c1`.

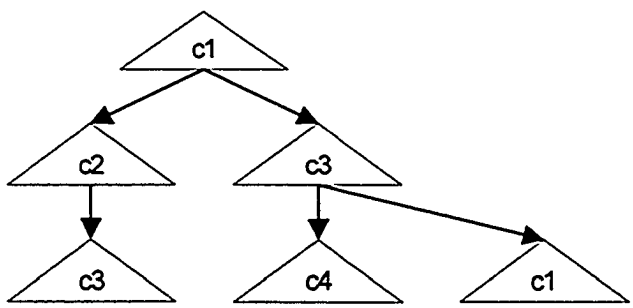


Figure 3-1: Karçali and Tai's CSPE Test Sequence Set Tree Graph Representation

Karçali and Tai offer two versions of the solution [25], an optimal algorithm and an (approximated) heuristic solution. The optimal version of the algorithm is not feasible

since it exhibits exponential time complexity, and Karçali and Tai use the heuristic solution in their case study. Both versions of the algorithm involve two steps: construction of the test sequence set tree graph, and pruning of the test sequence set tree graph.

Step 1 Construct Tree: The tree is grown breadth first, appending method sequence constraints of a given criterion with corresponding preceding and succeeding methods. Growth is terminated once all of the constraints have been covered in the tree, such that each constraint is covered at least once.

Step 2 Prune Tree: The tree is pruned to reduce the total number of methods required in the test suite. This is done by evaluating the paths (terminating at a leaf node or earlier) according to their lengths and number of constraints which they cover, and then removing any redundant nodes.

When generating test sequences for testing of concurrent program the objective is to construct short chains of events regardless of the total number of events (see [25] for additional details). The resulting tree is composed of a large number of short paths, i.e., the test sequence set is composed of a large number of short sequences of constraints. On the other hand, in the context of component testing all method sequence constraints should be covered in as few method invocations as possible. This requires that the composed tree minimize the number of nodes regardless of the number of paths and path

lengths. For this reason, the breadth first approach in growing the test sequence set tree graph is not necessarily optimal.

Test sequences generated according to the solution adapted from Karçali and Tai's are not efficient for criteria that contain *Always Valid* and *Possibly Valid* constraints (as will be shown in Section 7.2). Test sequence tree graphs are constructed from a set of all valid constraints types with no regard for optimal constraint combinations. A more advanced combinatorial solution is required. Additionally the solution to automated generation of CSPE based test sequences presented by Karçali and Tai does not address targeting functionalities of the system under test (Section 2.3.3). The test sequences tree graphs are constructed from a set of all possible sequence constraints, and the algorithm is not able to target some parts of the component as required and others as not required.

3.3 Logical Testing Techniques

Different testing criteria have been defined in the literature to describe tests from Boolean expressions [2, 6]. Let C_p be the set of clauses (Boolean variables) of the predicate p. When testing the predicate p, the expected Boolean value of the predicate is determined by a tuple $t \in \mathbb{B}^{|Cp|}$ of Boolean values imposed on the clauses $c \in C_p$. Furthermore, p as a function of t can be expressed as mapping:

$$p(t) : \mathbb{B}^{|Cp|} \rightarrow \mathbb{B}.$$

A tuple example of Boolean values for a predicate $a \wedge (b \vee c)$ is (*true*, *true*, *false*) such that the predicate p evaluates to *true*.

A set of tuples t that satisfy a predicate testing criterion tc for a predicate p is denoted as $T_p(tc)$. Moreover, $T_p(tc)$ can be divided into: a subset of tuples $T_p^{true}(tc)$ where the predicate p evaluates to *true*, and a subset of tuples $T_p^{false}(tc)$ where the predicate p evaluates to *false*, such that

$$T_p(tc) = T_p^{true}(tc) \cup T_p^{false}(tc)$$

, where

$$T_p^{true}(tc) = \{\forall t \in T_p(tc) \mid p(t) = true\}$$

, and

$$T_p^{false}(tc) = \{\forall t \in T_p(tc) \mid p(t) = false\}.$$

A comprehensive review of logical testing techniques including *Predicate Coverage* (PC) and *Combinatorial Coverage* (CoC) was done by Ammann and Offutt in [2], and the DNF based criteria *Implicant Coverage* (IC) and *Prime Implicant Coverage* (PIC) where presented in [6].

Logical testing techniques are used to define CSPE-based testing criteria in this thesis. This section presents the techniques in three sections. First, all the definitions are presented in Section 3.3.1. Next, examples are provided for the defined techniques in Section 3.3.2, and last, a discussion of the techniques selected is found in Section 3.3.3.

3.3.1 Definitions

Logical testing techniques which are directly applicable to testing of CSPE constraint predicates are *Predicate Coverage* (PC) and *Combinatorial Coverage* (CoC).

Definition 3-1 Predicate Coverage (PC): Predicate Coverage requires that the tuples $t \in T_p(PC)$ test the predicate p twice, when it evaluates to *true* and *false*, respectively. In other words $|T_p(PC)| = 2$.

Definition 3-2 Combinatorial Coverage (CoC): Combinatorial Coverage requires that the tuples in $T_p(CoC)$ test the predicate p for all possible combinations of clauses' truth values, such that $|T_p(CoC)| = 2^{|Cp|}$.

Another two logical testing techniques selected are *Implicant Coverage* (IC) and *Prime Implicant Coverage* (PIC). IC and PIC are based on the following definitions for the *Disjunctive Normal Form* (DNF), *implicants*, and *nonredundant prime implicants.*

Definition 3-3 Disjunctive Normal Form (DNF): A predicate p is in disjunctive normal form if it is a fundamental conjunction or it is a disjunction of at least two disjuncts, or fundamental conjunctions, none of which is a subconjunction of any of the others.

Definition 3-4 Implicants: Implicants of a predicate p in DNF are disjuncts, or fundamental conjunctions, that determine the truth value of the entire predicate p.

Note that implicants are clauses of C_p, or fundamental conjunctions of clauses of C_p. If an implicant of a predicate in DNF form is *true*, then the predicate is also *true*; but if all implicant are *false*, then the predicate is *false*. For example, the predicate $a \vee (b \wedge c)$ is in

DNF form and a disjunction of a and a fundamental conjunct ($b \wedge c$); each of these is an implicant of the predicate and if either a or ($b \wedge c$) are *true*, then the predicate is also *true*, but if both are *false* then the predicate is *false*. A fundamental conjunct is a clause containing no disjunctions, e.g., ($b \wedge c$) is a fundamental conjunct while ($b \wedge (c \vee d)$) is not (d is some additional clause).

A basic technique of logical testing based on DNF representation of the predicate is to use tuples that will impose *true* values on implicants of the predicate p such that the predicate is expected to evaluate to *true*. In order to also test the predicate p such that the predicate is expected to evaluate to *false*, the complement of the predicate, $\neg p$, is expressed in DNF and implicants of $\neg p$ are also used. Let I_p be the set of implicants of the predicate p; divided into a subset of implicants that are derived from the literal predicate p, I_p^{true}, and a subset of implicants that are derived from the predicate $\neg p$, I_p^{false}; such that,

$$I_p = I_p^{true} \cup I_p^{false}.$$

Based on the complete set of implicants I_p, the definition of IC is as follows.

Definition 1 Implicant Coverage (IC): Implicant Coverage requires that the tuples $t \in T_p(IC)$ test the predicate p such that each implicant of I_p is true at least once.

The IC testing criterion is weak because it is possible that a single test case can satisfy more than one implicant. For example, in testing the predicate $a \vee (b \wedge c)$ with the tuple

(*true*, *true*, *true*) both implicants of I_p^{true}, a and $(b \wedge c)$, are tested with one tuple. If the implementation of $(b \wedge c)$ is erroneous (e.g., $(\neg b \wedge c)$ is implemented instead) the fault is masked by the first implicant being true. Nonredundant prime implicants are defined to allow testing of predicate p with one implicant at a time avoiding the masking effect.

Definition 3-5 Nonredundant Prime Implicants: Prime implicants are implicants that contain no proper subterms that are themselves implicants of the predicate, where a proper subterm is an implicant with one or more clauses omitted. Furthermore, nonredundant prime implicants are prime implicants that contain no redundant implicants.

A non redundant prime implicant compose nonredundant DNF representations of p. Whereas, given a redundant implicant in a redundant DNF representation, the predicate can be written without it through Boolean transformations (e.g., distributive laws, absorbtion laws). For example, in the Boolean expression $(a \wedge b) \vee (a \wedge c) \vee (\neg b \wedge c)$, the implicant $(a \wedge c)$ is redundant, since through a simplification law it could be expressed as $(a \wedge b) \vee (\neg b \wedge c)$.

Similar to the definitions of implicant set I_p, a set of nonredundant prime implicants is defined as $PI_p = PI_p^{true} \cup PI_p^{false}$. Prime implicant coverage, a logical testing technique stronger than implicant coverage, is defined based on nonredundant prime implicants.

Definition 3-6 Prime Implicant Coverage (PIC): Prime Implicant Coverage requires that the tuples $t \in T_p(PIC)$ test the predicate p such that as each implicant of

PI_p^{true} is set to *true* while all other implicants of PI_p^{true} are *false*, and as each implicant of PI_p^{false} is set to *true* while all other implicants of PI_p^{false} are *false*.

3.3.2 Examples

A predicate $(a \vee b) \wedge (a \vee c)$ and its truth table, Table 3-3, are used to demonstrate all of the criteria defined in the last subsection. The column t of Table 3-3 refers to tuples by their indices (1 through 8) which are used in this section to identify adequate test sets.

Table 3-3: Truth Table for Example Predicate $(a \vee b) \wedge (a \vee c)$

t	a	b	c	$(a \vee b) \wedge (a \vee c)$ $= a \vee (b \wedge c)$
1	*false*	*false*	*false*	*false*
2	*false*	*false*	*true*	*false*
3	*false*	*true*	*false*	*false*
4	*false*	*true*	*true*	*true*
5	*true*	*false*	*false*	*true*
6	*true*	*false*	*true*	*true*
7	*true*	*true*	*false*	*true*
8	*true*	*true*	*true*	*true*

From the table there are 15 possible test cases $T_p(PC)$ satisfying PC, e.g., {1,4}, where each test case contains two tuples that test predicate p once when it evaluates to *true* and once for *false*. For CoC, there is one possible test case $T_p(CoC) = \{1,2,3,4,5,6,7,8\}$ such that the predicate is tested for all possible combinations of its clauses' Boolean values.

The predicate $(a \vee b) \wedge (a \vee c)$ is not in DNF since it is a conjunction of two clauses $(a \vee b)$ and $(a \vee c)$. It can however be easily be transformed into $a \vee (b \wedge c)$ (using a distributive law) which is in DNF. We will now refer to the same predicate in its DNF

form, $a \vee (b \wedge c)$. Note that the same truth table (Table 3-3) is used for this predicate, i.e., the clause values and resulting predicates are the same.

Before we can determine which subset of test cases satisfy the IC and PIC criteria we have to find the implicants of p and $\neg p$. With the predicate already in nonredundant DNF form, these are:

$$I_p^{true} = \{a, bc\} \text{ and } I_p^{false} = \{\neg a \neg b, \neg c\}$$

Based on I_p^{true} and I_p^{false}, a possible test case $T_p(IC)$ that satisfies IC is {1,8}. As explained in the last subsection, the IC criterion has a weakness: a masking effect may occur such that a failure in one of the other implicants of the predicate may not be detected. This is the case in predicate test case 8 (same example as used in the last subsection).

For the last criterion PIC a possible test cases $T_p(PIC)$ = {5,4,1,7}, where {5,4} are used to test the I_p^{true} implicants, and {1,7} is used to test the I_p^{false}.

3.3.3 Discussion

In this section we discuss the four logical testing techniques presented in terms of strength (effectiveness in error detection), cost (number of test cases required and complexity in their generation), and why they are chosen for application in this thesis. The PC criterion is a weak testing criterion, while on the other hand CoC is the strongest possible, exhaustive testing criterion. These are a natural choice of logical testing criteria that may be used under weakest and strongest, respectively, testing requirements.

The two additional criteria IC and PIC are used to fill the gap between the first two; either is more effective than PC, but not nearly as expensive as CoC. Because it considers the masking effect described in Section 3.3.1, the PIC criterion is stronger than IC. The PIC testing criterion goes by another name *Variable Negation* in [47], where it has shown to be a cost effective criterion.

Another point of interest for deriving test cases based on IC and PIC, is that DNF and nonredundant prime implicants can be derived from any predicate expression by means of Karnaugh maps. These types of methods are well suited for instrumentation: see Section 5.2 of [37] where a solution that can deal a very large number of terms ($2^n - 1$, where n is number of Boolean variable) is presented.

The defined criteria can be characterized in terms of subsumption. In context of logical testing techniques, when one criterion is said to subsumes another, it implies that any test case of the former (subsuming criterion) will satisfy the latter (subsumed criterion). The subsumption hierarchy for the four logical testing techniques is shown in Figure 3-2: IC subsumes PC, PIC subsumes IC (and therefore also PC), and CoC subsumes all the other three.

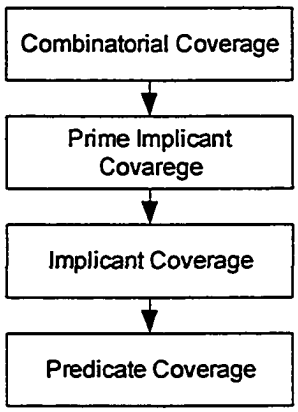


Figure 3-2: Logical Testing Techniques' Subsumption Hierarchy

For example, given the predicate p from the previous section (Section 3.3.2), a set of test cases that satisfy the IC criterion is $T_p(IC)=\{1,8\}$. Since this set of test cases tests the predicate for *true* and *false*, it also satisfies the PC criterion. In the case of CoC, its set $T_p(CoC)$ includes all possible test cases and satisfies any other criterion; hence CoC subsumes all other criteria.

3.4 Interface Mutation Testing Technique

The interface mutation method proposed by Gosh and Mathur [23] is motivated by the difficulty of testing of large systems. As the authors explain, white-box testing coverage criteria are useful in assessing testing of small systems, but are too expensive on larger systems as their cost grows exponentially with the size of the system. The issue posed by testing of large systems is very similar to that of testing COTS component without available source code. Gosh and Mathur demonstrate in an experiment that interface mutation testing is an efficient testing criterion. First, a brief definition of mutation-based testing techniques is in order.

Mutation-based testing techniques [13] are used to measure the effectiveness of test suites. Mutants are used to simulate faults in software; faults are source code mistakes or system errors that result in the software's failure. Mutants are seeded using mutation operators, a sample of these can be found in [27] and [26]. The mutated programs (containing one mutant each) are then tested with the intended test suites: mutants that are detected by the test suite are referred to as *killed mutants*, and the ratio of killed mutants and the total number of seeded mutants is defined as the *mutation score*. Test suites are then generated in an iterative manner by measuring their mutation score and enhancing them based on the mutants they failed to kill. The authors of [23] applied mutants to system interfaces and used them in generating system test suites.

Gosh and Mathur use IDL (Interface Definition Language) to demonstrate how the mutants are applied to interfaces. An IDL interface consists of a set of component method signatures. Each of the method signatures specifies the method name, its parameters, the return value, and zero or more exceptions that the method may throw. Gosh and Mathur define a set of eight mutation operators. The mutation operators mutate the method signatures found in the interface through: modification of parameter type, swapping of parameter values, modification of parameter values, passing of null objects and substituting the return values. An example of one of the proposed mutation operators follows.

Original IDL: `mkEntry(in String lname, in String fname)`

Mutant IDL: `mkEntry(in String fname, in String lname)`

Parameter values, `lname` and `fname`, are swapped such that the method should either run into a run-time error or return an erroneous value.

Gosh and Mathur conducted experiments to see how the interface mutation criteria stand up against two other more traditional ones, namely: *all methods coverage*, *all exception coverage*, and *code coverage*. They observed that interface mutation base test suites revealed more faults at a lower cost (measured as the number of test cases in the adequate test suite). The reader interested in the details of the experiments is referred to [23].

All methods coverage and all exceptions coverage testing techniques are of interest in this these because they do not require source code or design documentation. They are presented below as they are defined in [23].

All Methods Coverage, AC_{met}: A test suite is adequate with respect to method coverage if it leads to 100% method coverage.

All Exceptions Coverage, AC_{ex}: a test set is adequate with respect to exception coverage if it leads to 100% exception coverage, i.e. methods of the component are invoked such that every exception is raised.

3.5 Use of Component Metadata to Support Testing

Orso, Harrold and Rosenblum [34] describe a general type of component metadata that can be applied to support automated component testing. As Orso et al. point out, any software engineering artifact used in component based development can be a metadatum

for a given component as, as long as 1) the component vendor is involved in its production, 2) it is packaged with the component in a agreed upon (between the vendor and user) way, and 3) it is processable by automated development tools and environments.

Component metadata is common practice and allows component vendors to include with their components: deployment information, textual descriptions, version information and other information as prescribed by the particular component model. The metadata described by Orso, Harrold and Rosenblum is more generic than the metadata commonly used and, more importantly, extensible. In describing their proposed metadata implementation model Orso et al. address two issues: what format to use for the metadata, and how the data is accessed.

The main goal of the component implementation model is to allow for extensibility of the types of metadata attached to the components. In order to achieve this Orso et al. suggest the adaptation of a MIME (Multi-purpose Internet Mail Extensions) format specification. Email attachments are identified through MIME by a pair of slash delimited character strings, e.g. `"image/jpeg"` identifies the attachment as an image of the JPEG format. Similarity components, their interface methods, and other artifacts can be marked with metadata which itself is marked with a domain and a specific type.

The actual metadata can be static or dynamic. Static metadata includes a fixed value prescribed during development of the component, or initialized during component deployment. Dynamic metadata includes internal states of the component and

information calculated at run-time, e.g., current component state, code coverage achieved during execution. Ability to include dynamic metadata may prove useful in component testing and analysis.

Attaching and accessing metadata of the component would have to be standardized such that component vendors, component users, and component-based case tools be able to attach and read the metadata in a systematic way. As suggested by Orso et al. this is the one part of the implementation model that would have to remain fixed. Although the technical report does not detail how the data would be attached or how it should be stored, it does give example of how a user or automation tool would retrieve the metadata (See [34] for details).

Orso, Harrold, and Rosenblum also include examples in their technical report of applications of metadata in components. One example demonstrates how metadata could be used to execute run-time checks on method pre- and post-conditions, and the other suggests how metadata could be used to help in program slicing. For instance, in the former case operation contracts for a component could be retrieved from metadata type `"selfcheck/constract"` [34].

Orso et al. work should be considered when developing a COTS component testing automation framework. As proposed by Orso et al., XML should be used instead of MIME, and a XML schema that encompasses all aspects of the COTS component testing strategy should be defined. This requires that the static metadata remain outside of the compiled portion of the component. The static metadata should be packaged with the

component and accessed through part of the framework and not built-in interface methods as described in [34]. Dynamic metadata has to be accessed through built-in interface methods, and the COTS component testing framework should define precisely how these methods are developed and how they are discovered by the component user.

3.6 Statechart-based Component Testing

The work by Beydeda and Gruhn [5] presents a method for modeling a component-based system with statecharts, transforming these into a control flow graph and consequently using the graph to generate test cases. Their method addresses testing of COTS components of the component-based system, non-COTS components of the system, the deployment platform (specifically, the component container), and the integration of all three.

In [5], the example used demonstrates how a COTS component, a non-COTS component, and the component container are modeled with *component state machines* (CSM). CSMs are similar to UML state charts [21] and are composed of states and transitions. A transition is defined as a 5-tuple composed of: the source state, the target state, transition event, transition guard, and transition action. These state machines assume detailed knowledge of the implementation in the case of non-COTS component, and in the case of the COTS component are only based on the component specification, and are therefore very abstract.

The state transitions of all the CSMs are then transformed into control statements and used to construct a *component-based software flow graph* (CBSFG). The CBSFG models

information gathered from both component specification and source code, e.g., control flow and data flow information. The CBSFG contains two subgraphs for each of the non-COTS components: one is based on the specification and another is based of the available source code. The COTS component is represented only with one subgraph, based on the specification, since the source code for this component is not available.

As the last step of the technique presented in [5], the CBSFG is used to generate test cases. In [5], only the data flow criterion is used to generate the test sequences, specifically the *all-definitions* [6] criterion. Again, in the case of the COTS component, white-box type testing cannot be carried out for the lack of source code, and only black-box type testing is performed.

This technique is potentially effective in testing non-COTS components and integration testing of the component-based system, but is not applicable in testing of COTS components. It does not provide a method for extracting information out of the COTS component, a task required of the component vendor and described in Section 2.3.1. The COTS component of the example given in [5] is only tested based on its interactions with the system's non-COTS component and the component container.

3.7 Graph-based Test Sequence Generation

The work of Denise, Gaudel and Gouraud [14] presents a method for automatically generating test sequences and test inputs based on a statistical method and coverage criteria.

Software structures (or behavior) are modeled with directed graphs and can be directly derived from state charts, Petri-nets, or control flow graphs. Nodes are execution states of the software under test, and arcs are units of execution. For a description of a software system D and some coverage criterion C, the authors define $E_C(D)$ as the set of elements that are required to be covered.

The contributions of their work include [14]: a method by which to statistically determine the probability that an arc (unit of execution), i.e., elements of $E_C(D)$, will be covered, a method for routing the execution of a test suite from node v_s (start of test execution) to v_e (end of test execution) such that the optimal number of elements of $E_C(D)$ are covered, and a method that detecting unfeasible paths in the tests based on the constraints of the arcs (units of execution).

Figure 3-3 shows an example system D represented in a graph. Eight nodes (execution states) are labeled arbitrarily 0 through 7, and nodes v_s and v_e are the start and end points of the test. Arcs a through k are the units of execution of the system and each has a calculated probability of covering elements of $E_C(D)$, and constraints which it must satisfy during execution. Test suites maximizing the number of elements covered in $E_C(D)$ are generated by routing feasible (with respect to the constraints) paths between v_s and v_e.

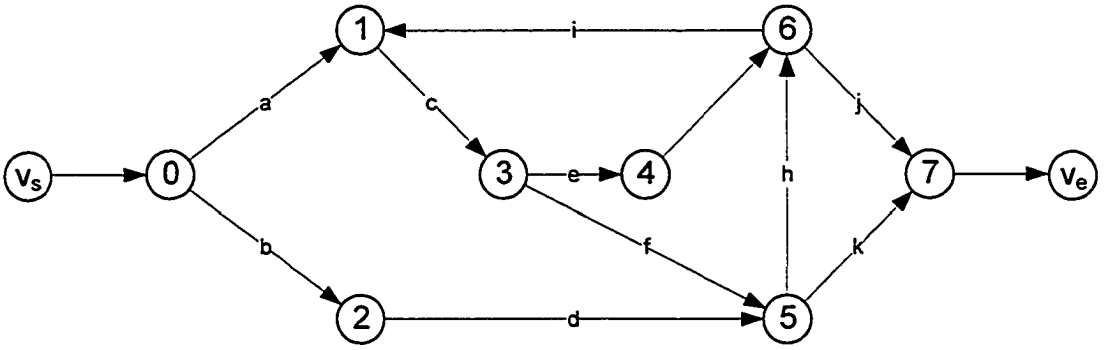


Figure 3-3: A Directed Graph with a Start Node and End Node

In relation to the work in this thesis, the technique of Denise, Gaudel and Gouraud [14] is an example use of combinatorial optimization in automated test suite generation.

Chapter 4
COTS Component Test Strategy

This chapter presents a testing strategy that fulfills all of the requirements outlined in the COTS component testing requirements presented in Section 2.3: 1) a testing technique for component's without source code (including identification of information to be used by vendor and user), 2) testing criteria to satisfy a wide range of testing costs and fault detection effectiveness, and 3) a method for efficiently generating adequate test suites. The chapter includes: an overview of the strategy where the roles and responsibilities of the vendor and user are outlined (Section 4.1), an example `Queue` component used for illustrating the strategy (Section 4.2), an adaptation of the CSPE technique to COTS component testing (Section 4.3), and a description of how the user can tailor the approach to only exercise the component functionalities that he/she is interested in (Section 4.4). Automation of the strategy, and in particular the generation of adequate test cases, is the purpose of the next chapter.

4.1 Strategy Overview

COTS component testing requires a systematic approach to procuring test requirements, generating test suites and executing the test suites. The proposed strategy adapts a technique of Constraints on Succeeding and Preceding Events (CSPE), described in Section 3.1. CSPE is a black-box technique previously used in testing concurrent software [9], and shown to be effective in other testing contexts such as class unit testing [12]. Figure 4-1 is a UML activity diagram depicting the overview of the proposed COTS

component testing strategy. It illustrates the roles and responsibilities of the component vendor and user (the two swim lanes of the activity diagram) in terms of the information that is generated and used for component testing purposes. Recall that in a UML activity diagram [21] rounded boxes denote activities, squares boxes denote objects (data, documents, or artifacts in our case), plain arrows denote control flow, and dashed arrows denote object flow.

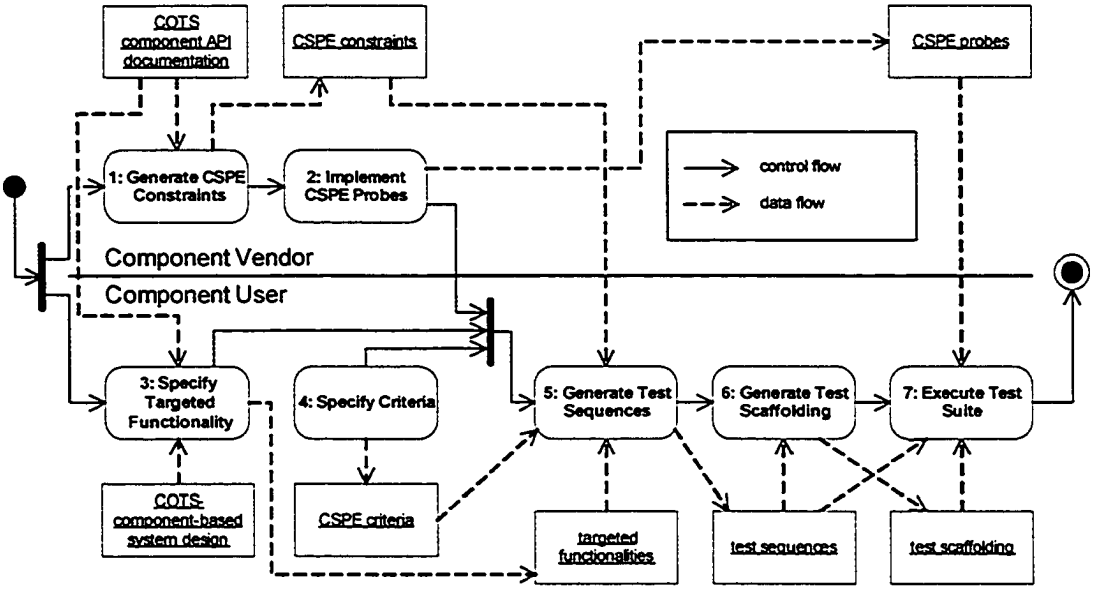


Figure 4-1: COTS Component Testing Strategy Overview (UML Activity Diagram)

The COTS component testing strategy overview depicted in the activity diagram of Figure 4-1 contains 8 objects (artifacts or data) and seven activities, which are described in Sections 4.1.1 and 4.1.2, respectively. We then discuss which activities in the strategy are addressed in this thesis (Section 4.1.2).

4.1.1 Artifacts

The following is a description of the eight artifacts of the COTS testing strategy depicted in the activity diagram of Figure 4-1. The dashed arcs of the activity diagram denote dependency, and in this context represent the flow of data, directed from the producing activity to the data, or from the data to a consuming activity.

COTS component API documentation is provided by the component vendor to the component user for use in designing a component-based system. The documentation includes method preconditions and postconditions which are used to derive the CSPE constraints.

CSPE constraints are produced by the vendor and then used by the component user to derive the test sequences and test scaffolding. They are also used by the component vendor to implement CSPE probes (defined in a bit).

CSPE criteria are the testing criteria that specify how much testing based on the CSPE constraints should be done.

COTS-component-based system design is the design of the software system developed by the user in which the COTS component is being integrated; along with the COTS component documentation it is used to specify the functionality of the component that will be under test.

Targeted functionalities are the subset of the COTS component functionalities that are actually used by the user's component-based system. These allow the component user to minimize test suite size by limiting the component testing to these only the functionalities that are of interest.

Test sequences are the actual method call sequences that are used to test the COTS component.

Test scaffolding represents the implementation of the test suites based on the test sequences. Test scaffolding includes: test drivers, test inputs, and test oracles.

CSPE probes are built-in methods provided by the component vendor and used by the component user in order to control and observe the component when needed during testing. In the context of CSPE-based testing, these may be used to observe the CSPE constraint predicates involved in Possibly Valid constraint types.

4.1.2 Strategy Steps

The following is the description of the seven activities of the COTS component testing strategy depicted in the activity diagram of Figure 4-1. The solid arcs of the activity diagram represent the control flow, i.e., the sequence in which the activities are completed (also denoted in the numbering of the activities). The two horizontal swim lanes indicate which activities are the responsibility of the component vendor (2 of the 7

activities) and which are of the component user (remaining 5 of the 7). In the special notation found within, the activity diagram also specifies that in order to proceed with activity 5 (Generate Test Sequences), the two vendor activities 1 and 2 (Generate CSPE Constraints and Implement CSPE Probes), as well as the two first user activities 3 and 4 (Specify Component Partition and Specify Criteria) must be completed. The activities of the strategy are listed in the order in which they would normally be followed.

Activity 1 Generate CSPE Constraints: Component method preconditions and postconditions, in the form of OCL constraints, are used to derive the CSPE constraints. Since the component vendor has access to the component source code and design documentation required to derive the details of the constraints, they perform this step of the strategy. If the OCL pre and post condition are not available (not part of the component vendor practices), other source of information, such as plain textual documentation, can be used.

Activity 2 Implement CSPE Probes: Based on the CSPE constraint predicates, CSPE probes (built-in methods) are added to in the component to support control and observability of the component states that are part of the CSPE constraint predicates.

Activity 3 Specify Criteria: Based on the testing requirements and constraints, the component user specifies the CSPE-based testing criteria (see Section 4.3.2).

Activity 4 Specify Targeted Functionality: Using the COTS component API documentation and the COTS-component-based system design, the component user specifies the targeted component functionality. The specification of the targeted functionality allows the component user to test only the parts of the component that are actually used.

Activity 5 Generate Test Sequences: The CSPE constraints, CSPE-based testing criteria, and targeted component partition are used to generate test sequences. Test sequences are generated such that the CSPE-based testing criteria are satisfied (adequate test cases) at a minimum testing cost.

Activity 6 Generate Test Scaffolding: The generated test sequences (adequate test cases) are then used to generate the test scaffolding (drivers, inputs, oracles).

Activity 7 Execute Test Suite: The complete test suite is then executed with the support of the CSPE probes.

4.1.3 Discussion

The COTS component test strategy forms an important contribution of this thesis. It captures the framework to be used when reasoning on component vendor and user responsibilities to ease component testing in the context of a particular component deployment (by an individual user). It is worth noting that this framework, although described in the specific context of CSPE-based testing, is generic enough and can be

used with other component oriented testing techniques. Although vendors and users may consider another testing technique, the following activities are still required: the vendor must provide testing information (activity 1) and built-in test support (activity 2); the user must be able to target a specific subset of the component functionalities and use the information provided by the vendor to generate test cases from testing criteria and execute them (activities 3 through 7).

Addressing all the issues (providing strategies for all activities) is a long term effort and this thesis is only concerned with a subset of them. In this thesis we address the definition of component testing criteria by adapting CSPE criteria, the definition of schemes used in specifying targeted functionality of the component, and the generation of optimal adequate test sets.

Issues that will be the purpose of future work are: the automation of the derivation of CSPE constraints from OCL contracts or similar artifacts; the automation of test scaffolding generation; and definition of CSPE probe construction and use.

4.2 Example `Queue` Component

The `Queue` class example was used in [25] to demonstrate how the CSPE technique could be adopted in the context of class unit testing. This `Queue` class example is reused, with some modifications, in this thesis to illustrate the component testing technique. The `Queue` class can be regarded as a simple component. Unlike the `Queue` class, the `Queue` component requires deployment in a component container before it is exercised through the *provides* interfaces, shown in Figure 4-2.

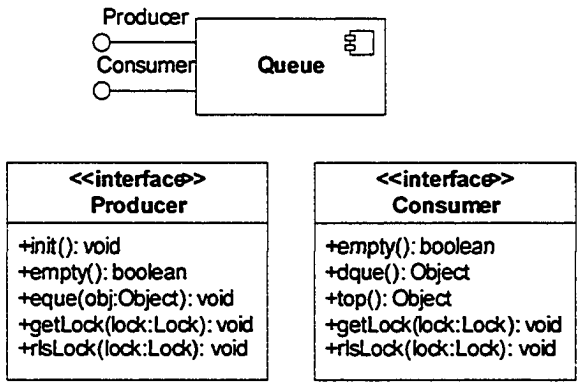


Figure 4-2: Queue Component UML 2.0 Class Diagram, from the Perspective of the Component User

The `Queue`, depicted in Figure 4-2, is a component that wraps and provides access to a linked list of arbitrary objects. The `Queue` component requires initialization before use and has virtually limitless capacity. The `Queue` component's seven interface methods are `init`, `empty`, `eque`, `dque`, `top`, `getLock`, and `rlsLock`. Method `init` initializes the `Queue` component before it can be used, `empty` checks if the `Queue` is empty, `eque` and `dque` add and remove an item to the queue respectively, and `top` fetches the last item off the queue without removing it.

The `Queue` component exposes two provides interfaces: `Producer`, and `Consumer`. Each is an interface to a subset of the seven interface methods of the component. The `Producer` interface is meant to be used by a client program that initializes the `Queue` component and en-queues to it, while the `Consumer` interface is meant to be use by a client program that checks for the status of the queue and de-queues from it. Some interface methods are exclusive to each of the interfaces (`eque`, `dque`), while others are accessible from both (`empty`, `getLock`, `rlsLock`).

Methods `getLock` and `rlsLock`, not found in the version of `Queue` in [25], are used to get and release a synchronization lock on the `Queue` instance. Given that the `Queue` implementation is not thread safe, `getLock` and `rlsLock` are used in a multi-threaded environment to assure synchronous access to the `Queue` component; methods that modify the `Queue` component, `eque` and `dque`, require that the `Queue` component be in either an unlocked state (for a single-threaded client), or that the calling thread hold the current lock (for a multi-threaded client).

After a client thread invokes the `getLock` method, the private data member `lock` holds the `Lock` instance belonging to that client thread; the client thread that owns the `Lock` object then has exclusive access to the `eque` and `dque` methods. The client thread that owns the `Lock` object must first release the lock, by invoking `rlsLock()` before other threads can modify the `Queue` instance. These two methods have been added to better illustrate *Possibly Valid* constraints.

The design of the `Queue` component from the perspective of the component vendor is shown in the class diagrams of Figure 4-3 and the interface methods of the `Queue` component are contractually specified through pre- and postconditions listed at the end of this section, in Table 4-1.

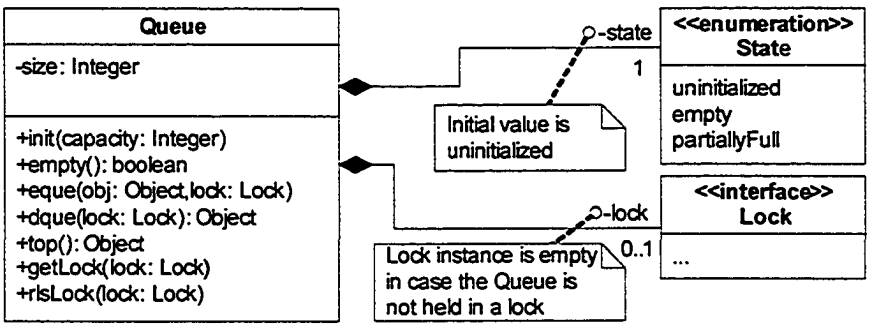


Figure 4-3: Queue Component UML 2.0 Class Diagram, from the Perspective of the Component Vendor

Enumeration `State` indicates the possible states of any `Queue`: `uninitialized`, before the queue's data structure has been initialized via the `init` method; `empty`, after the queue's data structure has been initialized via the `init` method but is empty; `partiallyFull`, after the queue has had object en-queued and is not empty. The `Queue` state is defined in a `state` private data member of the `Queue` and its initial enumeration value is `uninitialized`. The `Queue` class may have an associated instance of a `Lock` object owned by the client thread that has the current lock; the client thread that owns the `Lock` of the synchronized `Queue` is denoted in the OCL as `currentThread`[3].

[3] `currentThread` is the shorthand notation of a construct specific to the given programming environment. For example, in the standard Java synchronization mechanism getting `currentThread` would entail calling the static method `currentThread()` of class Thread.

Table 4-1: Queue Component Interface Method OCL Contracts

context Queue::init(c:Integer) **pre :** state = #uninitialized **post:** state = #empty	**context** Queue::top():Object **pre :** state <> #uninitialized and state <> #empty **post:** result = top()

context Queue::empty():Boolean
pre : state <> #uninitialized
post: if state = #empty then
result = true
else
result = false
endif
	context
	Queue::getLock(l:Lock):Object
	pre :
	post: if lock->isEmpty()@pre then
	lock = currentThread.lock
	and
	result = lock
	endif

context Queue::eque(obj:Object,l:Lock) **pre :** state <> #uninitialized and lock = currentThread.lock **post:** size = size@pre + 1 and top() = obj	**context** Queue::rlsLock(l:Lock):Object **pre :** lock = currentThread.lock **post:** lock->isEmpty()

context Queue::dque(l:Lock):Object
pre : state <> #uninitialized
and
state <> #empty
and
lock = currentThread.lock
post: size = size@pre - 1
and
result = top()@pre
and
if size = 0 then
state = #empty
else
state = #partiallyFull
endif	

4.3 Adaptation of CSPE to COTS Component Testing

The Constraints on Succeeding and Preceding Events (CSPE) technique is used to test COTS component without source code or design documentation. Recalling the

requirements for a COTS component testing strategy from Section 2.3.1, CSPE-based testing allows for: 1) a systematic method by which testing metadata is derived, 2) a systematic method for using the derived metadata in generating test suites, and 3) a set of criteria, covering a wide cost range, according to which tests suites are assessed for adequacy. CSPE constraints is adapted by reformulating the definition of CSPE constraints in the context of COTS component testing (Section 4.3.1), and redefining CSPE test adequacy criteria to better suit COTS component testing by addressing the issues identified in Section 3.2.2 (Section 4.3.2).

4.3.1 CSPE Constraints

The definition of CSPE constraints by Karçali, Daniels and Tai in [9] and [12], and reviewed in Section 3.2.1 of this thesis, requires re-formulation in adapting the technique for use in COTS component testing. The re-formulation of CSPE constraint type notations and definitions is done to address the points made at the end of Section 3.2.1: inconsistency in the constraint notation, and a redundant constraint *Possibly Invalid.* In addition to re-formulation of the CSPE constraint definition, this section also includes a brief description on how OCL may be used for deriving CSPE constraints.

4.3.1.1 Types of Constraint and Constraint Notation

Table 4-2 contains the re-formulation of CSPE constraints for use in context of COTS component testing. Note that the differences between the definitions found in Table 4-1 and those found in [9, 12] (reviewed in Section 3.2.1) are: change to a more consistent notation, and removal of a constraint type *Possibly Invalid.* Additionally, all of the

definitions are specific to component testing, i.e., events are invocations of interface methods (whereas, they were synchronization events in the context of concurrent programs and class methods in the context of class unit testing).

The new CSPE constraint notation, used in Table 4-2, is more consistent than the previous notation found in [9, 12]. The previous notation consisted of four different forms, one for each of the constraint types: *a*[M1; → M2] for *Always Valid*, ~[M1; → M2] for *Never Valid*, *pT*[M1; → M2]K for *Possibly Valid*, and *pF*[M1; → M2](*not* K) for *Possibly Invalid*. The new constraint notation defined in this thesis consists of a tuple of arity three: (M1,M2,P) where M1 and M2 are the preceding and succeeding interface methods, and P is the constraint predicate. This new notation is used to express all three different types of CSPE constraints such that the literal value `true` is used in place of P for *Always Valid* constraints, the literal value *false* is used in place of P for *Never Valid* constraints, and an actual constraint predicate is assigned to P for *Possibly Valid* constraints.

The second difference in re-formulation of the CSPE constraint definition is in the removal of the *Possibly Invalid* constraint type. As already described in Section 3.2.1, *Possibly Invalid* constraints are redundant since they can be expressed with a *Possibly Valid* constraint and the complement of the predicate P. Table 4-2 contains the three constraint types as defined in this thesis, including the notation and definition specific to the component testing (events are interface methods and predicate P is based on the state of the component).

Table 4-2: Re-formulated CSPE Sequence Constraints

Constraint Type and Notation	Definition
Always Valid (M1,M2,true)	Invoking the succeeding interface method M2 immediately after the given preceding interface method M1 is always valid. This happens when the postcondition of the preceding interface method M1 implies the precondition of the succeeding interface method M2. The constraint is denoted as (M1,M2,true), where true denotes the always valid nature of the constraint.
Never Valid (M1,M2,false)	Invoking the succeeding interface method M2 immediately after the given preceding interface method M1 is always invalid. This happens when the postcondition of the preceding interface method M1 contradicts the precondition of the succeeding interface method M2. The constraint is denoted as (M1,M2,false), where false denotes the never valid nature of the constraint.
Possibly Valid (M1,M2,P)	Invoking the succeeding interface method M2 immediately after the given preceding interface method M1 is valid only when the predicate P is satisfied, and invalid otherwise. This happens when the postcondition of the preceding method M1 does not imply the precondition of the succeeding method M2, but the conjunction of the preceding method's postcondition and the predicate P (state of the component derived from its interface methods' pre- and postcondition) implies the precondition of the succeeding method.

Additionally, since testing sequences begin with a single interface method invocation to an instantiated component, CSPE constraints without preceding interface methods are required. These CSPE constraints involve only one interface method M2, the succeeding

interface method, and in place of the preceding method we use a *null* interface method denoted as #. Below are examples of the notation for the different types of CSPE constraint. The example CSPE constraints are a sample derived from the OCL contracts of the Queue component found in Table 4-1.

```
c1=(#,init,true),
c8=(init,init,false),
c10=(init,eque,p1),
```

, where

```
p1=(lock->isEmpty()) or (lock = currentThread.lock)
```

Constraint c1 with no preceding method and succeeding method init is of an *Always Valid* since init is the only method that can begin a test sequence; recall that # denotes the *null* interface method, i.e., no preceding method. Constraint c8 with preceding method init and succeeding method init is *Never Valid* since the init method can only be called once in the lifetime of the instantiated Queue component. Constraint c10 with preceding method init and succeeding method eque is *Possibly Valid* and is valid when p10 evaluates to *true* and invalid otherwise; this happens when the Queue has no lock or when its lock is owned by the accessing client.

Although details of how predicates are implemented and used in the component test strategy are not part of this thesis, they are discussed briefly here. The CSPE predicates exposed to the component user do not necessarily represent the internal (implementation) states of the component. Choosing which internal states of the component are exposed, how they are exposed, and how they are documented is up to the discretion of the

component vendor. For example, the component vendor may choose to directly expose the internal states of the component if they feel they do not represent proprietary information, and also document these in detail so that the use can use them effectively. In a converse example, the component vendor may choose to abstract internal states, aggregate multiple internal states, or not represent some internal states at all in the exposed predicates. In this example the component vendor limits the provided predicate in order to protect their proprietary information, and may even provide anonymous predicates (referred to only by their literal, e.g., "p_3") without much documentation.

4.3.1.2 Deriving Constraints

CSPE constraints are modeled from the documentation of the interface methods of the component. For a set of component interface methods *IM* there should be one CSPE constraint for each ordered pair of interface methods, and one pair where the interface method is the first method of the sequence (and the preceding method is the null # method). Precisely, for a set of CSPE constraint *C*, the size of the set *C* is defined as

$$|C| = |IM|^2 + |IM|$$

, where the $|IM|^2$ term accounts for the constraint of ordered pairs of methods and the $|IM|$ terms accounts for the constraints with a preceding null method #.

CSPE constraints are derived from the interface methods' preconditions and postconditions. The interface methods' precondition dictates what conditions must be satisfied before the interface method is invoked, and the interface method's postcondition defines what condition holds after the interface method invocation. The postcondition of

a preceding method M1 and the precondition of a succeeding method M2 are used to derive the CSPE constraints according to the definitions of the three types of CSPE constraints in Table 4-1.

One of the benefits of using the CSPE technique is that CSPE constraints may be derived from a number of different sources such as: contracts in OCL [20] (or any formal language), sequence and collaboration diagrams, state charts, or even partially from textual documentation of how the component is to be used. Therefore, it can be used in different contexts and a variety of situations, and the constraints can be generated by the component vendor or user. We will assume in this thesis that contracts are expressed in the Object Constraint Language (OCL) and are used to derive CSPE constraints.

In order to get a complete list of CSPE constraints (one for each ordered pair of interface methods, and one for # and each interface method) a systematic method for deriving the type of constraint and the constraint predicate is required. An exact method is not part of this thesis, it is a problem for future work, but guidelines for deriving the three possibly CSPE constraints types from OCL are provided. The guidelines are presented below along with examples based on the OCL contracts of the Queue component found in Table 4-1, and also include a special case of deriving constraints with a preceding null method #.

In deriving the CSPE constraint every ordered pair of interface methods is identified, then for each pair three possible cases considered (one for each of the different CSPE constraints of Table 4-2). An exception to this are the constraints with a preceding null

method #, in which case one such constraint is required for each method in the component interface.

First case is of the *Always Valid* constraint type and the ordered pair of Queue interface methods init and emtpy. For this constraint type, the postcondition of the preceding method needs to imply the precondition of the succeeding method. The postcondition of init reads state = #empty, and the precondition of empty reads state <> #uninitialized. Since the former implies the latter, invoking empty after init is always valid. The constraint is identified as *Always Valid* and denoted as (init,emtpy,true).

Second case is of the *Never Valid* constraint type and the ordered pair of Queue interface methods init and dque. For this constraint type, the postcondition of the preceding method needs to contradict the precondition of the succeeding method. The postcondition of the init method reads state = #empty, and a conjunction of the precondition of the dque method reads state <> #empty. Since the former contradicts the latter, invoking dque after init is always invalid. The constraint is identified as *Never Valid* and denoted as (init,dque,false).

Third case is of the *Possibly Valid* constraint type and the ordered pair of Queue methods dque and top. This constraint is more difficult since it involves deriving the predicate P (state of the component under which the invocation of top is valid), which in conjunction with the postcondition of the preceding method will imply the precondition of the succeeding method. An excerpt of a conjunction of the precondition of the dque

reads "`if size = 0 then state = #empty`", and a conjunction of the precondition of `top` reads `state <> #empty`. From this we deduce that the invocation of `top` after `dque` is valid when `size <> 0`, and assign this to `p5` (found in Table 4-4). The constraint is identified as *Possibly Valid* and denoted as (`dque, top, p5`).

The special case to consider is when the preceding method is the null method `#`, i.e., the constraint (and therefore its succeeding method) is the first in the test sequence. The null method represents the newly instantiated state of the component and therefore has no postcondition per se. The constraint type is derived from the precondition of the succeeding method and the component documentation that describes the state of the newly instantiated component. This means that if the vendor does not provide already derived CSPE constraints, they need to provide the user with enough documentation outside of OCL to derive such constraints.

An example of this is the use of the `init` method as the first method in the `Queue` test sequence. The component documentation (UML diagram of Figure 4-3) specifies that the newly instantiated component has a `state` set to `uninitailized`. Since this implies the precondition of the `init` method, which reads `state = #uninitialized`, invoking `init` on a new component is always valid. The constraint is identified as *Always Valid* and denoted as (`#, init, true`).

Table 4-3 contains all of the `Queue` component's CSPE constraints (`c1` through `c56`), and Table 4-4 contains all of the related constraint predicates (`p1` through `p6`).

Table 4-3: Queue Component Method Sequence Constraints

c1=(#,init,true)	c20=(empty,getLock,p2)	c39=(top,dque,p1)
c2=(#,empty,false)	c21=(empty,rlsLock,p3)	c40=(top,top,true)
c3=(#,eque,false)	c22=(eque,init,false)	c41=(top,getLock,p2)
c4=(#,dque,false)	c23=(eque,empty,true)	c42=(top,rlsLock,p3)
c5=(#,top,false)	c24=(eque,eque,p1)	c43=(getLock,init,false)
c6=(#,getLock,p2)	c25=(eque,dque,p1)	c44=(getLock,empty,true)
c7=(#,rlsLock,p3)	c26=(eque,top,true)	c45=(getLock,eque,p1)
c8=(init,init,false)	c27=(eque,getLock,p2)	c46=(getLock,dque,p4)
c9=(init,empty,true)	c28=(eque,rlsLock,p3)	c47=(getLock,top,p5)
c10=(init,eque,p1)	c29=(dque,init,false)	c48=(getLock,getLock,false)
c11=(init,dque,false)	c30=(dque,empty,true)	c49=(getLock,rlsLock,p6)
c12=(init,top,false)	c31=(dque,eque,p1)	c50=(rlsLock,init,false)
c13=(init,getLock,p2)	c32=(dque,dque,p4)	c51=(rlsLock,empty,true)
c14=(init,rlsLock,p3)	c33=(dque,top,p5)	c52=(rlsLock,eque,p2)
c15=(empty,init,false)	c34=(dque,getLock,p2)	c53=(rlsLock,dque,p3)
c16=(empty,empty,true)	c35=(dque,rlsLock,p3)	c54=(rlsLock,top,p5)
c17=(empty,eque,p1)	c36=(top,init,false)	c55=(rlsLock,getLock,p2)
c18=(empty,dque,p4)	c37=(top,empty,true)	c56=(rlsLock,rlsLock,false)
c19=(empty,top,p5)	c38=(top,eque,p1)	

Table 4-4: Queue Component Method Sequence Constraint Predicates

Predicate Literal	Constraint Predicate
p1	(lock->isEmpty()) or (lock = currentThread.lock)
p2	(lock->isEmpty())
p3	(lock->isEmpty()) and (lock = currentThread.lock)
p4	(size <> 0) and ((lock->isEmpty()) or (lock = currentThread.lock))
p5	(size <> 0)
p6	(lock = currentThread.lock)

4.3.2 CSPE-based Test Adequacy Criteria

CSPE-based test adequacy criteria are used to generate and validate CSPE-based test suites tailored to a testing budget and required component reliability. CSPE-based testing criteria need to account for CSPE constraint and their related CSPE constraint predicates. The CSPE testing criteria defined by Daniels and Tai in [12] (reviewed in Section 3.2.2) only address CSPE constraints, and have ambiguous definitions.

Our CSPE-based test adequacy criteria are defined in two stages: first, CSPE constraint criteria specify which type of constraints are required to be covered in the component tests; next, in order to account for the different ways to satisfy CSPE constraint predicates, combinations CSPE constraint criteria and constraint predicate criteria specify how the constraint predicates of the *Possibly Valid* type criterion are to be covered.

The CSPE constraint criteria are based on the three different types of CSPE constraints: *Always Valid*, *Never Valid*, and *Possibly Valid*. The CSPE criteria are defined with respect to coverage of the three CSPE constraint types. Table 4-5 contains the definitions of the CSPE constraint criteria in the context of COTS component testing, i.e., events are invocations on the interface methods of the component.

Table 4-5: CSPE Constraint Criteria

Criterion (shorthand)	Definition
Always Valid Coverage (A)	Requires that every *Always Valid* constraint be covered at least once.
Always/Possibly Valid Coverage (AP)	Requires that every *Always Valid* constraint be covered at least once, and every *Possibly Valid* constraint be covered according to a CSPE constraint predicate coverage criterion. Only the subset T_p^{true} of the test cases required by the predicate testing criteria is used.
Never Valid Coverage (N)	Requires that every *Never Valid* constraint be covered at least once.
Never/Possibly Valid Coverage (NP)	Requires that every *Never Valid* constraint be covered at least once, and every *Possibly Valid* constraints be covered according to a CSPE constraint predicate coverage criterion. Only the subset T_p^{false} of the test cases required by the predicate testing criteria is used.
Always/Never/Possibly Valid Coverage (ANP)	Requires that every *Always Valid* and every *Never Valid* constraints be covered at least once, and *Possibly Valid* constraints be covered according to a CSPE constraint predicate criterion.

Always Valid Coverage (A) and *Always/Possibly Valid Coverage* (AP) criteria require covering constraints that are valid; and AP subsumes A (criteria subsumption is described in Section 4.1.3). *Never Valid Coverage* (N) and *Never/Possibly Valid Coverage* (NP) criteria require covering constraints that are invalid; and NP subsumes N. The *Always/Never/Possibly Valid Coverage* (ANP) covers all of the constraints and subsumes all other criteria. Figure 4-4 depicts the subsumption hierarchy within the CSPE constraint criteria.

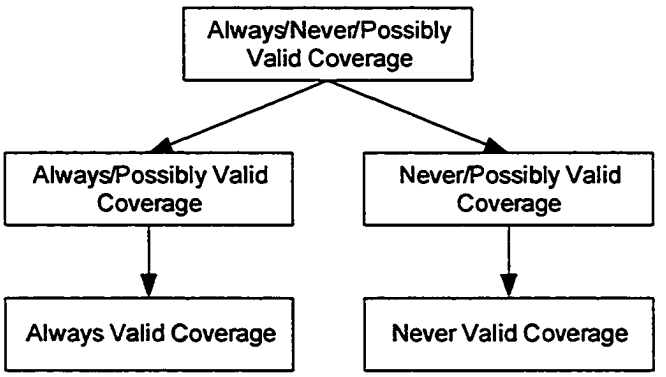


Figure 4-4: Subsumption Hierarchy for CSPE Constraint Criteria

Criteria that require only the *Always Valid* or *Never Valid* constraints (criteria A and N) are considered complete since they do not involve specifically testing CSPE constraint predicates. Note that test sequence generated according to these to criteria may include *Possibly Valid* constraint (test sequences may not be feasible without them), but in this case the CSPE constraint predicate is set arbitrarily. In the case of all the other criteria (AP, NP, and ANP), the constraint predicate of a *Possibly Valid* constraint needs to be properly tested.

For a *Possibly* Valid constraint, given that its CSPE constraint predicate consists of more than one clause, there are many ways to test the predicate and therefore many ways to cover that constraint. If we want testing to be complete in terms of exercising constraint predicates, the *Possibly Valid* constraint must be tested once for each of the test cases of its predicate. The test cases of the CSPE constraint predicates are derived according to the logical testing techniques defined in Section 3.3.

In order to thoroughly test *Possibly Valid* constraints, and allow for the tailoring of the effort spent on each CSPE constraint predicate, we propose the use of logical testing techniques reviewed in Section 3.3. The four logical testing techniques used in testing of CSPE constraint predicates are: *Predicate Coverage* (PC), *Combinatorial Coverage* (CoC), *Implicant Coverage* (IC), and *Prime Implicant Coverage* (PIC).

The three CSPE constraint criteria (A, AP, N, NP, and ANP) are combined (in a cross product of the two sets) with the four logical testing criteria to produce the four complete CSPE-based test adequacy criteria shown in Table 4-6.

Table 4-6: Complete Definition of CSPE-based Test Adequacy Criteria

CSPE Constraint Criteria	CSPE Constraint Predicate Coverage	Complete CSPE Criteria
Always Valid (A)	N/A	Always Valid (A)
Always/Possibly Valid (AC)	Requires that all *Possibly Valid* constraints are covered once for each tuple $t \in T_p^{true}(tc)$, where *tc* is one of the predicate criteria: PC, IC, PIC, CoC.	Predicate Coverage (AP–PC)
		Implicant Coverage (AP–IC)
		Prime Implicant Coverage (AP–PIC)
		Combinatorial Coverage (AP–CoC)
Never Valid (N)	N/A	Never Valid (N)
Never/Possibly Valid (NP)	Requires that all *Possibly Valid* constraints are covered once for each tuple $t \in T_p^{false}(tc)$, where *tc* is one of the predicate criteria: PC, IC, PIC, CoC.	Predicate Coverage (NP–PC)
		Implicant Coverage (NP–IC)
		Prime Implicant Coverage (NP–PIC)
		Combinatorial Coverage (NP–CoC)
Always/Never/Possibly Valid (ANP)	Requires that all *Possibly Valid* constraints are covered once for each tuple $t \in T_p(tc)$, where *tc* is one of the predicate criteria: PC, IC, PIC, CoC.	Predicate Coverage (ANP–PC)
		Implicant Coverage (ANP–IC)
		Prime Implicant Coverage (ANP–PIC)
		Combinatorial Coverage (ANP–CoC)

The definitions of the *Always/Possibly Valid* (AP) and *Never/Possibly Valid* (NP) criteria in Table 4-5 state that only a subset of the test cases required by the predicate testing criteria is used. In the case of the AP criterion the subset includes only those predicate test cases that result in the predicate evaluating to *true*, and conversely in the case of the NP criterion the subset includes only those that result in the predicate evaluating to *false*. These subsets correspond to the T_p^{true} and T_p^{false} subsets as defined in Section 3.3.

For instance, referencing back to the truth table Table 3-3, the CSPE constraint criteria AP would requires that the subset of clauses' Boolean value combinations where the predicate *p* is expected to evaluate to *true* (combinations 6, 7 and 8) be considered for testing; NP would require that only the subset of clauses' Boolean value combinations where the predicate *p* is expected to evaluate to *false* (combinations 1 through 5) be considered for testing; and ANP would require that all the clauses' Boolean value combinations be considered for testing.

4.4 Targeting Functionalities

Recall that one of the requirements that we identified for a component testing technique is to allow the component user to target the testing and only test the component functionality of interest, i.e., the component functionalities that are actually exercised in the component-based system being developed by the user. Indeed, with a large selection of COTS components to choose from and each catering to users with widely varying needs, the component user may select a component that provides more functionalities than what is actually required. It is thus important to be able to target only those

component functionalities that are of interest, i.e., to partition the component into what will be tested by the user and what will not.

In this thesis, three different schemes for targeting component functionalities are defined: *Required Method* (Section 4.4.1), *Required Constraint* (Section 4.4.2), and *Required Constraint Predicates* (Section 4.4.3). The three schemes are defined in a hierarchy of increasing precision allowing the component user to vary the scope of the targeted functionalities. They are used together in combination to enable the user to best tailor the tests. The *Required Method* scheme is the most course, and the *Required Constraint Predicate* the most precise.

Figure 4-5 shows an abstract model of how the three schemes are used to target the component functionalities (shown in gray) in terms of methods, constraints, and constraint predicates. The model is an excerpt of a COTS component under CSPE-based testing and includes: three methods (`m1`, `m2`, and `m3`) that are represented by rectangles, four constraints (`c1`, `c2`, `c3`, and `c4`) that are represented by circles, and a set of test cases for constraint predicates for three of the four constraints (smaller circles in the constraint circles. Note that the number of possible constraint predicate test cases depends on the selected predicate criterion. Note also, that constraint `c2` has no constraint predicate test cases because it is either an *Always Valid* (where the predicate is *true*) or a *Never Valid* (where the predicate is *false*) constraint: the other three are Possibly Valid constraints.

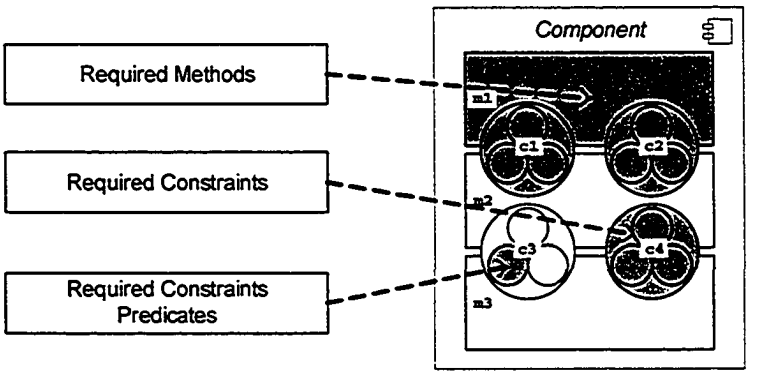


Figure 4-5: Component Partition Specification Schemes and Demonstration on Anonymous Component

The three schemes are used to target component functionalities by selecting a subset of CSPE constraints and CSPE constraint predicates that will be used in testing of the component. When targeting according to the *Required Methods* scheme, the user selects both a CSPE constraint criterion and a predicate criterion (Section 4.4.1). Two more restrictive schemes are also considered. Following the *Required Constraint* scheme, the component user selects specific constraints (they may not correspond to a CSPE criterion) as well as the accompanying predicate criterion (Section 4.4.2). With the *Required Constraint Predicates* scheme (Section 4.4.3), the component user selects actual test cases for testing constraints (they may not correspond to a CSPE constraint or predicate criterion). We end the section with a discussion on these schemes and provides examples (Section 4.4.4).

4.4.1 Required Methods Scheme

The *Required Methods* is the most coarse scheme for specifying component partitions. The component user specifies a subset of all of the component interface methods from the

component-based system design documentation. Formally, for a component with a set of interface methods $M = \{m_1, m_2, \ldots, m_n\}$, where $|M| = n$, the component user specifies a required set $M_R \subseteq M$.

This set of required component methods can be derived by the user simply according to his/her understanding of how the component will actually be used, i.e., which component interface method will be invoked. This information can be automatically derived from UML design documents showing in diagrams (e.g., sequence diagrams) how the component will be integrated in the component based system. Such a UML diagram is shown in Figure 4-6 for our Queue example.

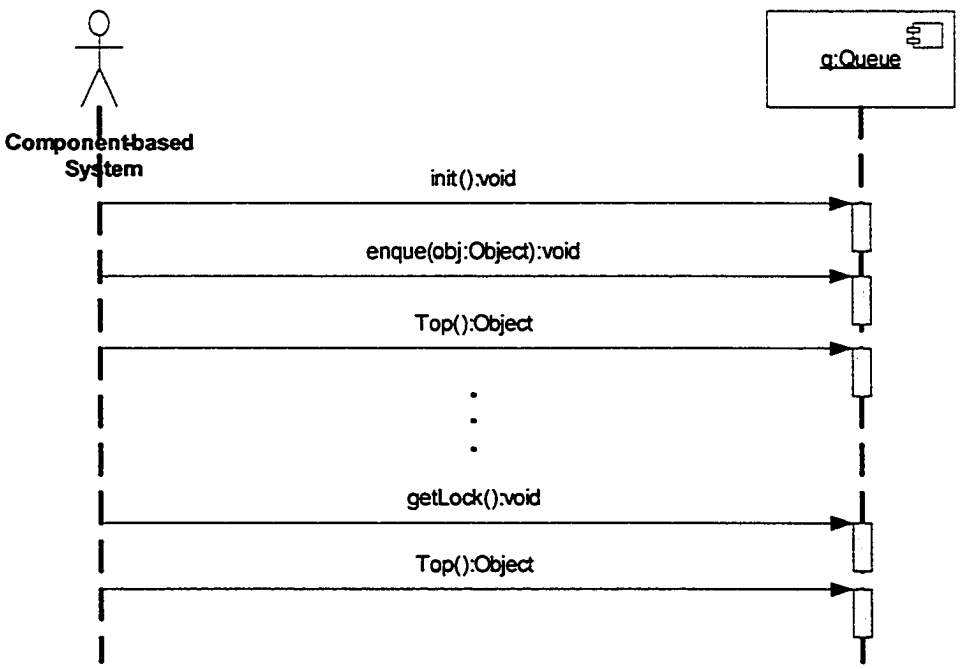


Figure 4-6: Example Design Documentation for a System that uses the Queue Component

Assuming that the sequence diagram of Figure 4-6 is part of the design documentation for a system that uses the `Queue` component, the user (or algorithm) identifies that the component methods in use are: `init`, `eque`, `top`, and `getLock`.

The *Required Methods* scheme then selects the subset of CSPE constrains used in testing, to which the CSPE constraint and predicate criteria are then applied. Specifically, from the set of all of the constraint for the component, only the constraints that involve any of the specified required methods are selected. Formally, identifying methods m_i and m_j as required methods, the required constraints (set C_R) are selected as follows:

$$\forall m_i, m_j \in M_R \cup \{\#\} \quad (m_i, m_j), (m_j, m_i) \in C_R .$$

Note that the formulation above does take into account the null method # which is always required: in order to test the component, the component must first be initialized.

For example, from the sequence diagram in Figure 4-6 and the set of all the `Queue` component's CSPE constraints of Table 4-3, and excerpt of the ones selected is `{(init,eque,p1), (init,top,false), (eque,eque,p1), ...}`.

4.4.2 Required Constraint Scheme

The *Required Constraint* is a more precise (and therefore restrictive) scheme for specifying component partitions. The component user selects, with no correlation to the CSPE constraint criterion, a subset of constraints $C_R \subseteq C$, where C is the set of all the component's CSPE constraints.

The CSPE constraints, which are basically sequences of pairs of the component's interface methods, can be specified manually from the component documentation or automatically from UML diagrams describing how the component is used. For example, from the sequence diagram of Figure 4-6, we can precisely select the constraints that will be required based on the sequence of method invocations. From the sequence of methods invocations [`init, eque, top, ... , getLock, top`], the following required constraints are selected: `c1=(#,init,true), c10=(init,eque,p1), c26=(eque,top,true), c47=(getLock,top,p5).`

4.4.3 Required Constraint Predicates Scheme

The *Required Constraint Predicates* is the most precise (and therefore most restrictive) of the defined schemes. The component user selects, with no correlation to the CSPE constraint or predicate criterion, the constraints and for each *Possibly Valid* constraint the exact constraint predicates. This enables the component user to specify the component partition they would like to test with the greatest detail.

The *Required Constraint Predicates* scheme allows the component user to specify for which combinations of predicate clause values the *Possibly Valid* constraints are tested. Formally, first, the user selects a subset of required constraints $C_R \subseteq C$ (e.g., using the previous scheme); then, for all the *Possibly Valid* constraints of that subset ($C_{PV} \subseteq C_R \subseteq C$), the user also selects a set of test cases T_p for each $c \in C_{PV}$.

For example, consider the constraint `c10=(init,eque,p1)`, a constraint in the selected subset C_{PV}. The predicate `p1` reads `(lock->isEmpty()) or (lock =`

`currentThread.lock`) and can be represented in Boolean expression $a \vee b$. Assuming that the user is specifically interested in testing this constraint when the `lock` is not empty and is held by the current thread (such that the predicate evaluates to *true*) the only selected test case for this *Possibly Valid* constraint would be T_{p1} = {(a=*true*, *b*=*true*)}.

4.4.4 Discussion and Example

The *Required Constraint Scheme* and *Required Constraint Predicate Scheme* are used to select CSPE constraints with no correlation to the CSPE constraint criteria (and constraint predicate criteria in the case of the latter) defined in Section 4.3.2. As intended in the proposed COTS component testing strategy (Section 4.1), CSPE constraint and predicate criteria are used in combination with targeted component functionalities to generate component tests, and the use of one does not preclude the other.

Also, it is possible to use combinations of the three targeting schemes, or use no schemes at all. By default, if no targeted functionalities are specified by the user, the entire component is tested. This is the same as selecting the entire set of methods as required methods, $M_R = M$. If the user does specify a subset of methods M_R (according to the *Required Methods Scheme*), such that only the constraints that involve methods of M_R are selected, s/he may also select a few which they particularly want tested but in which methods not in M_R are involved (Required Constraint scheme). Similarly when specifying tests with a subset of constraints C_R (according to the *Required Constraints Scheme*), the user may target the predicates of some of the constraint with specific test cases (Required Constraint Predicate Scheme).

To better explain the three targeting schemes and the discussion above, a complete example of application of the CSPE criteria and targeting schemes based on the `Queue` component is provided. The scenario is the following: `Queue` component user would like to test only the `init` and `eque` interface methods of the component, and also the `dque` method but only when the queue has been locked and therefore only by the client that locked it. The user's component-based system uses the `Queue` as a sort of produce-only queue, where consume can be performed only by an administrator. The component user would like to be thorough and therefore tests the targeted functionalities according to the ANP-PIC criterion. The user selects the targeted functionalities according to the *Required Methods Scheme* (select methods `init` and `eque`) and the *Required Constraint Predicate* (select specific `dque` use). Below is the subset of constraint from Table 4-3, derived according to *Required Methods Scheme* and the ANP CSPE constraint criterion.

```
c1=(#,init,true)
c3=(#,eque,false)
c8=(init,init,false)
c10=(init,eque,p1)
c22=(eque,init,false)
c24=(eque,eque,p1)
```

Next, the predicate `p1` has to be tested according to the PIC constraint predicate testing criterion. The predicate `p1` reads `(lock->isEmpty()) or (lock = currentThread.lock)` and can be represented as Boolean expression $a \vee b$. A test set that satisfies the PIC criterion for predicate `p1` is:

$$T_{p1}(\text{PIC}) = \{(a=true,b=false), (a=false,b=true), (a=false,b=false)\}$$

Last, the user selects the specific constraint and constraint predicate that will test the dque method invoked by the administrator (i.e., a client that has the lock on the queue). For this the following constraint and constraint predicate test case are selected according to the *Required Constraint Predicate Scheme*: `c46=(getLock,dque,p4)`, where `p4 = (size <> 0) and ( (lock->isEmpty()) or (lock = currentThread.lock) )`, is expressed as $a \wedge b \vee c$ and tested with T_{p4} = {(a=*true*, *b=false*, *b=true*)}.

Chapter 5
Automation Framework

This chapter describes our approach for the automated generation of component CSPE based adequate test sets. We first provide an overview of our test sequence generation framework (Section 5.1) and then describe how the problem to be solved can be expressed as a graph problem (Section 5.2): more specifically, our problem is similar to the Directed Rural Postman Problem [15]. We then provide some common terminology related to graph theory (Section 5.3). Section 5.4 then details how we adapted a solution to the Directed Rural Postman Problem to our problem, and Section 5.5 provides a discussion on the approach.

5.1 Framework Overview

This thesis addresses part of the test suite generation depicted in Figure 5-1, and part of the larger strategy described in Section 4.1. The proposed method generates test sequences based on the component metadata and test specification. The component metadata is specified in the form of CSPE constraints and constraint predicates (Section 4.3.1). The test specification is provided in the form of the targeted component functionalities (Section 4.4) and test criterion (4.3.2).

The component metadata includes the component's OCL contracts, CSPE constraint, and the CSPE constraint predicates (use of which is described in Section 4.3.1.2). The test specification includes the targeted functionalities (Section 4.4) and the CSPE testing

criterion (Section 4.3.2). How these are provided in the framework and their format is an implementation detail (the prototype tool of Chapter 5 uses input files in XML format).

The generated test sequences are then use to generate the test scaffolding and execute the test suite. Deriving the test scaffolding (test driver, test inputs, and test oracle) is left for future work.

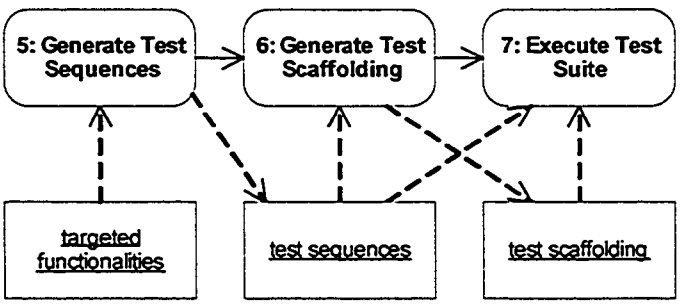


Figure 5-1: CSPE-based Test Automation Overview (excerpt of Figure 4-1)

5.2 Building CSPE Adequate Test Sets: A Graph-Based Problem

Let us consider the following abstract example (Figure 5-2) where a component has four interface methods (`m1` to `m4`) which executions are constrained by 20 CSPE *Always Valid*, *Always Invalid* and *Possibly Valid* constraints. Assuming the selected criterion is AP (Section 4.3.1), the test set must exercise all the *Always Valid* constraints as well as all the *Possibly Valid* constraints. One solution to finding an adequate test set is to adopt a tree based approach like the one presented in [25] (reviewed in Section 3.2.3). This is illustrated in Figure 5-3 (a) and results in 13 method executions (not accounting for #) since each tree path (from the root to the leaf node) is seen as a test case. However, even in this small example, test cases show some level of redundancy: e.g., method sequence

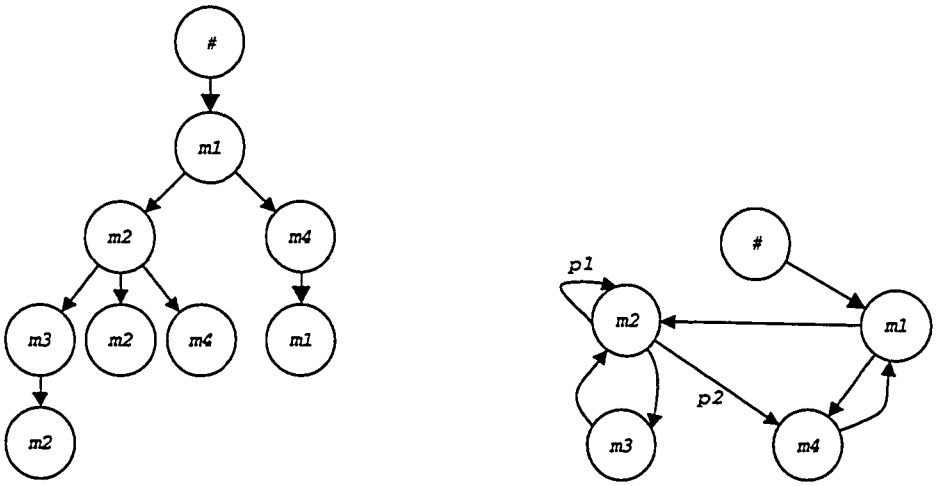
`(m1, m2)` is executed three times, in three different tests cases, i.e., for three different paths in the tree.

In order to avoid these redundancies, that can produce prohibitive test sets in more complex systems, the only solution is to take advantage of the cycles that are induced by the constraints. For instance, an adequate test set for the example in Figure 5-2 could contain the following unique test case (8 method executions instead of 13): `#, m1, m2, m2, m3, m2, m4, m1, m4`. Finding such a test case (or in general a set of test cases) is modeled by traversing a graph whose nodes are the component interface methods and arcs represent the CSPE constraints. In our case, the graph would be the one in Figure 5-3 (b): It contains five nodes (one for # and one for each of the component interface method), and arcs representing the *Always Valid* and *Possibly Valid* constraints. (Note that this is not the notation that we will eventually adopt but is only used for illustration purposes here.) In this graph, finding an adequate test set for criterion AP amounts to finding a path that goes through all the arcs.

Interface methods	Constraints			
	Always Valid	Never Valid		Possibly Valid
m1, m2, m3, m4	(#, m1, true)	(#, m2, false)	(m3, m4, false)	(m2, m2, p1)
	(m1, m2, true)	(#, m3, false)	(m3, m1, false)	(m2, m4, p2)
	(m1, m4 true)	(#, m4, false)	(m3, m3, false)	
	(m2, m3 true)	(m1, m3, false)	(m4, m2, false)	
	(m3, m2, true)	(m1, m1, false)	(m4, m3, false)	
	(m4, m1, true)	(m2, m1, false)	(m4, m4, false)	

Figure 5-2 Component Interface Methods and Constraints: Abstract Example

If on the other hand the selected criterion is A (Section 4.3.1), then an adequate test set can be made of the two following test cases: #, m1, m2, m3, m2 and #, m1, m4, m1. This corresponds to another kind of traversal of the graph where only certain arcs are taken, i.e., the ones for the *Always Valid* constraints. A shorter test set, in terms of number of method calls (i.e., number of arcs), can be: #, m1, m2, m3, m2, m4, m1, m4 (8 calls instead of 9 for the two test cases above). It is also adequate for criterion A (i.e., it exercises all the *Always Valid* constraints) but this time we allow arcs that are not required by the criterion to be used (arc from node m2 to node m4) in order to obtain a cheaper solution. Using in the solution test set arcs that are non-required by the selected criterion can be very helpful when trying to reduce the cost of an adequate test set.



(a) Tree-based Approach [25] **(b) Graph-based Approach**

Figure 5-3 Representing Constraints: Tree vs Graph

If we now turn our attention to criterion N, that requires that every *Never Valid* constraint be exercised, test cases must first ensure that the preceding methods of each *Never Valid* constraint be reached: for instance, in order to exercise constraint (m4, m3, false) in

the above example, a test case must first reach m4, for instance by executing sequence #, m1, m4, and then execute m3; the complete test case is #, m1, m4, m3. Once the *Never Valid* constraint is triggered and an exception is (hopefully) caught, the test case must end. Otherwise, if we continue, trying for example to cover constraint (m3, m2, true) and thus building test case #, m1, m4, m3, m2, we are not really exercising constraint (m3, m2, true) because method m3 has not executed successfully. Additionally, one could consider that a test case like #, m1, m4, m3, m2, m3, although not exercising constraint (m3, m2, true), exercises constraint (m2, m3, true). However, because m3 has not executed successfully we cannot foresee the result of executing m2 and then of executing m3 again. As a result, in case of a *Never Valid* constraint such as (m4, m3, false), the driver should start a new test case, starting again from node labeled #.

As we can see, building an adequate test set for a CSPE criterion is modeled by traversing a graph where some (required) arcs must be taken, and other (not required) arcs are optional. (It is even possible that required arcs be only reachable by traversing non-required arcs.) The required arcs are identified according to the selected component functionalities under test (Section 4.4) and the CSPE testing criterion (Section 4.3.2). This problem definition is exactly the definition of the Directed Rural Postman Problem (DRPP) [15]. In the DRPP only the required arcs need to be traversed at least once, and all the other arcs do not necessarily have to be traversed, but may be traversed if they are needed to create the shortest possible path in the solution to the problem. We will see precisely in Section 5.2.1 how we represent constraints as a graph, and illustrate this on

the Queue example in Section 5.2.2. We then discuss possible cost measures for arcs that can be used to identify what an optimal solution is (Section 5.2.3).

5.2.1 Graph Representation of CSPE Constraints

Our representation of component interface methods and CSPE constraints is a graph $G = [N,A]$ where nodes in N are component interface methods and arcs in A are CSPE constraints. More precisely, each component interface method involved in a CSPE constraint (including starting method labeled #) is represented by a node, labeled with the method name.

Each *Always Valid* constraint c = (*m1*, *m2*, *true*) is modeled as an arc from node labeled *m1* to node labeled *m2*, and the arc is labeled *c*. Each *Never Valid* constraint c = (m1, m2, false) is modeled as an arc from node labeled *m1* to an error node for method *m2*, labeled $\underline{\overline{\text{m2}}}$, and the arc is labeled *c*. There is only one way to leave an error node, which is to follow an arc from this node to node #. This arc is labeled '~' and represents the re-initialization of the component. In order to facilitate the identification of short test cases, such re-initialization arcs are added to all the nodes in the graph[4]. Then, during the traversal of the graph representing an adequate test set, those arcs can be taken if it means reducing the overall cost of the test set.

---

[4] For instance, assuming that the test case being constructed currently ends with method m_k and constraint (m_i, m_j, p) has to be exercised, it might be cheaper to exercise (m_k, #, true), that is the re-initialization arc, followed by a path from # to m_i (followed by m_j), than a path from m_k to m_i.

Each *Possibly Valid* constraint $c = (m1, m2, p)$ leads to two different arcs because both valid and invalid outcomes might need to be exercised according to the selected criterion. Additionally, *Possibly Valid* constraints are exercised possibly several times according to the selected predicate criterion, for different combinations of Boolean variable values leading to either *true* or *false* values of the predicate. As a consequence, for each selected combination of Boolean variable values resulting in a *true* value for the predicate there is one arc, labeled "*c* [*true*, *n*]" where *n* is the integer value of the Boolean variable value combination[5], from node labeled *m1* to node labeled *m2*. Similarly, for each selected Boolean variable value combination resulting in a *false* value of the predicate there is one arc, labeled "*c* [*false*, *n*]", from node *m1* to (error) node $\underline{\overline{m2}}$, where *n* is the integer value of the Boolean variable value combination[5].

5.2.2 The Queue Example

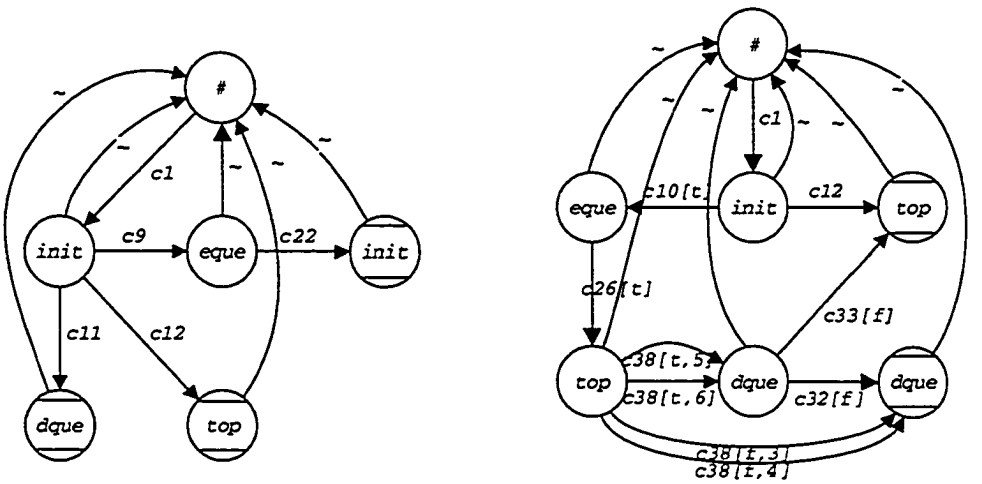
Graphs in Figure 5-4 are two different excerpts of the graph representing constraints for the `Queue` example. The first one, Figure 5-4 (a) illustrates the graph representation of *Always Valid* and *Never Valid* constraints, whereas the second, Figure 5-4 (b) illustrates the graph representation of *Possibly Valid* constraints.

In Figure 5-4 (a), as an example, *Always Valid* constraint `c9=(init, eque, true)`, with preceding method `init` and succeeding method `eque` is modeled as an arc

---

[5] If three Boolean variables are involved in the predicate and values 0, 1, and 1 (for the three variables respectively) is selected according to the predicate criterion, the integer representation is 3 (0.2^2 + 1.2^1 + 1.2^0).

directed out of node `init` and into node `eque`, labeled *c9*. Figure 5-4 (a) also shows three Never Valid constraints, namely constraints `c11=(init, dque, false)`, `c12=(init, top, false)` and `c22=(eque, init, false)`. Each constraint's succeeding method is modeled as an error node: $\overline{\text{dque}}$, $\overline{\text{top}}$, and $\overline{\text{init}}$, respectively. Those nodes have only one outgoing arc, a re-initialization arc labeled '~' and leading to node labeled #. As discussed before, and in order to facilitate the construction of cheaper test cases, nodes labeled `init` and `eque` also have such outgoing re-initializing arcs.



(a) – Always and Never Valid Constraints

(b) – Possibly Valid Constraints

Figure 5-4 Graph Representations of Constraints: the `Queue` Example

Figure 5-4 (b) additionally shows nodes and arcs for *Possibly Valid* constraints on one example, namely constraint `c38=(top, dque, p)` when `p` is:

```
(Queue.state <> State::empty)
and ( (lock->isEmpty()) or (lock = currentThread.lock) ).
```

Assuming that the selected predicate criterion is Prime Implicant Coverage (PIC) (Section 4.3.2), and considering that the predicate of constraint `c38` is of the form `A and (B or C)`, two combinations of Boolean variable values must be selected for the *true* value of the predicate, and two combinations of Boolean variable values must be selected for the *false* value of the predicate[6]. This results in two arcs from node `top` to node `dque`, labeled `c38 [t, 5]` and `c38 [t, 6]`, and two arcs from node `top` to node $\overline{\texttt{dque}}$, labeled `c38 [f, 3]` and `c38 [f, 4]`.

Note that constraints `(dqeue, dqeue, p)` and `(dque, top, p)` are *Possibly Valid* constraints. They are however represented as only two arcs in Figure 5-4 (b), between nodes `dque` and $\overline{\texttt{dque}}$, and `dque` and $\overline{top}$, as we decided to illustrate *Possibly Valid* constraints only on one example, constraint `c38`.

5.2.3 Arcs' Costs Measures

As succinctly mentioned before, the construction of an adequate test set from a selected CSPE constraint criterion should be driven by the cost of the test set. Indeed, several adequate test sets can be constructed and one (or an algorithm) should select the cheapest one. Additionally, solutions to the DRPP account for arc's cost values. One obvious cost

---

[6] Predicate $a \wedge (b \vee c)$ has a DNF form $(a \wedge b) \vee (a \wedge c)$. The inverse of the predicate also has a DNF form which is $\neg a \vee (\neg b \wedge \neg c)$. Applying the Prime Implicant criterion, for each expression, each prime implicant (they are also the implicants) should be set to *true* and the other implicant set to *false*. This leads to Boolean variable values combination of (*true,true,false*) and (*true,false,true*), that is integer values 6 and 5, for the predicate, and Boolean variable values combinations of (*false,true,true*) and (*true,false,false*), that is integer values 3 and 4, for the negation of the predicate.

measure that one can think of, and that has already been mentioned is to count the number of method calls in the test set. In terms of graph, this amounts to associating the cost of 1 to all the arcs of the graph. However, other more accurate cost measures can be associated with arcs. We discuss two of them in the following subsections: cost of method executions (Section 5.2.3.1) and cost of method pair executions (Section 5.2.3.2). We then illustrate the method cost on the Queue example (Section 5.2.3.3).

5.2.3.1 Method Cost

The execution cost of a component interface method may be the measure in terms of execution time, complexity of the algorithm, or amount of interactions between the component and its environment (e.g., interaction with a database, with the network). Using such information, the cost of an arc in the graph is the cost of the succeeding method in the corresponding constraint: for instance the cost of the arc from node `init` to node `eque` (i.e., constraint `c9`) in Figure 5-4 (a) is the cost of executing method `eque`. This cost measure is more accurate than counting methods without requiring too much additional effort on the part of the component provider. This cost information could also be part of the metadata provided along with the component by the component provider.

Note that with such a cost measure, as well as when counting methods or accounting for method pairs (next section), we can take special care of re-initializing arcs. Those arcs have a cost that may be far from being identical to any of the method (or method pair) costs because the component needs to be re-initialized. These arcs can be assigned the

same value according to the component vendor understanding of how much it costs to re-initialize the component.

5.2.3.2 Method Pair Cost

Let us consider three methods `m1`, `m2`, `m3` with the following *Always Valid* and *Possibly Valid* constraints: `c1 = (m1, m2, true)` and `c2 = (m3, m2, p)`. These mean that `m2` can always be executed after `m1` but one has to ensure that predicate `p` is true in order to execute `m2` after `m3`. Ensuring that the predicate is satisfied may have a cost (e.g., updating the contents of a database), leading to two different costs for arcs labeled `c1` and `c2`, although they correspond to the execution of the same method `m2`. The cost of an arc in the graph can then be the sum of the cost of satisfying the predicate and the cost of executing the succeeding method (see previous section).

This allows for very fine optimization of the generated test suites in terms of cost, but potentially requires considerable additional effort on the part of the component vendor or user, in order to derive these costs.

5.2.3.3 The Queue Example

The method cost approach of assigning costs to constraints/arcs is used in the `Queue` example. Because the `Queue` example is very simple, is only used for illustration purposes (e.g., we have not implemented it), we decided to use as an example an ordinal scale for identifying method costs. This allows us to distinguish which of the methods is more expensive than another. For the `Queue` example, methods `empty` and `top` are the least expansive ones as they only return simple data, whereas method `init` is the most

expansive one as the data structure needs to be initialized. On the other hand, methods `eque`, `dque`, `getLock` and `rlsLock` are of average cost. In terms of an ordinal scale, this analysis could result in the costs reported in Table 5-1.

Table 5-1: Approximation of the `Queue` Component's Interface Method Execution Cost

Method Name	Cost
`init`	3
`empty`	1
`eque`	2
`dque`	2
`top`	1
`getLock`	2
`rlsLock`	2

5.3 Background on Graph Theory

This section provides a brief review of all of the graph theory definitions used in this thesis. Because the DRPP and its solution are based on directed graphs, the definitions and problems are specific to these types of graph. Graph based problems and algorithms to solve the DRPP will be introduced when necessary in the rest of this Chapter.

5.3.1 Graph Theory Definitions

A directed graph G is defined as $G=(N,A)$, where N is a set of *nodes*, and A is a set of *directed edges* (referred to as *arcs*). Arcs are defined by an ordered pair of nodes (i,j), such that the arc is directed out of node i and into node j, i.e., from i to j. Arcs can also be assigned a cost, denoting their cost of traversal: the cost of an arc (i,j) is denoted as $c_{(i,j)}$. The example graph of Figure 5-5 is a directed graph containing a set of nodes

$N=\{n_1,n_2,n_3\}$ and a set of arcs $A=\{a_1,a_2,a_3,a_4\}$. The cost of each of the arcs is delimited by a colon, for instance the cost of traversing arc a_1, from node n_1 to n_3, is 1: $c_{(n1,n3)} = 1$.

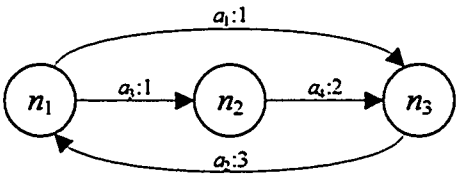


Figure 5-5: A Directed Graph

As arcs of a graph are traversed they trace a *path* on the graph. A path $v_{(1,n)}$ on a graph is a sequence of nodes $n \in N$ and is denoted as $[n_1, n_2, \ldots, n_n]$ such that (n_1,n_2), (n_2,n_3), ..., (n_{n-1},n_n) are arcs of the graph. The set of possible paths from node i to node j is denoted as $V_{(i,j)}$. Each path $v \in V_{(i,j)}$ has a cost which is the sum of all arc costs in the path: $cost_{v \in V_{(i,j)}} = \sum_{n=i}^{j} c_{(n,n+1)}$. The paths of $V_{(i,j)}$ are uniquely identified by the sequence of nodes (or arcs) that they traverse. For example, referring once again to Figure 5-5, paths $v \in V_{(1,3)}$ may be defined as sequence of nodes $[n_1, n_2, n_3]$ (arcs $[a_3, a_4]$), or as a sequence of nodes $[n_1, n_3]$ (arcs $[a_1]$). The cost of path $[n_1, n_3]$ is 1, which is lower than the cost of $[n_1, n_2, n_3]$ (i.e., 1 + 2 = 3), and $[n_1, n_3]$ is therefore referred to as the *lowest cost path* between nodes i and j.

If for every pair of nodes (i, j) there is a path from i to j or from j to i, then the graph is said to be *connected*, otherwise it is *disconnected*. If paths exist from i to j and from j to i for every pair (i, j) then the graph is strongly connected. The graph of Figure 5-5 is

strongly connected since the arcs allow for a path to be traced between any of the nodes n_1, n_2 and n_3.

Additionally, a graph may be partitioned into subgraphs, referred to as *components* $C_1,\ldots,C_n$, where C_i is also used to denote the set of nodes of the i component. The components are connected but the whole graph is disconnected. For example, the graph of Figure 5-6 is partitioned into two components, $C_1 = \{n_1, n_2, n_3\}$ and $C_2 = \{n_4, n_5, n_6\}$. A graph is said to be *component disconnected* when the graph is made of disconnected components. The component themselves may or may not be strongly connected. In the example graph of Figure 5-6, where component C_1 is strongly connected but C_2 is not, the graph is component disconnected, i.e., there is no path from any of the $C_2 = \{n_4, n_5, n_6\}$ to $C_1 = \{n_1, n_2, n_3\}$.

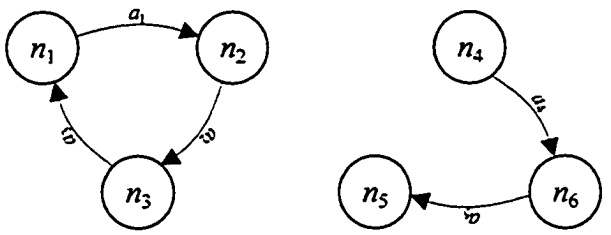


Figure 5-6: A Component Disconnected Graph

A node i may have any number of arcs directed in, defined as the *in degree* and denoted as $d_{in}(i)$. Similarly, a node i may have any number of arcs directed out, defined as the *out degree* and denoted as $d_{out}(i)$. A node with equal in degree and out degree, i.e., $d_{in}(i) = d_{out}(i)$, is said to be *symmetric*. A graph G is *symmetric* if all of its nodes are symmetric: $\forall i \in N \mid d_{in}(i) = d_{out}(i)$.

Furthermore, a graph that is both connected and symmetric is a special kind of graph referred to as an *Eulerian graph.* (Note that this also means the graph is strongly connected.) A property of an Eulerian graph is that a path can be traced on the graph starting from any node $i \in N$, traversing all of the arcs exactly once, and ending at the same node i. Such a path is referred to as an *Euler path.* The graph of Figure 5-7 is an Eulerian graph, and an example of an Euler path starting at node n_1 is traced by arcs [a_1, a_5, a_3, a_2, a_4, a_6].

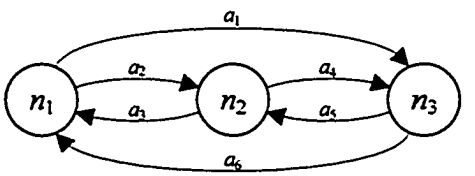


Figure 5-7: An Eulerian Graph Example

5.4 Directed Rural Postman Problem and CSPE Adequate Test Sets

The Directed Rural Postman Problem is an extension of the Chinese Postman Problem, which in turn is similar to the well-known Traveling Salesman Problem[7]. The book edited by Mosh Dror [15] provides a comprehensive review of such graph routing problems and solutions, and the reader is referred to it for further details on the above-mentioned problems.

[7] The Traveling Salesman Problem is to find the shortest way of visiting all the cities from a graph showing cities as nodes and indicating the travel costs between cities on nodes. In the Chinese Postman Problem, on the other hand, it is required that all the arcs of graph be traversed at the lowest cost possible.

In the formulation of the Directed Rural Postman Problem (DRPP) the directed graph denoted by $G=(N,A)$ contains a set of nodes N, and a set of arcs A of costs $c_{ij} \geq 0$. (Note that this constraint on costs is satisfied in our application problem.) The set of arcs A contains a subset of arcs $A_R \subseteq A$ that are referred to as required arcs. In the DRPP only the required arcs need to be traversed at least once, and all the other arcs do not necessarily have to be traversed, but may be traversed if they are needed to create the shortest possible path in the solution to the problem.

Figure 5-8 is a small example of a graph demonstrating the Directed Rural Postman Problem. The graph contains four arcs, a_1 through a_4, and three nodes, n_1 through n_3. Arcs a_1 and a_3 are the only two required arcs, displayed as solid lines. An example path solution of a Directed Rural Postman Problem on this graph is [a_1, a_4, a_2, a_3]. Note that although the arcs a_2 and a_4 are not required they are used in the solution in order to be able to traverse all the arcs that are required (arcs a_1 and a_3).

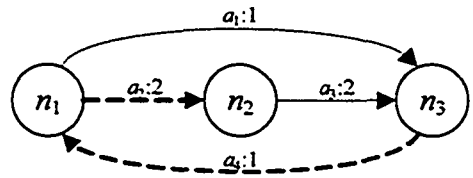


Figure 5-8: Directed Rural Postman Problem Example Graph

The Directed Rural Postman Problem (DRPP) is $\mathcal{NP}$-Complete [28], and as such does not have a polynomial time solution and is generally not easy to solve. The solution to the DRPP used in this thesis is adapted from [10], which is first detailed in Section 5.4.1. Our adaptation of Christofides' solution is then detailed in Sections 5.4.2, 5.4.3 and 5.4.4.

5.4.1 Christofies' Solutions to the DRPP problem

A heuristic solution and an exact algorithm for solving the DRPP are presented in [10][8]. Using the exact algorithm, Christofides et al. were able to solve DRPP for |N| ranging from 13 to 80, and |A| ranging from 24 to 180.

We decided to adapt the heuristic solution, which is based on graph theory, instead of the exact solution, in order to have a more general (scalable), although not always optimal, solution, especially since, according to experimental evidence reported by Christofides, the heuristic graph-based solution is reasonably close to the optimal. (The reader is referred to [10] for further details on this experimental evidence and on the integer linear programming problem specification.) We will however consider in future work the exact solution to the DRPP by Christofides [10], as well as other more recent solutions[9] to the DRPP [11].

The objective of the DRPP solution by Christofides is to identify which of the non-required arcs will be used in the final solution to the DRPP, and how many times each of the arcs of the entire set of arcs will be traversed.

The graph based solution consists eventually in finding an Euler tour in an Eulerian graph. In order to do so, the initial graph (which is our case represents CSPE constraints) has to undergo a set of four initial transformations (Section 5.4.2) and then be made

---

[8] This DRPP solution was applied to printed circuit board manufacturing [4].

[9] The article by Corbéran et al. presents an advanced solution to the mixed general routing problem, where the DRPP is a special case of such a problem.

Eulerian (Section 5.4.3). Section 5.4.4 then shows how an Euler tour can be determined and why the Euler tour is a solution to the DRPP problem.

Broadly speaking, the graph transformations below aim at promoting non-required arcs to the set of required arcs because, based on observation provided in [10], they will be part of any solution to the DRPP. Then the graph containing only those required arcs (either part of the original set of required arcs or promoted) is ensured to be Eulerian and an Euler tour is found.

5.4.2 Initial Transformations

First, recall that the required arcs, denoted as A_R, are identified by the targeted functionalities and the selected CSPE predicate coverage criterion. Any node in the graph that is not the source or the target of at least one required arc is not a required node. There is one exception to this rule, namely the node labeled #. This node corresponds to the initialization of the component and will start any test case used to test the component, i.e., it must be part of the solution. Node labeled # is thus an always required node. Additionally, since the only arcs leaving error nodes are those labeled '~', those arcs are required arcs. We thus identify in $G=(N,A)$ a set of required nodes A_R and a set of required nodes N_R.

Second, it is worth mentioning that because of our definition of the initial graph (Section 5.2.1), i.e., the way we build nodes and arcs, and especially the re-initialization arcs, the initial graph is strongly connected. Additionally, if only considering the required arcs, the initial graph may be component disconnected.

The purpose of the initial transformations is to find paths in the original graph, made of required and non-required arcs, that will facilitate (because they are the shortest path between two nodes) the identification of a nearly optimal solution. In order to ease reading in this section, the graph transformation steps numbers are indicated as superscripts to G (the graph), N (the set of nodes) and A (the set of arcs): e.g., graph $G^1=(N^1,A^1)$ is the result of the first transformation on graph $G=(N,A)$.

Transformation Step 1—Prune Non-required Nodes and Arcs:

Remove all nodes that do not have at least one adjacent $a \in A_R$ and all arcs that are not in A_R. The resulting graph is denoted as $G^1 = (N_R^1, A_R^1)$. ($N_R^1 \subseteq N$ and $A_R^1 = A_R$.)

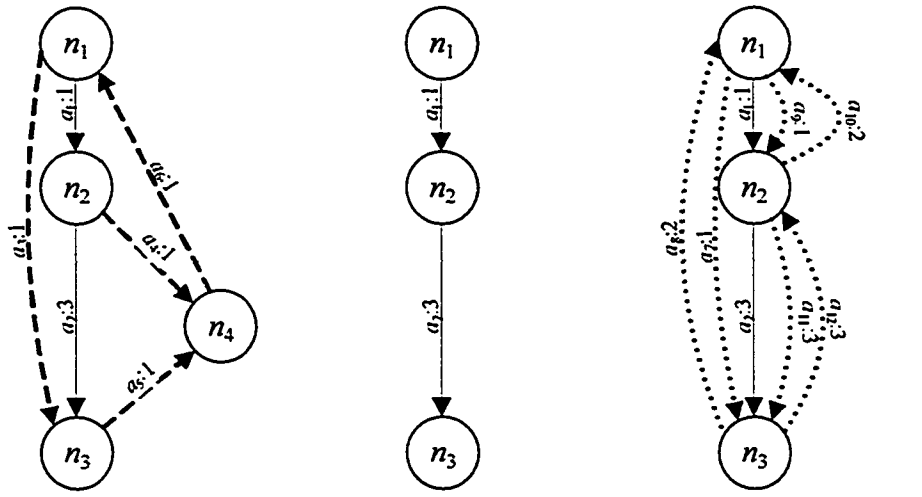
The next step of the transformation adds additional arcs, which are referred to as *virtual arcs*, to the directed graph G^1. Virtual arcs are used to record the cheapest path in G (the original graph) between every ordered pair of nodes in G^1 (i.e., required nodes), and can represent either a single arc or a path formed by a sequence of arcs of graph G. These paths can potentially help finding a cheap solution to the DRPP problem: it may be cheaper to traverse non-required nodes and arcs (i.e., a path in G) rather than traversing only required nodes and arcs (i.e., in G^1).

Transformation Step 2—Add Virtual Arcs[10]*:*

Construct a graph G^2 by adding virtual arcs between every ordered pair of nodes $(i, j) \in N_R^1$. The cost of virtual arc $c_{(i,j)}$ is equal to the lowest cost path $V_{(i,j)}$ in G between required nodes i and j. The resulting graph is denoted as $G^2 = \left(N_R^2, A^2\right)$, where $N_R^2 = N_R^1$, $A^2 = A_R^1 \cup A_V^2$, and A_V^2 is the set of new virtual arcs.

The graphs of Figure 5-9 illustrate the first two steps of the graph transformation. Required arcs are drawn with solid lines, non-required arcs are draw with dashed lines, and virtual arcs are drawn with dotted lines. Figure 5-9 (a) shows the original graph G. In step 1, non-required arcs (a_3, a_4, a_5, a_6) and non-required node (n_4) are removed from G. The result is graph G^1 shown in Figure 5-9 (b). Virtual arcs are then added to graph G^1, based on the shortest paths in the original graph G, and the resulting graph G^2 is shown in Figure 5-9 (c). For example: Virtual arc a_6 (cost 1) corresponds to path [a_1] of cost 1 in G (there are other paths in G from n_1 to n_2 but of higher costs); Virtual arc a_8 (cost 2) corresponds to path [a_5, a_6] (both cost 1); Virtual arc a_{12} (cost 3) corresponds to path [a_5, a_6, a_1] in G (all cost 1).

[10] Note that the mapping between virtual arcs and the corresponding path of non-required (and/or required) arcs in the original graph is recorded. Eventually, if a virtual arc is used in the solution to the DRPP, it will be expanded and replaced by the corresponding path in the original graph to form a complete, concrete test case.



(a) Original Graph G (b) Graph G^1 after Step 1 (c) Graph G^2 after Step 2

Figure 5-9: The First Two Steps of Graph Transformation

In the third transformation, redundant virtual arcs are identified. A virtual arc is redundant when we can find a path in G^2 between the same two nodes or a required arc in parallel of the same cost.

Transformation Step 3—Remove Redundant Virtual Arcs:

From the set of virtual arcs A_V^2 of graph G^2 remove:

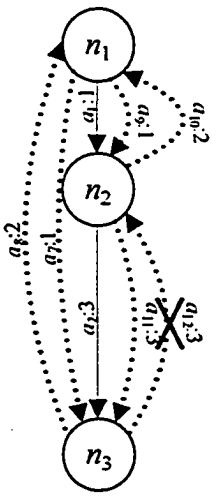
(a) any arc $(i,j) \in A_V^2$ for which there exists $k \in N_R^2$ such that $c_{ij} = c_{ik} + c_{kj}$;

(b) any arc $(i,j) \in A_V^2$ such that there exists a required arc in A_R^2 from node i to node j.

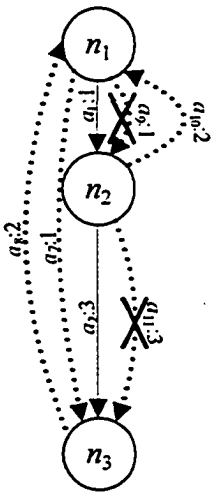
The resulting graph is denoted as $G^3 = \left(N_R^3, A^3\right)$, where $N_R^3 = N_R^1$, $A^3 = A_R^1 \cup A_V^3$, and A_V^3 is the reduced set of virtual arcs.

The two graphs of Figure 5-10 illustrate transformation 3 from graph G^2 in Figure 5-9 (c). In Figure 5-10 (a), an arc that meets condition (a) is identified (an 'X' crosses the arc): arc a_{12} is redundant with the sequence of arcs a_8 and a_1. In Figure 5-10 (b), arcs that

meet condition (b) are identified: arc a_3 is redundant with required arc a_2, and arc a_9 is redundant with required arc a_1.



(a) Transitive-equivalent Arcs Removed



(a) Arcs Equivalent to Required Arcs Removed

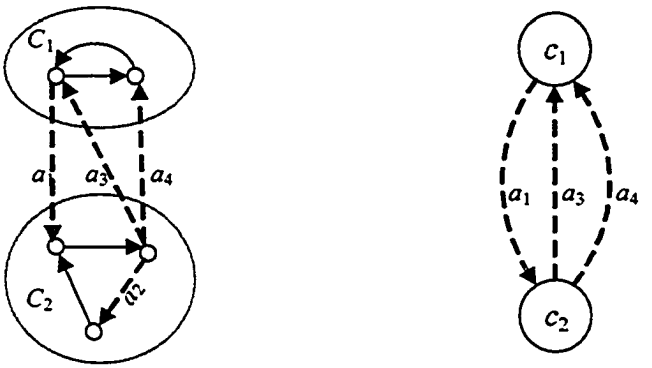
Figure 5-10: Third Step of the Graph Transformation

After redundant virtual arcs have been removed, between each pair of required nodes there is either (a) a required arc, or (b) a virtual arc representing the shortest path in G made of required and non-required arcs between the two nodes, or (c) both a required arc and a virtual arc in which case we have found a cheapest way than the required arc to go from its source to target node.

The resulting graph $G^3 = \left(N_R^3, A^3\right)$ is strongly connected. The graph induced by the required set of arcs only, i.e., $\left(N_R^3, A_R^1\right)$, may be *component disconnected.* We refer to this graph as being component disconnected with respect to (w.r.t.) arcs and we denote it as G_{cd}^3. In the rest of the Chapter, unless otherwise specified, a graph that is component disconnected w.r.t. the required arcs is simply called component disconnected. The

component disconnected graph G_{cd}^3 is partitioned into components $C_1,\ldots,C_p$ and can be represented in a condensed way by graph $G_C^3 = \left(N_C^3, A_{VC}^3\right)$, such that the node $n_i \in N_V^3$ of such graph represents the component C_i in graph G_{cd}^3, and the only arcs between those nodes correspond to some of the virtual arcs of the not condensed graph.

The example graph of Figure 5-11 (a) is a component disconnected graph (G_{cd}^3) made of two components C_1 and C_2. Component C_1 is strongly connected, while component C_2 is not. The corresponding condensed graph is shown in Figure 5-11 (b) where only the virtual arcs spanning the two components C_1 and C_2 are involved.



(a) Component Disconnected (w.r.t. required arcs) Graph G_{cd}^3 **(b)** Condensed Graph G_C^3

Figure 5-11: Component Disconnected Graph and Corresponding Condensed Graph, an Example

The last step of the graph transformation promotes virtual arcs from set A_V^3 to required arcs set A_R^3, because they will be involved in any solution to the DRPP, and is based on two observations presented in [10]. First, if for two components *Ci* and *Cj* in the component disconnected graph the unique arc from *Ci* to *Cj* is a virtual arc, then this

virtual arc must be in any solution to the DRPP. Second, if the only way to reach or leave a required node in a component is to take a virtual arc, then again, this virtual arc must be in any solution to the DRPP. The reason for these arc promotions is that only required arcs are traversed in the final solution, if there is only one arc connecting a component or node then it must be a required arc.

Transformation Step 4—Promote Virtual Arcs:

Virtual arc (i, j) is promoted from A_V^3 to the set of required arcs if it meets any one of the following three conditions:

(a) there is a partition of G^3 into two components C_i and C_j such that (i,j) is the unique arc of C_i to C_j;

(b) (i,j) is the only arc directed out of node $i \in N_R^3$;

(c) (i,j) is the only arc directed into node $j \in N_R^3$.

The resulting graph is $G^4 = \left(N_R^4, A^4\right)$, where $N_R^4 = N_R^1$, $A^4 = A_R^1 \cup A_R^{4new} \cup A_V^4$, and A_R^{4new} is the set of promoted virtual arcs ($A_R^{4new} \cup A_V^4 = A_V^3$). A_V^4 is the reduced set of virtual arcs. The set of required arcs in G^4 is $A_R^4 = A_R^1 \cup A_R^{4new}$.

For example, in the graph of Figure 5-11 (a), two virtual arcs a_1 and a_2 are promoted into the set of required arcs according to transformation step 4. Arc a_1 is the only arc that is directed out of component C_1 and into component C_2 whereas arc a_2 is the only arc directed into its target node.

The promotion of virtual arcs into required arcs in the last step does not guarantee that G^4 is not component disconnected w.r.t. to required arcs. We however have to ensure

it is strongly connected w.r.t. required arcs and symmetric, i.e., it is Eulerian. Making the graph strongly connected and symmetric is the objective of the following section.

5.4.3 Connected and Symmetric (i.e., Eulerian) Graph

These additional transformations of the graph, to make it Eulerian, are mentioned in [10] but without any details as to how to proceed. Graph theory can help us find problems, and thus algorithms, to transform G^4 into a connected (Section 5.4.2) and symmetric (Section 5.4.3) graph with respect to required arcs.

5.4.3.1 Towards a Connected Graph: The Shortest Spanning Arborescence Problem

The *shortest spanning arborescence problem* (SSA) [16] is based on a directed graph $G=(N,A)$. The problem is stated as follows: starting from a root node $t_\alpha \in N$, a set of arcs $A_{sol} \subset A$ must be chosen such that there exists paths from the root node to every other node of the graph with the lowest possible cost of the whole arborescence[11]. (Note that this does not necessarily mean that the paths from the root node of the arborescence to the leaf nodes are the shortest.) The selection of the root node t_α from the set of nodes N depends on the context in which the SSA problem is applied to. An example of a shortest spanning arborescence is depicted in Figure 5-12, where the set of arcs A_{sol} is highlighted. The solution to the SSA is the set of arcs $A_{sol} = \{a_1, a_2, a_4, a_6, a_7\}$ with a total

[11] The solution to the SSA was first presented by Edmunds in [16], and a better alternative solution, based on the work of Tarjan [44], can be found in [8].

cost of 5. A set of arcs $\{a_1, a_3, a_5, a_8, a_9\}$ also connects the arborescence rooted at t_α, but is not a SSA since its total cost is 13.

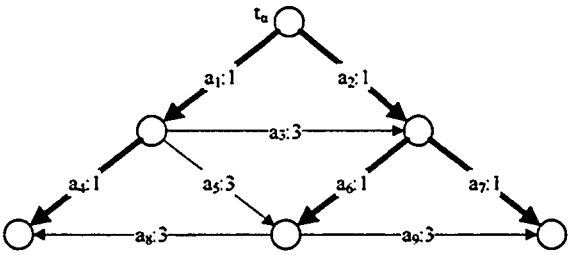


Figure 5-12: Example of Shortest Spanning Arborescence

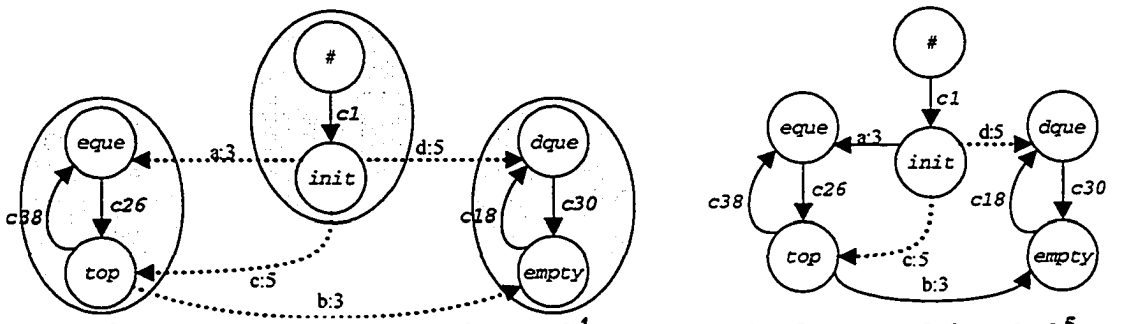
Acknowledging that graph G^4, obtained after the four initial transformations is potentially component disconnected (w.r.t. required arcs), our use of the SSA is to find a cost-effective way of making it connected. We will see that this is required for the next step of the transformations, i.e., to make the graph symmetric. In the case G^4 is already connected then the current step is not required. Otherwise, as already mentioned, G^4 can be represented by a condensed graph G_C^4.

We determine the SSA on G_C^4, with root node t_α as the component that contains node labeled #. This is a heuristic, based on a theory that node labeled # is the most connected node, and as already mentioned, node labeled # will start any of the test cases produced. The arcs of the SSA solution, A_{sol}, are then promoted from the set of virtual arcs to the set of required arcs. The SSA is used to tell us which arborescence is the cheapest, i.e., which virtual arcs in G_C^4 will likely be part of the oprimal solution, and thus should be promoted as required arcs.

Transformation Step 5—Promote Virtual Arcs from the SSA solution:

Determine the Shortest Spanning Arborescence on the condensed graph G_C^4, generated from G^4, and promote the virtual arcs that are part of the solution to the SSA problem. The resulting graph is $G^5 = (N_R^5, A^5)$, where $N_R^5 = N_R^4$, $A^5 = A_R^4 \cup A_R^{5new} \cup A_V^5$, and A_R^{5new} is the set of promoted virtual arcs because part of the SSA solution ($A_R^{5new} \cup A_V^5 = A_V^4$). A_V^5 is the reduced set of virtual arcs. The set of required arcs in G^5 is $A_R^5 = A_R^4 \cup A_R^{5new}$.

The graph of Figure 5-13 (a) is a component disconnected graph modeling a possible test suite of the `Queue` component. We assume here that the component user selected three component use case scenarios: initialization of the component, en-queuing of the component followed by checking what has been en-queued, and de-queuing of the component until it is empty. After the first four initial transformations, the graph G^4 resembles the graph in Figure 5-13 (a). It is component disconnected graph and Figure 5-13 (a) shows what the components are. The *virtual arcs* labeled *a* through *d* (identifying them in this example, as otherwise they are not labeled) are the arcs used to find the Shortest Spanning Arborescence. In this simple example the shortest spanning arborescence rooted at the node containing # is formed by the arcs *a* and *b*. These arcs are then promoted to *required arcs* making the graph G^5 connected (w.r.t. required arcs): Figure 5-13 (b).



(a) Component Disconnected Graph G^4 **(b) Connected Graph G^5**

Figure 5-13: Connecting (w.r.t. required arcs) using SSA (re-initializing arcs are omitted)

5.4.3.2 Towards a Symmetric Graph: The Minimum Cost Maximum Flow Problem

Recall that our overall objective is to build a graph made of required arcs and required nodes that is Eulerian (connected and symmetric). The subgraph of G^5 containing only required arcs and nodes is connected (this is ensured by the previous transformation) but not necessarily symmetric. If it is symmetric, then the transformation described in the current section is not necessary.

In this section we first introduce the Minimum Cost Maximum Flow (MCMF) problem (Section 5.4.3.2.1). Then show how we use it to make our graph symmetric with respect to required arcs (Section 5.4.3.2.2), i.e., the in-degree and out-degree when only accounting for required arcs are equal for all the required nodes.

5.4.3.2.1 Minimum Cost Maximum Flow

The *minimum cost maximum flow* (MCMF) problem is a variation of the maximum flow family of problems reviewed in [1]. The variation of the MCMF problem used in this thesis is known as the *supply and demand* problem. In this kind of problem, a directed

graph is seen as a flow network and used to solve problems about flow of goods between locations (i.e., nodes) where those goods are produced (supply) and where they are used (demand). The solution accounts for quantities that are produced and used by the supply and demand nodes (through a solution to the maximum flow), but also for the cost it takes to send trucks on specific routes (by making the flow minimum cost).

In other words, we have a directed graph $G=(N,A)$, where the nodes are either of the *source* type, $n \in N_r$, associated with a positive value (*supply*), or *sink* type, $n \in N_s$, associated with a negative value (*demand*). In the *supply and demand* problem solution, it is required that arcs are assigned a flow value such that source nodes have a net flow directed out equal to their supply (producers of goods sell all their stock), and sink nodes have a net flow directed in equal to their demand (users receive what they need), at the lowest possible cost.

The graph G = (N_0, A_0) specifying the flow problem contains supply and demand nodes. These nodes are labeled by the quantities produced and used: e.g., nodes n_1 and n_2 in Figure 5-14. The arcs are labeled with how much a unit of flow costs: e.g., arc between node n_1 and node n_2 in Figure 5-14. In order to solve the *supply and demand* problem this graph defining the flow is modified into graph $G_1 = (N_1, A_1)$. First, two additional arcs are added to the set of nodes: root source node r and root sink node s. Then, so called utility arcs (set A_u) are added from r to all the supply nodes and from all the demand nodes to s. The utility arcs are labeled by capacities, indicating the maximum flow that can go through them: the capacity is thus set to the arc's target node quantity (for an arc leaving

the root source node) or the arc's source node quantity (for an arc going to the root sink node). The utility arcs are not assigned any cost (rather the cost is 0) and the original arcs have an unlimited capacity.

An *r-s* flow is then defined as a set of paths $V_{(r,s)}$ from *r* to *s*. A flow value $f_{(i,j)}$ is defined as the number of times arc (i,j) appears in the *r* to *s* flow, i.e., in the paths of the flow. The cost of the *r-s* flow that is to be minimized is defined as the sum of the products of each of the arc's flow value and cost: $cost_{r\text{-}s} = \sum_{(i,j)\in A_o} f_{(i,j)} c_{(i,j)}$. At the end of the solution of the *supply and demand* problem, root source node *r*, root sink node *s*, and their adjacent arcs are removed, such that source nodes are left with a net flow directed out and sink nodes a net flow directed in.

An example of the MCMF problem and solution is illustrated in Figure 5-14, which shows graph G_1. Nodes with positive values (supply) are the source nodes $\{n_1, n_2, n_3\}$, and nodes with negative values (demand) are the sink nodes $\{n_4, n_5\}$. Arcs a_1 through a_7 are the arcs of A_o, and the arcs adjacent to nodes *r* and *s* are the arcs of A_u. A solution to the MCMF problem of Figure 5-14 is an *r-s* flow where the path $[r, n_2, n_5, s]$ appears twice, path $[r, n_1, n_4, n_5, s]$ appears three times, path $[r, n_3, n_4, s]$ appears 4 times, and path $[r, n_3, n_4, n_5, s]$ once. The flow on the arcs of A_o is: $\{f_{a1}=0, f_{a2}=0, f_{a3}=3, f_{a4}=0, f_{a5}=2, f_{a6}=5, f_{a7}=4\}$, and each of the node's supply or demand is met.

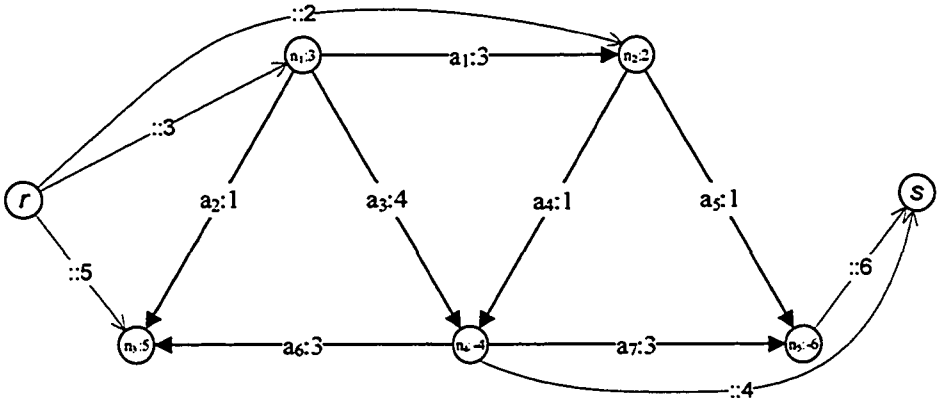


Figure 5-14: Directed Graph of a Minimum Cost Maximum Flow Problem

A solution to the MCMF problem was presented by Busackar and Gowen in [7]. The algorithm by Busackar and Gowen is advantagous because of its simplicity, but requires that all arc costs be non-negative (which is our case for building CSPE component test cases). Other fast solutions to the problem include those by Ahuja, Golderg, Orlin, and Tarjan [1]; and Orlin [33].

5.4.3.2.2 Use of MCMF to Make a Graph Symmetric

In order to make a non-symmetric graph symmetric w.r.t. to the required arc set we need to duplicate some of the existing arcs and possibly promote some more of the virtual arcs to the required arc set such that every node as an equal degree of in and out degree required arcs. The difficulty with this is that duplicating arcs in order to make one node of the graph symmetric has an effect on the net degree of the node on the other end of the arc. This indeed becomes a difficult problem. The MCMF is used to tell us which arcs have to be duplicated in order to make the graph symmetric and do this with a minimum cost of the arcs duplicated.

Graph G^5, result of the previous transformation (Section 5.4.3.1), is used to solve a MCMF problem. Recall that the nodes of G^5 are required nodes, and that its arcs are required arcs or virtual arcs. Costs associated with arcs in G^5 are the cost associated with the corresponding CSPE constraints: either the cost of a required arc in G, i.e., the arc in G^5 corresponds to one CSPE constraint, or the sum of the costs of the path in G represented by a virtual are or a promoted virtual arc in G^5.

Source and sink nodes, and their quantities (either supply or demand), are determined with respect to only required arcs. For each node n in G^5 we compute $D(n) = d_{in}(n) - d_{out}(n)$. If $D(n)<0$ then node n is considered a sink with demand equal to $D(n)$. If $D(n)>0$ then node n is considered a source with supply equal to D(n). Otherwise, i.e., if D(n)=0, the node is already symmetric (with respect to required arcs): it is neither a sink or source. Figure 5-15 illustrates this. In Figure 5-15 (a), node n has two incoming required arcs (i.e., $d_{in}(n)=2$) and one outgoing required arcs (i.e., $d_{out}(n)=1$), making it a supply node of quantity 1 (i.e., $d_{in}(n)-d_{out}(n)=2-1$). Node m has one incoming required arc (i.e., $d_{in}(m)=1$) and three outgoing required arcs (i.e., $d_{out}(m)=3$), making this node a demand node of quantity 2 (i.e., $d_{in}(m)-d_{out}(m)=1-3$). Note that the presence of a virtual arc from node m to node n in Figure 5-15 (b) does not change the kind of those nodes and their quantities as only required arcs are accounted for. Figure 5-15 (a) and Figure 5-15 (b) are the two main situations that we can encounter and are worth discussing here: i.e., a source node is linked to a sink node. Other situations like the one in Figure 5-15 (c), where two symmetric nodes appear, also in the end entail a source node and a sink node.

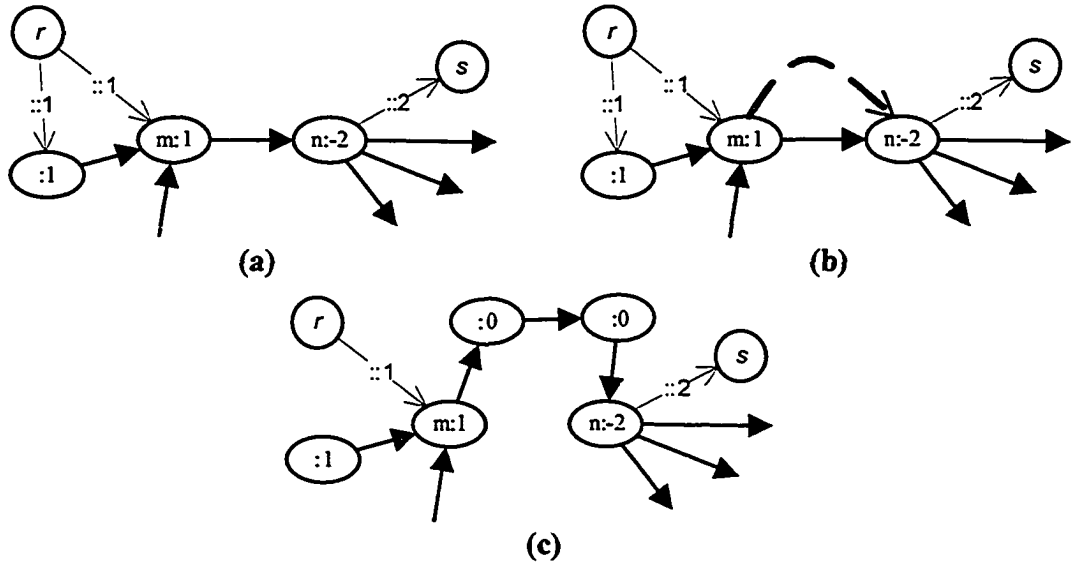


Figure 5-15 Preparing the graph for MCMF

When solving the MCMF problem, we connect sink nodes to the root sink node and the root supply node to supply nodes (Figure 5-15) and we account for the whole graph, i.e., we account for both required and virtual arcs, and consider that arcs have infinite capacities. This means that the maximum flow solution can involve required arcs as well as virtual arcs.

Figure 5-16 illustrates the MCMF solution from Figure 5-15 (a). A first flow of capacity 1, labeled *f1* starts from root node *r*, goes through nodes *m* and *n* and stops at root node *s*: Figure 5-16 (a). This results in the supply of node *m* being reduced by 1 (it is now 0) and the demand of node *n* augmented by 1 (it is now -1). Another flow, flow labeled *f2* in Figure 5-16 (b) is necessary to meet the demand of *n*. This flow of capacity 1 starts from *r*, goes through the unlabeled node and then *m* and *n*, and stops at *s*. This results of the supply of the unlabeled node reduced by 1 and the demand of node n augmented by 1 (it is now 0). (Note that the supply of *m* has not changed because *m* is

only traversed by the flow, instead of being its source.) In the end, the total flow through arc *m-n* is 2.

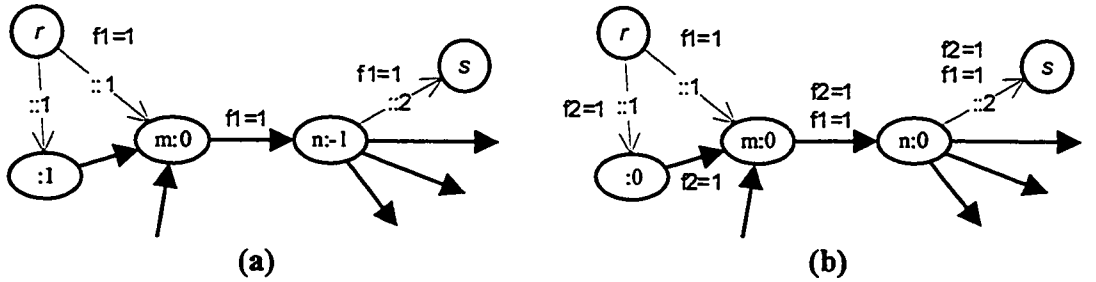


Figure 5-16 MCMF flows from Figure 5-15 (a)

Figure 5-17 illustrates the MCMF solution from Figure 5-15 (b) in which we have added costs for the two arcs between *m* and *n*: the required arc costs 2 whereas the virtual arc costs 1. This time, because MCMF tries to minimize the cost of the maximum flow, the two flows f1 and f2 go through the virtual arc, instead of the required arc. This results in a flow for required arc *m-n* of 0 and a flow of 2 for virtual arc *m-n*.

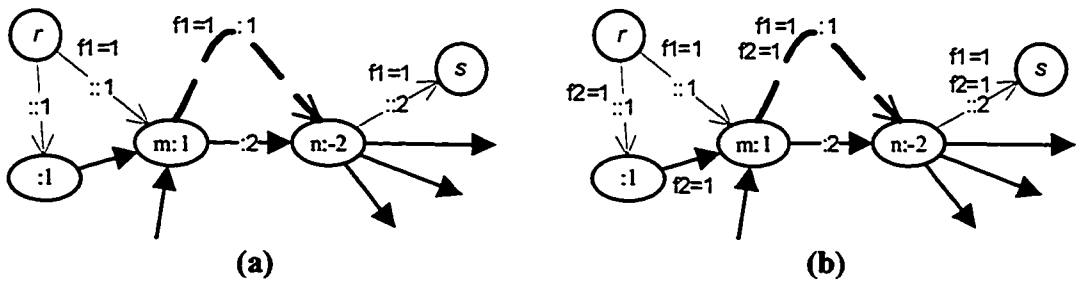


Figure 5-17 MCMF flows from Figure 5-15 (b)

In the solution to the MCMF problem, the flow on (required and virtual) arcs indicate where to add arcs, which arcs have to be added, and how many arcs have to be added. Consider for instance the flow result in Figure 5-16 (b). The flow on arc *m-n* is two, indicating that two new arcs (i.e., the arc flow) has to be created between *m* and *n*.

Indeed, if two other arcs exist from *m* to *n*, then *n* has three incoming required arcs and three outgoing required arcs. Additionally, *m* then has three outgoing required arcs and three incoming required arcs, because one incoming required arc to *m* has a flow of one indicating that a new incoming arc to *m* is needed. In this case, the arcs to be added between *m* and *n* are necessarily duplicates of the required arcs because *m* and *n* are only linked by a required arc[12]. (We cannot create arcs that do not correspond to CSPE constraints.) Consider now the flow result in Figure 5-17 (b). The flow on required arc *m-n* is 0 which, according to what precedes, indicates that required arc *m-n* is not duplicated. The flow of two is instead on the virtual arc *m-n* is chosen because the virtual arc has a lower cost than the required arc. In such a case, the virtual arc is promoted to a required arc and duplicated once (i.e., the virtual arc flow minus one). This again results in a total of three arcs between *m* and *n*, making them symmetric. Note the number of duplicates if the arc flow minus one because we already promote it once, i.e. the nodes to which it is adjacent have already been adjusted once.

As a result of the last transformation, summarized below, the graph made of the required nodes and required arcs is connected and symmetric, i.e., it is Eulerian.

[12] Note that in the case of the example of Figure 5-15 (c), the flow of all the required arcs in the path from m to n will be two, leading to two duplications of these required arcs. All the nodes are then symmetric.

Transformation Step 6—Duplicate and Promote Arcs from the MCMF solution:

Determine the MCMF solution from graph G^5 according to the following settings:

- A sink node n is a node for which $D(n)=d_{in}(n)-d_{out}(n)<0$, where d_{in} and d_{out} only account for required arcs.
- A source node n is a node for which $D(n)=d_{in}(n)-d_{out}(n)>0$, where d_{in} and d_{out} only account for required arcs.
- Account for the whole graph, i.e., all its nodes and arcs in the MCMF solution.
- Arcs have infinite capacities.

From the solution to the MCMF:

1. A required arc with a flow of x is duplicated x times.
2. A virtual arc with a flow of x is promoted and duplicated x-1 times.

The resulting graph is $G^6=\left(N_R^6,A^6\right)$, where $N_R^6=N_R^5$, $A^6=A_R^5\cup A_R^{6new}$, A_R^{6new} is the set of duplicated and promoted from the MCMF solution, and the remaining virtual arcs have been removed.

5.4.4 Eulerian Paths

The graph transformations described in previous sections produce the graph that we labeled $G^6=\left(N_R^6,A^6\right)$. A sub-graph of G^6, made of only required nodes and required arcs is Eulerian. An Euler path, i.e., a path through the graph which starts and ends at the same node and includes every arc exactly once, can be easily found manually since all nodes of the graph are symmetric, and there exist algorithms to automate the process (e.g., [17]). It is worth mentioning that for a given Eulerian graph, many Euler paths can be built.

As mentioned before, an Euler path in this graph, started with the node labeled #, solves our problem. An Euler path is in fact one long adequate test for the selected

criterion: #, m1, ... #. Because of possible re-initializations that make the test case cheaper, node (method) labeled # may occur several times in the method test sequence: i.e., the long adequate test case looks like #, m1, ..., #, m2, ..., #, m3, ..., #. This can be interpreted as several test cases in one adequate test set: each occurrence of method labeled # indicates the start of a new test case because this indicates that the component has to be re-initialized.

5.5 Discussion

Suppose the component under test has three public methods in its interface, namely `m1()`, `m2()` and `m3()`. Additionally, suppose `m1()` postcondition is to set a component attribute `attr` to 1 (`attr=1`); `m2()` postcondition is to increase the attribute by one (`attr=attr@pre+1`); and `m3()` precondition requires that the attribute equals to 4 (`attr=4`). Further assume that, considering the partition and the selected CSPE predicate criterion, Possibly Valid constraint `(m1, m3, p)`, where `p` is obviously `attr=4`, has to be exercised. This result in a required arc between node labeled `m1` and node labeled `m3` in graph G (the graph built before the transformations) and thus in graph G^6. The test set generated by the Euler tour thus contains method call sequence `m1(), m3()`. In order to execute this sequence and actually trigger the constraint, the driver needs mechanisms to set the attribute value to 4 (i.e., to satisfy the predicate). It cannot resort to method `m2()`, and execute sequence `m1(), m2(), m2(), m2(), m3()`, since then `(m1, m3, p)` constraint is not triggered.

The approach described in this Chapter for the automatic construction of CSPE predicate coverage adequate test sets thus assumes that some built-in test support is available in the component interface. If no such support is available, then other techniques have to be devised to build adequate test sets (e.g., how to automatically discover that we have to execute `m2()` three times after `m1()` in order to successfully execute `m3()`?). Moreover, some Possibly Valid constraint may become Never Valid constraint. Finding such techniques requires complicated analyses of OCL expressions involved in pre and postconditions and will be considered in future work.

Chapter 6
Prototype Tool

This section presents the prototype test sequence generation tool `PrestoSequence` implemented as part of this thesis. The prototype tool reads the component metadata and test specification provided by the component vendor and component user and outputs test sequences based on the DRPP solution (Chapter 5).

The tool is briefly discussed in two section: the implementation of the tool with references to the algorithms of Chapter 5 (Section 6.1), and description of the XML format input files representing the component metadata and test specification of the proposed framework of Section 4.1 (Section 6.2).

6.1 Implementation

The `PrestoSequence` prototype tool is implemented in Java. It contains 29 classes implemented in 1200 (un-commented) lines of code. Some of the features of this tool are:

- Use of XML format for reading input files, metadata and test specification
- Intermediate graph visualization using the Graphviz toolkit [3]
- Steps of the algorithm are interchangeable and allow for easy extension

Use of XML format was a natural choice for the tool as it has become a standard for human readable software interchange format. The component vendor and users can manually (or through tools of their own) manipulate the files based on a specified XML Schema. Another benefit of using the XML format is that the XML Schemas may be easily extended to include needs of the COTS testing framework beyond automation.

The intermediate graphs, i.e., graphs after the consecutive steps of the algorithm, are very large. Analyzing these graphs during debugging is not practical, and in order to effectively debug the tool a way to visualize the graphs was necessary. The Graphviz toolkit [3] was used to visualize the graphs. The `PrestoSequence` tool provides graph output files which in the Graphviz format, and Graphviz may then be used to generate (automatically layout) graphs is a wide variety of graphics formats.

The tool was designed to make it easy to extend the algorithms used by allowing the steps of the algorithm to be interchanged. This would, for instance, enable the implementation of the Christofides' exact algorithm (mention in Section 5.4.1) to be implemented and added into the tool with little impact on the existing implementation.

In order to run the tool the user invokes the following command line:

```
java squall.prestoseq.PrestoSequence [TEST_SUITE]
```

, where `TEST_SUITE` is the name of the test suite and `PrestoSequence` expects to find two input files named after `TEST_SUITE`. The tool, illustrated in the static structure UML diagram of Figure 6-1, is partitioned into five packages:

- `squall.prestoseq`
- `squall.prestoseq.input`
- `squall.prestoseq.solver`
- `squall.prestoseq.solver.algorithms`
- `squall.prestoseq.output`

Additionally, the tool depends on the Gravisto graphing framework [46], used for its basic graph construction and modification functionality[13]. This is shown in Figure 6-1 as the `org.gravisto` package.

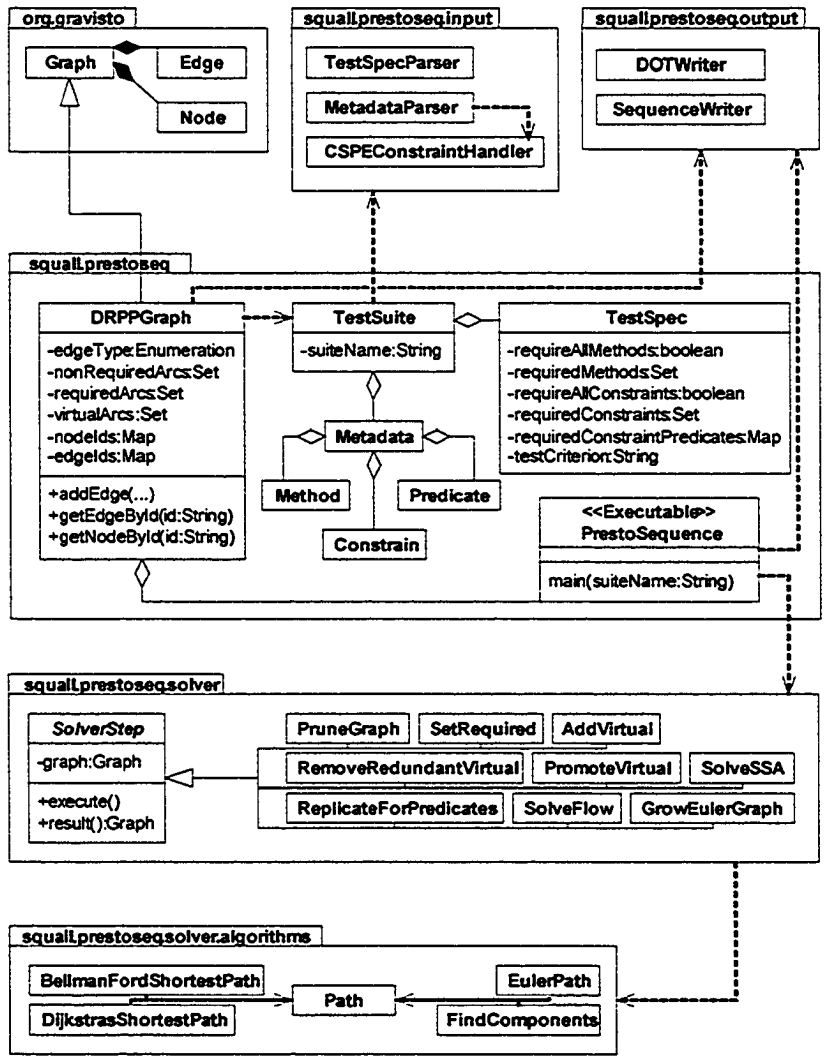


Figure 6-1: UML Diagram of the Implemented Prototype Test Sequence Generation Tool, `PrestoSequence`

[13] The framework was used to get a quick start on the implementation, but eventually a very small subset of the framework's functionality was used and the overly-complex API that it exposes is not required in *PrestoSequence*. A simple implementation of an adjacency list graph would suffice.

6.1.1 Package squall.prestoseq

This is the main package of the PrestoSeq tool and it also includes the main entry point of the tool execution, class PrestoSequence. PrestoSequence begins execution by loading the entity class TestSuite from the XML inputs files using the squall.prestoseq.input package described in the next section.

TestSuite and all of its aggregate classes (Metadata, Method, Constraint, Predicate, and TestSpec) are the entity classes that model the corresponding elements of the CSPE testing framework (read from inputs files).

6.1.2 Packages squall.prestoseq.input and squall.prestoseq.output

These two packages handle the input and output facilities of the tool. The input classes are based on Java API for XML Processing (JAXP) [42] and includes: TestSpecParses responsible for reading in the test specification file, and MetadataParser and CSPEConstraintHandler responsible for reading in the component metadata file.

The ouput package includes two classes: DOTWriter is used to output graphs parts of the DRPP algorithm in Graphviz format, and SequenceWriter is used to oupt the final component test sequence. The Graphviz format graphs generated during the tool execution include: the original graph (model of the CSPE test), intermediate graphs between steps of the algorithm, and final Euler graph (on which the test sequence is found).

6.1.3 Packages `squall.prestoseq.solver` and `squall.prestoseq.solver.algorithms`

The classes of the `squall.prestoseq.solver` package are the classes that actually implement the algorithm of Chapter 5. In order to facilitate interchange and possible future extension of the algorithm steps, all of the implemented steps are based on the abstract `SolverStep` class. The main execution block of the `PrestoSequence` deals with references to `SolverStep` and is therefore only loosely coupled with the actual implementations. All of the concrete classes that extend from `SolverStep` (e.g., `PruneGraph`, `SetRequired`, `AddVirtual`) are the implementation of the algorithms described in Chapter 5.

Algorithms that where not directly related to the solution of Chapter 5, but were required by the algorithms of the classes in `squall.prestoseq.solver`, were implemented separately in `squall.prestoseq.solver.algorithms`. These include two shortest path algorithms `BellmanFormShortestPath` and `DijkstrasShortestPath`, an algorithm for finding Euler paths `EulerPath`, an algorithm for finding disconnected components of a graph `FindComponents`, and a utility classes used by both packages `Path`.

6.2 Testing Input Files

The `PrestoSequence` user supplies component metadata and test specification to the tool in the form of XML format input files. As part of the `PrestoSequence`

implementation XML Schemas were designed for both the component metadata (shown Figure 6-2) and test specification (shown in Figure 6-3).

The CSPE-based metadata is provided in XML file based on the XML Schema shown in Figure 6-2. The main elements of the schema (and their content) are: `contracts` (include the OCL contracts for each of the interface methods of the component), `constraints` (include the derived CSPE constraints), `predicates` (include the CSPE constraint predicates), and `probes` (define the built-in methods provided by the vendor). The schema was designed to include the element `probes`, but its use in the framework and tool is part of future work.

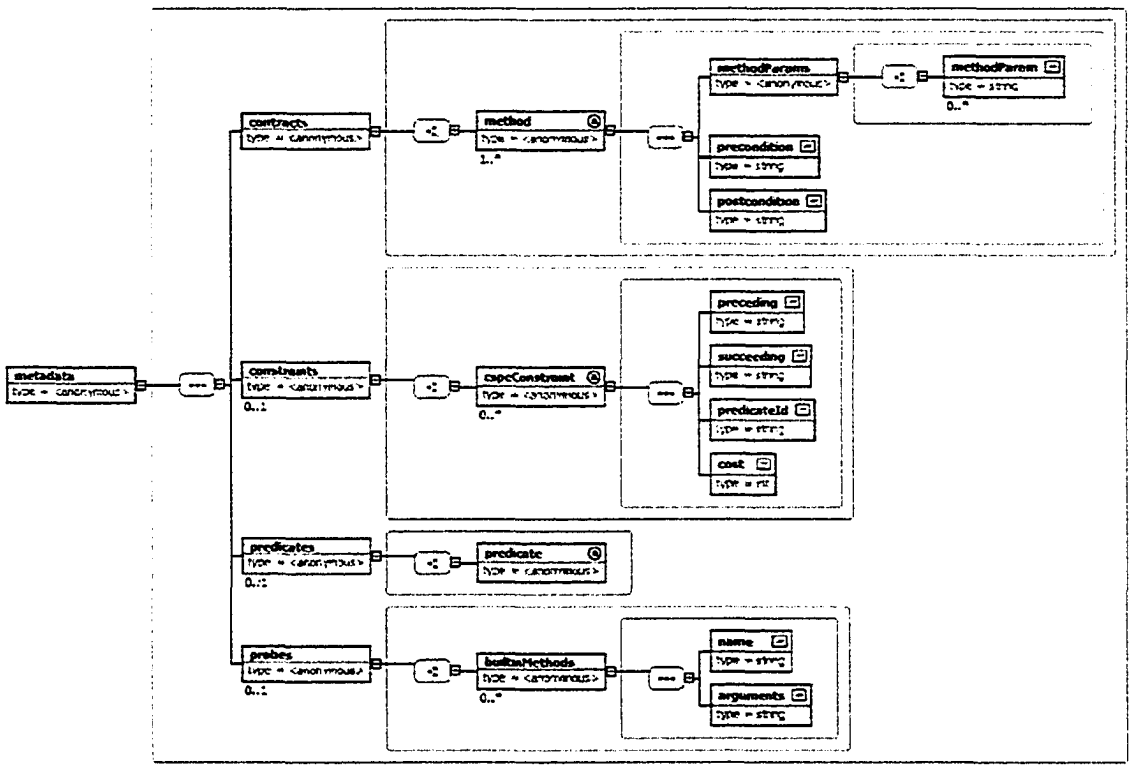


Figure 6-2: Graphical Representation of the XML Schema for Component Metadata

The CSPE-based test specification XML file is based on the XML Schema shown in Figure 6-3. The test specification is provided to the tool in terms of three elements (each containing): `requiredMethod` (the method specified according to *Required Methods Scheme* of Section 4.4.1), `requiredConstraint` (the constraints specified according to *Required Constraints Scheme* of Section 4.4.2), and `requiredConstraintPredicates` (the predicate test cases specified according to *Required Constraint Predicates Scheme* of Section 4.4.3), and `testCriterion` (the CSPE-based test criterion selected from the set defined in Section 4.3.2). The required methods, constraint, and predicates are specified using references from the component metadata file described above.

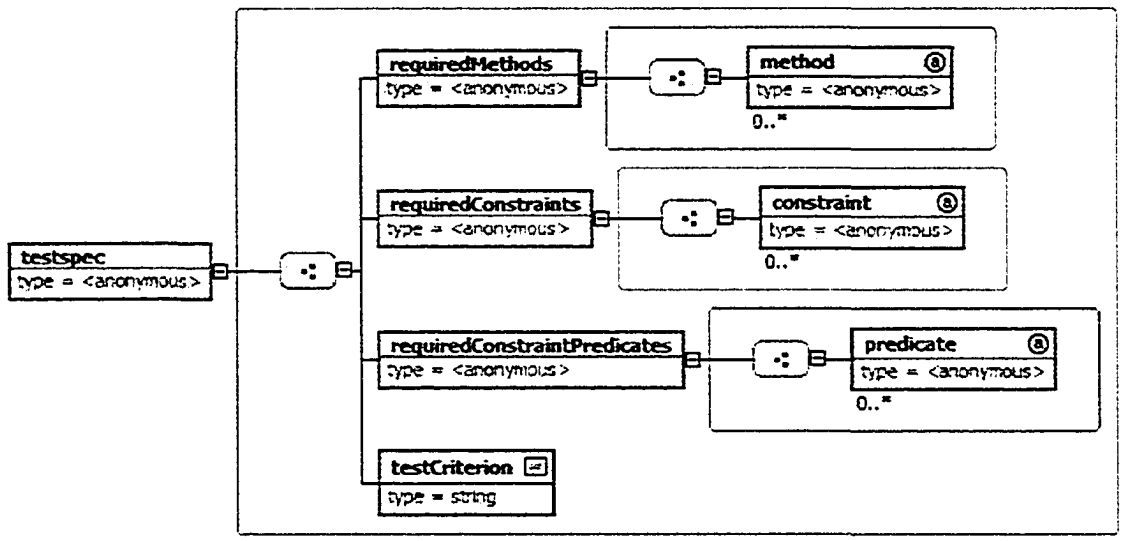


Figure 6-3: Graphical Representation of the XML Schema for Testing Specification

Chapter 7
Case Studies

The two main objectives of this section are to investigate the benefits of using the DRPP-based algorithm presented in Chapter 5, and to generate CSPE test sequences and investigate the fault detection effectiveness of the CSPE coverage criteria we proposed in Section 4.3.2. Section 7.1 describes a new component we will be used for our case studies: `Petstore` [29]. Section 7.2 uses the `Queue` and `Petstore` components to investigate the cost of DRPP test cases when compared to the technique proposed by Karçali and Tai. Section 7.3 uses the `Petstore` component to investigate how one of our main CSPE criteria performs in terms of fault detection. And lastly, Section 7.4 discusses the results.

7.1 Petstore System

The Petstore system is part of Sun's J2EE Blueprints collection [39]. The component presented here is based on an excerpt of the Petstore system used as an example in the book "JUnit in Action" [29] and made available by the authors for download from [30]. The component is the order fulfillment module of the system. It is responsible for handling the creation of new orders and their processing until fulfillment. In this section the component is modeled with UML from the perspective of the component vendor and the component user. It is described through a representative use case scenario, UML class and component diagrams, and OCL contracts which are defined for each of the component's interface methods. An overview of the implementation is also provided.

7.1.1 Use Case Scenarios

The functionality of the Petstore component is described in this section with the most representative use case scenario that describes how the system is used to order and purchase dog food.

Purchase Dog Food Use Case Scenario

1. The customer logs onto the web-based Petstore system in search of dog food.
2. The customer, after browsing the online catalogue, orders the dog food:
   - Triggers a `createOrder` method of the Petstore component; an order is created in the database with the specified item and current date
   - The state of the order is set to UNVERIFIED
3. The customer receives an email confirmation of the order, reads it over, and continues with the purchase following an email link.
   - Triggers the `verifyOrder` method of the Petstore component
   - The state of the order is set to VERIFIED
4. The pet store owner (system administrator) logs into the Petstore system, checks for new orders and prints them out.
   - Triggers the `orderCount` and `printOrder` interface methods of the Petstore component
5. The pet store owner charges the customers credit card and ships the dogfood.
6. The pet store owner logs into the Petstore system, pulls up the customer order and inputs in all of the shipping confirmation numbers and marks order fulfilled.
   - Triggers the `fulfillOrder` method of the Petstore component
   - The state of the order is set to FULFILLED
7. The customer receives the dogfood and feeds his/her dog.

7.1.2 UML Models

The Petstore order fulfillment component from the perspective of the component vendor is modeled in the following UML diagram.

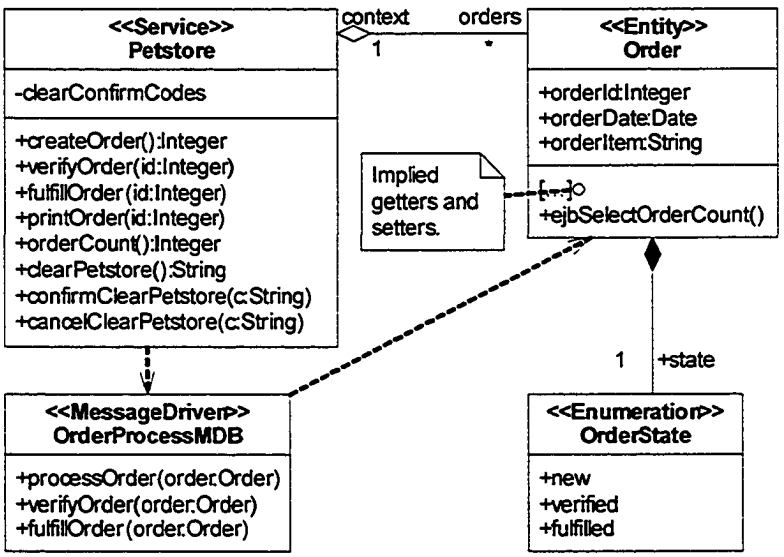


Figure 7-1: Petstore Order Fulfillment Component from the Perspective of the Component Vendor

The main class of the `Petstore` component is the `Petstore` bean. The `Petstore` class creates the order, and initiates each of the order processing steps until fulfillment. The system is capable of creating and processing multiple orders concurrently, and for this purpose it is designed to include a work queue implemented in a message-driven bean `OrderProccessMDB`. Each of the order processing operation initiated by the `Petstore` bean is queued and processed by the `OrderProccessMDB` at the earliest available processing time slot. Orders are stored in a database and accessed via a data access bean `Order`. The `Order` bean stores the properties of the order: id, date, item description, and state. The state of the order is encapsulated in the `OrderState` enumeration and changes with the consecutive processing steps of the order fulfillment. The `Petstore` bean may navigate all of the orders as well as get the number of orders in the system using the

`orderCount` method. The Petstore order fulfillment component from the perspective of the component user is modeled in the following UML diagram.

The Petstore can also be cleared, i.e., all of the orders are removed from the database. In order to clear the Petstore the client first invokes the `clearPetstore` method of the component and stores the confirmation code returned by the invocation. The client then finalizes the clear Petstore operation by invoking `confirmClearPetstore` with a valid confirmation code, or revokes the clear Petstore operation by invoking `cancelClearPetstore` with a valid code. A valid code generated on the first invocation to `clearPetstore` may only be used in one transaction ending in clear Petstore confirmation or cancellation.

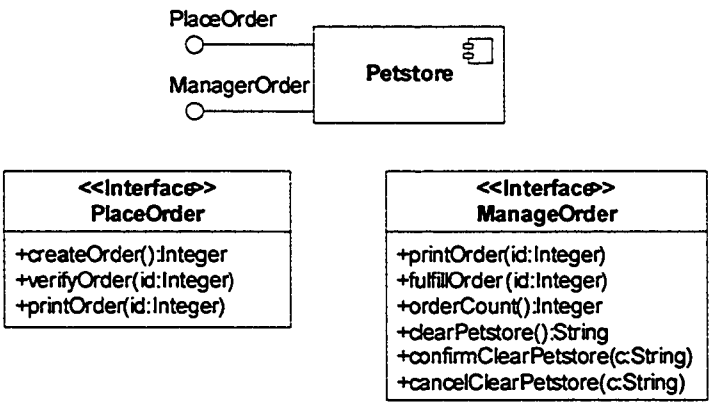


Figure 7-2: Petstore Order Fulfillment Component from the Perspective of the Component User

Similarly to the interfaces of the `Queue` component of Section 4.2, the provided interfaces of the Petstore component are tailored to the role the client accessing the component has in the system. The `PlaceOrder` interface is used by the customer's client application to create the order, verify the order, and check on its state. The

ManageOrder interface is used by the employee's client application to process the orders until fulfillment, get the number of orders in the system, and clear the Petstore in preparation for new inventory.

7.1.3 Interface Contracts

To employ CSPE-based test strategies, it is necessary to have contract descriptions for the component interface methods. Here preconditions and postcondition are defined in OCL [20]. Table 7-1, below, contains the listing of the OCL contracts for the Petstore component.

Table 7-1: Petstore Component Interface Method OCL Contracts

```
context: Petstore::createOrder(orderDate:Date,orderItem:String):Integer
pre:    --
post:   let order : Order.allInstances->select(o | o.orderId = orderId),
        Order.allInstances->exists(o | o.orderId = orderId)
        and
        order.orderDate = orderDate
        and
        order.orderItem = orderItem
        and
        order.state = OrderState::UNVERIFIED
        and
        Order.allInstances->size() = Order.allInstances->size()@pre + 1
```

```
context: Petstore::verifyOrder(orderId:Integer)
pre:    let order : Order.allInstances->select(o | o.orderId = orderId),
        Order.allInstances->exists(o | o.orderId = orderId)
        order.state = OrderState::UNVERIFIED
post:   let order : orders->select(o | o.orderId = orderId),
        order.state = OrderState::VERIFIED
```

```
context: Petstore::fulfilOrder(orderId:Integer)
pre:    let order : Order.allInstances->select(o | o.orderId = orderId),
        Order.allInstances->exists(o | o.orderId = orderId)
        and
        order.state = OrderState::VERIFIED
post:   let order : Order.allInstances->select(o | o.orderId = orderId),
        order.state = OrderState::FULFILLED
```

```
context: Petstore::printOrder(orderId:Integer):String
pre:    Order.allInstances->exists(o | o.orderId = orderId)
post:   let order : Order.allInstances->select(o | o.orderId = orderId),
        return.contains(orderId)
        and
        return.contains(order.orderDate)
        and
        return.contains(order.orderItem)
```

```
context: Petstore::orderCount():Integer
pre:    --
post:   return = Order.allInstances->size()
```

```
context: Petstore::clearPetstore():String
pre:    --
post:   clearConfirmCodes->exists(return)
```

```
context: Petstore::confirmClearPetstore(code:String)
pre:    clearConfirmCodes->exists(code)
post:   Order.allInstances->size() = 0
        and
        not clearConfirmCodes->exists(code)
```

```
context: Petstore::cancelClearPetstore(code:String)
pre:    clearConfirmCodes->exists(code)
post:   not clearCofirmCodes->exists(code)
```

7.1.4 CSPE Constraints

We list in Table 7-2 all of the CSPE constraints for the Petstore Component as we will be using them to generate CSPE-based test sequences. Similarly, Table 7-3 shows all of the related CSPE constraint predicates.

Table 7-2: Petstore Component CSPE Constraints

c1=(#,createOrder,true)	c25=(fulfilOrder,createOrder,true)	c49=(clearPetstore,createOrder,true)
c2=(#,verifyOrder,p1)	c26=(fulfilOrder,verifyOrder,p1)	c50=(clearPetstore,verifyOrder,p1)
c3=(#,fulfilOrder,p2)	c27=(fulfilOrder,fulfilOrder,p2)	c51=(clearPetstore,fulfilOrder,p2)
c4=(#,printOrder,p3)	c28=(fulfilOrder,printOrder,p3)	c52=(clearPetstore,printOrder,p3)
c5=(#,orderCount,true)	c29=(fulfilOrder,orderCount,true)	c53=(clearPetstore,orderCount,true)
c6=(#,clearPetstore,true)	c30=(fulfilOrder,clearPetstore,true)	c54=(clearPetstore,clearPetstore,true)
c7=(#,confirmClearPetstore,p4)	c31=(fulfilOrder,confirmClearPetstore,p4)	c55=(clearPetstore,confirmClearPetstore,p4)
c8=(#,cancelClearPetstore,p4)	c32=(fulfilOrder,cancelClearPetstore,p4)	c56=(clearPetstore,cancelClearPetstore,p4)

c9=(createOrder,createOrder,true)	c33=(printOrder,creatOrder,true)	c57=(confirmClearPetstore,createOrder,true)
c10=(createOrder,verifyOrder,p1)	c34=(printOrder,verifyOrder,p1)	c58=(confirmClearPetstore,verifyOrder,p1)
c11=(createOrder,fulfilOrder,p2)	c35=(printOrder,fulfilOrder,p2)	c59=(confirmClearPetstore,fulfilOrder,p2)
c12=(createOrder,printOrder,p3)	c36=(printOrder,printOrder,p3)	c60=(confirmClearPetstore,printOrder,p3)
c13=(createOrder,orderCount,true)	c37=(printOrder,orderCount,true)	c61=(confirmClearPetstore,orderCount,true)
c14=(createOrder,clearPetstore,true)	c38=(printOrder,clearPetstore,true)	c62=(confirmClearPetstore,clearPetstore,true)
c15=(createOrder,confirmClearPetstore,p4)	c39=(printOrder,confirmClearPetstore,p4)	c63=(confirmClearPetstore,confirmClearPetstore,p4)
c16=(createOrder,cancelClearPetstore,p4)	c39=(printOrder,cancelClearPetstore,p4)	c64=(confirmClearPetstore,cancelClearPetstore,p4)
c17=(verifyOrder,createOrder,true)	c41=(orderCount,createOrder,true)	c65=(cancelClearPetstore,createOrder,true)
c18=(verifyOrder,verifyOrder,p1)	c42=(orderCount,verifyOrder,p1)	c66=(cancelClearPetstore,verifyOrder,p1)
c19=(verifyOrder,fulfilOrder,p2)	c43=(orderCount,fulfilOrder,p2)	c67=(cancelClearPetstore,fulfilOrder,p2)
c20=(verifyOrder,printOrder,p3)	c44=(orderCount,printOrder,p3)	c68=(cancelClearPetstore,printOrder,p3)
c21=(verifyOrder,orderCount,true)	c45=(orderCount,orderCount,true)	c69=(cancelClearPetstore,orderCount,true)
c22=(verifyOrder,clearPetstore,true)	c46=(orderCount,clearPetstore,true)	c70=(cancelClearPetstore,clearPetstore,true)
c23=(verifyOrder,confirmClearPetstore,p4)	c47=(orderCount,confirmClearPetstore,p4)	c71=(cancelClearPetstore,confirmClearPetstore,p4)
c24=(verifyOrder,cancelClearPetstore,p4)	c48=(orderCount,cancelClearPetstore,p4)	c72=(cancelClearPetstore,cancelClearPetstore,p4)

Table 7-3: Petstore Component CSPE Constraint Predicates

Predicate Literal	Constraint Predicate
p1	let order : Order.allInstances->select(o | o.orderId = orderId), Order.allInstances->exists(o | o.orderId = orderId) and order.orderState=OrderState::UNVERIFIED
p2	let order : Order.allInstances->select(o | o.orderId = orderId), Order.allInstances->exists(o | o.orderId = orderId) and order.orderState=OrderState::VERIFIED
p3	Order.allInstances->exists(o | o.orderId = orderId)
p4	clearConfirmCodes->exists(code)

7.1.5 Implementation

The `Petstore` order fulfillment component is implemented in an Enterprise Java Bean (EJB) component model [38], and deployed on the JBoss Application Server [24]. The component consists of three EJBs, a set of utility classes, and the client interfaces. The three EJBs are `OrderEJB`, `OrderProcessorMDB`, and `PetstoreEJB`.

The `OrderEJB`, an entity bean, contains all of the order information and stores it in a database via a persistence mechanism provided by the component container (JBoss). It provides accessor methods (get and set type methods) for each of the order attributes (date, item, and state), as well as a mechanism for retrieving an order based on its id.

The `OrderProcessorMDB`, a message driven bean, receives order processing requests from the `Petstore` component and executes them on the orders stored in the database. For instance, on receiving a new order request, the bean retrieves the order according to the id provided in the request, sets the state of the order to `UNVERIFIED` and sends out the email notification to the customer. Similarly, it is responsible for setting the state of the order to `VERIFIED` and then `FULFILLED` and executing the processing tasks in each case, respectively.

The `PetstoreEJB`, a service bean, is the main entry point of the component. It exposes all of the interface methods which the clients use to exercise the component. These include: `createOrder`, `confirmOrder`, `fulfillOrder`, `printOrder`, `orderCount`, `clearPetstore`, `confirmClearPetstore`, and `cancelClearPetstore`. Some of these methods access the `OrderEJB` object directly

(e.g., `printOrder`, `orderCount`), while others send requests to the `OrderProcessorMDB` (e.g., `createOrder`, `verifyOrder`).

Utility classes implemented as part of the Petstore component include: `JMSUtil`, `JNDIUtil`, `JNDINames`, `PetstoreException`, and `OrderState`. `JMSUtil` is a utility class used to simplify access to the Java Message Service (JMS) [40], a service used to send requests to the `OrderProcessorMDB`. `JNDIUtil` and `JNDINames` are utility classes used to access the beans and other component resources using the Java Naming and Directory Interface (JNDI) [41]. `PetstoreException` is a Java exception that is thrown when the component is not correctly used, and the `OrderState` is an enumeration class representing the different states of the order.

In addition to the EJBs and utility classes, the EJB client interfaces are also implemented according to the EJB specification. These include object interfaces and home interfaces for the `OrderEJB` and the `PetstoreEJB` (for more details refer to [38], or preferably a good book on the subject [35]).

7.2 Cost Evaluation of DRPP Test Suites

Ideally, we would like to assess the reduction in cost brought by the DRPP algorithm. However, this raises two questions. Since we do not have cost data for test suites, what surrogate measure can we use? Second, assuming we have such a measure, what do we compare the results of DRPP with? First, we will use test suite size, measured as number of executed methods, as a surrogate measure for cost. It is reasonable to expect the cost of developing and running test suites to be proportional to the number of methods executed.

Though different methods usually entail varying costs, this is expected to average out over an entire test suite and is therefore a good approximation at a test suite level. Second, we will compare DRPP test suites with the algorithm by Karçali and Tai (KT) reviewed in Section 3.2.3. The test suites generated by the `PrestoSequence` tool will be analyzed on both the `Queue` and `Petstore` components.

Sequences were generated for all the different CSPE constraint criteria for a fixed CSPE constraint predicate criterion: Predicate Coverage. Since KT's approach does not address the coverage of predicates and simply covers Possibly Valid constraints once, this corresponds to our Predicate Coverage criterion. By fixing the CSPE constraint predicate criterion to Predicate Coverage, we thus make sure that we can compare the two techniques and investigate whether the DRPP algorithm is good at minimizing test sequences. Using more complex predicate criteria would only increase the relative improvement of DRPP over KT[14]. Sequences were therefore generated for A-PC, AP-PC, N-PC, NP-PC, and ANP-PC. Figure 7-3 is a chart comparing the size of the DRPP-based approach to the size of the KT algorithm on `Queue` in terms of number of methods executed in the test suites. Note that one test suite was automatically generated for each technique.

---

[14] We could have adapted KT's approach to handle other predicate coverage criteria, such as Prime Implicant Coverage (Section 3.3), by following the same tree based approach (Section 3.2.3). This would have however resulted in much more redundancy than what we have already observed on simple examples and would have disadvantaged KT's approach over ours.

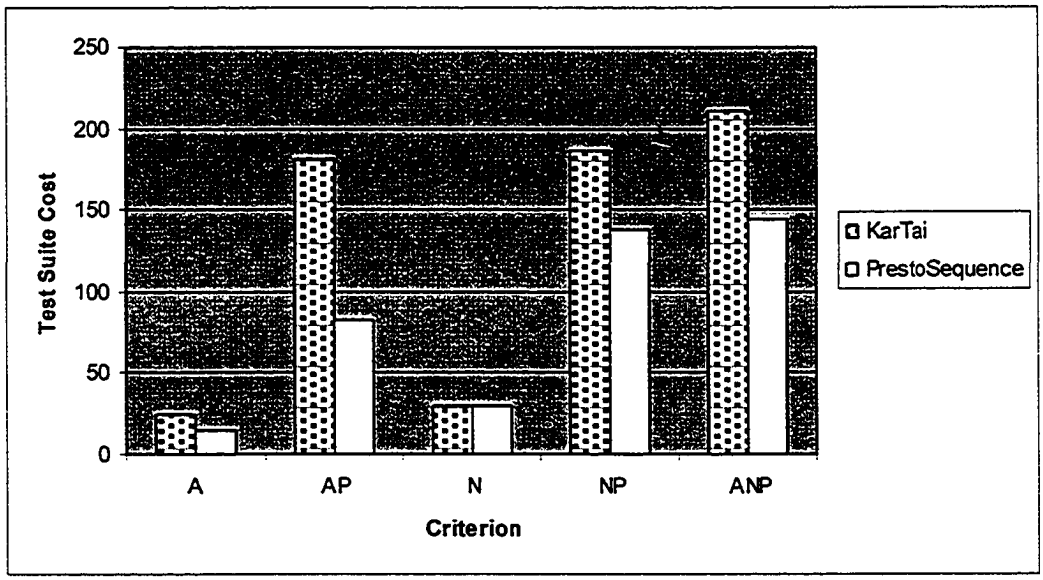


Figure 7-3: Costs of DRPP-based vs. KT Generated Test Sequences for Queue

The DRPP-based test sequences clearly show a significantly lower size for all but one CSPE criterion (N). In that particular case, the tree-based breadth-first approach taken by the KT algorithm is able to grow the paths on the tree such that each path terminates with a Never Valid constraint and the total size is optimal. The relative test sequence sizes between the CSPE criteria vary greatly, and Figure 7-4 shows the relative improvement brought by DRPP. If we exclude N, improvement percentages are substantial and vary between 25% and 54%.

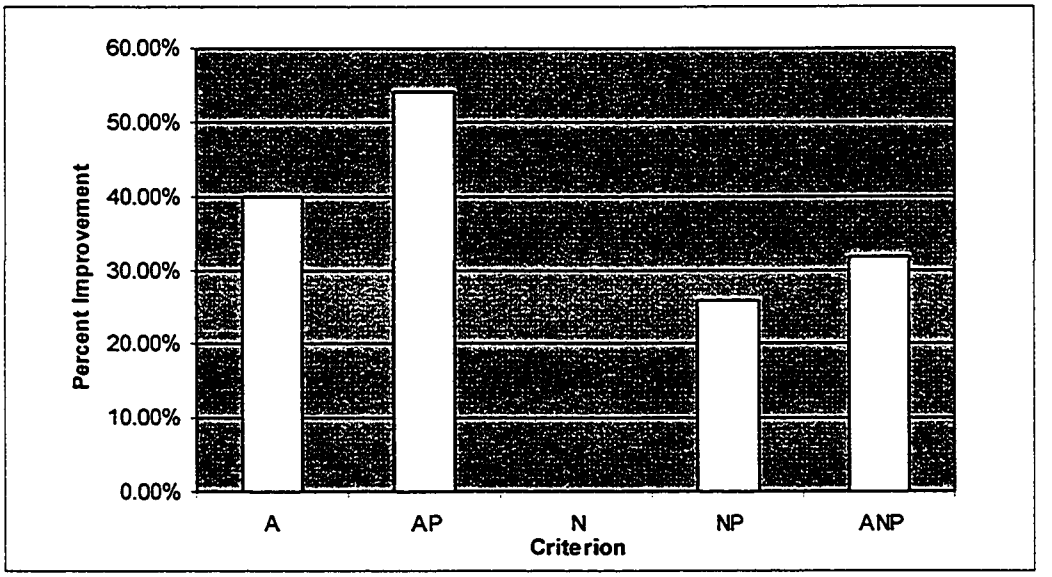


Figure 7-4: Improvement of the DRPP-based Approach over the KT Agorithm for `Queue`

Similar charts are shows in Figure 7-5 and Figure 7-6 for the Petstore component. Here the N CSPE criterion does not apply but we can see that results are very consistent with those of `Queue`. Relative improvements range from 26% to 53%.

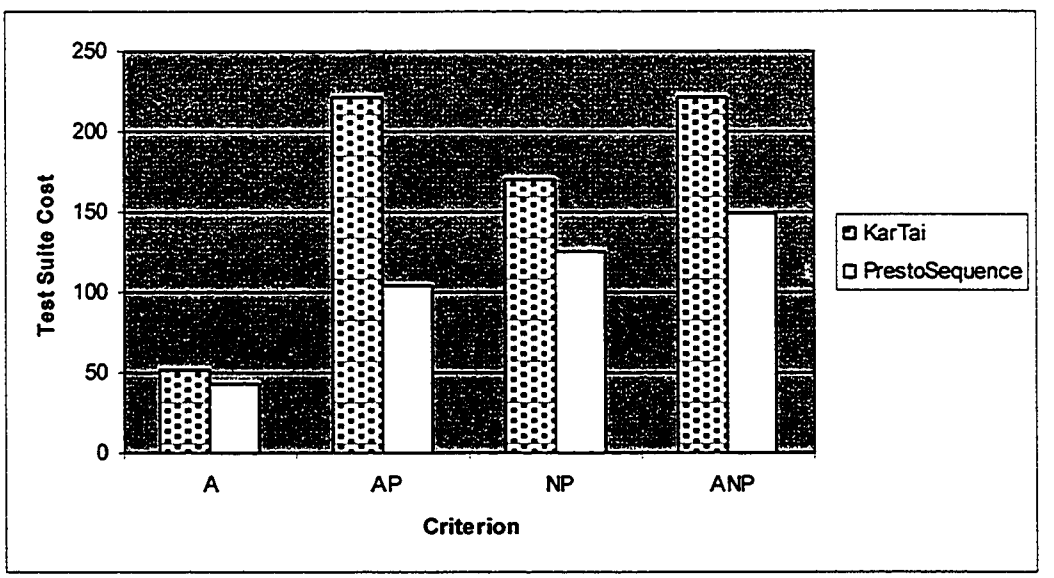


Figure 7-5: Costs of DRPP-based vs. KT Generated Test Sequences for `Petstore`

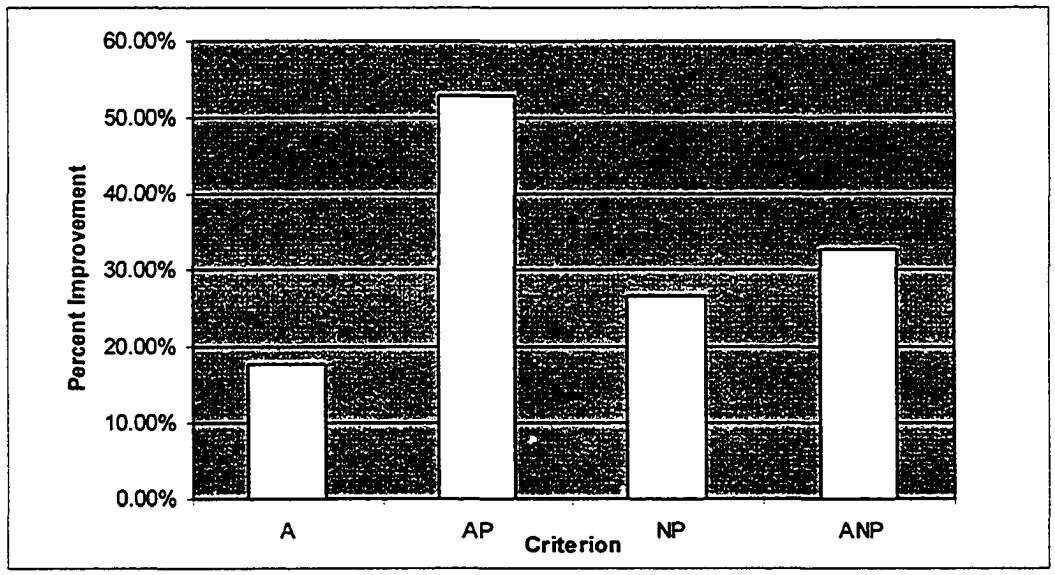


Figure 7-6: Improvement of the DRPP-based Approach over the KT Algorithm for `Petstore`

7.3 Analyzing the Fault Detection Effectiveness of Test Suites

In this section our goal is to analyze the effectiveness at detecting faults of the CSPE coverage criteria based on the `Petstore` component, which is complex enough to enable such an investigation. From a practical standpoint we want to determine whether deriving such test suites based on interface method contracts brings any significant advantage over simpler test strategies. In order to investigate this question, a number of issues have to be addressed. Since we have no known faults on the components we use, how do we seed faults to allow us to experiment (Section 7.3.1)? Second, with what other test strategy should we compare our CSPE results (Section 7.3.2)? Third, how do we automate the experimental process (Section 7.3.3).

7.3.1 Mutants

It is common in testing research to resort to seeding faults to investigate the effectiveness of testing techniques [32]. The main reason is practical: researchers rarely have access to large numbers of actual faults. The question is now how to seed such faults in a systematic and unbiased manner? To do so, most researchers have used mutation operators [27], that is a set of change operators modeling small, atomic changes in the code.

For the `Petstore` case study, we assumed that the component container properly used the component interface and that this interface was correct to start with. In a component-based context, test cases (in our case CSPE-based test cases) are derived to investigate whether the component can work correctly on the selected platform, i.e., to investigate the impact of defects in the component platform (or other components) on the component under investigation. This may also be seen as testing the robustness of the component to faults in the container and platform. To do so, the most straightforward solution is to seed mutants in the component platform (or other components). This turns out to be very difficult because of the complexity of the deployment platform, or even impossible, depending on the actual deployment platform used[15]. Another simpler solution is to simulate platform failures (e.g., the platform fails to throw an exception). Two alternatives were considered: (i) create a wrapper between the component under test

[15] In our case, since the deployment platform we use (i.e., JBoss) is open source, this solution could be considered, although JBoss's complexity would make it difficult to implement. With other deployment platforms which are not open source, this solution is simply not possible.

and the platform that simulates the failures, (ii) simulate the failures directly in the component. We chose the latter as it requires less test scaffolding.

A set of mutation operators were then selected and faults were seeded manually and randomly through the entire component code (thus generating mutant programs). The mutation operators were selected so as to potentially simulate component deployment failures. So, for example, interface mutation operators [23] were not selected as we assumed that the component container used correctly the component interface and that this interface was correct. In our experiment, we used both generic mutation operators [27][16] and others specifically defined for Java [26].

Ideally, the Petstore mutation operators should simulate only plausible deployment platform failures. Since the deployment platform involves the component container and a database, we selected the mutation operators reported in Table 7-4 [26, 27] for the following reasons:

AOR: This operator can be used in the component to change data values sent to the plateform, thus simulating a lost message between the component and the platform.

RSR, SDL: These operators can be used in the component to change requests sent to a database, or answers received from it, thus simulating the database failing to accurately update some data.

[16] Although the mutants presented in [27] were developed for use in the Fortran language, they are based on basic programming principles and can be easily adapted to other imperative languages.

CRP, ROR, UOI, and SVR: These operators can be used in the component to affect the control flow (e.g., case, if statements) and thus simulate the platform failing to deliver messages in the right order or at the right moment (e.g., the component expects an acknowledgement but receives another message).

EHR: This operator can be used in the component to avoid handling an exception, thus simulating the platform failing to throw it.

Table 7-4: Mutation Operators Used in the Petstore Case Study Selected from [26, 27]

Type	Description
AOR	*Arithmetic Operator Replacement*: An arithmetic operator is replaced by another operator.
ROR	*Relational Operator Replacement*: A relational operator is replaced by another operator.
UOI	*Unary Operator Insertion*: An arithmetic expression is negated.
RSR	*Return Statements Replacement*: A statement-is replaced by the return statement.
SDL	*Statement Deletion*: A statement is deleted.
CRP	*Constant Replacement*: A constant value is modified, e.g., a numeric constants incremented/decremented by a fixed value, or a character of a string is replaced by another.
SVR	*Scalar Variable Replacement*: A scalar variable is modified, e.g., a number variable in incremented/decremented by a fixed value.
EHR	*Error Handler Removal*: An error handler (the catch block) is removed.

Using the mutation operators of Table 7-4, 102 mutants where seeded in the Petstore component source code. When seeding the mutants, any generated mutants that resulted in source code that did not compile were discarded.

A couple of examples of the mutants created with the mutant operators of Table 7-4 are shown below. A mutant is seeded in the `PetstoreEJB.java` source file using the Relational Operator Replacement (ROR) mutation operator, shown in Figure 7-6. The correct and mutated source codes read:

Original source: `if (orderState == OrderState.UNVERIFIED) {`

Mutant source: `if (orderState >= OrderState.UNVERIFIED) {`

Figure 7-7: An Example ROR Mutant

The mutant is applied such that relation operator == is replaced with another relation operator >=. This mutant results in a failure since even though only the order state `UNVERIFIED` should be admitted, any order state with constant value greater than that of `UNVERIFIED` is erroneously admitted (`VERIFIED` or `FULFILLED`).

Another example of a seeded mutant is in the `OrderProcessorMDB.java` source file, shown in Figure 7-8. The Statement Deletion (SDL) type mutation operator removes a `break` statement from a Java `switch` block:

Correct source:

```
switch (orderState) {
case OrderState.NEW:
   proceedOrder(order);
   break;
case OrderState.UNVERIFIED:
   verifyOrder(order);
   break;
case OrderState.VERIFIED:
   fulfillOrder(order);
   break;
case OrderState.FULFILLED:
   logOrder(order);
   break;
}
```

Mutant source:

```
switch (orderState) {
case OrderState.NEW:
   proceedOrder(order);
   break;
case OrderState.UNVERIFIED:
   verifyOrder(order);

case OrderState.VERIFIED:
   fulfillOrder(order);
   break;
case OrderState.FULFILLED:
   logOrder(order);
   break;
}
```

Figure 7-8: An Example SDL Mutant

As a result of this mutant, execution of the `switch` block where the `orderState` is UNVERIFIED will result in the program falling through to the second `case`, and in addition to verifying the order (`verifyOrder` method) the order will also be fulfilled (`fulfillOrder` method). A full listing of all of the mutants seeded in the case study is found in Appendix A.

7.3.2 Test Suites

As a baseline of comparison, we decided to use the *all methods and exceptions* (AME) test strategy.

> *Definition 7-1 All Methods and Exceptions (AME)*: Requires that all of the methods and all of the exception instances be covered at least once by a compliant test suite. An exception instance is defined as an exception type thrown by a method, i.e., the same exception type may be thrown by a number of methods and each one is an exception instance.

AME is a simple but thorough testing technique applicable in a system without available source code, and therefore a good baseline to compare the CSPE technique against. It is based on the two test coverage criteria AC_{met} and AC_{ex} [23] defined in Section 3.4.

For example, the AME testing technique would require that all the interface methods of the `Petstore` component be tested at least once, and that each method that throws the `PetstoreException` be tested at least once to verify the exception instance. An exception instance is tested (verified) by executing the test suite such that the exception is

thrown. Then, if the test suite catches the exception the system is considered to be correct or else if the exception is not caught this is identified as a failure.

Two test suites were therefore developed for the Petstore case study, one for each of the techniques employed: CSPE and AME. Test drivers are implemented using the Apache Cactus framework [45], based on the popular Java testing framework JUnit [31]. Each test suite consists of a set of test cases, each of these consisting of a method sequence. In addition to test cases themselves, test drivers include scaffolding for each test case: test inputs, test oracles. A test suite is contained in a single Java class extending the Cactus `ServletTestCase` class.

The CSPE Cactus test suite is based on the test cases (method sequences) generated using the *PrestoSequence* tool (Chapter 6). There is no specific targeted functionalities of the component specified for testing so that the entire component is tested. The CSPE constraint and predicate criterion employed is Always Never Possibly Valid – Predicate Coverage (ANP–PC). We used ANP as this is the most complete CSPE criterion and we use the weakest predicate coverage criterion (PC) since this happens to detect (Section 7.4.1) all our mutants and no stronger criterion is therefore required.

An AME test suite includes an interface method for every test case of the test suite, as well as two methods shared by all the test cases used to initialize the test case and tear it down after completion.

For both AME and CSPE test suites, the test inputs are instances of mock Petstore objects (order dates, order items, orders). The test oracles are JUnit assertion statements,

e.g., `assertTrue`, `assertEquals`. When a test case (Cactus test suite method) encounters an assertion that fails, a failure is reported. The test cases in a test suite are executed in an arbitrary order (Cactus implementation detail) and a report is produced on the number of test cases that passed or failed; reports of failed test cases are accompanied by detailed exception traces.

Below is an example of a `createOrder` interface method and its scaffolding, an excerpt found in both CSPE and AME base test suites.

```
Date[] dates = { new Date(234234), new Date(123123), new Date(345345) };
String[] items = { "item 1", "item 2", "item 3" };
Integer orderId0 = store.createOrder(dates[0], items[0]);
Thread.sleep(100);
OrderLocal order0 = orderHome.findByPrimaryKey(orderId0);
assertEquals((Integer)(dates[0].hashCode()
      + items[0].hashCode()),orderId0);
assertEquals(new Integer(OrderState.UNVERIFIED),
      order0.getOrderState());
```

The first two statements are the test inputs, an array `dates` of order dates (pseudo random) and an array `items` of item names. The second statement uses the initialized `store` EJB object to create the order, and return the order id stored in `orderId0`. The `Thread.sleep(100)` statement is inserted into the scaffolding of the component to process the new order through the message-driven bean `OrderProcessorMDB`.[17] Up to this point the scaffolding consists of test input and test drivers parts. The last three statements are the test oracles of the test case. The `orderHome` interface is used to

directly retrieve the `Order` object identified by `orderId0`. Then, the first `assertEquals` statement verifies that the order id was correctly set, and the next `assertEquals` statement verifies that the order is in the `UNVERIFIED` state (as would be expected of a newly created order).

7.3.3 Test Infrastructure

Test suites are executed using the Apache Cactus testing framework [45]. The specific feature in Cactus that requires its use (over JUnit on which it is based) is that it enables the testing of Enterprise JavaBeans from within the component container. All of the 102 mutants of the Petstore component are tested.

Instead of maintaining a version of the entire source code for each of the mutants, the mutants are stored as patch files[18]. In order to test the component for the given mutant, the correct source code was patched from a repository of mutants (patch files). An example of a patch file representing the mutant of Figure 7-7 is shown below in Figure 7-9.

---

[17] Without the sleep statement allowing the component to fully complete the order creation the test case would not represent real world use of the component, i.e., no software client in real use would invoke the interface methods of the component one right after the other as in the case of the test case.

```
*** junitbook/ejb/service/PetstoreEJB.java~  2005-08-14 02:14:34.000000000 -0400
--- junitbook/ejb/service/PetstoreEJB.java   2005-08-14 02:21:09.000000000 -0400
***************
*** 51,57 ****

                    // catch if we've got the state right
                    Integer orderState = order.getOrderState();
!                   if (orderState == OrderState.UNVERIFIED) {
                            JMSUtil.sendToJMSQueue(JNDINames.QUEUE_ORDER, order
                                    .getOrderId(), false);
                    } else {
--- 51,57 ----

                    // catch if we've got the state right
                    Integer orderState = order.getOrderState();
!                   if (orderState >= OrderState.UNVERIFIED) {
                            JMSUtil.sendToJMSQueue(JNDINames.QUEUE_ORDER, order
                                    .getOrderId(), false);
                    } else {
```

Figure 7-9: An Example Patch File for ROR Type Mutant

Each mutated component (Mutant) was tested and the test suite results reported in XML format. The XML file contains a header with a summary of the test cases that passed or failed, and a set of more detailed reports for each of the test cases.

The patch file and XML report format mechanisms allowed for the testing process to be automated with basic scripting facilities available on the testing platform. At a high-level the testing process consisted of the following steps:

1. The correct version of the Petstore component source code was recovered from a previously mutated version.
2. The given mutant was applied by patching the source code.
3. The mutated Petstore component was compiled and deployed in the component container along with the precompiled Cactus test suites.

---

[18] Patch file commonly created using the `diff` utility, part of most Unix/Linux environments and available for Windows.

4. The resulting test reports are stored in XML files, one for each of the mutants tested, named consistent with the mutant id.

The set of XML files was then processed using XSL Transformations (XSLT) the use of which facilitated compilation of the test reports into the results presented in the next section.

7.4 Results

The results of the case study were compiled by collecting all of the test suite output XML files for each of the mutants and determining the state of the mutant. A mutant is killed if one of the test cases of the test suite fails. As a heuristic, a mutant is considered equivalent if neither test suites fail and after verification of the code. Comparisons of the mutants killed by the CSPE to those killed by the AME test suites were performed to evaluate the automated CSPE technique developed in this thesis. In this subsection we first analyze and compare mutant scores of test suites (Section 7.4.1). Then, the average mutant detection rates of individual test cases are compared (Section 7.4.2) and the practical implications of the results are discussed in Section 7.4.3.

7.4.1 Analysis of Mutant Scores

Out of the entire set of 102 mutants, 22 were identified as equivalent, 81 were killed by the CSPE test suite, and 76 were killed by the AME test suite. Therefore, the CSPE test suite killed all non-equivalent mutants, five more than the AME test suite.

The size of the CSPE test suite is 149 methods contained in 45 test cases, and the size of the AME test suite is 22 methods contained in 5 test cases. The sizes of the test

suites were measured by counting only the method invocations on the component and do not include test input instantiations or oracles (test suite scaffolding). A summary of the test suites' effectiveness and size is found in Table 7-5.

Table 7-5: Quantitative Results of the Petstore Case Study

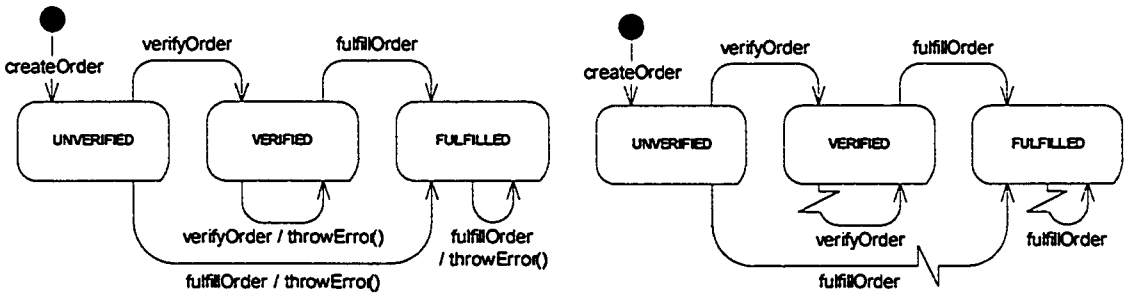
	CSPE	AME
Mutants Killed	81	76
Number of Test Cases	45	5
Test Suite Size	149	22

From the results in Table 7-5, we can therefore conclude that CSPE test suites are more effective at catching faults but are much more expensive. It is difficult to a priori determine whether the faults missed by AME should be expected to be critical and common in object-oriented software. We therefore analyzed the location and nature of these five mutants killed only by CSPE.

These mutants were all found in two specific areas of the `Petstore` component, that is where the order processing functionality (1 mutant) and the `Petstore` clear and confirm/cancel functionality (4 mutants) of the component are implemented. Both of these areas in the component share a common characteristic in that they implement state-based behavior, that is behavior that depends on the state of the component.

The order processing functionality of the component is found in the `OrderProcessMDB` class. The message driven bean receives order processing requests

from the component, and processes the order according to its state. Figure 7-10 (a) shows a state diagram of an order as it is (correctly) processed: only three states are necessary to describe the behavior of orders and transition labels are the component methods involved in order processing. Mutants were seeded in the source code of the `Petstore` component and effectively produced failures in each of the transitions of the order processing functionality. One of these mutants was only killed by the CSPE test suite. Figure 7-10 (b) shows the incorrect processing of orders resulting from the mutant.



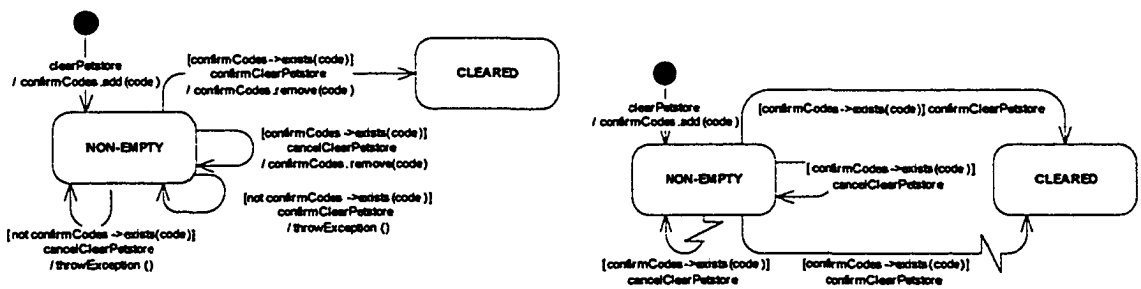
(a) Correct behavior **(b) Incorrect behavior**

Figure 7-10: State Diagram of the Order Processing Functionality

This mutant enables an already fulfilled order to be once again fulfilled, which is erroneous, without an exception being thrown. The correct source code represented in Figure 7-10 (a) of the component is able to properly check the requested operation, and in case it is erroneous throw the appropriate exception (`throwError()` in the diagram). The mutated source code represented in Figure 7-10 (b) fails to properly check the requested operation and the result is an erroneous state of the component. This mutant is in fact the one shown in Figure 7-7.

The reason that the CSPE technique killed all of the mutants seeded in the order processing functionality of the component is that all possible transitions are modeled in the derived CSPE constraints. The constraint `c27` found in Table 7-2 defines a constraint that covers the invocation of `verifyOrder` twice in a sequence. Of note, is that the AME technique was able to kill all the other mutants in the invalid transitions. This is because exception instance coverage addresses invalid invocations.

The clear and confirm/cancel functionality of the component is another other stateful part of the component. It is used to clear the Petstore database of all of the orders. This is done through the `clearPetstore`, `confirmClearPetstore`, and `cancelClearPetstore` methods. The `clearPestore` method returns a confirmation code, and clearing the database requires that the same confirmation code be provided in the `confirmClearPetstore` method. In the same way, the clearing of the database can be revoked by calling the `cancelClearPetstore` method with the confirmation code. A set of confirmation codes is maintained in the component. This behavior is described by the state diagrams in Figure 7-12.



(a) Correct behavior (b) Incorrect behavior

Figure 7-12: State Diagram of the Petstore Clearing Functionality

In a correct scenario represented in Figure 7-12 (b), `clearPetstore` is first called on a non-empty `Petstore` component and the provided confirmation code is stored in `confirmCodes`. Then, `confirmClearPetstore` triggers the removal (`confirmCodes.remove(code)` action) of the confirmation code. When the client tries to invoke the `confirmClearPetstore` method again with that same confirmation code, the confirmation code is not found in the set off codes and an exception is thrown.

The incorrect behavior, represented in Figure 7-12 (b), begins with a call to either `confirmClearPetstore` or `cancelClearPetstore`. As a result of the mutants neither of the methods properly remove the valid confirmation codes from the set of maintained codes after their execution. The following invocations which then result in failures, represented by two transitions denoted with broken type arrows, successfully use the same "stale" confirmation code.

This allows the Petstore to be cleared multiple times with the same confirmation code, or the clearing to be canceled with the same confirmation code multiple times. The CSPE-based suite tested the abovementioned specific method invocations with the constraints below.

```
c63=(confirmClearPetstore,confirmClearPetstore,p4)
c64=(confirmClearPetstore,cancelClearPetstore,p4)
c71=(cancelClearPetstore,confirmClearPetstore,p4)
c72=(cancelClearPetstore,cancelClearPetstore,p4)
```

CSPE therefore ensured that the faulty transitions in Figure 7-12 (b) were exercised whereas the AME technique only verifies the basic `Petstore` clearing functionality by

asserting that the database is empty after a `confirmClearPetstore` with valid confirmation code, that it has been unaltered after a `cancelClearPetstore` with valid confirmation code, or that it has been unaltered and an exception has been thrown after a `confirmClearPetstore` with an invalid confirmation code.

7.4.2 Analysis of Test Case Effectiveness

Another way to look at differences between CSPE and AME is to look at the effectiveness of individual test cases in killing mutants. This is of interest for two reasons. In practice, test cases may take a substantial time to run and verify. If test cases for CSPE turn out to have a higher probability to detect faults, then failures would be triggered earlier during the test process and more time would then be available for debugging, correction, and testing. Furthermore, if for any practical reason, it turned out not to be possible to run the entire test suite on a component, then high detection probability for individual test cases would help retain more of the complete test suite detection capability.

For each mutant, we first compute for both test suites the percentage of test cases killing the mutant. Let ET_{CSPE} and ET_{AME} denote the ratio of the number of effective test cases for the CSPE and AME. These ratios are estimates of the average test case detection probability for each test technique. For all non-equivalent mutants, we then compare the ET_{CSPE} and ET_{AME} distributions by using a paired t-test {Devore, 1991 #55}. The test is one-tailed as we expect CSPE to perform better than AME for reasons that have been described in Section 7.3.2. The results are shown in Table 7-6 and clearly

suggest a very significant difference (p-value < 0.0001) of 12% in average fault detection probability among test cases.

A plot of the differences between the paired samples of ET_{CSPE} and ET_{AME} is shown in Figure 7-11. The 81 mutants are plotted on the x axis, and on the y axis is the value of ET_{AME} - ET_{CSPE} for each mutant. The horizontal lines in the plot of Figure 7-11 represent, from top to bottom: the 0 difference point, the upper limit of the standard deviation of the mean difference, the mean difference, and the lower limit of the standard deviation of the mean.

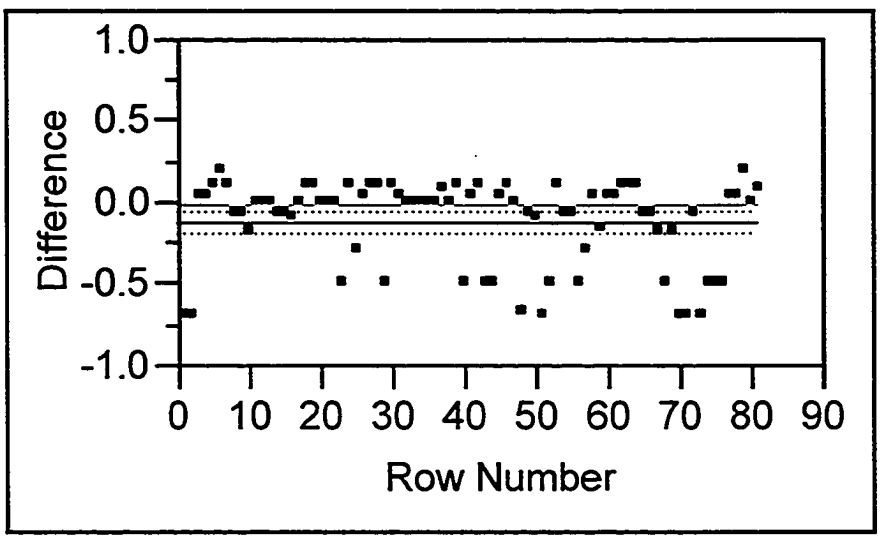


Figure 7-11: The Difference in the Paired Samples of ET_{CSPE} and ET_{AME}

Because test cases in CSPE and AME test suites tend to be of different size (Section 7.4.1), we also computed the detection ratios above but by accounting for the number of methods in test cases. Let EM_{CSPE} and EM_{AME} be the set of mean weighted ratios of effective test cases for the CSPE and AME techniques, respectively. A weighted ratio is defined as the ratio of the number of methods involved in effective test cases to the total

number of methods in the test suite. We also perform a one-tailed t-test and the results are presented in Table 7-6. We can tell that the results are very similar to the previous test though the average difference in ratios is slightly smaller: 9%. A similar diagram to Figure 7-11 graphically depicting differences is shown in Figure 7-12. We can therefore conclude, based on the Petstore component, that CSPE test cases have a higher probability to kill mutants than AME test cases, whether weighted according to test case size or not.

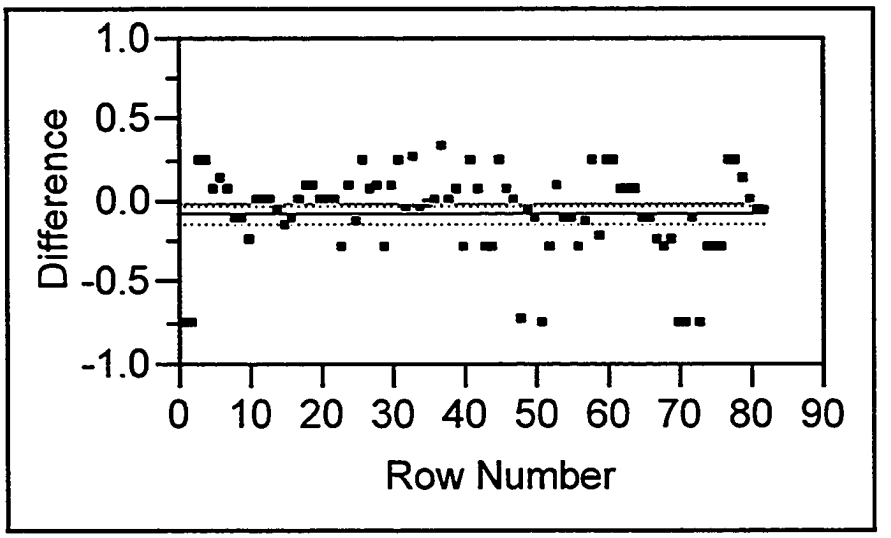


Figure 7-12: The Difference in the Paired Samples of EM_{CSPE} and EM_{AME}

Table 7-6: The Mean and Mean Difference for the Two Set of Ratios of Effective Test Cases in CSPE and AME

	ET_{CSPE}	ET_{AME}	EM_{CSPE}	EM_{AME}
Mean	0.5391	0.4173	0.5492	0.4609
Mean Difference	0.1218		0.0883	
***t* Value**	-4.2470		-2.9796	
Probability < *t*	< 0.0001		0.0019	

7.4.3 Analysis of Test Suite Subset Effectiveness

Prompted by the results in the previous section on the effectiveness of the individual CSPE-based test cases, a heuristic was devised to select a subset of the CSPE-based test cases and evaluate their effectiveness on the set of mutants. The test case subset was selected such that it was of equal or lower cost than the total cost of the AME test suite.

The effectiveness of the test suite subsets was measured with a simulated scenario based on the original set of 81 effective mutants. The scenario considered is: a user selects a subset of CSPE test cases based on a set of mutants and then uses this subset to find actual faults. In order to simulate this we split our set of 81 mutants into two subsets: one represents the user's set of mutants (training mutants), and the other represents actual faults in the system (evaluation mutants). We split the 81 mutants randomly into a *training* set and an *evaluation* set. Because they split unevenly, the larger subset (containing 41 mutants) is also chosen randomly.

The size of the selected subset was determined by calculating the average cost of a CSPE-based test case (3.311 methods per test case) and considering that the total cost of the AME-based test suite is 22 methods. From this we determine the subset should be of size 6, i.e., the sum of the cost of 6 CSPE test cases should be no more than 22 methods, which is the size of the entire AME-based test suite. The heuristic for selecting an effective subset of test cases is as follows:

1. Evaluate each test case according to the training mutants it kills (*score*).
2. Order the test cases in descending score order.
3. Select the first 6 test cases.

The test case subsets selected according to the heuristic and the training mutant sets were then tested on the corresponding (the compliment set of mutants) evaluation mutant sets. Over 10 training/evaluation sets the average number of evaluation mutants killed by AME and CSPE were 37.9 and 40.5, respectively. Given that 40.5 is the maximum possible average (recall that the 81 mutants split unevenly into training and mutant sets), 100% of the mutants were systematically killed by our subsets of 7 CSPE test cases whereas AME killed an average of 94% of the mutants. A statistical test of significance, whether we use a paired t-test or a non-parametric Wilcoxon signed rank test, indicates that the difference is significant with a p-value < 0.0001. Furthermore, in terms of cost, the average number of methods in CSPE subsets was smaller than in the AME test suite: 19.2 and 22, respectively. We can therefore conclude that the heuristic presented above can be effective at selecting a subset of CSPE test cases so as to significantly reduce its cost while retaining its fault detection capability.

7.4.4 Discussion

From Section 7.4.1, we have seen that five more mutants (out of 81) were killed by the CSPE test suite than by the AME test suite. Now the question is whether such faults would be of importance in practice. We have determined that these faults were related to the state behavior of parts of the component. They were not detected as this state behavior was not fully exercised by AME. From what we know on testing OO systems [6], detecting state behavior-related faults is crucial as many critical functionalities in such systems exhibit such state-driven behavior. Many test techniques relying on detailed design information or code are state-based [6] and are defined in terms of state model coverage. This clearly reflects the fact that detecting such faults is of practical importance. We can therefore conclude that CSPE is likely to be useful in detecting a type of faults, namely state-based faults, that is going to be of practical importance.

The issue of repeatability is addressed, i.e., whether the CSPE-based test suite mutation score and cost results are repeatable when suites are generated again based on the same test specification (targeted functionality and criterion). When building an adequate test suite, random variations can come from the construction of the initial Eulerian graph and from the computation of the Euler tour (e.g., there are different ways to traverse the Eulerian graph to produce the Euler tour). As a result, since there exist many ways to build an adequate test suite for any of our criteria, is the mutation score and suite cost we observe always comparable? A first answer is that, using various randomly generated problems of varying sizes, the heuristic solution to the DRPP problem we use was shown to be very often close to the optimal in [10]: more precisely, the heuristic

based solution is on average within 1.4% of the optimal solution. This implies that the cost of adequate test suites will not vary substantially across adequate test sets. As for the effectiveness at detecting faults, we computed nine other Euler tours, thus building a total of 10 adequate test suites for ANP-PC. We observed that the fault-detection results we report above (Section 7.4.1) are consistent over those 10 sets and can therefore be trusted as representative.

If we now turn our attention to the effectiveness ratios of individual test cases, we have also seen (Section 7.4.2) that there is roughly a difference in detection probability of 10% between CSPE and AME test cases (from roughly 45% to 55%). Now the question is whether this would help, in practice, the detection of failures earlier in the test process and with smaller test suites, and whether this would make a practically significant difference. This question is addressed in Section 7.4.3, where we show that using a simple heuristic it is possible to select subsets that are more effective and less costly than the entire AME-based test suite. In that same section these result were also shown to be statistically significant. As for practical significance: given that component user takes the time to evaluate the entire CSPE-base test suite (all test cases) on a training set of mutants, it would in fact be possible to detect failures earlier in the test process and with smaller test suites. Specifically, as the first option, the component user would be able to sort the entire test suite according to the heuristic described in Section 7.4.3 and then receive reports of detected faults earlier than if they were to run the entire test suite in arbitrary order. As the second option, the component user would be able to use only a subset of the test suite (a relatively small number of test cases) selected according to the

same heuristic and subsequently only use that small number of test cases to benefit from the CSPE technique with none of the associated high cost.

Chapter 8
Conclusion

The problem addressed by this thesis (Chapter 2) is that of testing of Commercial-off-the-Shelf (COTS) component by the component user in a typical situation where the component vendor does not provide the user with the component's source code and design documentation. The lack of source code and design documentation limits the component users with respect to the information they can use in adequately testing the component in their specific environment.

The problem was approached in this thesis by first proposing a framework describing component vendor and user responsibilities and tasks which would serve as groundwork for a COTS component testing strategy (Section 4.1). Then, an existing black box testing technique (Constraint on Succeeding and Preceding Events (CSPE)) and predicate testing techniques were adapted and combined in our definition of testing criteria for COTS components (Section 4.3). In another contribution, we described how CSPE-based test suites can be modeled with a directed graph, and how a graph-based optimization technique can be applied to automatically generate efficient CSPE-based test sequences. A test sequence generation prototype tool `PrestoSequence` was implemented (Chapter 6). The `PrestoSequence` tool was implemented in Java and takes as input XML files describing the component metadata and test specification, and generates adequate test sequences.

The case studies conducted were based on two example components. The two case studies were designed to evaluate: a) the cost of adequate test sets for criteria based on CSPE by comparing our graph-based approach with a simpler approach available in the literature; and b) the effectiveness of the generated CSPE-based test suites at detecting faults. The results showed that our graph-based test sequence generation approach consistently generated test sequences of lower cost (point a)): up to 54% improvement in some cases. The effectiveness of the CSPE-based test suites suggested in the second case study (point b)) that there are some mutants which are only likely to be killed by our approach and require using specific method and states sequences during test execution: 5 of the 81 mutants we seeded in the program were only killed by the CSPE suite and not the AME one. Based on a careful analysis of these mutants, it was determined that in practice such types of faults are likely to be of critical importance. Additionally, it was shown with statistical significance that the individual CSPE test cases were more likely to kill mutants than the test cases of the AME technique, and that it is possible to select subsets of CSPE test suites that are more effective and less costly than the entire AME-based test suite.

Some of the areas for future work based on this thesis are: a closer examination of effectiveness of the technique on a larger COTS component, investigation into the application of the technique in integration testing of component-based systems, and further expansion of the testing framework including support of full automation. Although the Petstore case study was sufficient to highlight the effectiveness of the

technique, a large COTS component (ideally an actual commercial product) might give more insight into how this technique should be further developed.

The CSPE-based technique should also be evaluated for use in integration testing of component-based systems. For instance, it could be used to test a set of COTS components, where the set can be seen as one large COTS component itself. Such an application of the technique would require a method by which the interaction of the composed components is tested. Similarly to the work presented in Section 3.6, the technique could also be used to test the integration of COTS components with other non-COTS components and the component container.

The parts of the COTS testing framework that were not addressed in this thesis and should be addressed in future work include: automation of CSPE constraint generation and predicate generation, automation of test scaffolding generation. Additionally, the exact version of the graph-based algorithm should be considered as an alternative algorithm that may provide more improvement on the cost of generated test sequences.

References

[1] R. K. Ahuja, T. L. Magnanti and J. B. Orlin, *Network flows : theory, algorithms, and applications*, Prentice Hall, 1993.

[2] P. Ammann, J. Offutt and H. Huang, "Coverage criteria for logical expressions," *Proc. Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on*, pp. 99-107, 2003.

[3] AT&T Reasearch, Graphviz: Graph Visualization Software, http://graphviz.org/, (Last accessed August 27 2005)

[4] M. O. Ball and M. J. Magazine, "Sequencing of Insertions in Printed Circuit Board Assembly," *Operations Research*, vol. 36 (2), pp. 192-201, 1987.

[5] S. Beydeda and V. Gruhn, "An integrated testing technique for component-based software," *Proc. Computer Systems and Applications, ACS/IEEE International Conference on. 2001*, pp. 328-334, 2001.

[6] R. V. Binder, *Testing Object-Oriented Systems*, Addison-Wesley, 1999.

[7] R. G. Busacker and P. J. Gowen, "A Procedure for Determining a Family of Minimal-Cost Network Flow Patterns," John Hopkins University, O.R.O. Tech. Paper 15, 1961.

[8] P. M. Camerini, L. Fratta and F. Maffioli, "Ranking arborescences in O(Km log n) time," *European Journal of Operational Research*, vol. 4 (4), pp. 235-242, 1980.

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

[10] N. Christofides, V. Campos, A. Corberan and E. Mota, "An algorithm for the rural postman problem on a directed graph," *Mathematical Programming Studies* (26), pp. 155-66, 1986.

[11] A. Corbéran, G. Mejía and J. M. Sanchis, "New Results on the Mixed General Routing Problem," *Operations Research*, vol. 53 (2), pp. 363-376, 2005.

[12] F. J. Daniels and K. C. Tai, "Measuring the effectiveness of method test sequences derived from sequencing constraints," *Proc. Technology of Object-Oriented Languages and Systems, 1999. TOOLS 30. Proceedings*, pp. 74-83, 1999.

[13] R. A. DeMillo, R. J. Lipton and F. G. Sayward, "Hints on test data selection: help for the practicing programmer," *Computer*, vol. 11 (4), pp. 34-41, 1978.

[14] A. Denise, M.-C. Gaudel and S.-D. Gouraud, "A Generic Method for Statistical Testing," *Proc. Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, pp. 25-34, 2004.

[15] M. Dror, *Arc Routing: Theory, Solutions and Applications*, Kluwer Academic Publishers, 2000.

[16] J. Edmonds, "Optimum branchings," *J. Res. Nat. Bur. Standards Sect. B*, vol. 71B, pp. 233--240, 1967.

[17] Fleury, "Deux problemes de geometrie de situation," *Journal de mathematiques elementaires*, pp. 257-261, 1883.

[18] E. Foundation, Eclipse Universal Tool Platform, http://www.eclipse.org, (Last accessed 4 Feb. 2005)

[19] U. FTF, UML 1.5 Specification, http://www.omg.org/cgi-bin/doc?formal/03-03-01, (Last accessed 4 Feb. 2005)

[20] U. O. FTF, UML 2.0 OCL Final Adopted Specification, http://www.omg.org/cgi-bin/doc?ptc/2003-10-14, (Last accessed 26 Jan. 2005)

[21] U. S. FTF, UML 2.0 Superstructure Final Adopted Specification, http://www.omg.org/cgi-bin/doc?ptc/2003-08-02, (Last accessed 26 Jan. 2005)

[22] J. Z. Gao, H.-S. Jacob Tsao, Ye Wu, *Testing and Quality Assurance for Component-based Software*, Artech House, 2003.

[23] S. Ghosh and A. Mathur, "Interface Mutation to assess the adequacy of tests for components and systems," *Proc. Technology of Object-Oriented Languages and Systems, 2000. TOOLS 34. Proceedings. 34th International Conference on*, pp. 37-46, 2000.

[24] JBoss, JBoss Application Server, http://www.jboss.com, (Last accessed 26 Jan. 2005)

[25] B. Karçali and K.-C. Tai, "Automated test sequence generation using sequencing constraints for concurrent programs," *Proc. Software Engineering for Parallel and Distributed Systems, 1999. Proceedings. International Symposium on*, pp. 97-108, 1999.

[26] S. Kim, J. A. Clark and J. A. McDermid, "Class Mutation: Mutation Testing for Object-Oriented Programs," *Proc. Net.ObjectDays*, Erfurt, Germany, October 9-12, 2000, 2000.

[27] K. N. King and A. J. Offutt, "A Fortran language system for mutation-based software testing " *Softw. Pract. Exper.* , vol. 21 (7), pp. 685-718 1991

[28] J. K. Lenstra and A. H. G. Rinnooy-Kan, "Complexity of vehicle routing and scheduling problems," *Networks*, vol. 11, pp. 221-227, 1981.

[29] V. Massol and T. Husted, *JUnit in Action*, Manning Publications Co., 2004.

[30] V. Massol and T. Husted, JUnit in Action Source Code, http://www.manning.com/books/massol/source, (Last accessed 13 Feb. 2005)

[31] I. Object Mentor, JUnit Java Testing Framework, http://www.junit.org/index.html, (Last accessed 26 Jan. 2005)

[32] A. J. Offutt, "A practical system for mutation testing: help for the common programmer," *Proc. Proceedings of International Test Conference, 2-6 Oct. 1994*, Washington, DC, USA, Int. Test Conference, pp. 824-30, 1994//, 1994.

[33] J. B. Orlin, "A polynomial time primal network simplex algorithm for minimum cost flows,", *Proceedings of the seventh annual ACM-SIAM symposium on Discrete algorithms*. Atlanta, Georgia, United States, Society for Industrial and Applied Mathematics, pp. 474-481, 1996.

[34] A. Orso, M. J. Harrold, D. Rosenblum, G. Rothermel, M. L. Soffa and H. Do, "Using component metacontent to support the regression testing of component-based software," *Proc. Software Maintenance, 2001. Proceedings. IEEE International Conference on*, pp. 716-725, 2001.

[35] E. Roman, R. P. Sriganesh and G. Brose, *Mastering Enterprise Java Beans Third Edition*, Wiley, 3rd Edition, http://www.theserverside.com/books/wiley/masteringEJB/index.tss, 2005.

[36] D. S. Rosenblum, "Adequate testing of component-based software," Department of Inormation and Computer Science, University of California Technical Report 97-34, 1997.

[37] W. G. Schneeweiss, "Finding all Prime Implicants of a Boolean Function (Section 5.2), and Minimization (Section 5.3),", *Boolean Functions with Engineering Applications and Computer Programs*. Berlin, Springer-Verlag, pp. 98-117, 1989.

[38] Sun Microsystems, Enterprise JavaBeans Technology, http://java.sun.com/products/ejb/, (Last accessed 26 Jan. 2005)

[39] Sun Microsystems, Java Blueprints, http://java.sun.com/reference/blueprints/, (Last accessed 26 Jan. 2005)

[40] Sun Microsystems, Java Message Service, http://java.sun.com/products/jms/, (Last accessed 26 Jan. 2005)

[41] Sun Microsystems, Java Naming and Directory Interface, http://java.sun.com/products/jndi/, (Last accessed 26 Jan. 2005)

[42] Sun Microsystems, Java API for XML Processing, http://java.sun.com/xml/jaxp/index.jsp, (Last accessed 26 Jan. 2005)

[43] C. Szyperski, with Dominik Gruntz and Stephen Murer, *Compnent Software – Beyond Object-Oriented Programming*, Component Software, Addison-Wesley, 2nd ed. Edition, 1999.

[44] R. E. Tarjan, "Finding optimum branchings," *Networks*, vol. 7 (1), pp. 25-35, 1977.

[45] The Apache Software Foundation, Cactus Testing Framework, http://jakarta.apache.org/cactus/, (Last accessed 26 Jan. 2005)

[46] University of Passau and Institute of Plant Genetics and Crop Plant Research Gaterleben, Gravisto: Graph Visualization Toolkit, http://www.gravisto.org/, (Last accessed August 27 2005)

[47] E. Weyuker, "Automatically Generating Test Data from a Boolean Specification," *IEEE Trans. on Software Engineering*, vol. 20 (5), pp. 353-363, 1994.

Appendix A

Petstore Mutants

The table below contains full listing of all of the mutants seeded in the Petstore case study. The table contains the mutant id (which also identifies the type of mutant), the source file, the correct source code, and the mutated source code. In addition to the source file, the second column also specifies the line range where the mutant was seeded. Empty cells in the correct source column denote that the mutant is a newly added line, and empty cells in the mutated source denote that the mutant was a deleted line.

Mutant	Source File and Location	Correct Source	Mutated Source
AOR-1	OrderEJB.java 46,52	`uid = orderDate.hashCode() + orderItem.hashCode();`	`uid = orderDate.hashCode() - orderItem.hashCode();`
AOR-2	OrderEJB.java 46,52	`uid = orderDate.hashCode() + orderItem.hashCode();`	`uid = orderDate.hashCode() * orderItem.hashCode();`
CRP-10	PetstoreEJB.java 51,57	`if (orderState == OrderState.UNVERIFIED) {`	`if (orderState == OrderState.VERIFIED) {`
CRP-11	PetstoreEJB.java 52,58	`JMSUtil.sendToJMSQueue(JNDINames.QUEUE_ORDER, order`	`JMSUtil.sendToJMSQueue(JNDINames.ORDER_LOCALHOME, order`
CRP-12	PetstoreEJB.java 68,74	`.lookup(JNDINames.ORDER_LOCALHOME);`	`.lookup(JNDINames.QUEUE_ORDER);`
CRP-13	PetstoreEJB.java 73,79	`if (orderState == OrderState.VERIFIED) {`	`if (orderState == OrderState.UNVERIFIED) {`
CRP-14	PetstoreEJB.java 74,80	`JMSUtil.sendToJMSQueue(JNDINames.QUEUE_ORDER, order`	`JMSUtil.sendToJMSQueue(JNDINames.ORDER_LOCALHOME, order`
CRP-15	PetstoreEJB.java 93,99	`.lookup(JNDINames.ORDER_LOCALHOME);`	`.lookup(JNDINames.QUEUE_ORDER);`
CRP-16	PetstoreEJB.java 102,108	`return (orderId.toString() + ": " + date.toString() + ", " + item`	`return (orderId.toString() + ": " + date.toString() + ", " + item.substring(3)`
CRP-17	PetstoreEJB.java 109,115	`.lookup(JNDINames.ORDER_LOCALHOME);`	`.lookup(JNDINames.QUEUE_ORDER);`
CRP-18	PetstoreEJB.java 127,133	`if (clearConfirmCodes.contains(confirmCode)) {`	`if (clearConfirmCodes.contains(confirmCode.substring(3))) {`
CRP-19	PetstoreEJB.java 129,135	`.lookup(JNDINames.ORDER_DATASOURCE);`	`.lookup(JNDINames.QUEUE_ORDER);`
CRP-1	OrderEJB.java 38,44	`return 0;`	`return 3;`
CRP-20	PetstoreEJB.java 132,138	`.prepareStatement("DELETE FROM ORDERS");`	`.prepareStatement("DELETE FROM ORDERSSS");`
CRP-21	PetstoreEJB.java 136,142	`clearConfirmCodes.remove(confirmCode);`	`clearConfirmCodes.remove(confirmCode.substring(3));`
CRP-22	PetstoreEJB.java 149,155	`if (clearConfirmCodes.contains(confirmCode)) {`	`if (clearConfirmCodes.contains(confirmCode.substring(3))) {`

CRP-23	PetstoreEJB.java 150,156	clearConfirmCodes.remove(confirmCode);	clearConfirmCodes.remove(confirmCode.substring(3));
CRP-24	JNDIUtil.java 21,27	object = getInitialContext().lookup(name);	object = getInitialContext().lookup(name.substring(3));
CRP-25	JNDINames.java 1,7	public static final String QUEUE_CONNECTION_FACTORY = "ConnectionFactory";	public static final String QUEUE_CONNECTION_FACTORY = "ConnectionFactorysss";
CRP-26	JNDINames.java 3,9	public static final String QUEUE_ORDER = "queue/petstore/Order";	public static final String QUEUE_ORDER = "queue/petstore/Ordersss";
CRP-27	JNDINames.java 5,11	public static final String ORDER_DATASOURCE = "java:DefaultDS";	public static final String ORDER_DATASOURCE = "java:DefaultDSsss";
CRP-28	JNDINames.java 7,13	public static final String ORDER_LOCALHOME = "ejb/petstore/Order";	public static final String ORDER_LOCALHOME = "ejb/petstore/Ordersss";
CRP-29	JNDINames.java 9,13	public static final String PETSTORE_HOME = "ejb/petstore/Petstore";	public static final String PETSTORE_HOME = "ejb/petstore/Petstoresss";
CRP-2	OrderProcessorMDB.java 21,27	.lookup(JNDINames.ORDER_LOCALHOME);	.lookup(JNDINames.QUEUE_ORDER);
CRP-30	JMSUtil.java 25,31	.lookup(JNDINames.QUEUE_CONNECTION_FACTORY);	.lookup(JNDINames.QUEUE_ORDER);
CRP-31	JMSUtil.java 28,34	QueueSession.AUTO_ACKNOWLEDGE);	QueueSession.CLIENT_ACKNOWLEDGE);
CRP-3	OrderProcessorMDB.java 46,52	order.setOrderState(OrderState.UNVERIFIED);	order.setOrderState(OrderState.VERIFIED);
CRP-4	OrderProcessorMDB.java 51,57	order.setOrderState(OrderState.VERIFIED);	order.setOrderState(OrderState.UNVERIFIED);
CRP-5	OrderProcessorMDB.java 56,62	order.setOrderState(OrderState.FULFILLED);	order.setOrderState(OrderState.VERIFIED);
CRP-6	PetstoreEJB.java 30,36	.lookup(JNDINames.ORDER_LOCALHOME);	.lookup(JNDINames.QUEUE_ORDER);
CRP-7	PetstoreEJB.java 32,38	order.setOrderState(OrderState.NEW);	order.setOrderState(OrderState.VERIFIED);
CRP-8	PetstoreEJB.java 34,40	JMSUtil.sendToJMSQueue(JNDINames.QUEUE_ORDER, order.getOrderId(),	JMSUtil.sendToJMSQueue(JNDINames.ORDER_LOCALHOME, order.getOrderId(),
CRP-9	PetstoreEJB.java 46,52	.lookup(JNDINames.ORDER_LOCALHOME);	.lookup(JNDINames.QUEUE_ORDER);
EHR-10	PetstoreEJB.java 151,157	throw new PetstoreException();	
EHR-11	PetstoreEJB.java 153,159	throw new EJBException("Error clearing the database...");	
EHR-12	JNDIUtil.java 11,18	throw new RuntimeException("Failed to create InitialContext [" + e.getMessage() + "]");	
EHR-13	JNDIUtil.java 23,30	throw new RuntimeException("JNDI lookup failure for [" + name + "]. Reason [" + e.getMessage() + "]");	
EHR-1	OrderEJB.java 34,41	e.printStackTrace();	
EHR-2	OrderProcessorMDB.java 40,46	throw new EJBException("Error processing order...");	
EHR-4	PetstoreEJB.java 58,64	throw new PetstoreException();	

EHR-5	PetstoreEJB.java 78,84	`throw new PetstoreException();`	
EHR-8	PetstoreEJB.java 137,143	`throw new PetstoreException();`	
EHR-9	PetstoreEJB.java 139,145	`throw new EJBException(e);`	
ROR-1	JNDIUtil.java 7,13	`if (jndiContext == null) {`	`if (jndiContext != null) {`
ROR-2	JMSUtil.java 35,41	`if (sender != null) {`	`if (sender == null) {`
ROR-3	JMSUtil.java 38,44	`if (session != null) {`	`if (session == null) {`
ROR-4	JMSUtil.java 41,47	`if (cnn != null) {`	`if (cnn == null) {`
ROR-5	JMSUtil.java 44,50	`if (ic != null) {`	`if (ic == null) {`
ROR-6	PetstoreEJB.java 51,57	`if (orderState == OrderState.UNVERIFIED) {`	`if (orderState >= OrderState.UNVERIFIED) {`
ROR-7	PetstoreEJB.java 71,77	`if (orderState == OrderState.VERIFIED) {`	`if (orderState >= OrderState.VERIFIED) {`
SDL-10	OrderProcessorMDB.java 33,39	`fulfillOrder(order);`	
SDL-11	OrderProcessorMDB.java 34,40	`break;`	
SDL-12	OrderProcessorMDB.java 36,42	`logOrder(order);`	
SDL-13	OrderProcessorMDB.java 37,43	`break;`	
SDL-14	OrderProcessorMDB.java 46,52	`order.setOrderState(OrderState.UNVERIFIED);`	
SDL-15	OrderProcessorMDB.java 51,57	`order.setOrderState(OrderState.VERIFIED);`	
SDL-16	OrderProcessorMDB.java 56,62	`order.setOrderState(OrderState.FULFILLED);`	
SDL-17	PetstoreEJB.java 32,38	`order.setOrderState(OrderState.NEW);`	
SDL-18	PetstoreEJB.java 34,41	`JMSUtil.sendToJMSQueue(JNDINames.QUEUE_ORDER, order.getOrderId(), false);`	
SDL-19	PetstoreEJB.java 52,59	`JMSUtil.sendToJMSQueue(JNDINames.QUEUE_ORDER, order.getOrderId(), false);`	
SDL-20	PetstoreEJB.java 74,81	`JMSUtil.sendToJMSQueue(JNDINames.QUEUE_ORDER, order.getOrderId(), false);`	
SDL-22	PetstoreEJB.java 120,126	`clearConfirmCodes.add(confirmCode);`	
SDL-23	PetstoreEJB.java 133,139	`dropTable.execute();`	
SDL-24	PetstoreEJB.java 134,140	`dropTable.close();`	
SDL-25	PetstoreEJB.java 135,141	`con.close();`	
SDL-26	PetstoreEJB.java 136,142	`clearConfirmCodes.remove(confirmCode);`	
SDL-28	PetstoreEJB.java 150,156	`clearConfirmCodes.remove(confirmCode);`	

SDL-2	OrderEJB.java 46,52	uid = orderDate.hashCode() + orderItem.hashCode();	
SDL-32	JMSUtil.java 33,39	sender.send(msg);	
SDL-33	JMSUtil.java 36,42	sender.close();	
SDL-34	JMSUtil.java 39,45	session.close();	
SDL-35	JMSUtil.java 42,48	cnn.close();	
SDL-36	JMSUtil.java 45,51	ic.close();	
SDL-3	OrderEJB.java 48,54	setOrderId(new Integer(uid));	
SDL-4	OrderEJB.java 49,55	setOrderDate(orderDate);	
SDL-5	OrderEJB.java 50,56	setOrderItem(orderItem);	
SDL-6	OrderProcessorMDB.java 27,33	proceedOrder(order);	
SDL-7	OrderProcessorMDB.java 28,34	break;	
SDL-8	OrderProcessorMDB.java 30,36	verifyOrder(order);	
SDL-9	OrderProcessorMDB.java 31,37	break;	
SVR-10	PetstoreEJB.java 50,56	Integer orderState = order.getOrderState();	Integer orderState = order.getOrderState() + 3;
SVR-11	PetstoreEJB.java 53,59	.getOrderId(), false);	.getOrderId() + 3, false);
SVR-12	PetstoreEJB.java 69,75	OrderLocal order = (OrderLocal) orderHome.findByPrimaryKey(orderId);	OrderLocal order = (OrderLocal) orderHome.findByPrimaryKey(orderId + 3);
SVR-13	PetstoreEJB.java 72,78	Integer orderState = order.getOrderState();	Integer orderState = order.getOrderState() + 3;
SVR-14	PetstoreEJB.java 75,81	.getOrderId(), false);	.getOrderId() + 3, false);
SVR-15	PetstoreEJB.java 94,100	OrderLocal order = (OrderLocal) orderHome.findByPrimaryKey(orderId);	OrderLocal order = (OrderLocal) orderHome.findByPrimaryKey(orderId + 3);
SVR-16	PetstoreEJB.java 98,104	orderState = order.getOrderState();	orderState = order.getOrderState() + 3;
SVR-17	PetstoreEJB.java 110,116	int count = orderHome.orderCount();	int count = orderHome.orderCount() + 3;
SVR-18	JMSUtil.java 27,33	session = cnn.createQueueSession(transacted,	session = cnn.createQueueSession(!transacted,
SVR-1	OrderEJB.java 32,38	return ejbSelectOrderCount();	return ejbSelectOrderCount() + 3;
SVR-2	OrderEJB.java 46,52	uid = orderDate.hashCode() + orderItem.hashCode();	uid = orderDate.hashCode() + orderItem.hashCode() + 3;
SVR-3	OrderEJB.java 48,54	setOrderId(new Integer(uid));	setOrderId(new Integer(uid + 3));
SVR-4	OrderEJB.java 50,55		orderItem = orderItem.substring(3);
SVR-5	OrderEJB.java 48,53		uid = uid + 3;
SVR-6	OrderProcessorMDB.java 23,29	OrderLocal order = orderHome.findByPrimaryKey(orderId);	OrderLocal order = orderHome.findByPrimaryKey(orderId + 3);

SVR-7	OrderProcessorMDB.java 25,31	`switch (orderState) {`	`switch (orderState + 3) {`
SVR-8	PetstoreEJB.java 36,42	`return order.getOrderId().intValue();`	`return order.getOrderId().intValue() + 3;`
SVR-9	PetstoreEJB.java 47,53	`OrderLocal order = (OrderLocal) orderHome.findByPrimaryKey(orderId);`	`OrderLocal order = (OrderLocal) orderHome.findByPrimaryKey(orderId + 3);`
UOI-1	PetstoreEJB.java 51,57	`if (orderState == OrderState.UNVERIFIED) {`	`if (!(orderState == OrderState.UNVERIFIED)) {`
UOI-2	PetstoreEJB.java 73,79	`if (orderState == OrderState.VERIFIED) {`	`if (!(orderState == OrderState.VERIFIED)) {`
UOI-3	PetstoreEJB.java 127,133	`if (clearConfirmCodes.contains(confirmCode)) {`	`if (!clearConfirmCodes.contains(confirmCode)) {`
UOI-4	PetstoreEJB.java 149,155	`if (clearConfirmCodes.contains(confirmCode)) {`	`if (!clearConfirmCodes.contains(confirmCode)) {`