Model Based Test Generation and Optimization

Mohamed Mussa A. Mussa

A Thesis
In the Department
Of
Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of
Doctor of Philosophy (Electrical and Computer Engineering) at
Concordia University
Montreal, Quebec, Canada

June 2015

CONCORDIA UNIVERSITY
SCHOOL OF GRADUATE STUDIES

This is to certify that the thesis prepared

By: Mohamed Mussa A. Mussa

Entitled: Model Based Test Generation and Optimization

and submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Electrical and Computer Engineering)

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

______________________ Chair
Dr. R. Dssouli

______________________ External Examiner
Dr. H. Sahraoui

______________________ External to Program
Dr. J. Rilling

______________________ Examiner
Dr. O. Ait Mohamed

______________________ Examiner
Dr. A. Hamou-Lhadj

______________________ Thesis Supervisor
Dr. F. Khendek

Approved by: ______________________
Dr. A.R. Sebak, Graduate Program Director

____
June 4, 2015 Dr. A. Asif, Dean
Faculty of Engineering & Computer Science

Abstract

Model Based Test Generation and Optimization

Mohamed Mussa A. Mussa, Ph.D.
Concordia University, 2015

Software testing is an essential activity in the software engineering process. It is used to enhance the quality of the software products throughout the software development process. It inspects different aspects of the software quality such as correctness, performance and usability. Furthermore, software testing consumes about 50% of the software development efforts. Software products go through several testing levels. The main ones are unit-level testing, component-level testing, integration-level testing, system-level testing and acceptance-level testing. Each testing level involves a sequence of tasks such as planning, modeling, execution and evaluation.

Plenty of systematic test generation approaches have been developed using different languages and notations. The majority of these approaches target a specific testing-level. However, only little effort has been directed toward systematic transition among testing-levels. Considering the incompatibility between these approaches, tailored compatibility-tools are required between the testing levels. Furthermore, several test models are usually generated to evaluate the implementation at each testing level. Unfortunately, there is redundancy among these models. Efficient reuse of these test models represents a significant challenge. On the other hand, the growing attention to the model driven methodologies bonds the development and the testing activities. However, research is still required to link the testing levels.

In this PhD thesis, we propose a model based testing framework that enables reusability and collaboration across the testing levels. In this framework, we propose test generation and test optimization approaches that at each level consider artifacts generated in preceding testing levels. More precisely, we propose an approach for the generation of integration test models starting from component test models, and another approach for the optimization of the acceptance test model using the integration test models. To conduct our research in rigorous settings, we base our framework on standard notations that are widely adopted for software development and testing, namely Unified Modeling Language (UML). In our first approach, component test cases are examined to locate and select the ones that include an interaction among the integrated components. The selected test cases are merged to generate integration test cases, which tackles the theoretical research issue of merging test cases. Furthermore, the generated test cases are mapped against each other to remove potential redundancies. For the second approach, acceptance test optimization, integration test models are compared to the acceptance test model in order to remove test cases that have already been exercised during the integration-level testing. However, not all integration test cases are suitable for the comparison. Integration test cases have to be examined to ensure that they do not include test stubs for system components.

We have developed two approaches and implemented the corresponding prototypes in order to demonstrate the effectiveness of our work. The first prototype implements the integration test generation approach. It accepts component test models and generates integration test models. The second prototype implements the acceptance test optimization approach. It accepts integration test models along with the acceptance test model and generates an optimized acceptance test model.

Acknowledgements

I would like to express my sincere gratitude to my supervisor Professor F. Khendek for his continued support, guidance, encouragement and constructive criticism. I would like to send great thanks to Professor Reinhard Gotzhein, in Germany, for his fruitful discussion and insightful advice. I would also like to thank all my friends at Concordia University and in my country who encouraged me.

During my PhD study, I passed through, and am still going through the hardest period in my life with mixed emotions of happiness and sorrow. I am glad we were able to get rid of our dictator in Libya. I am very grateful for the heroes who sacrificed their lives to bring us hope. Yet, negative impacts come along with the change. The unjustified suspension of my Libyan scholarship placed a heavy burden on my shoulders, since it made it harder for me to support my family. Losing beloved ones and worrying about the ones who are in danger has a deep impact on my life, family and my productivity. At the end, one has to face reality and keep going. Thanks to all martyrs and living heroes.

Words fall short when I attempt to express my love and gratitude to my parents, Zayinap and Mussa. This work would have never been completed without their love, encouragement and moral support. I would like to thank my wife and children for their understanding, encouragement and sacrifices to complete my study. I would like to thank my brothers and sisters for their support and prayers.

I would like to extend my thanks to the examining committee for their support during different stages of my PhD study and for their efforts in the evaluation process.

Finally, I would like to acknowledge the financial support of my country, Libyan-North America Program, the Natural Sciences and Engineering Research Council (NSERC) of Canada through Dr. F. Khendek’s Discovery Grant and Concordia University.

Table of Contents

List of Figures

List of Tables

List of Definitions

List of Algorithms

List of Abbreviations

ATC	Abstract Test Cases
BPEL	Business Process Execution Language
CBSD	Component-Based Software Development
CTM	Component Test Model
CUT	Component Under Test
DSLs	Domain Specific Languages
EATs	Executable Acceptance Test cases
FSMs	Finite State Machines
HLAT	High-Level Acceptance Test
IUT	Implementation Under Test
JUnit	Java Unit testing framework
MDA	Model-Driven Architecture
MDE	Model Driven Engineering
MSCs	Message Sequence Charts
OCL	Object Constraint Language
OMG	Object Management Group
PIM	Platform-Independent Model
PSM	Platform-Specific Model
PUT	Process Under Test
SDL	Specification and Description Language
SPL	Software Product Lines
SUT	System Under Test
SysML	Systems Modeling Language
TDD	Test Driven Development
TTCN-3	Testing and Test Control Notation version 3
U2TMP	UML 2.0 Testing and Monitoring Profile
UML	Unified Modeling Language
UTP	UML Testing Profile

UUID	Universal Unique IDentifier
Z	Z notation

Chapter 1
Introduction

1.1 Thesis Motivations

Software products are present in all aspects of our life. They control our vehicles, communications, house appliances, etc. They coexist in complex platforms, which are composed of hardware, operating systems, middleware and other software products, and collaborate together to serve our needs. However, the success of developing such software products depends on the principles of the software engineering. Software engineering is the use of systematic and disciplined processes/models for the development, the use and the maintenance of software products [1]. Known software processes include waterfall, spiral, w-model, prototyping, extreme programming and unified process. Software processes define the steps, activities and tools for the development of quality software products. In a software process, a software product progresses through several stages, from requirement, specification, design, implementation, testing, deployment to maintenance. In general, a software product is composed of several components that may be decomposed further to smaller units. Components are often handled, designed, implemented and tested independently. Components are then integrated, in iterations, to build sub-systems and ultimately build the complete software product. However, software development is an error-prone process [2]. Hence, software products have to be searched for defects that are introduced at different stages of the software process. This activity is referred to as the software testing.

Software testing consists of testing mechanisms, models and methods throughout the software process to detect software defects. Software products need to be continuously tested for their internal interoperability [1]. Accordingly, software products are tested during each stage of the software development process. The effort used for software testing is significant in terms of time and cost [3-5]. Software testing is composed of several levels that run in parallel with the software development process. The main levels are:

1. The unit-level testing
2. The component-level testing

3. The integration-level testing
4. The system-level testing
5. The acceptance- level testing

During the software development process, there is almost a complete separation between the development activity and the testing activity. Different tools and languages are used in each activity. Even within the software testing, different expertise is required for every testing level [4, 6, 7]. All these diversities make collaboration among stakeholders a challenging task.

Many software testing approaches have been proposed; they are developed to target different software domains such as information systems, real-time systems, embedded systems, and telecommunication. In practice, the majority of software testing approaches target a specific software testing level in a specific software domain. The lack of clear and systematic interactions among the software testing levels is a noteworthy problem in the software testing [4].

Software products should be exhaustively tested to improve their quality. However, exhaustive testing is an impractical task. The number of the tests increases proportionally with the size and complexity of the software products. Different techniques, such as test coverage [4, 8, 9], have been proposed to minimize the number of tests. However, the scope of such techniques is the reduction of the number of tests within the same testing level. The reduction of tests across the software testing levels has not been considered.

For decades, graphical models were used as passive assets, for documentation and communication purposes, in software engineering. Nowadays, graphical models are an essential part of the software development process, thanks to the model driven engineering (MDE) [10, 11]. MDE was introduced to handle the complexity of software products by increasing the level of abstraction. It enhances the software productivity by enabling the use of models described at a high-level of abstraction and enabling automatic transformations of such models to produce an executable code or model [12, 13]. The introduction of the model based testing (MBT) [14, 15] is an important progress in the software testing. Different modeling languages and notations have been proposed and used. While the existence of such diversity by itself is a healthy attribute, it weakens the collaboration within the software testing. Furthermore, the development of such MBT approaches is still targeting specific software testing levels, which keeps the collaboration problem across different software testing levels open.

Unified Modelling Language (UML) [16] is a widely accepted modelling notation in the software domain. However, it has no support for testing concepts. Recently, the Object Management Group (OMG) [17] standardized a UML Testing Profile (UTP) [18]. The profile was developed by a consortium of different institutes: academia, tool vendors and clients. It enables test concepts in UML models in order to create precise UML test models. The profile is a promising step toward using the same language among the software testing community. The literature shows an increase focus on UTP based approaches. Many approaches have been proposed based on UTP [19, 20, 23, 25, 58, 70, 93]. However, researchers are still focusing on the development of software testing approaches for specific software testing levels [19-25].

Software reusability improves the software development process by reducing the development time and lowering the cost. For decades, software reusability has been applied in several forms such as Component-Based Software Development (CBSD) [26], libraries and design patterns [27-29]. On the other hand, test models have been reused across test projects. However, systematic reuse of test models across different software testing levels is a challenging task [4].

1.2 Contributions

In this thesis, we propose a model based testing framework to enable collaboration, reusability and optimization across different software testing levels. The test models in the framework are based on a widely recognized modeling language, namely UML and its profile UTP. While our methodology is applicable for all well-formed test models, we express it using UTP test models. Using UTP test models has the following advantages:

1. UML is a widely recognized standard language in the software development domain. Using the same development language for testing enhances the collaboration and communication among the stakeholders.
2. UTP test models can be systematically transformed to a test execution code for a well-known test execution environment such as JUnit [30] and ITCN-3 [21, 31, 32]. It simplifies the transition among the testing tasks: design, implementation and execution.
3. There has been a lot of work for formalizing UML sequence diagrams and its ancestor Message Sequence Charts (MSCs) [33-35]. Deriving processes based on formal notations strengthen the approach.

The framework is composed of two approaches: the test generation approach and the test optimization approach. The test generation approach generates test models for the target software testing level by reusing component test models. It links software testing levels through the reuse of test models to generate subsequent test models. In this dissertation, we discuss the generation of integration test models from component test models. The approach enables test model reusability across different software testing levels. Furthermore, we have investigated the merging of test cases. While there is adequate research activities toward merging architectural models, rare research activities are devoted toward merging behavioral models. Test models are finite models. Hence, we developed a merging process that is specific to the software testing.

The test optimization approach optimizes test models by relating them to test models that have been already executed in the preceding software testing levels. It enhances test execution and improves the software testing. The approach links software testing levels by relating test models of different levels in order to eliminate redundancy of test executions across different software testing levels. In this dissertation, we discuss the optimization of acceptance test models by relating them to integration test models. Furthermore, we developed a model comparison process that is specific to the software testing. Finally, we have implemented prototypes of the two approaches to demonstrate the effectiveness of our framework.

1.3 Thesis Organization

The rest of this PhD thesis is structured as follows: In Chapter 2, we introduce the software testing and give a brief description of the modelling methodology. We conclude the chapter by surveying the literature and discuss the related work. In Chapter 3, we introduce our model based testing framework, and present a formal definition for the test model. In Chapter 4, we present the test generation approach, and discuss the generation of integration test models from component test models. In Chapter 5, we present the test optimization approach, and discuss the optimization of the acceptance test model using integration test models. In Chapter 6, we discuss the implementation of the two approaches, test generation and test optimization, followed by a case study to demonstrate the effectiveness of our framework. In Chapter 7, we summarize our contributions and discuss potential future work. Three appendices are attached at the end of the dissertation. In the first appendix, we investigate the commutative and associative properties of our integration test generation approach. The second appendix presents the system specification of our case study. We discuss the generation of the integration test models for the case study in more details in the last appendix.

Chapter 2
Background and Literature Review

In this chapter, we introduce briefly the essential knowledge that we use in this thesis. We discuss the main concepts of software testing in the first section. Next, we briefly introduce the model-based development methodology. Following that, the unified modeling language and its testing profile are discussed. The rest of the chapter is devoted to the literature review, which is spread across three subsections where we discuss model-based testing, model comparison and test-suite reduction, respectively.

2.1 Software Testing

Software testing is an integral part of the software development process. Development processes, such as waterfall, spiral and v-model, describe the software testing as an individual stage in the development process. In practice, software testing goes in parallel with the development activity. The w-model addresses explicitly the relation between the development and testing activities as shown in Figure 1. The software testing starts during the early stages of the software development process. The software testing consists of several tasks: planning, design, execution and evaluation. The first two tasks, planning and design, are performed in parallel to the software specification and design; while the other two tasks, execution and evaluation, are performed in parallel to the software implementation and maintenance. Real disasters, such as the European space shuttle Ariane 5, could be avoided if adequate robust testing was adopted [36].

There are two types of tests: black-box and white-box testing. Black-box testing, or functional testing, considers the implementation under test (IUT) as a black box and exercises tests through the external interfaces of the IUT. Test designers do not require a detailed knowledge of the IUT's internal design. On the other hand, white-box testing, structural testing requires a good knowledge of the IUT's integral design. It inspects the internal functionality and variables of the IUT. Access to internal variables and blocks is required. This kind of testing is usually performed by the software developers.

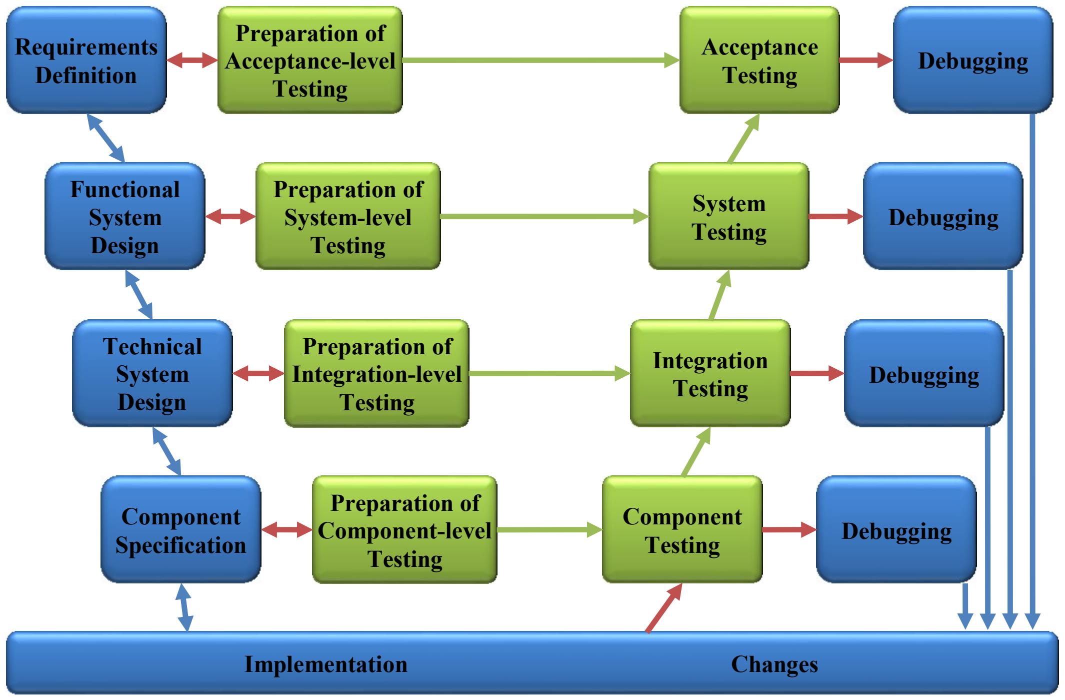


Figure 1. W-model

In order to test the implementation, one or more test models should be built. Test models describe the expected behavior of the implementation under test. Typically, test models are composed of two parts: structural and behavioral. The structural part defines the required test objects to execute the test and defines their relationships. The most important test objects are the IUT and the test control. The IUT is the software product under investigation; it can be a small piece of code, such as a method or a class, an individual system component or a complete system. The test control runs test scenarios and provides verdicts. The behavioral part defines the test cases, which will be exercised on the implementation. Each test case specifies a test scenario. A test scenario represents a set of steps with higher probability of finding defects that are not already detected. These steps can be a normal execution trace of the IUT or an invalid execution trace, called fuzz testing [37]. Test cases are accompanied by test stimuli. A test stimulus is composed of a set of inputs and expected results. The specification of test cases is usually extracted from the user requirements or the system design. There are several languages available to build test models, such as SDL [38], UML, and Z [39].

Software products are composed of several components. Components are developed first, and then integrated to build the targeted product. Different integration strategies can be adopted: bottom-up, top-down, bing-bang and ad-hoc. Components are integrated incrementally, in a certain order, to build intermediate sub-systems and eventually build the target product. There are many research activities based on choosing the optimum integration order [40-42]. To detect defects on early software stages, software testing consists of several stages/levels: unit-level, component-level, integration-level, system-level and acceptance-level testing. Unit-level testing is applied to small pieces of software such as classes and methods. It is performed frequently by the developers using white-box testing. Component-level testing is the first testing stage to be applied individually on the software's components. Component-level testing is performed when a component is fully implemented. This test examines the intended component's functionality. A test model is developed for each component. An intensive research has been done on this stage, and many testing approaches and tools were developed. A test environment is built for every test model. Test stubs and/or drivers are built to emulate the behavior of missing services and/or components during the test execution. Components, which have passed the Component-level testing, are forwarded to the integration-level testing. Integration-level testing investigates the compatibility, interoperability and consistency among the integrated components. It is conducted during the assembly of the software architecture to uncover defects associated with interfacing [43]. Components are combined to build sub-systems and then tested to see if they integrated properly. Components are added incrementally to the sub-systems, and then additional tests are applied on the lately built sub-systems. In case of test failure, the interfaces between the added component and the sub-system are debugged to detect errors and repair them. Tests are carried-out by testers using black-box testing. A test model is developed for each integration increment. A test environment is built for every test model; stubs are built to emulate the behavior of unavailable components during the test execution. System-level testing is performed to evaluate the system's conformance to the design. It is performed on complete systems before the deployment stage. Testers perform such test using black-box testing. A test model is usually generated systematically from the design model. A test environment is built for the test model to emulate required services that are not available during test execution. In most cases, system-level testing requires the construction of the system state-space. This construction is unpractical for testing complex systems. It leads to a well-known problem: state-space explosion [44]. Acceptance-level testing examines the final product against the requirements specification. It is the final stage in the software testing. The main source for building the acceptance test model is the user requirements.

A test model is developed by testers using black-box testing. The software product is often ready for deployment after successfully passing these testing levels.

2.2 Model-Driven Engineering - MDE

The cost of software testing rises up with the increase of system complexity. Fortunately, enhanced techniques of software development and testing have been introduced to meet today's requirements: system complexity, high quality and demand of change. Modeling languages have been introduced to build software artifacts. Graphical models are easier to understand and communicate than lines of code. Models have been used in the software testing for a long time, even before the introduction of the term Model-Driven Engineering; model notation like Finite State Machines (FSMs) [45] was proposed in 1956 to generate test artifacts [4]. Nowadays, model-driven engineering methodology has been widely adopted to develop software artifacts.

MDE paradigm aims at increasing the level of abstraction in the early stages of the software development process and eliminates barriers between modeling (documentation) and implementation (code). MDE separates the application logic from its specific-domain's details. Models are built at high-level of abstraction to provide a clear view for stakeholders and overcome system complexity. These models focus on the system functionality and are free from implementation details. The software development process starts with creating high-level models to simplify the system's complexity. Then, models are incrementally enriched with more details throughout the software development process to reach the implementation. Models become an implementation asset in addition to the documentation role. The well-known initiative is the Model-Driven Architecture (MDA) [46] adopted by Object Management Group. Figure 2 depicts a simplified MDA process. The process starts by developing an abstract design model, named Platform-Independent Model (PIM). The system functionality is specified in a model without any implementation related information. The PIM along with transformation rules are submitted to a transformation engine to generate a more detailed design model, named Platform-Specific Model (PSM). The transformation engine plays a role similar to traditional compilers. The transformation rules guide the engine during the mapping of PIM elements to PSM elements. This transformation can be more complicated and requires the generation of intermediate models before generating the PSM. Finally, The PSM is transformed to code. The figure shows a simple transformation relationship, one-to-one. In practice, this relationship can be many-to-one or one-to-many [12, 47, 48]. A many-to-one relationship can be presented by aspect-oriented programming where different aspects are modeled separately then merged at the PSM stage or the code stage. A one-to-many relationship can be presented by software systems composed of several parts that need to be distributed on different platforms.

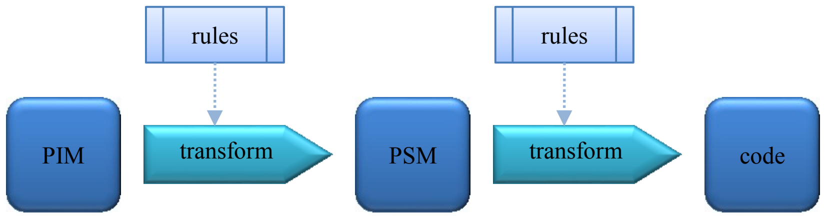


Figure 2. Basic MDA process

MDA promises full automation of the development process from specification to code. It appears that MDA has optioned-out software testing since early high-level models became essential assets in the development process and automation controls the transformation process. This assumption may be partially true for the long run. However, practice contradicts this assumption. Manual intervention is still required for the time being [49]. PIMs are developed by humans who can misinterpret the user specifications. Generated models are sometimes manually tailored for performance purpose or insufficiency of the transformation engine/rules. Furthermore, the MDA specification permits plugging-in code, built by traditional languages, into the transformation engine to facilitate the transformation process [12, 50]. Transformation rules are developed manually, which is an error-prone task. In addition to traditional testing, a new testing field was introduced to test the mapping process, called Model Transformation Testing [51, 52]. Hence, software testing is still required in the MDE paradigm and the consideration of model-based testing techniques grows up in the software testing. On the other hand, model-based testing has been used for a long time in the software testing, especially in the telecommunication domain [4]. Test models are built manually or generated systematically from the development models. They are transformed to concrete test models and enriched with test oracles in order to be transformed to executable test code. The test code is eventually exercised on the implementation and provides the verdicts. We devote a complete section for MBT.

2.3 The Unified Modeling Language - UML

UML is a widely accepted modeling language. It was standardized by the Object Management Group. It provides the stakeholders with visual representation of the system's aspects with different views. UML is used to specify, construct, document, and visualize the system's models during the software engineering process [53]. UML provides a high-level of abstraction by omitting implementation details that are not necessary for a certain design stage. UML consists of a wide range of diagrams that reflect different views of the system. These diagrams are categorized into two groups: structural and behavioral, as shown in Figure 3. The structural diagrams describe the static architecture of entities in a system, while the behavioral diagrams illustrate the dynamic activities of objects in a system. Four types of UML diagrams are used in this work: Package Diagram, Class Diagram, Composite Structure Diagram, and Sequence Diagram. UML package diagram describes a system with a high-level of abstraction. It describes the system in terms of its composed components and shows their relationships. UML class diagram defines the internal structure of the components. It clarifies the services provided by these components in terms of methods and regulates the relationships among them. UML composite structure diagram provides a snapshot of the communication among components during a run-time. UML sequence diagram describes the behavior of the objects during the partial/full lifetime of the system. Furthermore, UML offers extension mechanisms, such as tagged values, stereotypes, and constraints. These extensions can be grouped in packages to create UML profiles, which provide flexibility of applying these extensions to the UML models. A profile represents a certain aspect of the system, such as security, or extends UML to define domain specific languages (DSLs), such as SysML [54]. In 2007, OMG introduced a UML profile for facilitating testing aspects in UML, called UTP. UTP extends UML to support testing activities and artifacts by introducing test concepts, such as data representation, time concepts and evaluation mechanisms.

2.3.1 UML Testing Profile - UTP

UML testing profile extends UML to support testing activities and artifacts by introducing concepts, such as data representation, time concepts and evaluation mechanisms. UTP defines several test concepts to enable the building of precise test models in a systematic manner [5]. A UTP test model may consist of several diagrams as shown in Figure 4. The most significant ones are the test architecture diagram, the test package diagram, the test configuration diagram and the test case diagram. In this section, we briefly introduce these diagrams; for more information with walk-through scenarios, one can read Baker's book [5].

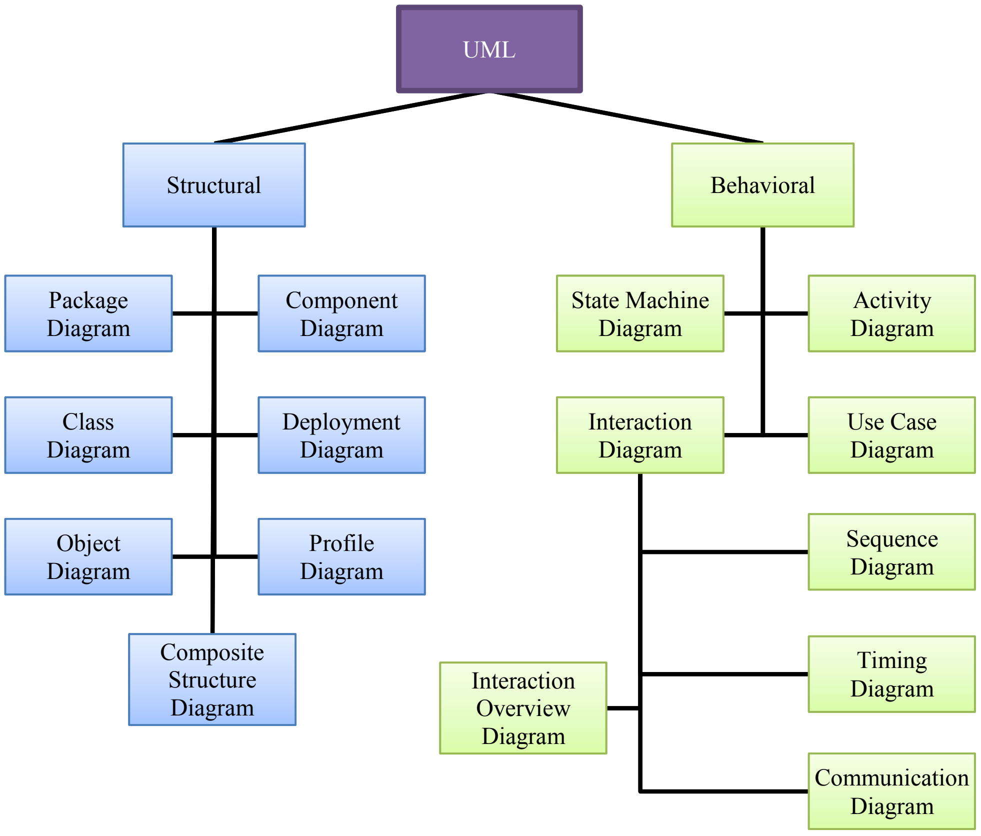


Figure 3. UML diagrams

The test architecture provides a high-level specification of the test model, as illustrated in Figure 4.a. The UML package diagram is used to describe the test architecture. The test architecture describes the relation between the test package, discussed in the next paragraph, and other packages that are required to realize the test. A mandatory package is the System Under Test (SUT). Optionally, test stubs and/or real packages may be imported to specify some environment functionalities, such as operating system APIs, which are required to execute the test.

A test package, as shown in Figure 4.b, defines the specification of the test objects and their relationships. UML class diagrams are used to describe the test package. Two test objects are mandatory: the test control and SUT. Optional test objects are test stubs and system environment. Test objects are represented by UML classes. These classes are identified by special stereotypes defined in UTP. Test controls are associated by UTP stereotype *TestContext*; test cases are defined as operations in the test control class. Test controls are responsible for executing test cases and provide verdicts. The system under test is associated by UTP stereotype *SUT*. Test stubs and system environment are associated by UTP stereotype *TestComponent*.

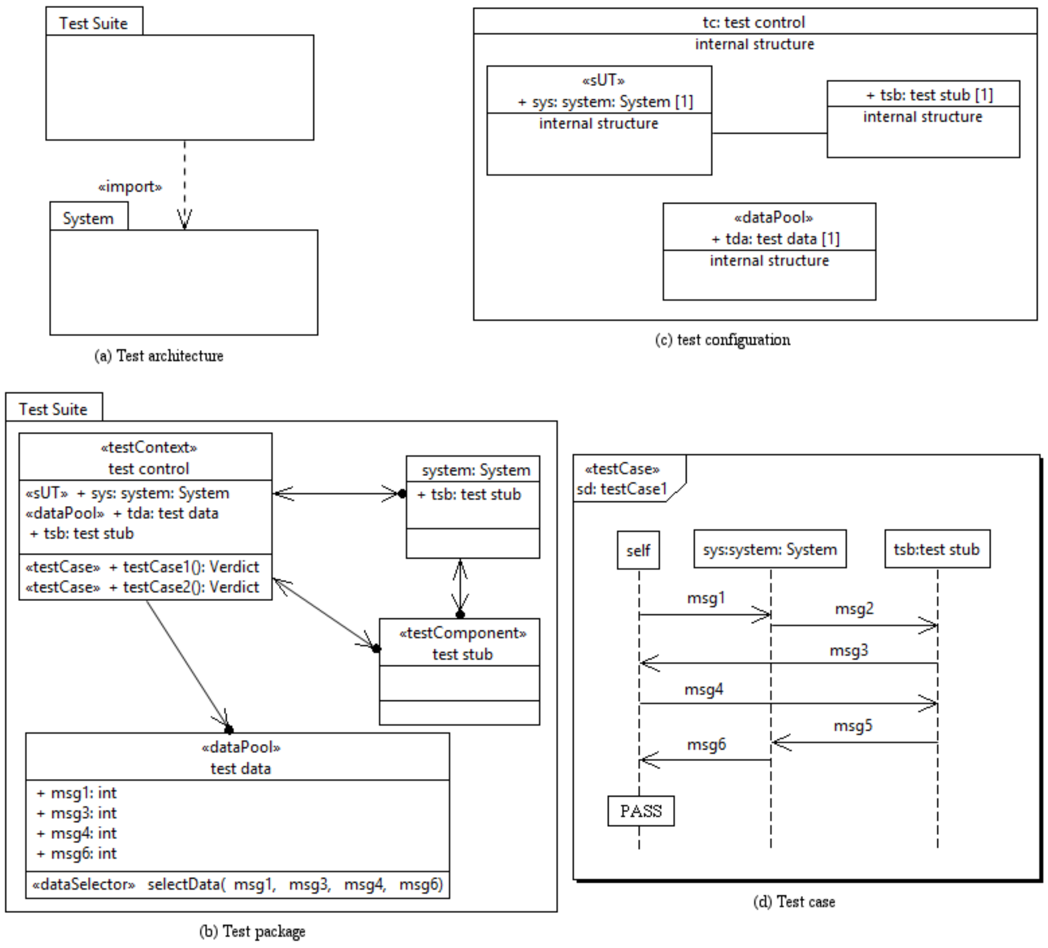


Figure 4. UTP test model

In addition to test objects, abstract test stimulus can be defined in the test package through three mechanisms: UTP data pool, UTP data selector and UTP data partition. UTP data pool works as a container/database to the test stimuli. It is defined as UML class in the test package with UTP stereotype *DataPool*. UTP data selector facilitates the implementation of different data selection strategies [18]. It is defined as UML operation in UTP data pool or UTP data partition and tagged with UTP stereotype *DataSelector*. UTP data partition allows the classification of data to subsets. It is known as an equivalent class in the software testing [43, 8]. It is defined as UML class with UTP stereotype *DataPartition*. UTP data partitions must be associated to UTP data pools or other UTP data partitions, which allows the existence of hierarchy in the data classification.

A test configuration, as shown in Figure 4.c, defines the test setup. It describes relationships among instances of the test objects. The UML composite diagram of the test control class is used for the test configuration. Different test configuration diagrams may be built to represent different test setups. Each test case, or set of test cases, is associated with a specific test configuration diagram.

The abovementioned UTP diagrams define the test structure; test behavior is defined through a set of test cases. Behavioral UML diagrams, sequence, activity and state machine, are used to express test cases; the test case shown in Figure 4.d is specified in UML sequence diagram. Test cases should be linked to their corresponding operations in the test control's operation compartment. UTP concepts are used to enrich these diagrams with the necessary test specification. There is one limitation in the current UTP specification, UTP 1.2, that the UTP metamodel does not include the test behavior; it is left out for future releases. Hence, test developers have to look around through UML metamodel to identify association among test objects in the test structure and their counterpart instances in the test behavior.

In addition to the aforementioned concepts, other concepts can be used to model precise test specifications. For example, testers can use time concepts to define a shared time zone among a group of test components, use data concepts to define wild cards for ignoring unimportant data in the test model, or use test arbiters to evaluate the test case verdict. A UTP test model can be mapped to test execution environments such as JUnit or TTCN-3 to execute the test cases and analyze the results [18]. The UTP specification provides mapping rules to the two test execution environments. Hence, the software development process can be handled exclusively using UML/UTP models. Figure 5 shows a UML and UTP centric software development process proposed by Baker et al. [5]. Using a widely accepted modeling language, UML, throughout the software development process enables robust collaboration among the software stakeholders.

2.4 Literature Review

In this section, we review related work. While our work is mainly devoted toward the domain of model based testing, we have also touched upon the domain of model comparison and merging. Furthermore, we have also considered test-suite reduction techniques to distinguish our optimization approach from them. Therefore, we structure related work into three subsections, one for each research topic. The first subsection presents related work in model based testing. The second subsection presents related work in model comparison and merging. The third and last subsection presents test-suite reduction and some approaches in this research topic.

2.4.1 Model Based Testing

Model based testing refers to the use of models defined in software constructs to build test models and drive the software testing [4]. The use of models in software testing goes back to the mid of the 20th century but recently it got a growing attention in the software development domain [4]. While a piece of code can be considered as a model, our focus is on graphical models that have formal or semi-formal specifications. The literature shows a diversity of MBT techniques based on several factors such as modeling notations, dependency on the development models, and degree of automation [4, 9, 55-56]. Our objective is to link different testing levels with enabled reusability and optimization. To the best of our knowledge, there is no existing systematic framework that links different testing levels with enabled reusability and optimization. In the rest of this subsection, we discuss standalone MBT techniques. First, we present MBT techniques based on UML notation; then, we present MBT techniques based on UTP, and we conclude by presenting MBTs based on non-UML notations such as FSMs.

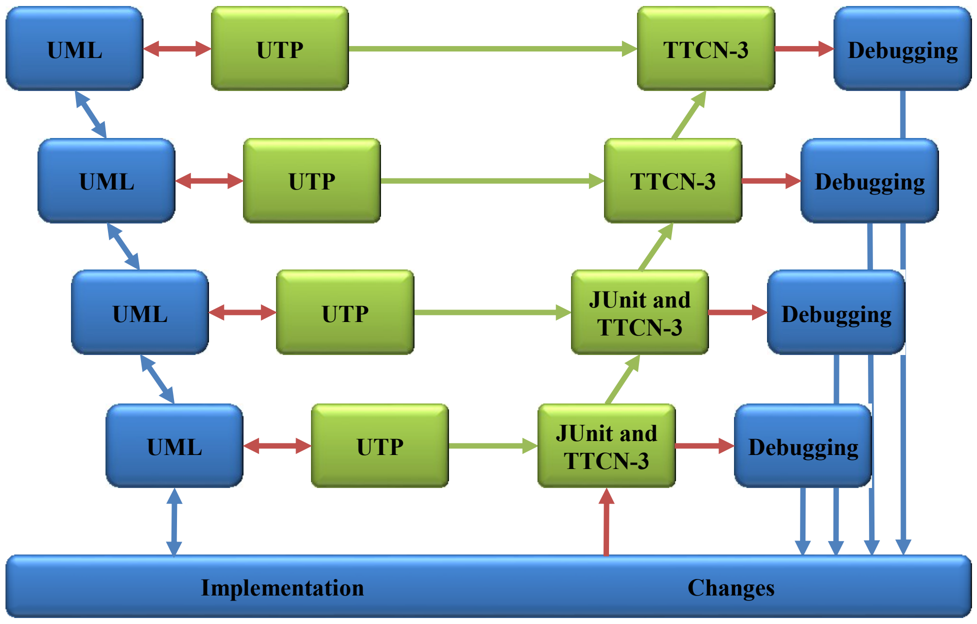


Figure 5. UML/UTP W-model [5]

2.4.1.1 MBT Approaches Using UML Models

Model-based testing approaches based on UML have been proposed for different testing levels; see for instance [20, 25, 57-61]. Moreover, several domains have been targeted including automotive, health, and telecommunications [23, 57, 62]. However, most of these studies focus on one stage of the software testing, mainly unit-level or system-level testing.

The closest work to our test generation approach was proposed by Le [63]. Le proposes a composition approach based on UML 1.x collaboration diagrams. Component test models are developed manually. The test model is composed of two test objects: the component under test (CUT) and the test control. The test control controls and performs the test suite, and simulates all necessary stubs and drivers. The author demonstrates the reusability of the component test models to build integration test models through introducing adaptors between the component test models. In this approach, the role of the test control becomes more complex since it is composed of the test management and the required stubs and drivers. Granularity is a significant characteristic in software engineering. Separating the test management from the test stubs improves the reusability and simplifies the test implementation. Real entities of test stubs may already exist; utilizing them provides testing results that are more accurate. The approach deals with the external interfaces of the two composed components. However, the author does not address the interconnection between the two composed components nor the synchronization between events of test behavior. Furthermore, the test case selection is not clear, since not the entire unit test cases are suitable for the integration-level testing.

Machado et al. [64] present a UML based approach for integration-level testing using Object Constraint Language (OCL) [65]. The authors illustrate a complete testing approach for integration-level testing. The component specification is described in UML class diagram and sequence diagram with OCL constraints. UML use case diagrams are used to describe the components' services (interfaces). To generate interaction test cases, a set of UML communication diagrams are created based on use case scenarios. However, the authors did not mention the synchronization of the events in the generated communication diagrams since there is no event ordering in the UML use case diagram. This order can be extracted from the provided sequence diagrams, but sequence diagrams may cover partial views of the integrated components and require a merging technique to get the global picture.

El-Attar and Miller [66] propose a framework for generating acceptance test cases from UML use case diagrams and robustness diagrams [67]. The system requirements are described using UML use case diagrams and the domain model diagram. The framework goes through three phases. In the first phase, high-level acceptance tests (HLATs) are generated for each UML use case. The UML use case along with the related domain model is inspected to generate these HLATs. HLATs are composed of semi-narrative text in use case syntax. In the second phase, a robustness diagram is created for every HLAT. The generation of robustness diagrams may require the update of the domain model diagram with missing objects and attributes. In the third phase, executable acceptance test cases (EATs) are generated from HLATs, UML use case diagram, domain diagram, and robustness diagrams. The Fit (Framework for Integrated Test) [68] format is proposed for EATs. A tool was developed to implement this approach. This approach applies MBT and Fit methodologies. However, the first two phases are carried manually since informal artifacts, UML use case (text part) and HLAT, are manipulated during these phases.

2.4.1.2 MBT Approaches Using UTP Models

Several MBT techniques based on UTP have been proposed. We discuss some of them in this paragraph. Busch et al. [22] present an MDA approach for generating test models from design models. PIMs are transformed into platform independent test models (PITs). Platform specific test models (PSTs) are generated either from platform specific models (PSMs) or from platform independent test models (PITs). Both PIT and PST are based on UTP. PST models are submitted to a test execution environment, TTCN-3, for evaluation. The approach focuses only on the system-level testing, while our generation approach focuses on the integration-level testing.

Lamancha et al. [20] propose an MDA approach to generate UTP models from the system design models. UTP models are built from UML use cases and sequence diagrams. UML design model is transformed to UTP model. However, this approach targets only system-level testing.

Liang and Xu [19] present Test Driven Development (TDD) [69] for component integration based on UML 2.0 Testing and Monitoring Profile (U2TMP) [19], which is a proposed extension to UTP to enable monitoring. The generated test cases are used to build a glue code between the integrated components. However, integration test cases are created manually, and there is no utilization of component test cases.

Yuan et al. present an automatic approach in [70] to generate test cases of a given business process of a web service. BPEL (Business Process Execution Language) [71] and UML activity diagrams are used to define the Process Under Test (PUT). The UTP and the TTCN-3 concepts are used to construct the test cases. The generated test model can be tailored to target any of the following test types: unit testing, component testing or system testing. The approach applies two automatic transformations to generate an executable test case set. The first transformation is used to build the Abstract Test Cases (ATC) from two models: the PUT model and the test case model. The test case model is based on UTP framework and TTCN-3 key concepts. The second transformation is applied on the ATC to generate executable test case scripts, which are executed in the TTCN-3 environment. The authors' approach presents a practical application of the UTP framework.

Baker and Jervis [21] present an approach that is similar to the previous one (i.e., relies on the UTP standard). In addition to generating test cases for validating the implementation, they provide a mechanism to validate the design model at early stages of the software development cycle. Timing and concurrency have also been handled by their approach. The approach has been successfully applied in many projects.

2.4.1.3 MBT Using Non-UML notations

Testing approaches based on FSMs are frequently used, such as in [72-75]. The component specification is given as an FSM. FSM models of the composed components are merged to create a global behavior model. Test cases are generated from the global model. However, the construction of the global behavior may lead to the well know problem of state space explosion [44]. New methods for avoiding this state space explosion and reducing the final number of test cases, such as C-Method [74], have been proposed.

Haugset and Hanssen [76] propose an acceptance test generation approach. The approach generates acceptance test cases based on Fit in agile processes. Fit shifts the acceptance testing to the customer side. Test cases are created for each story card. Customers build a table with the system inputs and the expected output for each story card. Developers, on their site, write a small code, called fixture, to link the table with the system. Fit tools execute the test suite and report the results. The major drawback of this initiative is the customer experience and applicability for complex systems.

2.4.2 Model Comparison and Merging

Model comparison and merging has been an essential part of the software development for decades [77, 78]. The first launch for such approaches targeted textual files to identify similarities and differences between two files. Mature tools are available to handle software versioning and clone-detection. With the introduction of model driven engineering, new approaches are required to manage graphical models since old approaches are line oriented and cannot deal with hierarchy and model semantics [79, 80]. Different approaches have been proposed to handle graphical models [79, 80]. Some are domain specific or modeling notation specific while others are more general and domain independent [80]. These approaches target different aspects of the software development lifecycle: Version Control Systems (VCS) [81], Model Cloning [78, 82], and Model Transformation Testing [51, 52]. They target a variety of model types: structural, behavioral and data-flow. For UML models, the research is devoted more on structural diagrams [83-85], particularly the class diagram, than on behavioral diagrams [86, 87]. As far as we know, there is no model comparison in the software testing. In model comparison, approaches are developed based on one characteristic: the assumption that the compared models have evolved from the same source model/fragment; it is usually called the base model. The approaches have been classified into two categories according to the information required to manage the comparison: three-way comparison and two-way comparison [77]. Three-way comparison techniques require the existence of a base model in addition to the two models. Two-way comparison techniques compare two models without external references; however, they are also based on the assumption of the existence of the base model. Furthermore, different measurements are used to evaluate the similarity factor between the elements of the compared models. While there is no formal classification of such measurements, we can list four of them that are recognized by published surveys in this field: unique identifiers, names, features and size. Approaches using the first measurement are language specific; they require that each element has a static Universal Unique Identifier (UUID). The second measurement uses element's names for comparison. Störrle [82] shows the effectiveness of such an approach on UML models. The third measurement enhances the second measurement by using the attributes of the elements in additional to their names. The last measurement uses the size of elements to compare large models.

The closest approach to our work is proposed by Liu et al. [87]. Their work was on model-cloning using UML sequence diagrams. The approach converts the sequence diagrams into an array. This array is represented as a suffix tree. Duplication is detected by traversing the tree and applying the longest common prefix algorithm. Our approach is different in two aspects. First, this approach handles only synchronous messages, while ours handles asynchronous messages as well. More importantly, this approach is restricted to contiguous behavior. It handles adjacent events. In our domain, the shared test events could be scattered across different test objects and could be split by un-matched test events.

Hélouët et al. [88, 89] propose a merging approach for MSC specifications. The approach covers both basic MSC (bMSC) and high MSC (HMSC). The approach merges all scenarios to build the global behavior of the system. It solves the non-local choice by creating a new object, called controller, which controls the merged scenario. The controller broadcasts a sensing message to ask all objects about the path they are willing to take. The chosen path of the first object to answer to the message will be taken, and the controller broadcasts his decision in order to be followed by all other objects. Our approach is different. We merge only two scenarios at a time and only a subset of given scenarios, which capture interactions between the integrated parties, are merged. Furthermore, bMSCs are integrated according to their relation in the corresponding HMSC specification: sequence, alternative, parallel, etc. Consequently, the integration may produce a non-local choice. However, our approach applies only merge composition. More importantly, we do not change the behavior of the given test specifications.

In this research, model comparison is used for the test generation approach and the test optimization approach, while model merging is used for test generation approach. However, we cannot assume that test models evolved from the same base model since they are often built independently.

2.4.3 Test-Suite Reduction

The execution of test models is time consuming. Test models generally consist of a large number of test cases. The reduction of such tests improves the software testing. There are ongoing research activities toward the reduction of tests [90, 91]. The effort of such research focuses on the reduction of test models from within the model itself. Additional data is required to describe each test case according to certain criterion such as test objectives and test coverage. Approaches analyze such data to detect shared criterion among test cases, e.g.: test cases that have the same coverage. Then approaches remove duplicated test cases, which share the same criteria.

Tallam et al. [92] propose a test-suite reduction approach. The approach requires a set of test cases and a set of test requirements. Each test case covers a set of test requirements. This information is provided as input in a table. The approach uses the table to select the minimum set of test cases that covers all the test requirements. This approach can be used as a first step in our approach to select the set of acceptance test cases that covers all the test requirements, if provided, before comparing them to integration test cases.

Chapter 3
Model Based Testing Framework

In this chapter, we present a model based testing framework for linking different testing levels. We discuss the necessity of such a framework and the modifications required on the traditional development lifecycle.

3.1 Framework

Software testing approaches have been proposed in standalone fashion for several years. Even with the introduction of model driven engineering, approaches are developed to target a specific testing level. In this dissertation, we present a model based testing framework linking testing levels for enabling reusability and optimization. While the framework can be applied on any well-formed test models, UTP test models are used in this dissertation. Figure 6 depicts the framework. The framework consists of two approaches: a test generation approach and a test optimization approach. The framework enables the reuse of test models to generate subsequent test models. The generation approach is used to link

- The component-level testing to the integration-level testing and
- The component-level testing to the system-level testing.

In this work, we focus on the generation of integration test models from component test models. In this thesis, a component is defined as a self-coherent piece of software that provides one or more services, and can interact with other components. Furthermore, the framework enables the optimization of test models by mapping them to the previously executed test models. The optimization approach is used to link

- the integration-level testing to the system-level testing,
- the integration-level testing to the acceptance-level testing and
- the system-level testing to the acceptance-level testing

We will briefly discuss these approaches in the following two sections and develop them in the following chapters.

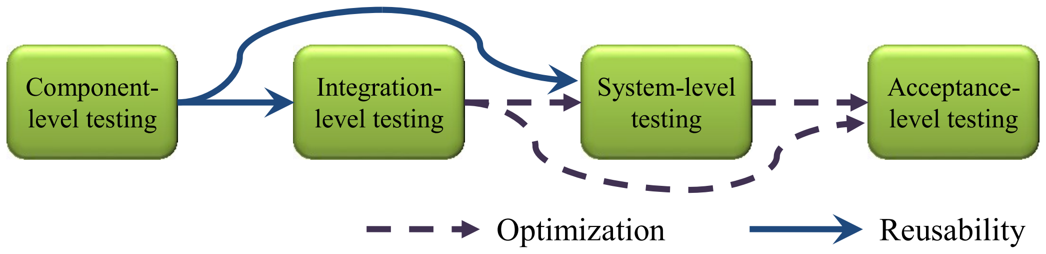


Figure 6. Model based testing framework

The framework conforms to the software development process as illustrated clearly in the w-model shown in Figure 7. The links that are caused by the generation approach are indicated by black solid arrows, while the links that are caused by the optimization approach are indicated by black dotted arrows. However, some changes have to be adopted. For the generation approach, the framework requires modification of the software testing by reordering the preparation sequence during the design stage. The traditional software testing begins by the preparation of the acceptance test model, then the system test model, then the integration test models, and finally the component test models as specified clearly in the w-model, as shown in Figure 1. In our work, the preparation of the acceptance test model is still at the head of the software testing. However, the preparation order of the rest of the test models is reversed: component, integration then system test model. Engineers begin by developing the component test models. There are many systematic MBT approaches for generating component test models based on UTP from the design models [19, 20, 23, 25, 58, 70, 93]; we have discussed some of them in Section 2.4.1. Next, integration test models and system test model are automatically generated from the component test models. For the optimization, no change is required in the software testing since the execution of the integration testing precedes the execution of the system-level and the acceptance-level testing.

3.2 Test Generation

Software reuse is a mature discipline in software engineering [29, 94-96]. Enormous research activities focus on software reuse; CBSD [97, 98] and Software Product Lines (SPL) [99] are well known practices of software reuse. However, the literature of software reuse does not provide any evidence of systematic reuse for test generation. To the best of our knowledge, there is no systematic test generation approach that relates and links the testing levels. Furthermore, generation of integration test models from component test models is a research challenge as it has been stated by Bertolino [4]: "What remains an open evergreen problem is the theoretical side of component-based testing: how can we infer interesting properties of an assembled system, starting from the results of testing the components in isolation?"

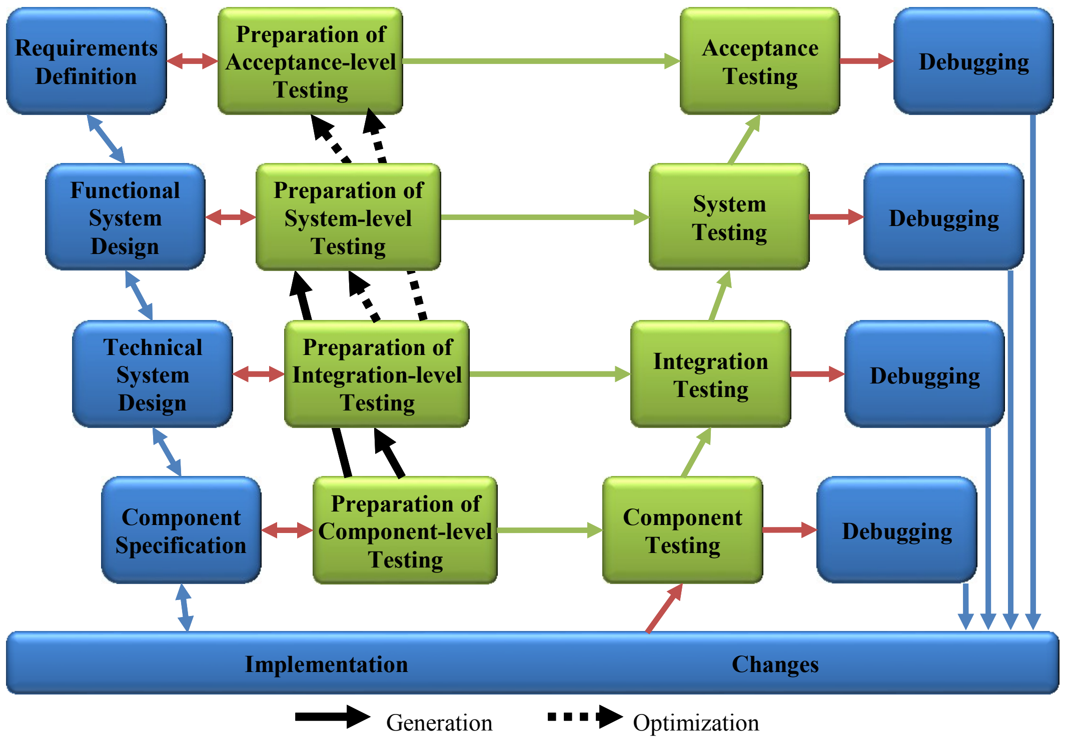


Figure 7. The MBT framework included in the w-model

In the literature, model based testing techniques follow the process described in Figure 8. In general, test models are generated from the design specification and transformed to test code, which is eventually exercised on the implementation. The process would work perfectly on component-level and system-level, but not during integration-level. Systems, nowadays, are very complex and the policy of divide and conquer is still applicable. To overcome this complexity, systems are divided into components, which are divided into small fragments, units and classes. The implementation starts by developing the small fragments, which are integrated to build components. In their turn, components are integrated to build the complete system. In parallel to that, component test models are generated from component design models; integration test models are generated from sub-system design models, and a system test model is generated from a system design model. From software testing perspective, generating integration test models from sub-system design models will check the functionality of the corresponding sub-systems. It should be called sub-system-level testing rather than integration testing. Integration-level testing focuses on checking the compatibility and inter-connectivity between the integrated components. Generating integration test models, in a systematic way, from sub-system design models is more complex and requires extra information. Engineers must specify explicitly in the design model the newly integrated component and its interfaces during each integration iteration. This can be accomplished by tailoring the design model and adding special tags or stereotypes, or providing a separate model with the required information linked to the design model. Hence, the integration test, by using such technique, is shifted from black-box testing to gray-box testing which is impractical.

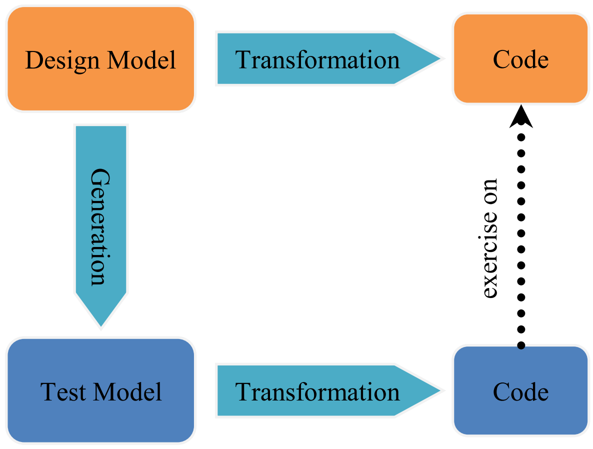


Figure 8. MBT process

We propose a test generation approach that enables reusability across the testing levels. Our approach reuses the component test models to generate the integration test models as well as the system test model. Figure 9 illustrates the test generation in our framework. The approach starts by generating the first integration test model, ITM_1, from the component test models, CTM_1 and CTM_2, of the integrated software components. The integration test model is exercised on the integrated components. Upon a successful test result, the current integration test model, ITM_1, is integrated with the component test models, CTM_3, of the next available component to generate a new integration test model, ITM_2. This process is repeated until all components are integrated and tested successfully. Finally, the system test model is generated by integrating the component test models. We discuss the integration test generation in more details in Chapter 4. The system test generation is left-out for future work.

3.3 Test Optimization

The software testing is time consuming. The size of test models is generally large for complex systems. The number of test cases grows rapidly by time due to software modifications; new test cases are added after every software fix or upgrade request. The need for test optimization triggered a research field in the software testing known as Test-Suite Reduction [90, 91]. Researchers, in this field, work on reducing the number of test cases in the test model by removing redundant test cases. Redundancy is calculated based on different aspects such as test coverage and test requirement. However, they do not take into account the optimization of test cases across the software testing. In this dissertation, we propose a complement approach that optimizes test models across testing levels. More specifically, we focus on optimizing test models across integration, system and acceptance testing. Here, we are targeting acceptance testing performed on the development site (alpha testing), not the one performed on the user site (beta testing).

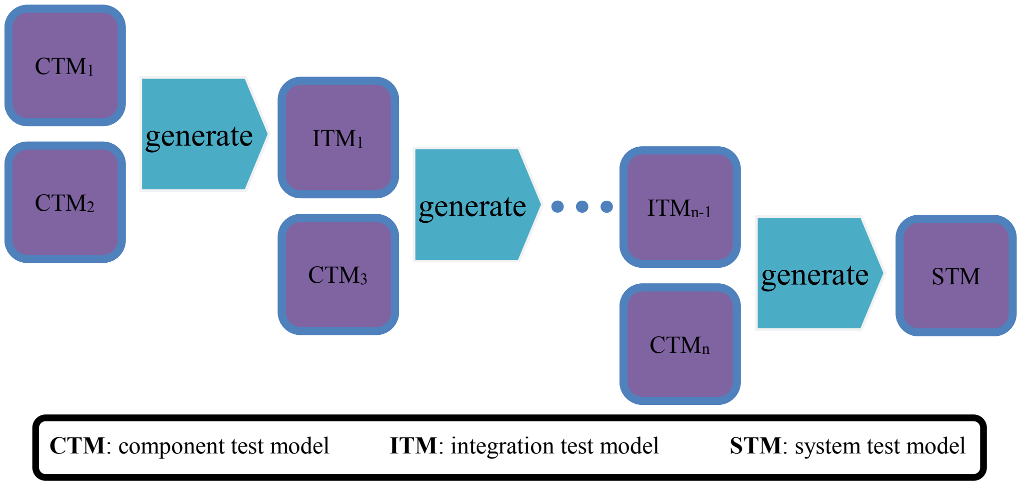


Figure 9. Test generation approach

Large numbers of test cases are generated and exercised during each testing level. These test cases are used to check the functionality of the system and discover bugs. Each test case is meant to examine a specific behavior or service of the system. Our goal is to prevent the execution of test cases that have been already executed on the system during previous testing levels. Hence, we aim to reduce the number of test cases in the test suite. The optimization of the acceptance test model using the system test model is understandable since both test models are applied on complete systems. However, the optimization of the acceptance test model and the system test model using the integration test models is more difficult since integration test models are applied on incomplete systems. We elaborate more on the later optimization approach in Chapter 5.

3.4 Test Model Definition

We conclude this chapter by introducing definitions that are used later for our approaches.

Definition 1. (Test Model)

A test model is represented as a double

$M = (P, T),$

Where

P is the test package

T is a set of test cases

Definition 2. (Test Package)

A test package is expressed as a tuple

$P = (tcn, tcm, sut),$

Where

tcn is the test control

tcm is a set of test components that are required to realize the test execution (test stubs)

sut is a set of components under test

Definition 3. (Test Case)

A test case is expressed as a tuple

$t = (I, E, R),$

Where

I is a set of instances

E is a set of events (defined further in Definition 4)

$R \subseteq (E \times E)$: is a partial order reflecting the transitive closure of the order relation between events on the same axis and the sending and receiving events of the same message

We classify events into three categories: message events, time events and miscellaneous events. Message events, the sending event and receiving event, represent the two ends of messages exchanged between two instances referred to as the sender and the receiver, respectively. In this dissertation, messages are instances of an execution trace. Hence, they are unique throughout a single system execution. Time events represent events related to timers. Each timer is associated with one instance. We classify the rest of event types, such as instance termination and UTP verdict, into the third category. Notice that the association between events and instances is part of the event definition in this work.

Definition 4. (Event)

We have three different kinds of events; hence, there are three definitions:

1. A message event E_{msg} is a tuple *(ty, nm, owner, msg, oIns)*, where
   - (a) *ty* ∈ {*send*, *receive*}
   - (b) *nm* is the event name
   - (c) *owner* is the instance where the event belongs to. *owner = (nm, st)*, where
     - (i) *nm* is the instance name
     - (ii) *st* is the UTP stereotype of the instance
   - (d) *msg* is the message the event is related to
   - (e) *oIns* is the other instance related to msg, *oIns = (nm, st)*, where
     - (i) *nm* is the instance name
     - (ii) *st* is the UTP stereotype of the instance
2. A time related event E_{time} is a tuple *(ty, nm, tm, owner, pd)*, where
   - (a) *ty* ∈ { *timeOutMessage, startTimerAction, stopTimerAction, readTimerAction, timerRunningAction* }
   - (b) *nm* is the event name
   - (c) *tm* is the timer name
   - (d) *owner* is the instance where the event belongs to, *owner = (nm, st)*, where
     - (i) *nm* is the instance name
     - (ii) *st* is the UTP stereotype of the instance
   - (e) *pd* is the timer value
3. A miscellaneous event E_{misc} is a tuple *(ty, nm, v, owner)*, where
   - (a) *ty* ∈ {*Action, Terminate, UTPverdict*}
   - (b) *nm* is the event name
   - *(c)* *v* is the value associated with the event. This value can be *pass, fail, inconclusive, error* in case *ty = UTPverdict.*
   - (d) *owner* is the instance where the event belongs to, *owner = (nm, st),* where
     - (i) *nm* is the instance name
     - (ii) *st* is the UTP stereotype of the instance

We use the test model specified in Figure 10 to illustrate our definitions. The test model is composed of a test package, *p*, that represents the test architecture and two test cases, *t1* and *t2*, that represent the test behavior. To distinguish between the sending and receiving events of the same message, we suffix the message name with the first letter of the corresponding action. We define this test model, *M*, as follows:

$M = (P, T)$

$P = (TC, \varnothing, \{CUT\})$

$T = \{ t_1, t_2 \}$

$t_1 = (\{tc,cut\}, \{m_{1s}, m_{2r}, m_{3s}, m_{4r}, ver, m_{1r}, m_{2s}, m_{3r}, m_{4s}\}, \{(m1s,m2r),(m2r,m3s),(m3s,m4r),(m4r,ver),(m2s,m3r),(m3r,m4s),(m1s,m1r),(m2s,m2r),(m3s,m3r),(m4s,m4r),(m1s,m3s),(m2r,m4r),(m2r,m3r),(m3s,ver),(m2s,m4s),(m3r,m4r),(m2s,m3s),(m3s,m4s),(m4s,ver),(m1s,m4r),(m1s,m3r),(m2r,ver),(m2r,m4s),(m2s,m4r),(m3r,ver),(m1s,ver),(m1s,m4s),(m2s,ver)\})$

$tc = ("tc", TestContext)$

$cut = ("cut", SUT)$

$m_{1s} = (send, "m_{1s}", tc, m_1, cut)$

$m_{2r} = (receive, " m_{2r}", tc, m_2, cut)$

$m_{3s} = (send, "m_{3s}", tc, m_3, cut)$

$m_{4r} = (receive, "m_{4r}", tc, m_4, cut)$

$ver = (UTPverdict, "ver", "pass", tc)$

$m_{1r} = (receive, "m_{1r}", cut, m_1, tc)$

$m_{2s} = (send, "m_{2s}", cut, m_2, tc)$

$m_{3r} = (receive, "m_{3r}", cut, m_3, tc)$

$m_{4s} = (send, "m_{4s}", cut, m_4, tc)$

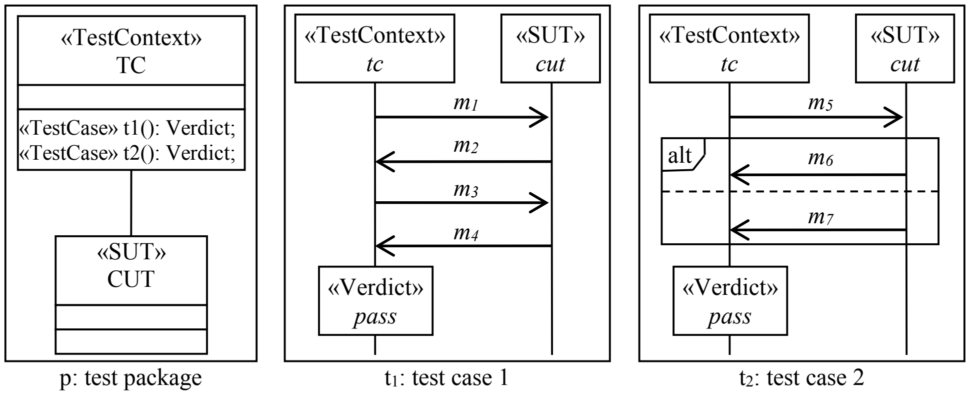


Figure 10. Test model (*M*)

$t_2 = (\{tc,cut\}, \{m_{5s}, m_{6r}, m_{7r}, ver, m_{5r}, m_{6s}, m_{7s}\}, \{(m_{5s},m_{6r}),(m_{5s},m_{7r}),(m_{6r},ver),(m_{7r},ver),(m_{5r},m_{7s}),(m_{5s},m_{5r}),(m_{6s},m_{6r}),(m_{7s},m_{7r}),(m_{5s},ver),(m_{5r},m_{7r}),(m_{5s},m_{7s}),(m_{6s},ver),(m_{7s},ver),(m_{5r},ver)\})$

$tc = ("tc", TestContext)$

$cut = ("cut", SUT)$

$m_{5s} = (send, "m_{5s}", tc, m_5, cut)$

m_{6r} = (receive, "m_{6r}", tc, m_6, cut)

m_{7r} = (receive, "m_{7r}", tc, m_7, cut)

ver = (UTPverdict, "ver", "pass", tc)

m_{5r} = (receive, "m_{5r}", cut, m_5, tc)

m_{6s} = (send, "m_{6s}", cut, m_6, tc)

m_{7s} = (send, "m_{7s}", cut, m_7, tc)

3.5 Conclusion

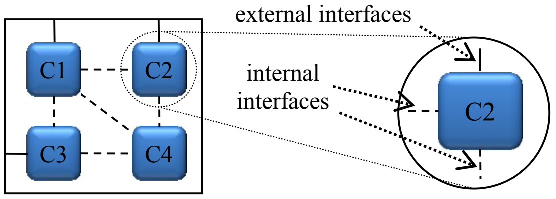
The main characteristics that differentiate our framework from existing work are reusability, optimization and smooth transition among the testing levels. We are linking testing levels by relating test models from one testing level to test models of preceding testing levels. Test models are reused to construct the subsequent test models. Acceptance test cases are reduced to improve the software testing. A standard modeling language, UML, is utilized throughout our framework.

Chapter 4
Integration Test Generation

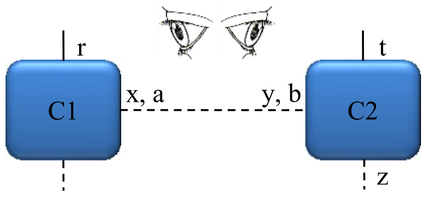
This chapter presents the integration-test generation approach. We discuss the generation of integration test models form component test models. The chapter is organized into three sections. We introduce the overall test generation approach in the first section and discuss decisions made during the development of our approach. In the second section, we discuss the integration test generation approach and the four processes that compose the approach: the identification process in Section 4.2.1, the selection process in Section 4.2.2, the generation process in Section 4.2.3 and the redundancy removal process in Section 4.2.4. In the Subsection 4.2.5, we discuss different strategies that have been used to carry on previously integrated test models. Finally, we conclude the chapter in Section 4.3.

4.1 Introduction and Overview

Test models are composed of a set of test cases, and these test cases capture the test behavior that is exercised on the targeted implementation. Test behavior in general reflects the behavior of the implementation under test. We believe that the collective test behavior of all component test models capture the system behavior. In practice, some research activities migrate system behavior across different development stages using test cases since test cases are finite and precise comparing to the system design models [6]. Component testing is a black-box testing; tests are exercised on components through their interfaces. These interfaces can be internal, to communicate with components, or external, to communicate with the system environment, as shown in Figure 11.a. During the component-level testing, several test cases are specified for the same interface; each test case is included in a different test model and corresponds to a different component, which uses this interface. While these test cases use the same interface, the specification may be different since it is taken from different views. In other words, while these test cases have different syntax, they describe the same system behavior. We illustrate this point using the example shown in Figure 11.b. The two components, *C1* and *C2*, exchange messages through the internal interface. Let us assume that the test models of the two components include a test case that covers this interface.



(a) Interfaces of system components



(b) Different views of the same interface

Figure 11. Component interfaces

The test case of the first component, *C1*, specifies the interface as follows:

- Component *C1* as CUT,
- Component *C2* as test stub,
- Messages *x* and *a* as inputs and
- Messages *y* and *b* as expected outputs.

On the other hand, the test case of the second component, *C2*, specifies the interface as follows:

- Component *C2* as CUT,
- Component *C1* as test stub,
- Messages *y* and *b* as inputs and
- Messages *x* and *a* as expected outputs.

As a result, we have two different test specification of the same system behavior. Hence, we conclude that test cases of different component test models may overlap. In our research, we focus on component interfaces to generate the subsequent test models.

Prior to introducing the generation approach, we have to emphasize on the characteristics/quality of the component test model. In order to generate test models from component test models, the component test cases must be well-formed and capture the following characteristics. In addition to testing the internal functionality of the components, component test models should include test cases that completely cover all the interfaces of the targeted component.

Every test case should cover complete services, which are provided by the corresponding component. Furthermore, there should be a consistency among the specifications of the component test models since they describe different components of the same system. The names of the components, interfaces and messages should be consistent among the test models.

Integration testing examines the consistency, interconnectivity and compatibility among the integrated components. Hence, performing integration testing on independent unrelated components is irrelevant. Applying appropriate integration strategy and order increase the efficiency of the integration testing. To generate integration test cases from component test cases, we need to search for component test cases that examine the same services on the same interfaces that connect the integrated components. This search has to be performed on the two component test models related to the integrated components. By examining these test cases, we may reveal an overlapping between their specifications from which we could generate integration test cases. To illustrate our point, we use test cases in Figure 12; the specification is based on the architecture in Figure 11.b. There are two services available on the internal interface between the two components, *C1* and *C2*. The components exchange messages *x* and *y* to perform the first service, and exchange messages *a* and *b* to perform the second service. Assume we have one test case from *C1* test model as shown in Figure 12.a and two test cases from *C2* test model as shown in Figure 12.b & 12.c that examine the internal interface between *C1* and *C2*. According to our required characteristics, there should be another test case in *C1* test model that covers the second service, but we omitted it just for simplicity. By comparing the specification of *C1* test case to the specification of *C2* test cases, one can see that there is a shared behavior, exchanging *y* and *x*, described in *C1* test case 1, Figure 12.a, and *C2* test case 2, Figure 12.c. Here, we do not count the test verdict, PASS, because it is not a system behavior but a test property. Hence, the specifications of the two test cases are overlapping. The specifications of the two test cases can be merged to produce an integration test case as shown in Figure 12.d.

System integration is an iterative process. Components are integrated into system context in incremental manner. During each iteration, test models are exercised on the integrated components to examine the consistency and interoperability among them. We support the integration of one component at a time; hence, the approach supports the most known software integration strategies, top-down, bottom-up and ad-hoc. Engineers can take different orders to integrate the system components. There are research activities that investigate the selection of the optimum integration order [40-42]. The integration order may separate adjacent components that have direct interactions. This issue may lead to the loss of integration information that is carried by the component test models. To accommodate different integration orders, we carry on component test models to the subsequent integrations as they may be used to generate additional integration test cases. We elaborate more on this issue in Section 4.2.5. In this case, our approach produces consistent results regardless of the integration order that would be taken. In Appendix A, we discuss the impact of selecting different integration strategies on the results of our generation approach.

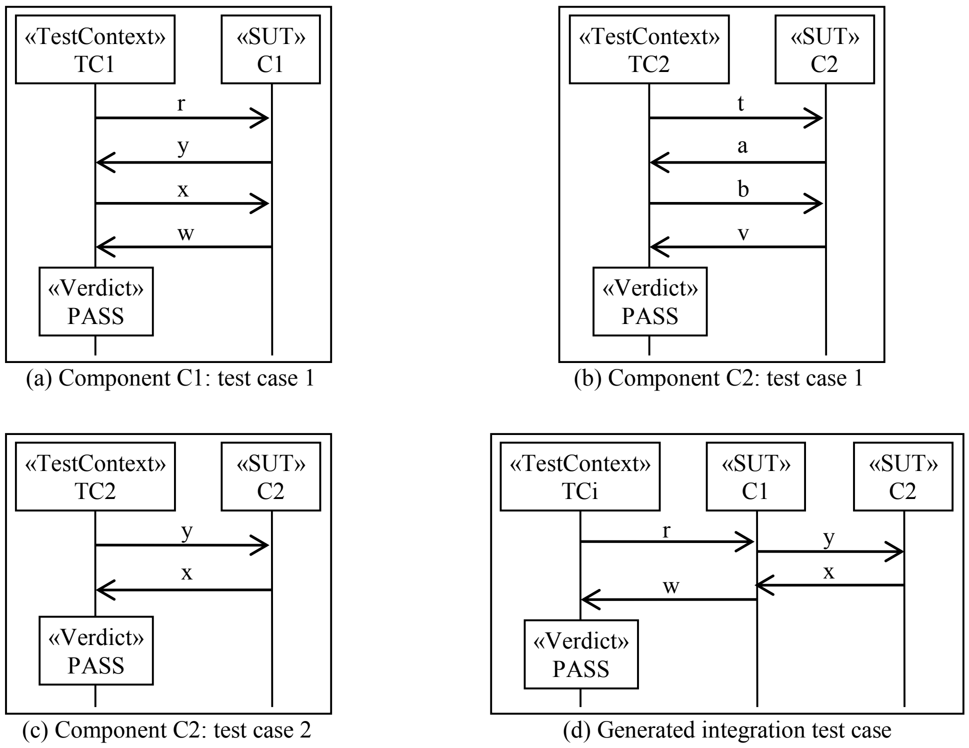


Figure 12. Overlapping test cases

4.2 Integration Test Generation Approach

The approach goes through an iterative process to generate integration test models corresponding to the development integration stages as described in Figure 13. In the first iteration, the approach begins by considering component test models of the first two components to be integrated to build a sub-system. The two component test models are examined and used to generate the integration test model. The test cases of the generated test model have to reflect interactions between the integrated components. The integration of the two components builds a sub-system that is eventually integrated with a third component of the system. In the second iteration, the former integration test model is used to generate the current integration test model along with the component test model of the third component. The component test models of the first two components are also examined to extract test cases that capture interactions with the third component and have not been carried on by the first integration test model. This process is repeated for the subsequent iterations to generate the subsequent integration test models until the integration of the component test model of the last component.

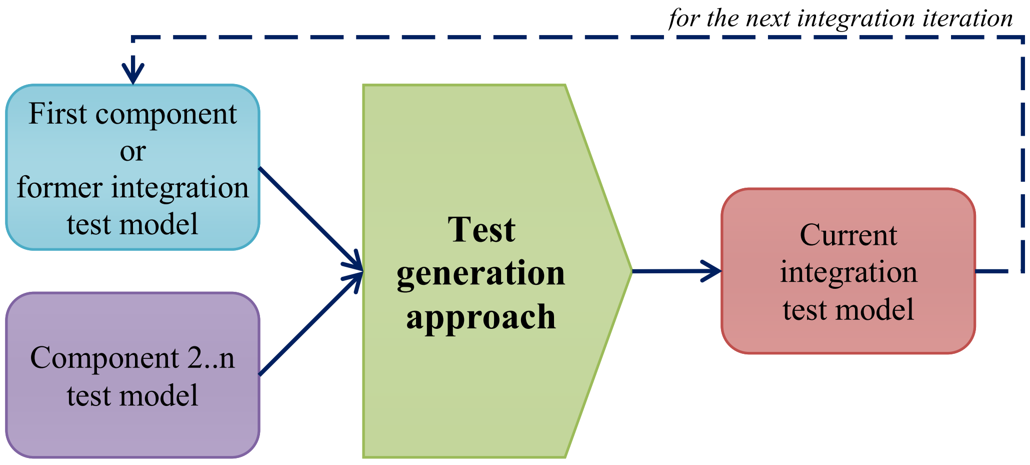

for the next integration iteration
First component or former integration test model
Component 2..n test model
Test generation approach
Current integration test model

Figure 13. Integration test generation approach

The generation approach depends on the quality of the component test models. As an input to our framework, component test models can be systematically generated by several techniques such as [19, 20, 23, 25, 58, 70, 93]. Component test models can be created by the same engineer or different engineers. They can be created on the same development site or on different development sites as in CBSD. Software cloning may be applied to parts of the component test models. In this work, we make no assumption about the creation of component test models; we treat each component test model as original work. However, we require some consistency among the component test models of the same system. The name convention of components, interfaces and messages should be consistent throughout the software testing. Furthermore, test cases of a component test model should completely cover the interfaces of that component. While our methodology can be applied on any well-formed test model, we developed our approach based on the UTP test model. The test architecture should be specified using UML class diagram and the test behavior should be specified using UML sequence diagram, which has been formally investigated [33-35].

Component test models have to be mapped against each other in order to extract integration test specification from them. There is a lot of work on comparing UML class diagrams, but rare work is devoted toward comparing UML sequence diagrams [87]. However, these techniques assume the evolving of the compared models from the same source. In our research, we assume that models are different and we need to look for similarities among them. Similarities can be captured from the existence of shared interfaces among the compared components, that is why we insisted on covering all interfaces of each component in the component-level testing. Excluding implementation under test, a test object can embed the behavior of several real entities. These real entities can be a system environment and/or system components that are not realized during the test execution. Therefore, we need to analyze these test objects prior to model comparison in order to compare each test object to its corresponding ones on the other test model whether they have a standalone specification or their specification is embedded in other test objects. Moreover, during the comparison, test behavior may overlap among different test cases of the compared test models. In this case, we need to merge this behavior to build an integration test behavior. Furthermore, redundancy may be found among the generated test cases. Test cases represent viewpoints of parts of the system behavior. Different component test models can capture the same viewpoint in their test cases. The generation of integration test cases from these component test models may produce redundant test cases. Thus, we need to compare the generated test cases against each other to remove any redundancy that may exist. In conclusion, we split our generation approach to four processes, as shown in Figure 14, to handle these issues. We devote a separate subsection to elaborate more on each process.

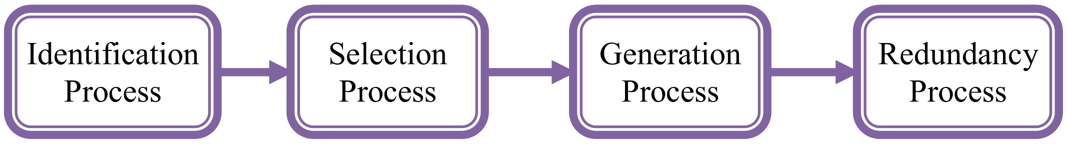


Figure 14. The different processes of the integration test generation approach

4.2.1 Test Object Identification

In order to generate integration test models from component test models, there should be a shared specification between the two component test models that reflects interactions between the corresponding system components. The shared specification belongs to certain test objects that are specified on both test models. Hence, we need to identify these test objects and this behavior prior to the generation of the integration test model. In general, test models are composed of a test architecture and a set of test cases. The test architecture describes test objects, which participate in the targeted test, and the relation among them. The test cases describe the test behavior of these test objects to examine a specific implementation behavior. To compare test models, we have to compare the behavior of each test object to the corresponding test object on the other test model. This task is not a straightforward operation since test objects may play a simple or a complex role in the test scenarios. Three kinds of test objects can be specified in test models: test control, test stub and IUT. The main role of the test control is to drive the tests specified in the test cases and to provide test verdicts. There is usually one test control per a test model. However, a test control can play an optional role by emulating system entities or a system environment that is not realized during the test execution. The optional role makes us uncertain about the real identity of the test control; it can be just the main role or a complex role with embedded behavior of other entities. Hence, we need to analyze the behavior of the test controls to identify their actual roles. On the other hand, test stubs are dummy objects that emulate a system environment or system entities that are not realized during the test execution. Test stubs are optional, and they are typically embedded in the test control. A test stub can emulate one or more of the actual entities. Hence, we need to analyze the behavior of test stubs too. The third test object is implementation under test. It represents different parts of the system depending on the testing level; it can be a system under test, a component under test, etc. However, we are confident that this is the only test object that represents a unique real entity. The first two test objects can represent a single or multiple real entities. In well-formed test models, entities are represented by one test object in each test model. Accordingly, to identify unknown test objects in one component test model, we have to compare them to known test objects of other component test models of the same system. We take into account that these test models may be generated by different testers. Table 1 summarizes the applied comparison pattern among the test objects. While we do compare test controls to each other in test case comparison, which is discussed on a subsequent section, we do not compare them for test object identification. Even when part of the behavior of two test controls is matched, we cannot conclude that these test controls emulate a system entity or system environment; the matched behavior could be a behavior to control the test, i.e.: test setup. The identification process goes through two phases. We analyze the test structure in the first phase and the test behavior in the second phase.

4.2.1.1 Phase I of the Identification Process

In the first phase, the process uses the specification of the test architecture of the two test models to identify test objects as illustrated in Figure 15.a. The process compares test objects of the two test structures using the comparison pattern in Table 1 and matches similar ones. Different methods can be adopted to measure the similarity among test objects. UML stereotypes can be used to define the identity of test objects. In this method, test objects are enriched with UML stereotypes that define the entities, which they represent. Suppose that we have test objects that emulate three real entities, say *x*, *y* and *z*, then we add three UML stereotypes, «*x*», «*y*» and «*z*» respectively, to the test specifications of the corresponding test objects. In this case, the process compares the UML stereotypes of test objects of the two test architectures and identifies similar test objects. The identification process is simple and fast. In addition, this method may eliminate the second phase of the identification process. However, this method requires additional information to be inserted to the component test models. We left-out this setup since one of our research objectives is to follow the standards and bring collaboration among software stakeholders. This method is not a standard methodology and may not be agreed upon by all stakeholders.

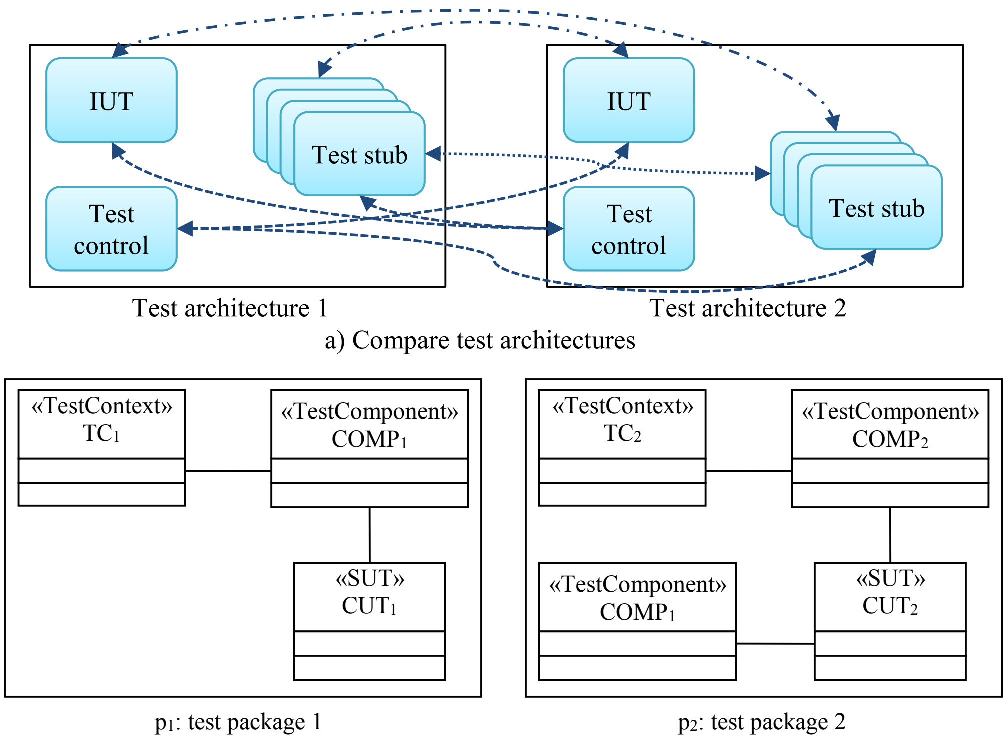


Figure 15. Identification process: phase I

Another method depends on the consistency of name convention among test models [82]. Test objects of component test models should be named according to their corresponding system components. Given that we are designing test models of the same system, names of the system components should be adopted to their corresponding test objects. We adopted this method since it does not impose any extra design regulation on the test models and gives more flexibility to the designer. We illustrate this method using the example given in Figure 15.b. In this example, we compare two test architectures, p_1 and p_2. The process identifies the shared test object $Comp_1$. Test object $Comp_2$ is unidentified in this phase; it could be emulated by the test control TC_1 or just required for the second test model. In this phase, we can identify test objects that correspond to single real entities. For test objects that emulate several real entities, we need to proceed to the next phase.

Table 1. Comparing test objects

Test objects to be identified	Test objects to be compared to		
	Test control	Test stub	IUT
Test control	✘	✓	✓
Test stub	✓	✓	✓

4.2.1.2 Phase II of the Identification Process

In the second phase, we try to identify shared test objects that have not been identified during the first phase using the test behavior. Test behavior is the largest portion of test models. It is composed of a set of test cases and each test case is composed of a set of instances of test objects with a finite behavior. The process locates the instances of unidentified test objects in one test model and compares their behavior to the behavior of the instances in the other test model as illustrated in Figure 16. Three cases are excluded in this comparison. In the first case, there is no comparison between the two test controls since both of them are unknown. In the second case, we do not compare instances of unidentified test object to instances of test objects, which are already specified in the same test case. In the third case, we do not compare unknown instances to instances of test objects that represent test stubs of the IUT of the first test model. The results of comparing any two instances may produce:

- No match,
- Partial match or
- Full match.

In the latter two cases, we can conclude that the test object related to the instance of the second test model emulates exclusively or partially the test object of the first test model regardless of their names.

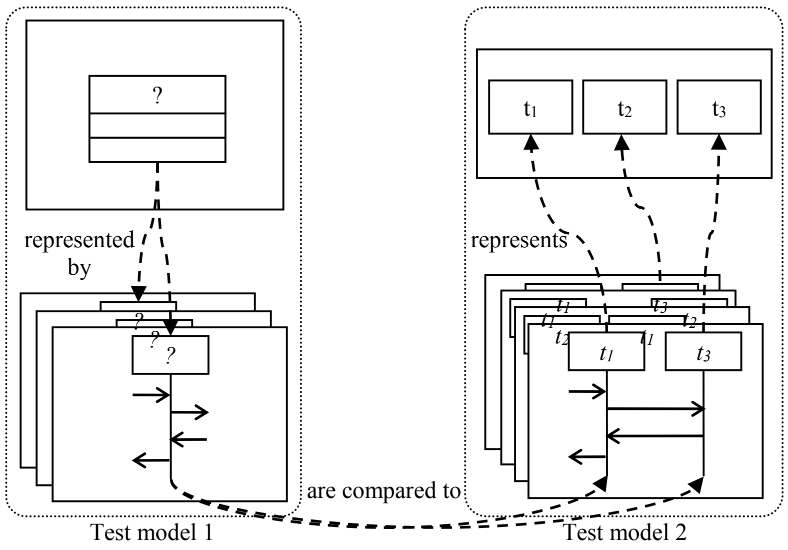


Figure 16. Compare test behaviors

To compare the behavior of two instances, we compare the events located on the lifelines of these two instances. The behavior of two instances may be shared if there are similar events located on both lifelines. Hence, we have to derive our definition of event similarity across two lifelines. We compare events of the same kind according to our classification in Definition 4. The easiest method is to compare event names. Störrle [82] shows the effectiveness of such an approach on UML models. This may be applicable in other fields such as clone-detection, but it may not work well in our case. While we strongly recommend the usage of a consistent naming convention, at least across the same project, test developers may use different naming conventions for different test models. Moreover, modeling tools may generate the same names for different events of different models. Furthermore, test stubs can be embedded in the test control; in this case, name matching is irrelevant. Hence, we use event attributes to define event matching $Match_{msg}$, $Match_{time}$ and $Match_{misc}$ for the case of E_{msg}, E_{time} and E_{misc}, respectively.

Definition 5. (Event Matching)

Let e_1 and e_2 be two events of the same kind from two different instances, then e_1 and e_2 match (and noted $e_1 = e_2$) if and only if:

1. $Match_{msg}(e_1, e_2) = \{ e_1 \in E_{msg}, e_2 \in E_{msg} \mid (e_1.ty = e_2.ty) \wedge (e_1.msg = e_2.msg) \wedge ((e_1.nm = e_2.nm) \vee (((e_1.owner.nm = e_2.owner.nm) \vee (e_1.owner.st \neq SUT) \vee (e_2.owner.st \neq SUT) \wedge ((e_1.oIns.nm = e_2.oIns.nm) \vee (e_1.oIns.st \neq SUT) \vee (e_2.oIns.st \neq SUT)) \}$.
2. $Match_{ime}(e_1, e_2) = \{ e_1 \in E_{time}, e_2 \in E_{time} \mid (e_1.ty = e_2.ty) \wedge (e_1.tm = e_2.tm) \wedge (e_1.pd = e_2.pd) \wedge ((e_1.nm = e_2.nm) \vee (e_1.owner.nm = e_2.owner.nm) \vee (e_1.owner.st \neq SUT) \vee (e_2.owner.st \neq SUT)) \}$.
3. $Match_{misc}(e_1, e_2) = \{ e_1 \in E_{misc}, e_2 \in E_{misc} \mid (e_1.ty = e_2.ty) \wedge (e_1.v = e_2.v) \wedge ((e_1.nm = e_2.nm) \vee (e_1.owner.nm = e_2.owner.nm) \vee (e_1.owner.st \neq SUT) \vee (e_2.owner.st \neq SUT)) \}$.

To proceed to the next process, the identification process should detect at least one test object that is specified in both test models. We call such test objects *shared test objects*. The existence of shared test objects reflects high probability of the existence of interactions among the integrated components. In case of no shared test objects found, we conclude that there is no interaction specified between the given test models and we stop the generation process for the current integration iteration. We believe that this issue can happen due to the use of an incorrect integration strategy when the two components do not have direct interface between them, or it could happen due to under-specified test models when test models do not cover all component interfaces. The tester should fix this issue to proceed with the generation.

4.2.2 Component Test Case Selection

The selection process searches the test cases of the given test models to locate interactions between the integrated system components. These interactions usually occur through the behavior of the shared test objects, which have been identified by the previous process. The interactions, between the integrated system components, can be direct or indirect through test stubs of other system components that have not been integrated. The existence of such interactions among test cases permits us to select them to be reused to generate integration test cases. In this process, we are looking for two patterns: single test cases or two complement test cases.

4.2.2.1 First Selection Pattern: Complete Integration Test Cases

For the first pattern, we search for individual test cases in both test models that contain an implicit/explicit emulation of the system component of the other test model. We call test cases of such pattern *complete integration test cases*. Test cases t_1 and t_3 in Figure 17 present explicit and implicit emulation of this pattern respectively. In test case t_1, a test stub of the system component $COMP_4$ is specified, which is identified during phase I of the identification process. In test case t_2, there is no explicit presentation of the system component $COMP_3$ but the test control TC_4 is embedded with the behavior of the system component $COMP_3$. Phase II of the identification process detects such behavior by comparing test cases t_2 and t_3. While both system components are specified in the test cases, we have to examine their behavior to ensure the existence of an interaction between the integrated system components. There must be at least one message exchanged directly/indirectly between the two system components. The two components have a direct interaction by exchanging messages (m_2, m_3) in the test case t_1, and they also have indirect interactions by exchanging messages (m_7, m_8) and (m_9, m_{10}) through $COMP_5$ in the test case t_3. We discuss our interaction-detection technique in the subsequent subsection, 4.2.2.3.

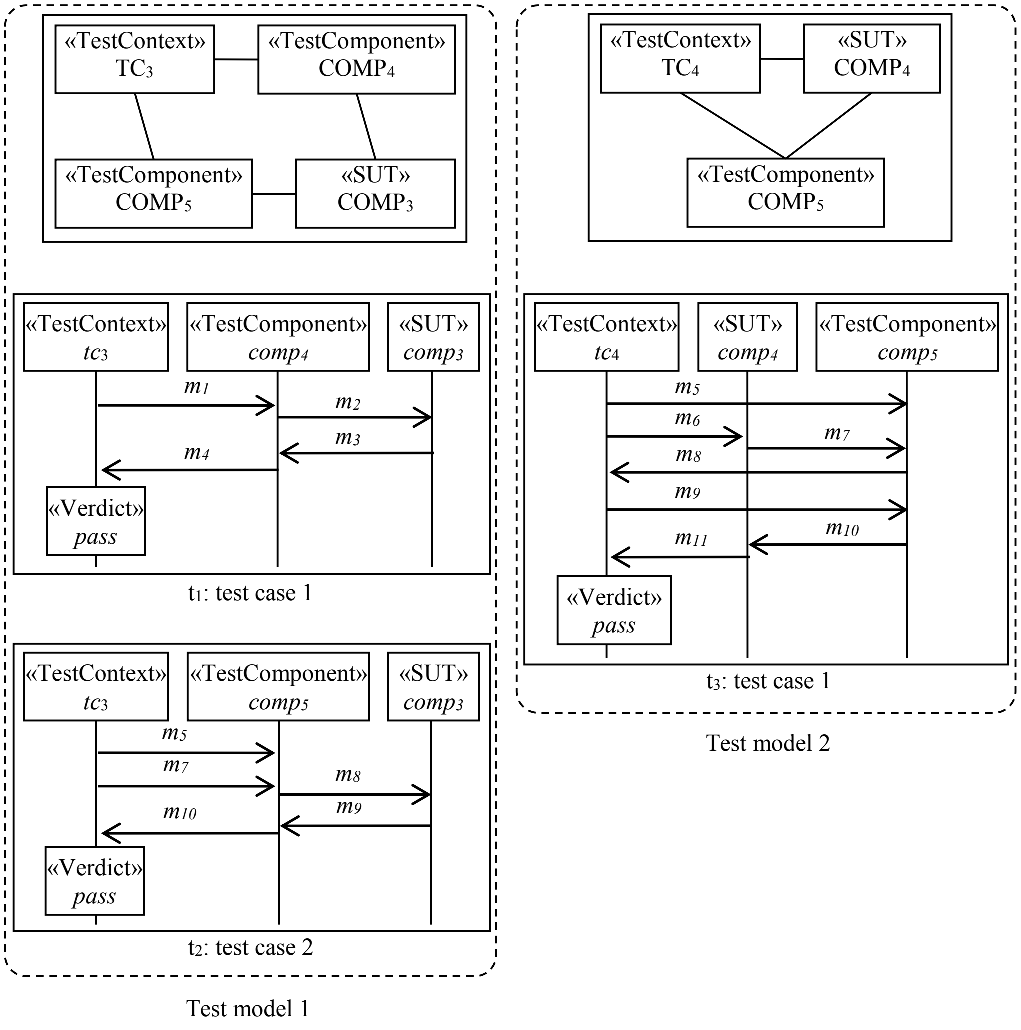


Figure 17. Selection patterns

4.2.2.2 Second Selection Pattern: Complement Integration Test Cases

The second pattern involves two test cases, one test case from each test model. The two test cases must share at least one test object in their specifications. We call such pairs of test cases *complement integration test cases*. This pattern can be illustrated by test cases t_2 and t_3 in Figure 17. The test object $COMP_5$ is specified in the two test cases. To select such test cases for generating integration test cases, we apply the interaction-detection technique discussed in Subsection 4.2.2.3. The two components have indirect interactions by exchanging messages (m_7, m_8) and (m_9, m_{10}) through $COMP_5$.

4.2.2.3 Event Dependency Tree (EDT)

The main objective of the integration testing is to check the inter-connectivity among the integrated components. In order to select component test cases for generating integration test cases, we have to guarantee that such component test cases specify interactions between the integrated system components. To prove the existence of such an interaction, we have to examine the execution traces specified by the participated test cases. We have developed an interaction detection technique by building what we called the Event Dependency Tree. The Event Dependency Tree presents the dependency order among the events of the participated test cases, as illustrated in Figure 18. Each node represents an event and each edge represents an order relation between the linked events. For readability reasons, we construct the event name from the message name followed by the first letter of the event type, e.g.: m_{1s} is the sending event of message m_1. The UTP verdict is given by *ver*. The construction of the EDT goes through two or three steps depending on the participating test cases, one or two test cases respectively.

In the first step, we create the nodes from the events set E in Definition 3 for each test case. Figure 18.a illustrates step 1 for the test case t_1 in Figure 17. In the second step, we create the edges using the relation R in Definition 3. Figure 18.b illustrates step 2 for the test case t_1 in Figure 17, while Figure 19.a illustrates step 2 for test cases t_2 and t_3 in Figure 17. The EDT construction is completed for the first selection pattern, *complete integration test cases*. However, we perform step 3 when there are two participating test cases, selected in the second selection pattern. In this step, we use Definition 5, for event similarity, and the results of the identification process to link the two test cases. We remove the duplication of similar nodes and redirect edges of the deleted nodes to their corresponding node if they do not already exist. To illustrate, step 3 is applied on the two graphs in Figure 19.a to produce the final EDT in Figure 19.b. In this example, the events of test case t_2 are completely captured in test case t_3. Two edges, (m_{5s}, m_{7s}) and (m_{10r}, ver), are redirected. These edges are related to the emulation of the test control tc_3 in t_2 to the system component $comp_4$ in t_3. The two pairs of relation exist implicitly in t_3.

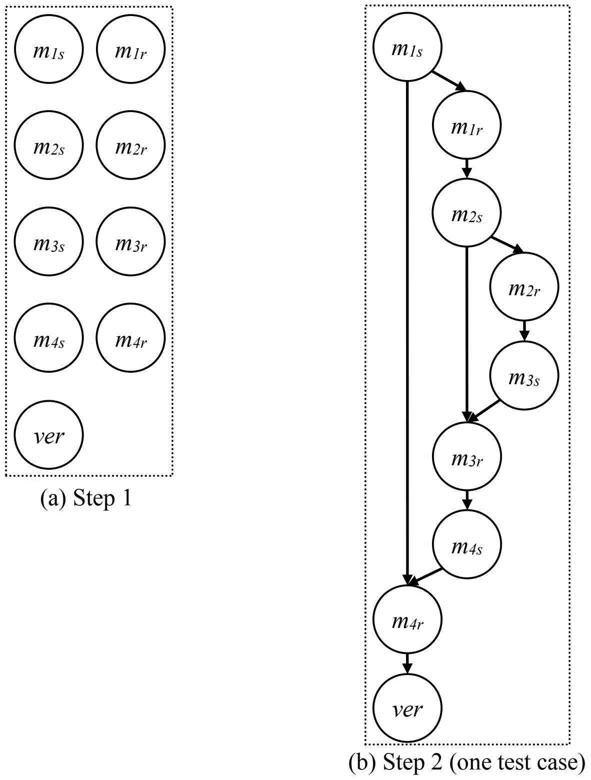


Figure 18. EDT Construction (1/2)

The EDT is used to detect interactions between the integrated components. We traverse the EDT to locate a node for a sending event of one of the integrated components. From that node, we search the branched paths for a reception event of the other integrated component. On the discovery of such a path between a sending and a receiving event, we stop the detection process and confirm the existence of an interaction among the integrated components in this test scenario. The test cases are selected to generate integration test cases, which will be covered in the following section. In the case of search failure, we resume our search for another sending event for one of the integrated components and perform the same process. The test cases are excluded from the selection if there is no path between any pair of sending and receiving events corresponding to the integrated components. Test cases may be reused again with different test cases to detect such interactions. Figure 20 illustrates the detection method using EDTs shown in Figure 18.b and Figure 19.b. There are two interactions in both EDTs. However, we are satisfied with just one interaction. In Figure

20.a, the detection method will always detect the first interaction (m_{2s}, m_{2r}) since both interactions are on the same path, and as you may have noticed the interaction is direct. However, the detection method for the EDT in Figure 20.b depends on the logic of the search method for selecting one of the two interactions since they reside on two different paths. The first interaction is reached through two paths: (m_{5s}, m_{7s}) and (m_{5s}, m_{6s}, m_{6r}, m_{7s}), and the second interaction is reached through one path: (m_{5s}, m_{6s}, m_{8r}, m_{9s}). As you may have noticed, the two interactions are indirect and go through component *comp*$_5$.

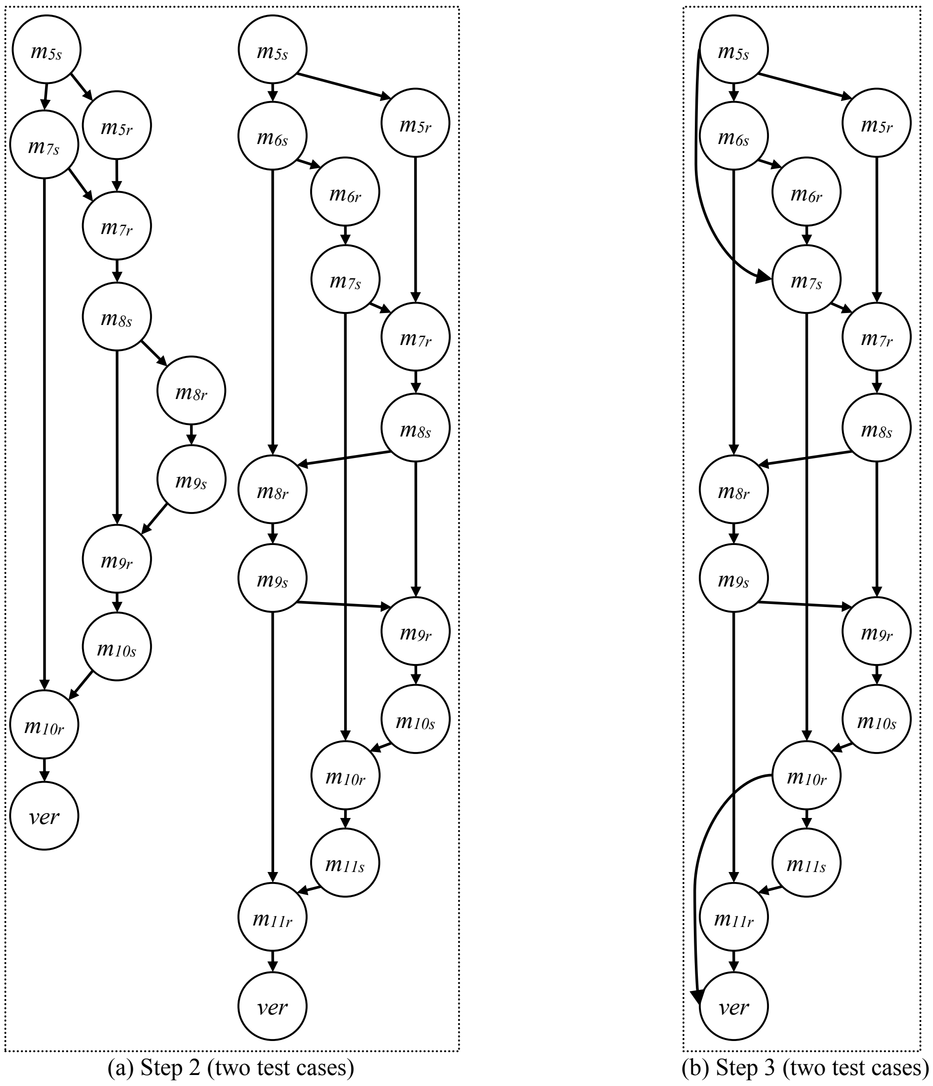


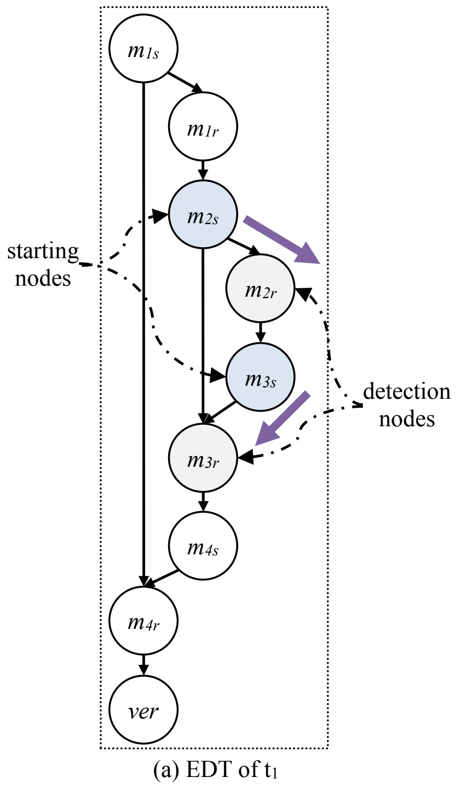
Figure 19. EDT Construction (2/2)

4.2.3 Test Model Generation

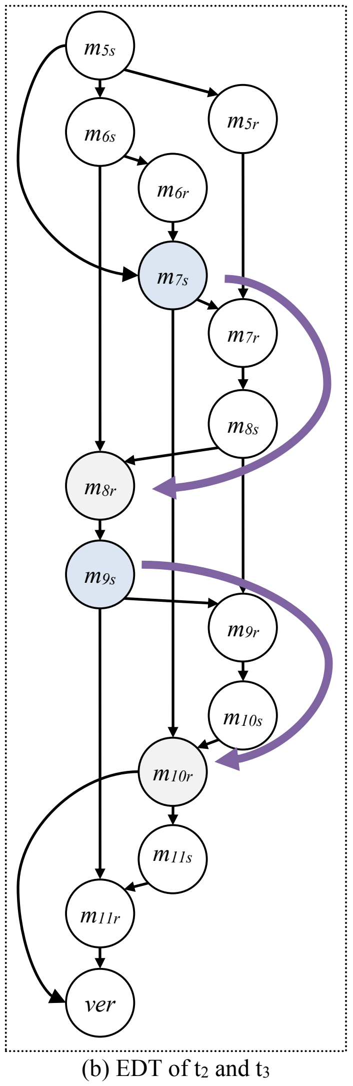
The process generates the integration test model in two stages. In the first stage, it generates the test behavior, and it generates the test architecture in the second stage.

4.2.3.1 Stage I of the Test Generation

The test behavior is generated from the selected test cases by the selection process. We can classify the selected test cases into two groups: complete integration test cases and complement integration test cases. The complete integration test cases are self-contained component test cases that include the integrated components in their specification. One of the integrated components is specified as a test stub. The process generates integration test cases from such component test cases by replacing the instances of the test stubs with the instances of their corresponding system components. In this group, the test behavior is not modified.



(a) EDT of t_1



(b) EDT of t_2 and t_3

Figure 20. Interaction detection using EDT

Figure 21.a is an example of generated integration test case from component test case t_1 shown in Figure 17. In the second group, the complement integration test cases are pairs of component test cases. Each pair is composed of two test cases, one from each test model. The test scenario of the two test cases represents an integration test scenario. Thus, the process merges the two component test cases to generate an integration test case. This step brings up the theoretical issue of merging test cases, which we discuss in the following subsection. We assume that the given test cases are completely different. The process searches the two test cases for shared elements. Test cases are specified using UML sequence diagram. The process focuses on particular elements of the UML sequence diagram that are related to our domain. The most important elements are lifelines, messages and end_messages. Furthermore, test cases are finite models, which makes them manageable.

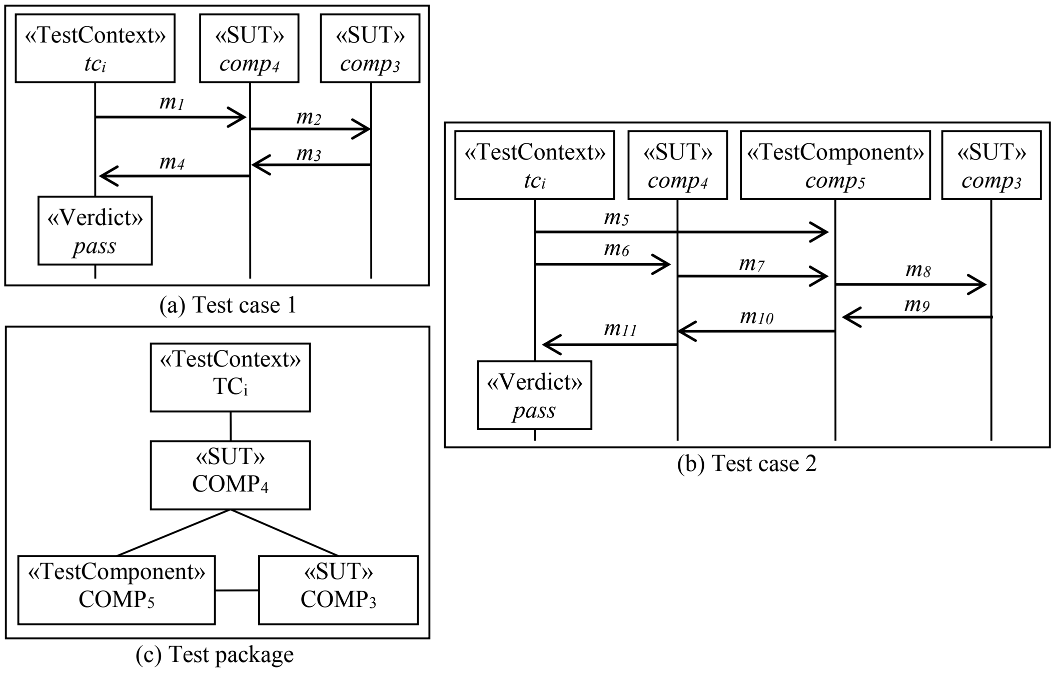


Figure 21. Generated test model

The process uses Definition 5 to detect a similar test behavior. The events of the two test controls are combined to build the behavior of the integration test control. Events, which are partially emulated by the test control or test stubs, are moved to their corresponding system component when they are added to the test case. We have at the same time to maintain the specification of both test cases; e.g.: if one test case specifies *n* instances of an entity and the other test case specifies *m* instances of the same entity, then the approach merges *min(n, m)* instances that have shared behavior. The test case in Figure 21.b is generated from the merging of the two component test cases, t_2 and t_3, in Figure 17.

4.2.3.2 Stage II of the Test Generation

Upon the completion of generating the test behavior, the process builds the test architecture. The test architecture is created from the specification of the test behavior. The given test architectures of the component test models are used to relate test objects to their external models, if found. We focus on the UTP test package in this dissertation. Table 2 summarizes the important mapping elements to generate test architecture from test behavior. The process traverses the generated test cases. It goes through the elements of each test case, and creates the equivalent elements in the test architecture. Internal references between elements of the test behavior and the corresponding elements of the test architecture are built. After that, the process compares the generated test objects, UML classes, to their corresponding test objects in the given component test cases. In case any test object has a reference to an external model, the process updates the corresponding generated test object with the same reference. The most important test object is the SUT, which is always externally referenced. Finally, the process plugs a reference to the UTP to enable its stereotypes in the generated test model.

Table 2. Mapping test behavior to test structure

Test Behavior	Test Architecture
UML Lifeline	UML Class
UML Message	UML Association
UML Sequence Diagram	UML Operation

4.2.3.3 Merging Test Cases

A test case captures only a portion of the IUT behavior, with a partial view. Some insignificant details may be omitted when designing test cases. Integrating two partial views and ordering the events is a challenging problem as discussed thoroughly in [88, 89, 100]. Different integration operators were proposed such as alternative, parallel, sequential and merging operator; and several approaches presented to integrate various behavioral models [77, 80, 88]. In this work, we are interested in the merging operator. We generate integration test cases by merging component test cases that share test objects. We call such component test cases complement integration test cases. Definitions 3 to 5 are used to derive our merging expression. Furthermore, the process generates the behavior of the integration test control by merging the behavior of the two component test controls; we name it tc_i. In order to merge the two test cases, we have to identify the shared events, which are located on the shared test objects' lifelines. The process uses such shared events as coordinate points in the merging process.

Definition 6. (Shared Events)

Let E_1 and E_2 be two sets of events of the two component test cases. Using Definition 5, the shared events are defined as

$$\text{se} = \{(e_1,e_2): e_1 \in E_1 \text{ and } e_2 \in E_2 \mid e_1 = e_2\}$$

Definition 7. (Merging Test Cases)

Let $t_1 = (I_1, E_1, R_1)$ and $t_2 = (I_2, E_2, R_2)$ be component test cases and se_{12} be the corresponding shared events. Then, the generated integration test case is produced by

$$\begin{aligned} t_{12} &= t_1 + t_2 \\ &= (g(I_1)\ U\ g(I_2),\ f(E_1)\ U\ f(E_2),\ f(R_1)\ U f(R_2)) \end{aligned}$$

Where

$g(i) : \{i: i \in I,\ \forall i \text{ if } i.st = TestContext, \text{ then } i = tc_i\}$. The function transforms component test controls to the integration test control.

$f(e) : \{e: e \in E \text{ and } (e_1, e_2) \in se, \text{ if } e = e_1 \text{ then } e = e_2\}$. The function replaces the first element of a pair in the shared events to the second element so that the approach eliminates the duplication of identical events. In other words, it relocates emulated events to their corresponding real components.

To illustrate our merging expression, we use the given component test cases t_2 and t_3 shown in Figure 17. Events are named by their corresponding messages suffixed with the first letter of the event type. UTP verdicts are named *ver*. Using Definition 3, we can express the two test cases as

t_2 = ({tc3,comp5,comp3}, {m5s,m7s,m10r,ver,m5r,m7r,m8s,m9r,m10s,m8r,m9s}, {(m5s,m7s), (m7s,m10r),(m10r,ver),(m5r,m7r),(m7r,m8s),(m8s,m9r),(m9r,m10s),(m8r,m9s),(m5s,m5r), (m7s,m7r),(m8s,m8r),(m9s,m9r),(m10s,m10r),(m5s,m10r),(m5s,m7r),(m7s,ver),(m5r,m8s), (m7r,m9r),(m7r,m8r),(m8s,m10s),(m9r,m10r),(m8r,m9r),(m7s,m8s),(m8s,m9s),(m9s,m10s), (m10s,ver),(m5s,ver),(m5s,m8s),(m5r,m9r),(m5r,m8r),(m7r,m10s),(m7r,m9s),(m8s,m10r),(m9r,ver), (m8r,m10s),(m7s,m9r),(m7s,m8r),(m9s,m10r),(m5s,m9r),(m5s,m8r),(m5r,m10s),(m5r,m9s), (m7r,m10r),(m8s,ver),(m8r,m10r),(m7s,m10s),(m7s,m9s),(m9s,ver),(m5s,m10s),(m5s,m9s), (m5r,m10r),(m7r,ver),(m8r,ver),(m5r,ver) })

$t_3 = (\{tc4,comp4,comp5\}, \{m5s,m6s,m8r,m9s,m11r,ver,m6r,m7s,m10r,m11s,m5r,m7r, m8s,m9r,m10s\}, \{$
$(m5s,m6s),(m6s,m8r),(m8r,m9s),(m9s,m11r),(m11r,ver),(m6r,m7s),(m7s,m10r),(m10r,m11s),$
$(m5r,m7r),(m7r,m8s),(m8s,m9r),(m9r,m10s),(m5s,m8r),(m6s,m9s),(m8r,m11r),(m9s,ver),$
$(m6r,m10r),(m7s,m11s),(m5r,m8s),(m7r,m9r),(m8s,m10s),(m5s,m9s),(m6s,m11r),(m8r,ver),$
$(m6r,m11s),(m5r,m9r), (m7r,m10s), (m5s,m11r),(m6s,ver),(m5r,m10s),(m5s,ver) \})$

The generated integration test control is named tc_i. The shared events relation is constructed using Definition 6

$se = \{(m5s,m5s),(m5r,m5r),(m7s,m7s),(m7r,m7r),(m8s,m8s),(m8r,m8r),(m9s,m9s),(m9r,m9r),$
$(m10s,m10s),(m10r,m10r),(ver,ver)\}$

The order of the events in each pair is very important if their owned instances are different: test control events are always put as the first element of the pair (domain) and SUT events are always put as the second element of the pair (range). By this arrangement, the process relocates emulated events to their corresponding test objects. The next step it to apply the transformation functions, *g()* and *f()*.

$g(I_2) = \{\underline{tci},comp5,comp3\}$

$g(I_3) = \{\underline{tci},comp4,comp5\}$

$f(E_2) = \{\underline{m5s},\underline{m7s},\underline{m10r},\underline{ver},\underline{m5r},\underline{m7r},\underline{m8s},\underline{m9r},\underline{m10s},m8r,m9s\}$

$f(E_3) = \{m5s,m6s,\underline{m8r},\underline{m9s},m11r,ver,m6r,m7s,m10r,m11s,m5r,m7r,m8s,m9r,m10s\}$

$f(R_2) = \{ (\underline{m5s},\underline{m7s}),(\underline{m7s},\underline{m10r}),(\underline{m10r},\underline{ver}),(\underline{m5r},\underline{m7r}),(\underline{m7r},\underline{m8s}),(\underline{m8s},\underline{m9r}),$
$(\underline{m9r},\underline{m10s}),(m8r,m9s),(\underline{m5s},\underline{m5r}),(\underline{m7s},\underline{m7r}),(\underline{m8s},m8r),(m9s,\underline{m9r}),(\underline{m10s},\underline{m10r}),$
$(\underline{m5s},\underline{m10r}),(\underline{m5s},\underline{m7r}),(\underline{m7s},\underline{ver}),(\underline{m5r},\underline{m8s}),(\underline{m7r},\underline{m9r}),(\underline{m7r},m8r),(\underline{m8s},\underline{m10s}),$
$(\underline{m9r},\underline{m10r}),(m8r,\underline{m9r}),(\underline{m7s},\underline{m8s}),(\underline{m8s},m9s),(m9s,\underline{m10s}),(\underline{m10s},\underline{ver}),(\underline{m5s},\underline{ver}),$
$(\underline{m5s},\underline{m8s}),(\underline{m5r},\underline{m9r}),(\underline{m5r},m8r),(\underline{m7r},\underline{m10s}),(\underline{m7r},m9s),(\underline{m8s},\underline{m10r}),(\underline{m9r},\underline{ver}),$
$(m8r,\underline{m10s}),(\underline{m7s},\underline{m9r}),(\underline{m7s},m8r),(m9s,\underline{m10r}),(\underline{m5s},\underline{m9r}),(\underline{m5s},m8r),(\underline{m5r},\underline{m10s}),$
$(\underline{m5r},m9s),(\underline{m7r},\underline{m10r}),(\underline{m8s},\underline{ver}),(m8r,\underline{m10r}),(\underline{m7s},\underline{m10s}),(\underline{m7s},m9s),(m9s,\underline{ver}), (\underline{m5s},\underline{m10s}),$
$(\underline{m5s},m9s),(\underline{m5r},\underline{m10r}),(\underline{m7r},\underline{ver}),(m8r,\underline{ver}), (\underline{m5r},\underline{ver}) \}$

$f(R_3) = \{(m5s,m6s),(m6s,\underline{m8r}),(\underline{m8r},\underline{m9s}),(\underline{m9s},m11r),(m11r,ver),(m6r,m7s),$
$(m7s,m10r),(m10r,m11s),(m5r,m7r),(m7r,m8s),(m8s,m9r),(m9r,m10s),(m5s,\underline{m8r}),$
$(m6s,\underline{m9s}),(\underline{m8r},m11r),(\underline{m9s},ver),(m6r,m10r),(m7s,m11s),(m5r,m8s),(m7r,m9r),$
$(m8s,m10s),(m5s,\underline{m9s}),(m6s,m11r),(\underline{m8r},ver),(m6r,m11s),(m5r,m9r),(m7r,m10s),$
$(m5s,m11r),(m6s,ver),(m5r,m10s),(m5s,ver) \}$

We have underlined the transformed events. The final step is to apply the union operator on the transformed sets and relations to generate the integration test case that is equivalent to the one shown in Figure 21.b.

$t_{23} = t_2 + t_3 =$ ({tci,comp3,comp4,comp5}, {m5r,m5s,m6r,m6s,m7r,m7s,m8r,m8s,m9r, m9s,m10r,m10s,m11r,m11s,ver}, { (m5s,m7s),(m7s,m10r),(m10r,ver),(m5r,m7r), (m7r,m8s),(m8s,m9r),(m9r,m10s),(m8r,m9s),(m5s,m5r),(m7s,m7r),(m8s,m8r), (m9s,m9r),(m10s,m10r),(m5s,m10r),(m5s,m7r),(m7s,ver),(m5r,m8s),(m7r,m9r), (m7r,m8r),(m8s,m10s),(m9r,m10r),(m8r,m9r),(m7s,m8s),(m8s,m9s),(m9s,m10s), (m10s,ver),(m5s,ver),(m5s,m8s),(m5r,m9r),(m5r,m8r),(m7r,m10s),(m7r,m9s), (m8s,m10r),(m9r,ver),(m8r,m10s),(m7s,m9r),(m7s,m8r),(m9s,m10r),(m5s,m9r), (m5s,m8r),(m5r,m10s),(m5r,m9s),(m7r,m10r),(m8s,ver),(m8r,m10r),(m7s,m10s), (m7s,m9s),(m9s,ver),(m5s,m10s),(m5s,m9s),(m5r,m10r),(m7r,ver),(m8r,ver), (m5r,ver),(m5s,m6s),(m6s,m8r),(m9s,m11r),(m11r,ver),(m6r,m7s),(m10r,m11s), (m6s,m9s),(m8r,m11r),(m6r,m10r),(m7s,m11s),(m6s,m11r),(m6r,m11s), (m5s,m11r),(m6s,ver),(m5s,m11s),(m8s,m11r),(m10s,m11s),(m7r,m11r), (m9r,m11s),(m5r,m11r),(m8s,m11s),(m7s,m11r),(m9s,m11s),(m7r,m11s), (m8r,m11s),(m5r,m11s),(m6s,m9r),(m6s,m10s),(m6s,m10r),(m6r,m7r),(m6r,ver), (m6r,m8s),(m6r,m9r),(m6r,m8r),(m6r,m10s), (m6r,m9s),(m6s,m11s),(m6r,m11r) })

4.2.3.3.1 Validating the Merging Process

The merging process relocates events from the lifeline of one instance to the lifeline of another instance. In order to validate the correctness of the implementation of such a process, the behavior of the generated test case should be identical to the overall behavior of the input test cases. We propose an on-the-fly validation method based on the EDT. To apply this method, we have to save the EDT, which has been constructed in the selection process. In this method, we construct an EDT for the generated test case then compare it to the previous one. We have to consider the shared test objects that were recognized during the identification process since some events of the first EDT are associated to deleted instances. The two EDTs should be identical; otherwise, the implementation of the merging process should be inspected to fix detected bugs.

4.2.4 Test Case Redundancy Removal

The generation approach is applied on different test models during every integration iteration as discussed in Section 4.1. First, the approach takes the component test model of the currently integrated component and the latest generated integration test model to generate an integration test model for the current integration iteration. Then, it takes the same component test model with one of the carried-on component test models from previous iterations to generate additional integration test cases, and so on. The number of the carried-on component test models, n, at iteration r can be calculated using the following equation:

$n = r \text{ for } r > 1.$

Hence, the approach is executed $r+1$ times at iteration r. The approach generates a set of integration test cases at each execution. There may be a redundancy among these sets of test cases. Therefore, we should investigate such sets to remove any redundancy. The redundancy process maps the currently generated test cases to the existing test cases. Test cases, whose specifications are completely included in the specifications of existing test cases, are removed. Using Definition 3, we define our test case inclusion.

Definition 8. (Integration test case inclusion)

Let $T_1 = (I_1, E_1, R_1)$ be an integration test case and $T_2 = (I_2, E_2, R_2)$ be another integration test case, then $T_1 \subseteq T_2$ if and only if the following conditions are satisfied:

1. $I_1 \subseteq I_2$
2. $E_1 \subseteq E_2$
3. $R_1 \subseteq R_2$

The first condition states that the instances specified in the first test case must be all specified in the second test case. The second condition states that the events specified in the first test case must be all specified in the second test case. The third condition checks that the order relation among the events of the first test case is respected in the second test case specification. The first test case, T_1, is removed from the generated test model only if the three conditions are satisfied.

Furthermore, redundancy could be produced within a single execution. This kind of redundancy is caused by the selection of the same test cases twice by the selection process. The selection process searches for two patterns of interaction between the integrated components as discussed in Section 4.2.2. The same component test case could be selected twice, once for each pattern. For the first pattern, an integration test case is generated by updating the component test case, while the component test case is combined with another component test case to generate an integration test case in the second pattern. Hence, the generated test case of the first pattern is identical or included in the generated test case of the second pattern. For example, the component test case t_3, Figure 17, can be selected twice. It can be selected for the first pattern since the test control emulated the CUT $comp_3$. The integration test case is generated by adding an instance for the second CUT $comp_3$ and relocating the events m_{8r} and m_{9s} from the test control to the CUT $comp_3$. The generation of the other test case is illustrated with more details in Sections 4.2.2 & 4.2.3

and shown in Figure 21. Consequently, the two generated test cases are identical. This kind of redundancy can be avoided by detecting it during the selection process and removing such test cases from the selection list of the first pattern.

4.2.5 Selective vs Cumulative Integration

During the development of the generation approach, we studied the effect of the integration strategy on our approach. System components are integrated using different strategies; some of them are well-known, such as top-down, bottom-up, big-bang and ad-hoc. The generated test models for the same set of system components should not depend on their integration strategy. In other words, if we have three components, *A*, *B* and *C*, then the generated test model should be consistent regardless of the different integration strategies that may be taken: *(A+B)+C*, *(A+C)+B* or *(B+C)+A*. Of course, the intermediate test models would be different since the integrated components are different: *(A+B), (A+C) or (B+C)*. However, due to the sequential execution of the integration process, important test information may be lost, which leads to the production of incomplete test models. To explain, let us take the system shown in Figure 22 and integrate its components using the following integration strategy: *((A+B) + C) + D*. Let us focus on the interface between *A* and *D*. In each integration iteration, the generation approach goes through a refinement process, which refines the component test models by focusing on certain interfaces that link the integrated component to the sub-system. In the first iteration, the approach focuses on interface *AB* and ignores the others. Therefore, the generated integration test model *(A+B)* would carry test information regarding *AB*; it may carry extra information regarding *AD* and/or *BC* but it highly depends on the given test cases. In the second iteration, the approach should examine the interface *BC* using the component test model of component *C* and the previously generated integration test model *(A+B)*. Nevertheless, the lastly generated test model *(A+B)* may not carry any information regarding the interface *BC*. The approach focuses on the test cases that specify interactions through the interface *BC*. Hence, there may be no integration test model for this iteration. In the last iteration *((A+B)+C)+D*, there is high probability of losing all information about the interface *AD* in the lastly generated test model *((A+B)+C)*. The approach may find test cases that cover the interface *DC* since the component C was the last to be integrated. However, test cases covering interface *AD* may be excluded during the previous iterations. Consequently, the integration testing is finished without examining the interface *AD*. Hence, we have to find a way to carry the information of component test models to the subsequent integration iterations.

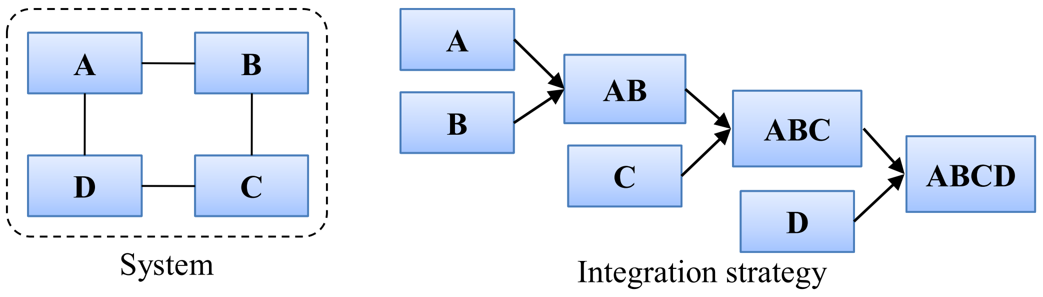


Figure 22. Integration strategy

We worked on two techniques, as shown in Figure 23, to carry test information of component test models to subsequent integration iterations: selective and cumulative. The two techniques apply the same set of the generation processes that we have discussed but differ in the order of applying these processes as shown in Figure 24.

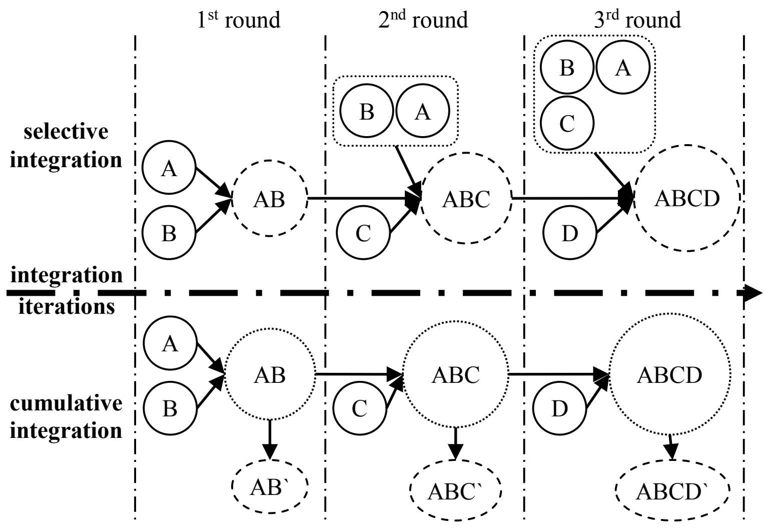


Figure 23. Cumulative & selective integration

The selective technique carries the component test models along with the generated integration test model to the subsequent integration iterations. The technique does not change the order of the processes of the approach. In each iteration, the approach is applied several times to generate the current integration test model. First, it uses the former integration test model, which is generated in the previous iteration, and the component test model of the currently integrated system component to generate the integration test model for the current iteration. Next, it uses the carried-on component test models of previously integrated components and the component test model of the currently integrated component to generate additional test cases. The generated integration test model and the component test models of the integrated components, including the currently integrated component, are carried to the subsequent integration iteration.

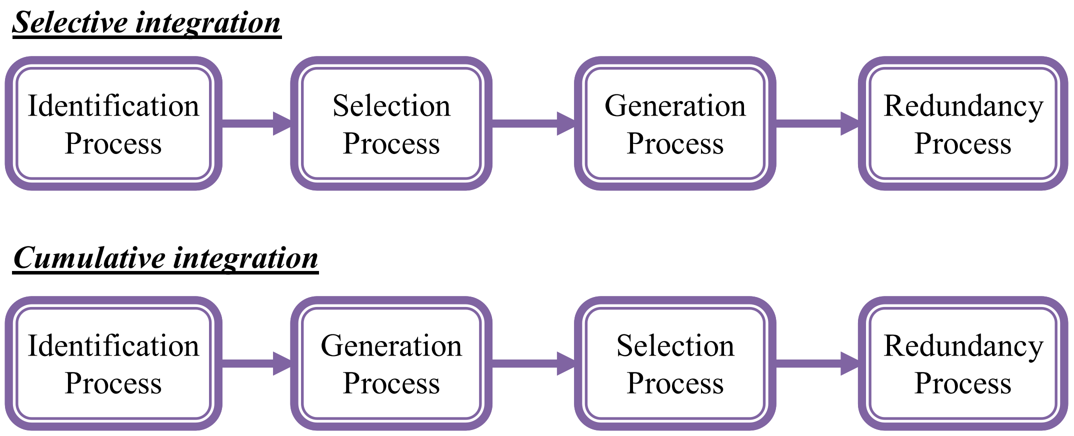


Figure 24. The order of the test generation processes

In the cumulative technique, we reorder the middle generation processes, the selection and the generation, of our generation approach to generate test cases, then to select the ones that involve interactions between the integrated components as shown in Figure 24. The approach generates test cases from the given component test cases without any restriction or filtration. The generated test cases are produced by merging the test cases of the two test models; i.e.: if we have *m* test cases in one test model and *n* test cases in the second test model then we generate $(m \times n)$ test cases. Some of the generated test cases do not reflect any interactions between the integrated components. Furthermore, the approach may merge test cases that do not have any shared behavior. The generated test cases are submitted to the selection process to select integration test cases that reflect interactions between the integrated components; these integration test cases are exercised on the integrated system. The complete set of generated test cases is carried-on to the next integration iteration. In this technique, we have reserved the complete information carried by the component test models in one test model.

We applied both techniques on our case studies. Subsequently, we observed that the cumulative integration may generate invalid integration test cases. The invalid test cases are generated from merging component test cases that do not hold interactions between the integrated components during previous integration iterations and are carried on to the current integration iteration. We illustrate this issue by applying the two techniques on the system specified in Figure 25. The system is composed of three components, *C1*, *C2* and *C3*, and provides three services, *A*, *B* and *C*. Messages are named after their corresponding system services. In this example, we focus on the test behavior and omit the test architecture. The test behavior is given in Figure 26 for the three components. In the following subsections, we apply the two techniques on the system components using the same integration strategy *(C1 + C2) + C3*.

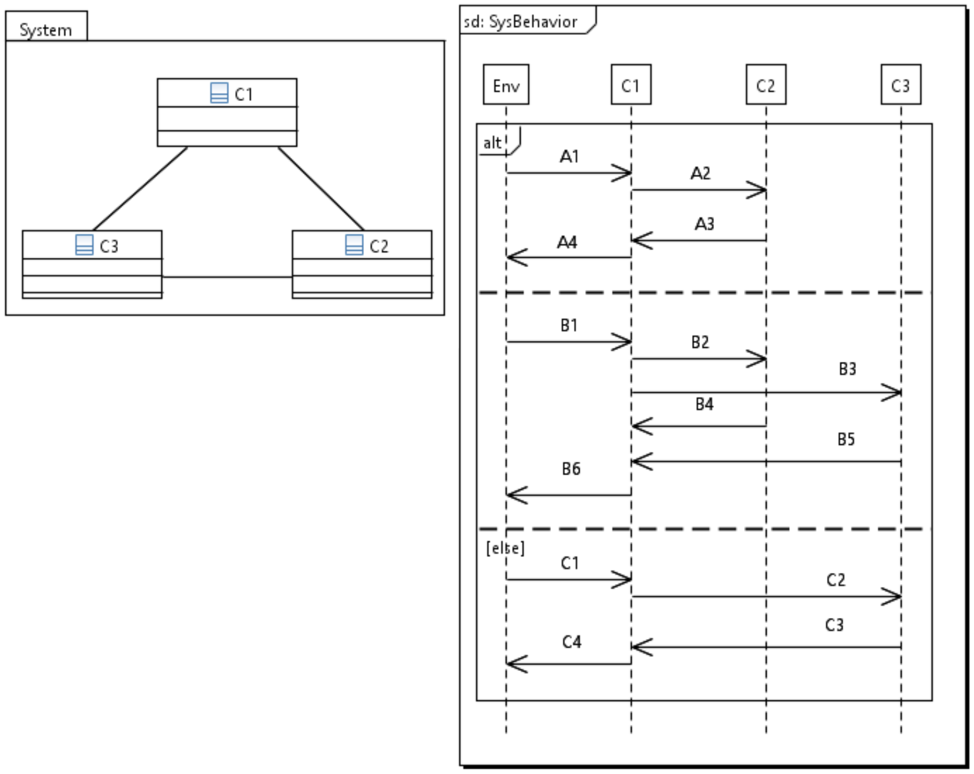


Figure 25. System specification

4.2.5.1 Selective Integration

In the first iteration of the selective integration, the approach examines the two sets of component test cases and selects the ones that capture interactions between the two CUTs, *C1* and *C2*. The results of this process are presented in Table 3. The approach selects the two component test cases that need to be merged to generate two integration test cases.

Table 3. Selective Integration: selected test cases in iteration 1

	C1 test cases	**C2 test cases**	**Exchanged messages**
1	C1_testCase1	C2_testCase1	A2, A3
2	C1_testCase2	C2_testCase2	B2, B4

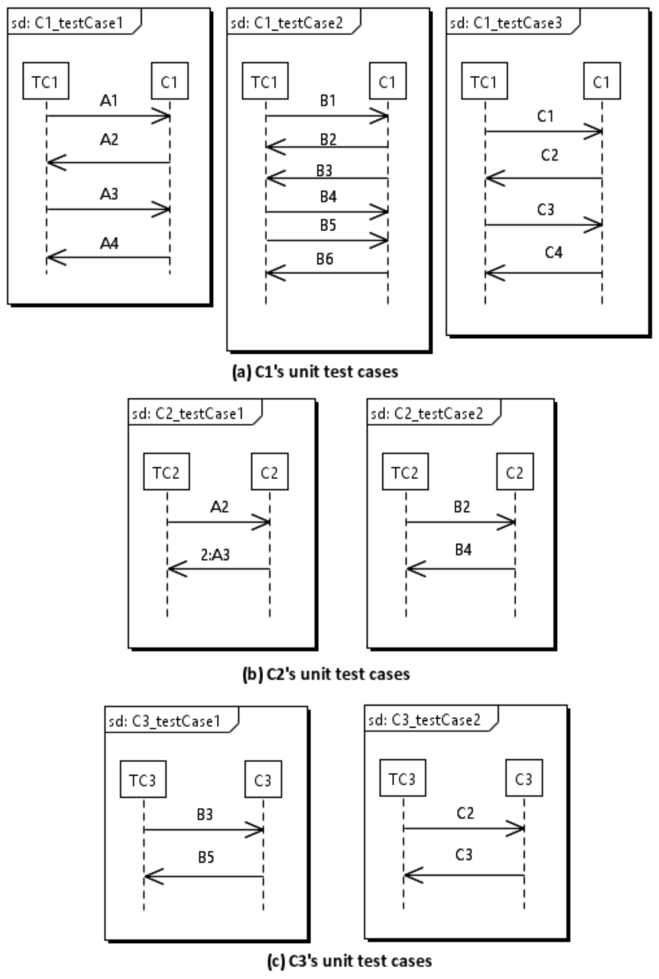


Figure 26. Component test cases

Figure 27 presents the generated integration test cases in iteration 1.

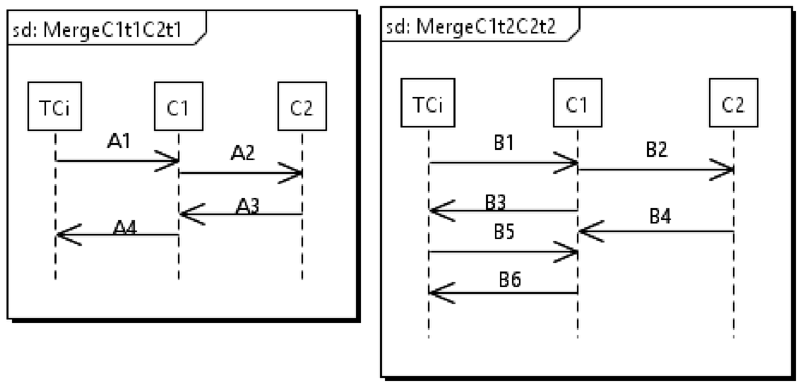


Figure 27. Selective integration: iteration 1 generated test cases

In the second iteration, the generated test cases are examined against the component test cases of the component *C3*. In addition, the approach examines the component test cases of the component *C3* against the component test cases of the integrated components of the sub-system, *C1* and *C2*.

Test cases, which include interactions between *C3* and the sub-system, are selected. Table 4 shows the selected test cases from the test models. The approach selected one test case from the generated test cases to be merged with the first *C3* component test case, and selected one component test case from the *C1* test model to be merged with the second *C3* test case.

Table 4. Selective integration: selected test cases in iteration 2

	C1C2 test cases	C3 test cases	Exchanged messages
1	MergeC1t2C2t2	C3_testCase1	B3, B5
2	C1_testCase3	C3_testCase2	C2, C3

Figure 28 presents the generated integration test cases on iteration 2.

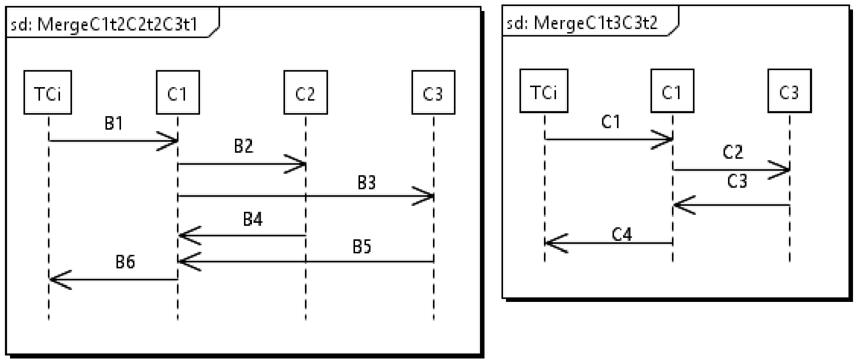


Figure 28. Selective integration: iteration 2 generated test cases

4.2.5.2 Cumulative Integration

In the first iteration, the approach merges the three component test cases of *C1* test model with the two component test cases of the *C2* test model. Figure 29 presents the generated test cases. Next, the approach examines the generated test cases for the existence of interactions between the two CUTs, *C1* and *C2*. Two test cases, *MergeC1t1C2t1* and *MergeC1t2C2t2*, are selected from the generated test cases to be exercised on the sub-system in this integration iteration. For the second iteration, the six generated test cases are forwarded to be merged with the component test cases of *C3* test cases.

In the second iteration, the approach merges the generated test cases with the component test cases of component *C3*. Figures 30-31 present the generated test cases. After that, the approach examines the generated test cases for the existence of interactions between the CUTs, (*C1* or *C2*) and *C3*. In this iteration, the approach selects four test cases out of twelve test cases to be exercised on the system: *MergeC1t2C2t1C3t1*, *MergeC1t2C2t2C3t1*, *MergeC1t3C2t2C3t2* and *MergeC1t3C2t1C3t2*.

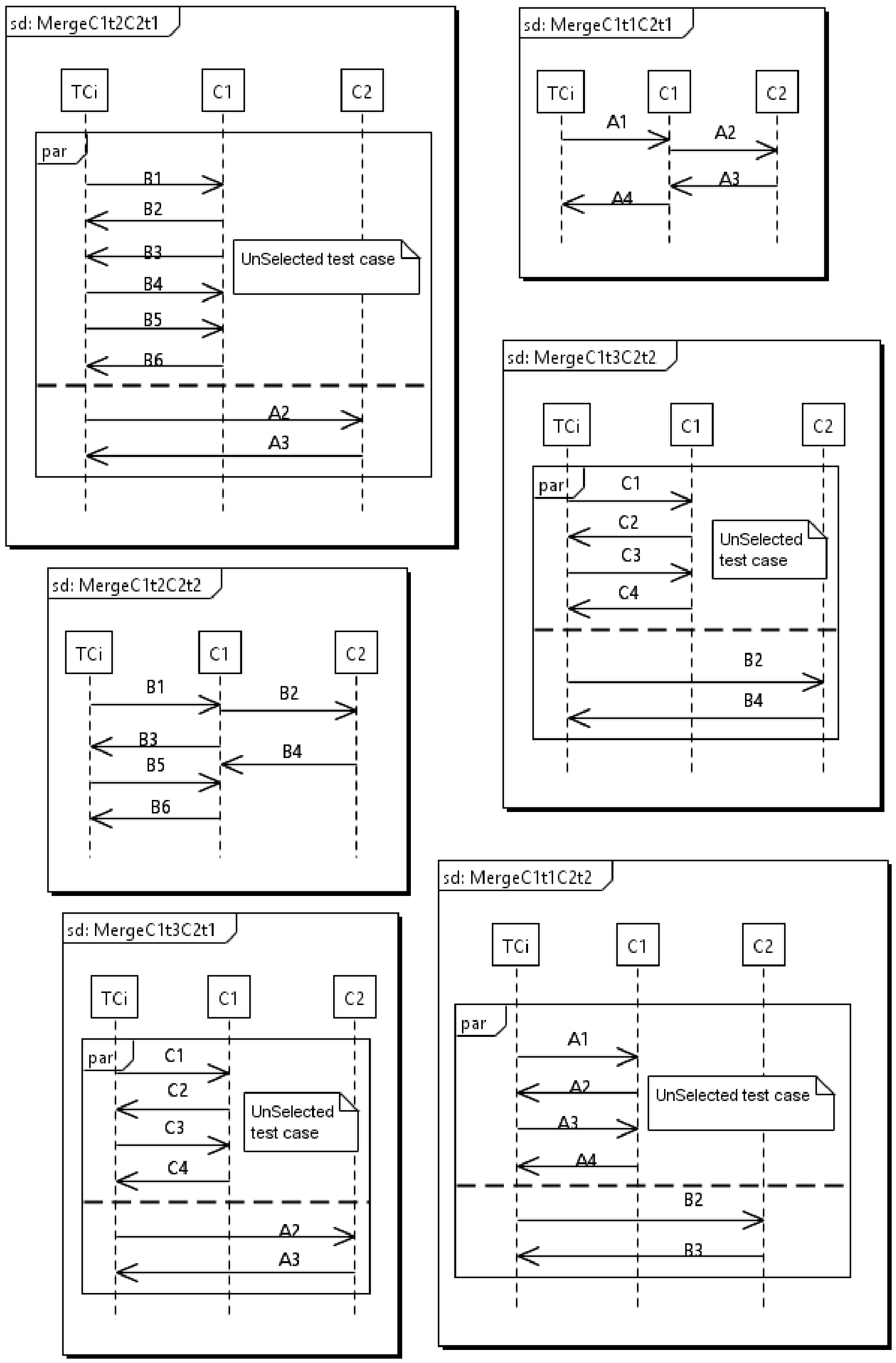


Figure 29. Cumulative integration: iteration 1 generated test cases

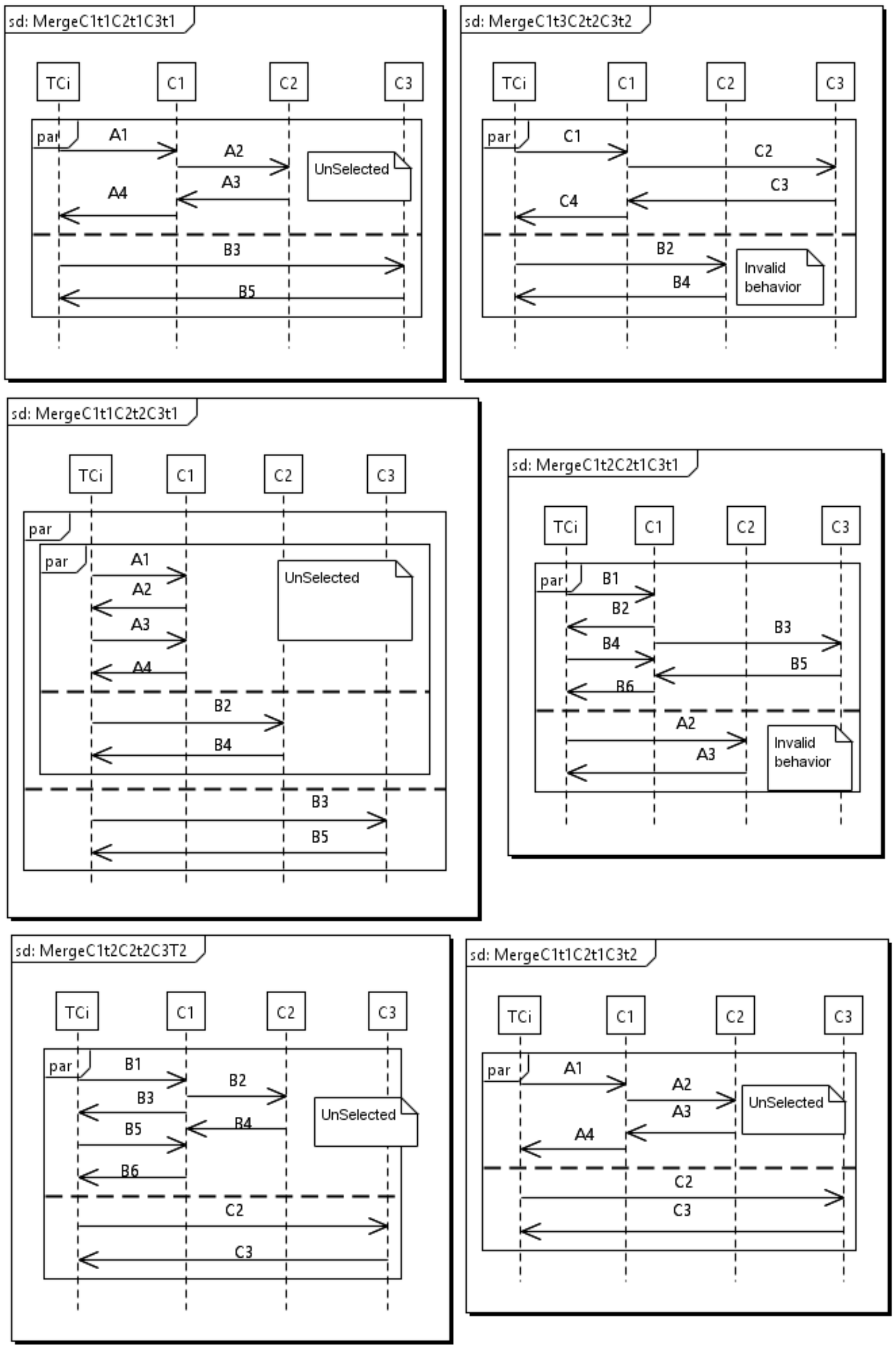


Figure 30. Cumulative integration: iteration 2 generated test cases (1/2)

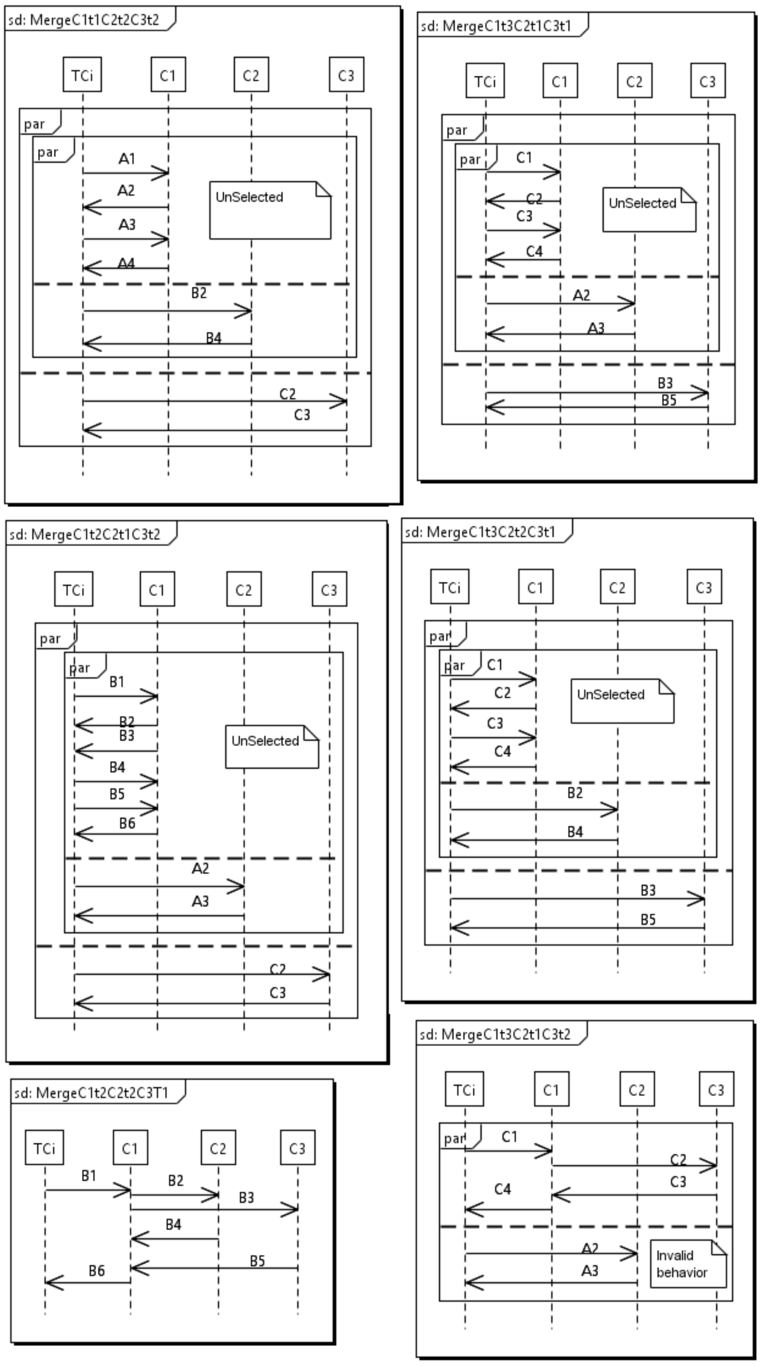


Figure 31. Cumulative integration: iteration 2 generated test cases (2/2)

4.2.5.3 Discussion

Let us study the results of the two techniques. The first remark is that the selective technique covers the three system services *A*, *B* and *C* in three test cases: *MergeC1t1C1t1*, *MergeC1t2C2t2C3t1* and *MergeC1t3C3t2* respectively; while the cumulative technique covers only two system services *A* and *B* in two test cases: *MergeC1t1C1t1* and *MergeC1t2C2t2C3t1* respectively. The second remark is that the two techniques have the same set of test cases during the first iteration, *MergeC1t1C1t1* and *MergeC1t2C2t2*. However, quantity wise, the cumulative technique generated three times more test cases than the selective technique. The final remark is that the second iteration of the cumulative technique selected and generated invalid test cases: *MergeC1t2C2t1C3t1*, *MergeC1t3C2t1C3t2* and *MergeC1t3C2t2C3t2*. Let us take test case *MergeC1t2C2t1C3t1* to clarify our argument. We can see that while the system is completely constructed, the test control *TCi* is still emulating an integrated component, *C1*, by sending message *A2* and receiving message *A3*. This behavior is incorrect for the following aspects. The first aspect, we are exercising part of a system service on the complete system, which may produce invalid results, as we will explain in the third aspect. The second aspect concerns the testability issue, which is a recognized problem in software testing, especially in embedded systems testing. Since the system is already integrated and the interfaces among the integrated components have been joined, the problem is if we can reach an individual component in order to control it by the test. As a last aspect, suppose that the interfaces of *C2* are reachable, so if we had exercised the test case and the test control sent message *A2*, according to the specification, component C2 would reply to the test control by sending message *A3*. However, component *C1* will receive message *A3* too since it is connected to that interface. What is going to be the reaction of *C1*? The system specification in Figure 25 and the specification of the test cases are silent on this situation. That means the reaction of *C1* will be interpreted as an invalid behavior and the test case will fail. Therefore, we ignored the cumulative technique and we adopted the selective technique for generating test models.

4.3 Conclusion

In this chapter, we presented a test generation approach. The proposed approach closes the gap among the testing levels. More precisely, it connects the component-level testing to the integration-level testing and the system-level testing. The approach also enables reusability across the software testing. It reuses the test models of the component-level testing to generate the test model of the subsequent testing levels. In this dissertation, we developed a test generation approach for the generation of the integration test models from the component test models. Several issues have been tackled in this research: test object identification, test case selection, test case merging and test case comparison. A prototype has been implemented and demonstrated in Chapter 6.

Chapter 5
Acceptance Test optimization

We discuss the optimization approach in this chapter. An acceptance test optimization approach is investigated throughout the chapter. The chapter is composed of three sections. We provide an overview of the optimization approach in the first section. The second section covers the selection of integration test cases that need to be mapped to the acceptance test cases. In the last section, we discuss the mapping of test cases to detect and remove redundant ones.

5.1 Introduction and Overview

The optimization approach maps test cases of the targeted testing level to test cases of previously performed testing levels. The mapping technique is based on the comparison of the semantics of the involved test cases. Techniques for comparing textual and graphical models are available and known as Model Comparison [52, 79]. These techniques are used by different methodologies such as Model-Cloning, Version Control Systems and Model-Transformation Testing. Furthermore, they are classified into two categories depending on the required information for the comparison: three-way comparison and two-way comparison [77]. Three-way comparison techniques require the existence of a base model in addition to the two models to be compared. Each model is compared separately to the base model. The differences in each model, from the base model, are identified and marked with one of the three flags: added, deleted or modified. Two-way comparison techniques compare two models without external references. One characteristic is common among all techniques in both categories that the compared models are evolved from the same source model. In this research, we assume that test models are built independently and that they are different. However, we also assume that part of these models may overlap since they describe the same system from different perspectives. Our idea is similar to the panorama technique in photography, where photos are taken independently, and then integrated to build the panorama, the big picture. Different methods are used to calculate the similarities and differences among the mapped models such as

- **Universal unique identifier (UUID)**: several modeling notations, including UML, assign unique identifiers to every created element in the model. These identifiers do not change once assigned. Some model comparison techniques use these identifiers to determine the similarities among the elements of the compared models. The two compared models have to be evolved from the same source. This method is not applicable to our domain since test models are created independently.
- **Name convention**: element's names are used to calculate the differences and similarities. Even though names are the most targeted attributes for changes in a distributed development environment, studies show the effectiveness of such methods [82]. While we request the consistency and the use of name convention among system components, interfaces and messages, we believe it is impractical to impose name convention in low-level elements such as message events.
- **Element properties**: In addition to the element name, model elements have several attributes that can be used in the comparison. However, these attributes differ from one element type to another. For example, UML classes have properties and operations, UML properties have type and multiplicity, and UML operations have passing parameters and return parameters. The use of all of the element's properties will increase the accuracy. However, it will hamper the performance and the computation speed. A wise selection of such properties is recommended.

In our approach, we use a mix of name convention and element properties methods to calculate the similarities and differences. We focus on software testing. Hence, our comparison approach is domain specific. Hence, we have selected certain element properties to calculate the similarities and differences among the test models. These element properties are related to the variables that are defined in the expressions of Definitions 1-4.

We propose an approach that optimizes the acceptance test model by relating it to the integration test models. We aim to reduce the acceptance test execution time by reducing the number of acceptance test cases. This can be achieved by eliminating acceptance test cases that have already been exercised on the system during integration-level testing. The approach maps the acceptance test cases to the integration test cases and excludes the ones that have already been exercised during the integration-level testing. However, Integration test cases are mostly applied on sub-systems. Usually, they emulate some of the system components that have not yet been integrated. Hence, they do not match with the acceptance and system test cases. However, there are two situations where the integration test cases are suitable to substitute acceptance and system test cases. The first situation includes test cases applied on the last stage of the integration-level testing. These test cases are exercises during the integration of the last component to the sub-system to build complete system. Therefore, the test cases are applied on complete systems. The second situation includes integration test models applied on sub-systems that fulfil the requirements of some of the system functionalities. Hence, test cases of such test models that examine these functionalities are actually applied on complete sub-systems. In other words, the test cases do not emulate system components.

The approach is composed of two processes: the selection process and the mapping process as shown in Figure 32. We present each process in the following sections. The approach can be applied to optimize the system test model in the same context without any modification.

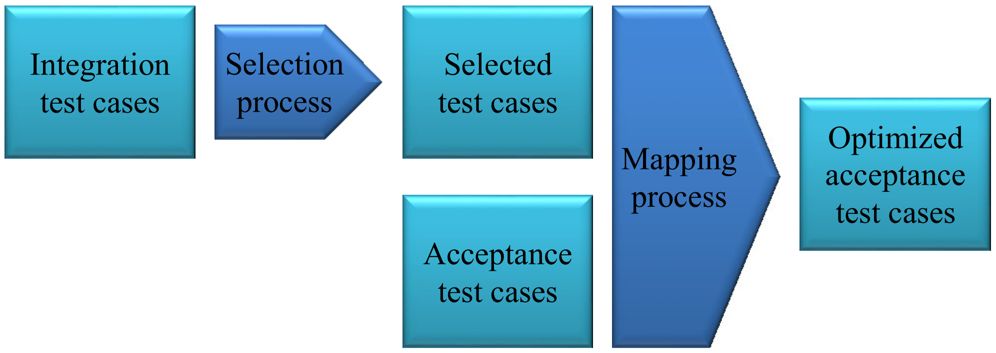


Figure 32. The optimization approach

5.2 Integration Test Case Selection

Integration testing is an iterative process. System components are sequentially integrated to build the complete system. An integration test model is developed and exercised during the integration of each component to check the compatibility among the integrated components. The development of integration test models usually includes the creation or use of test stubs of system components that have not been integrated yet to the system. The use of such test stubs in the integration test models disqualifies them from being compared to the acceptance test model. The acceptance test model must be exercised on the complete system without any emulation of any part of the system. Therefore, integration test models have to be free of any emulation of system components in order to be qualified for the comparison against the acceptance test model. We have to examine the given integration test models for the use of test stubs of system components. The test stubs may be specified in some test cases and not specified in other test cases of the same test model. To improve the accuracy of our approach, our examination will be on the level of the test cases instead of the level of the test model. The last integration test model is applied on the complete system when integrating the last system component to the sub-system. Hence, the test cases of the last integration test model are qualified to be mapped to the acceptance test cases. For the rest of the integration test models, we compare the behavior of their test stubs and test controls to the behavior of the CUTs of the subsequent integration test models as shown in Figure 33. More specifically, the approach compares the behavior of the test stubs and/or controls of each test case in a test model to the behavior of the CUTs of each test case in the subsequent test models. The selection algorithm is listed in Algorithm 1.

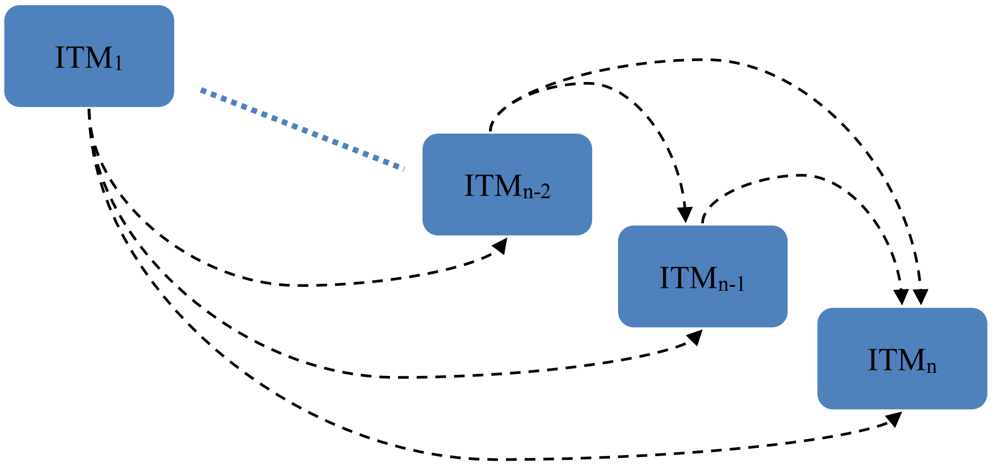


Figure 33. The selection process

The selection process selects the test cases that do not include test stubs of system components in their specifications. To formulate our selection condition, we use Definition 3 and Definition 5 to define the selection condition as follows:

Definition 9. (Selection condition)

Let $T_{kh} = (I_{kh}, E_{kh}, R_{kh})$ be the integration test case *h* at the integration iteration *k* and $T_{ij} = (I_{ij}, E_{ij}, R_{ij})$ be the integration test case *j* at integration iteration *i*, where $i > k$, then T_{kh} does not use a test stub for the CUT of T_{ij} if and only if the following condition is satisfied:

$$Sel_{kh} = \begin{array}{l} \forall (e_j, e_h).\, e_j \in E_{ij}, e_h \in E_{kh} \mid (e_j \neq e_h) \lor \\ \left((e_j = e_h) \land (e_j.owner.st \neq SUT)\right) \end{array}$$

Algorithm 1. The selection algorithm

```
read integration test models: TM[1..n]
initialize the set of selected test cases: SelectionSet = {}
for k = 1 to n-1 do
  traverse through test cases of TM[k]: T[h=1..m]
    isSelected = true
    for i = k+1 to n do
       traverse through test cases of TM[i]: T[j=1..w]
        evaluate Selkh
        isSelected = Selkh
        if isSelected = false then
          exit
        endif
    endfor
    if isSelected = true then
      SelectionSet.add( TM[k].T[h] )
    endif
  endfor
```

The selection process stops the comparison as soon as the condition is no longer satisfied, i.e.: it returns false. Consequently, the corresponding test case is excluded from the selection. We illustrate the selection process using the system shown in Figure 34. The system is composed of three components: *C1*, *C2* and *C3*, and it provides two services: *A* and *B*. Service *A* is handled by components *C1* and *C2*, and service *B* is managed by the three components. To distinguish between the two services, we have suffixed the names of the system messages with their corresponding services. To build the system, we integrate the components *C1* and C2 to build an intermediate sub-system; then we integrate the component *C3* to the sub-system to build the complete system. Consequently, we have to examine the integration twice, i.e.: two integration test iterations.

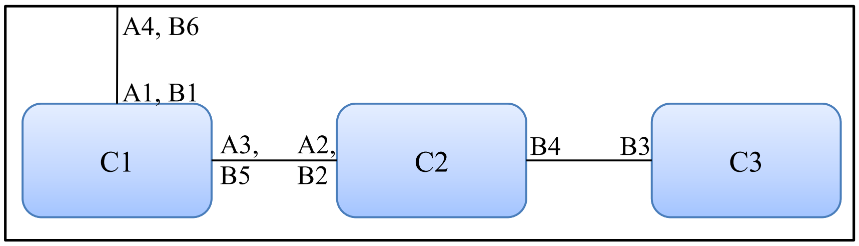


Figure 34. System specification

In the first integration test iteration, we apply the integration test cases shown in Figure 35 on the integrated components to examine the connectivity between the two components *C1* and *C2*. Using Definitions 3-4, we can express the given test cases as follows:

$tcase11 = (\{TC1, C1, C2\}, \{e_{k1},e_{k2},e_{k3},e_{k4},e_{k5},e_{k6},e_{k7},e_{k8},e_{k9}\}, \{(e_{k1},e_{k2}), (e_{k2},e_{k3}), (e_{k4},e_{k5}), (e_{k5},e_{k6}), (e_{k6},e_{k7}),$
$(e_{k8},e_{k9}), (e_{k1},e_{k4}), (e_{k5},e_{k8}), (e_{k9},e_{k6}), (e_{k7},e_{k2}), (e_{k1},e_{k3}), (e_{k4},e_{k6}), (e_{k4},e_{k8}), (e_{k5},e_{k7}), (e_{k6},e_{k2}), (e_{k8},e_{k6}),$
$(e_{k1},e_{k5}), (e_{k5},e_{k9}), (e_{k9},e_{k7}), (e_{k7},e_{k3}), (e_{k4},e_{k7}), (e_{k4},e_{k9}), (e_{k5},e_{k2}), (e_{k6},e_{k3}), (e_{k8},e_{k7}), (e_{k1},e_{k6}), (e_{k1},e_{k8}),$
$(e_{k9},e_{k2}), (e_{k4},e_{k2}), (e_{k5},e_{k3}), (e_{k8},e_{k2}), (e_{k1},e_{k7}), (e_{k1},e_{k9}), (e_{k9},e_{k3}), (e_{k4},e_{k3}), (e_{k8},e_{k3})\})$

$TC1 = \{\text{“}TC1\text{”},TestContext\}$

$C1 =\{\text{“}C1\text{”},SUT\}$

$C2 =\{\text{“}C2\text{”},SUT\}$

$e_{k1} = (send, \text{“}e_{k1}\text{”}, TC1, A1, C1)$

$e_{k2} = (receive, \text{“}e_{k2}\text{”}, TC1, A4, C1)$

$e_{k3} = (UTPverdict, \text{“}e_{k3}\text{”}, \text{“}pass\text{”}, TC1)$

$e_{k4} = (receive, \text{“}e_{k4}\text{”}, C1, A1, TC1)$

$e_{k5} = (send, \text{“}e_{k5}\text{”}, C1, A2, C2)$

$e_{k6} = (receive, \text{“}e_{k6}\text{”}, C1, A3, C2)$

$e_{k7} = (send, \text{“}e_{k7}\text{”}, C1, A4, TC1)$

$e_{k8} = (receive, \text{“}e_{k8}\text{”}, C2, A2, C1)$

$e_{k9} = (send, \text{“}e_{k9}\text{”}, C2, A3, C1)$

$tcase12 = (\{TC1, C1, C2\}, \{e_{k11},e_{k12},e_{k13},e_{k14},e_{k15},e_{k16},e_{k17},e_{k18},e_{k19},e_{k20},e_{k21},e_{k22},e_{k23}\}, \{(e_{k11},e_{k12}), (e_{k12},e_{k13}),$
$(e_{k13},e_{k14}), (e_{k14},e_{k15}), (e_{k16},e_{k17}), (e_{k17},e_{k18}), (e_{k18},e_{k19}), (e_{k20},e_{k21}), (e_{k21},e_{k22}), (e_{k22},e_{k23}), (e_{k11},e_{k16}), (e_{k17},e_{k20}),$
$(e_{k21},e_{k12}), (e_{k13},e_{k22}), (e_{k23},e_{k18}), (e_{k19},e_{k14})$
$,(e_{k11},e_{k13}),(e_{k12},e_{k14}),(e_{k12},e_{k22}),(e_{k13},e_{k15}),(e_{k16},e_{k18}),(e_{k16},e_{k20}),(e_{k17},e_{k19}),(e_{k18},e_{k14}),$
$(e_{k20},e_{k22}),(e_{k20},e_{k12}),(e_{k21},e_{k23}),(e_{k22},e_{k18}),(e_{k11},e_{k17}),(e_{k17},e_{k21}),(e_{k21},e_{k13}),(e_{k13},e_{k23}),$
$(e_{k23},e_{k19}),(e_{k19},e_{k15}),(e_{k11},e_{k14}),(e_{k11},e_{k22}),(e_{k12},e_{k15}),(e_{k12},e_{k23}),(e_{k16},e_{k19}),(e_{k16},e_{k21}),$
$(e_{k17},e_{k14}),(e_{k18},e_{k15}),(e_{k20},e_{k23}),(e_{k20},e_{k13}),(e_{k21},e_{k18}),(e_{k22},e_{k19}),(e_{k11},e_{k18}),(e_{k11},e_{k20}),$
$(e_{k17},e_{k22}),(e_{k17},e_{k12}),(e_{k21},e_{k14}),(e_{k13},e_{k18}),(e_{k23},e_{k14}),(e_{k11},e_{k15}),(e_{k11},e_{k23}),(e_{k12},e_{k18}),$
$(e_{k16},e_{k14}),(e_{k16},e_{k22}),(e_{k16},e_{k12}),(e_{k17},e_{k15}),(e_{k20},e_{k18}),(e_{k20},e_{k14}),(e_{k21},e_{k19}),(e_{k22},e_{k14}),$
$(e_{k11},e_{k19}),(e_{k11},e_{k21}),(e_{k17},e_{k23}),(e_{k17},e_{k13}),(e_{k21},e_{k15}),(e_{k13},e_{k19}),(e_{k23},e_{k15}),(e_{k12},e_{k19}),$
$(e_{k16},e_{k15}),(e_{k16},e_{k23}),(e_{k16},e_{k13}),(e_{k20},e_{k19}),(e_{k20},e_{k15}),(e_{k22},e_{k15})\})$

$TC1 = \{\text{“}TC1\text{”},TestContext\}$

$C1 =\{\text{“}C1\text{”},SUT\}$

$C2 =\{\text{“}C2\text{”},SUT\}$

$e_{k11} = (send, \text{“}e_{k11}\text{”}, TC1, B1, C1)$

$e_{k12} = (receive, \text{“}e_{k12}\text{”}, TC1, B3, C2)$

$e_{k13} = (send, \text{“}e_{k13}\text{”}, TC1, B4, C2)$

$e_{k14} = (receive, \text{“}e_{k14}\text{”}, TC1, B6, C1)$

e_{k15} = *(UTPverdict, "e_{k15}", "pass", TC1)*

e_{k16} = *(receive, "e_{k16}", C1, B1, TC1)*

e_{k17} = *(send, "e_{k17}", C1, B2, C2)*

e_{k18} = *(receive, "e_{k18}", C1, B5, C2)*

e_{k19} = *(send, "e_{k19}", C1, B6, TC1)*

e_{k20} = *(receive, "e_{k20}", C2, B2, C1)*

e_{k21} = *(send, "e_{k21}", C2, B3, TC1)*

e_{k22} = *(receive, "e_{k22}", C2, B4, TC1)*

e_{k22} = *(send, "e_{k23}", C2, B5, C1)*

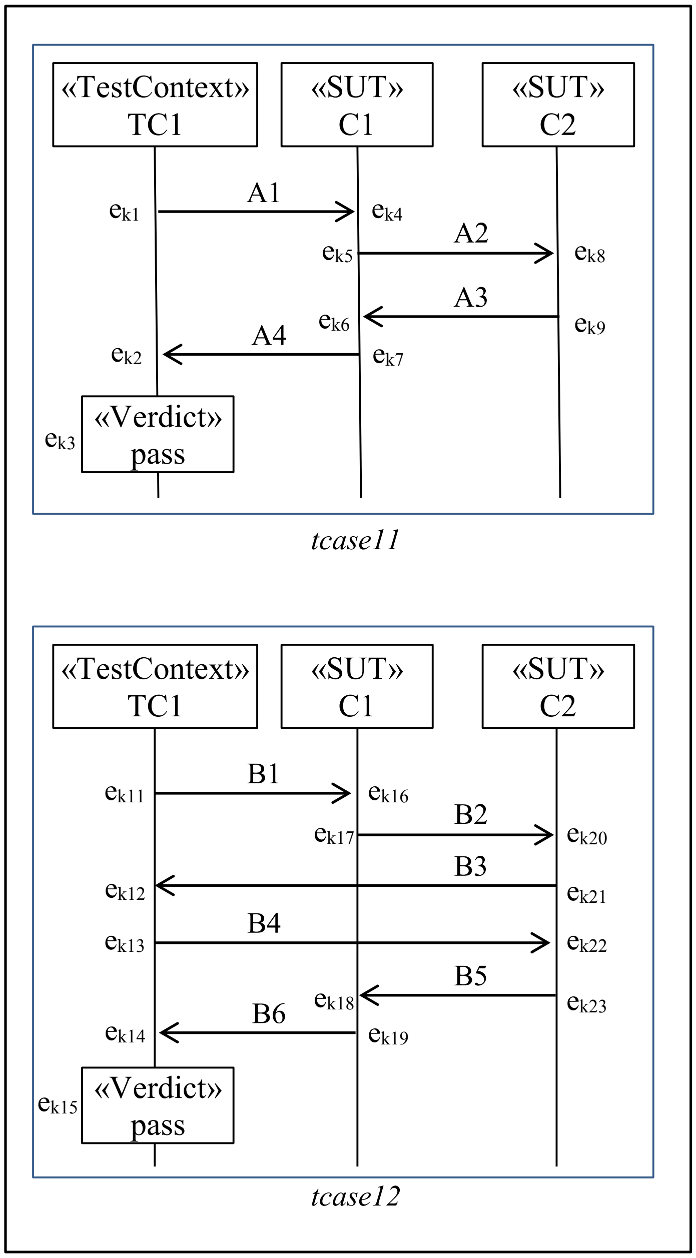


Figure 35. Integration test cases: first iteration

In the second integration test iteration, we apply the integration test cases shown in Figure 36 on the integrated components to examine the connectivity between the sub-system *SbSys*, which is composed of the integrated components *C1* and *C2*, and the component *C3*. Using Definitions 3-4, we can express the given test cases as follows:

tcase21 = ({TC2, SbSys, C3}, {ei1,ei2,ei3,ei4,ei5,ei6,ei7,ei8,ei9}, {(ei1,ei2), (ei2,ei3), (ei4,ei5), (ei5,ei6), (ei6,ei7), (ei8,ei9), (ei1,ei4), (ei5,ei8), (ei9,ei6), (ei7,ei2), (ei1,ei3), (ei4,ei6), (ei4,ei8), (ei5,ei7), (ei6,ei2), (ei8,ei6), (ei1,ei5), (ei5,ei9), (ei9,ei7), (ei7,ei3), (ei4,ei7), (ei4,ei9), (ei5,ei2), (ei6,ei3), (ei8,ei7), (ei1,ei6), (ei1,ei8), (ei9,ei2), (ei4,ei2), (ei5,ei3), (ei8,ei2), (ei1,ei7), (ei1,ei9), (ei9,ei3), (ei4,ei3), (ei8,ei3)})

TC2 = {"TC2",TestContext}

SbSys ={" SbSys",SUT}

C3 ={"C3",SUT}

ei1 = (send, "ei1", TC2, B1, SbSys)

ei2 = (receive, "ei2", TC2, B6, SbSys)

ei3 = (UTPverdict, "ei3", "pass", TC2)

ei4 = (receive, "ei4", SbSys, B1, TC2)

ei5 = (send, "ei5", SbSys, B3, C3)

ei6 = (receive, "ei6", SbSys, B4, C3)

ei7 = (send, "ei7", SbSys, B6, TC2)

ei8 = (receive, "ei8", C3, B3, SbSys)

ei9 = (send, "ei9", C3, B4, SbSys)

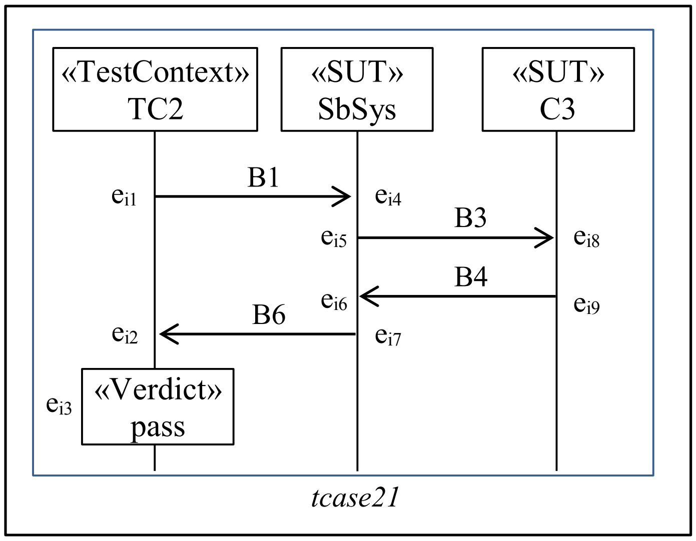


Figure 36. Integration test cases: second iteration

We submit the three integration test cases: *tcase11*, *tcase12* and *tcase21*, to the selection process to select the qualified ones to be compared to acceptance test cases. Since the integration test case *tcase21* was applied on the complete system during the final integration iteration, the selection process selects it to be forwarded to the mapping process. However, the selection process examines the other two test cases by relating them to the test case *tcase21* using the selection condition in Definition 9. We need to check if one of them emulates the system component *C3*. The test case *tcase11* passes the examinations since it has a different set of events and the term $(e_j \neq e_h)$ of the selection condition is always fail. However, the test case *tcase12* fails the examination because of two pairs of events that unsatisfied the selection condition: (e_{i8}, e_{k12}) and (e_{i9}, e_{k13}).

$$Sel_{kh} = (e_{i8} \neq e_{k12}) \vee \left((e_{i8} = e_{k12}) \wedge (e_{i8}.owner.st \neq SUT)\right) \xrightarrow{yields} false$$

$$Sel_{kh} = (e_{i9} \neq e_{k13}) \vee \left((e_{i9} = e_{k13}) \wedge (e_{i9}.owner.st \neq SUT)\right) \xrightarrow{yields} false$$

The selection process requires only one pair to exclude the test case *tcase12*. So, let us discuss the first pair. The two events match according to the event matching expression defined in Definition 5, i.e.: the term $(e_{i8} \neq e_{k12})$ fails and the term $(e_{i8} = e_{k12})$ passes. Hence, we focus on the second part of the selection expression, i.e.: $(e_{i8}.owner.st \neq SUT)$. The event e_{i8} is owned by the CUT *C3*, which falsifies the last portion of the second part ($(e_{i8} = e_{k12})$ *and* ($e_j.owner.st \neq SUT$)). That means the test control *TC1* is emulating the system component *C3* during the execution of the test case *tcase12*. Consequently, the whole expression fails and the test case *tcase12* is excluded. At the end of the selection process, two test cases *tcase11* and *tcase21* are qualified to be mapped to the acceptance test cases and are forwarded to the mapping process, which we cover in the following section.

The results of the selection process depend on the integration order. The usage of test stubs of system components depends on the integration order. We may not require any test stubs when we choose the right integration order. There is a lot of research work being done on the selection of the right integration order [40-42].

5.3 Mapping Acceptance Test Cases to Integration Test Cases

The mapping process compares the acceptance test cases against the selected integration test cases. The process removes acceptance test cases from the test model if their specifications are included in the specification of the selected integration test cases. The inclusion expression in Definition 8 cannot be used in this process because it examines the instances of both test cases. However, the acceptance-level testing has a different perspective of the system than the integration-level testing as shown in Figure 37. In the acceptance-level testing, we see the system as a solid block and we examine it through its external interfaces, while in the integration-level testing, we see fragments of the system, and we examine it through its external interfaces as well as through the internal interfaces of the currently integrated component. Consequently, the generated test cases are different with respect to the test objects described in each testing level. The mapping algorithm is listed in Algorithm 2.

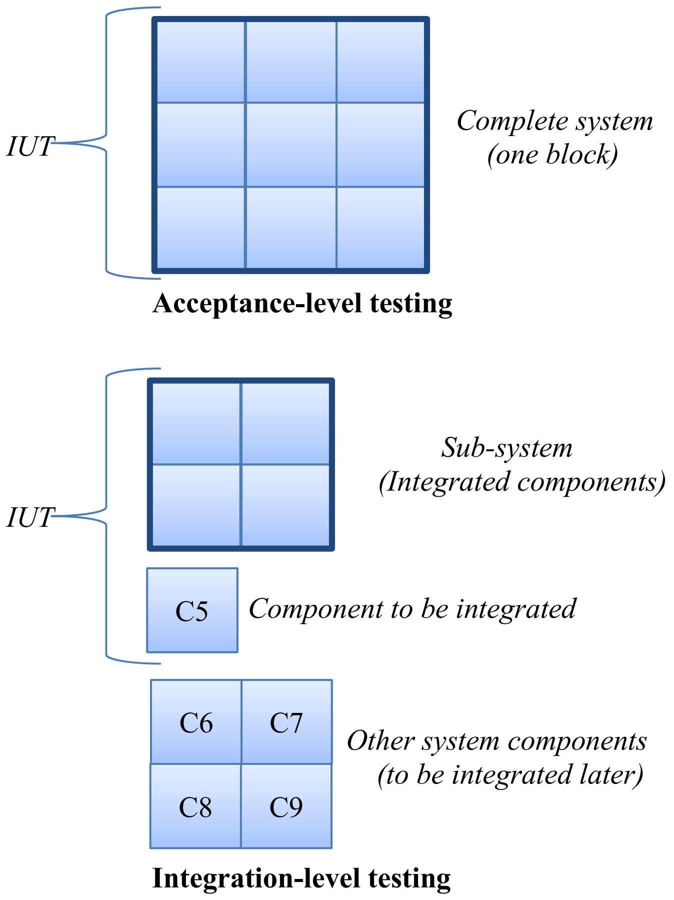


Figure 37. Testing levels with different views of the IUT

Furthermore, we have to take into account that the events specified on a lifeline of a test object in an acceptance test case may be distributed over several lifelines in the mapped integration test case as shown in Figure 38. Acceptance test cases are usually composed of two test objects, the test control (*TCa*) and the system under test (*Sys*), while integration test cases are composed of at least three test objects: the test control (*TCi*), the *CUT* and the sub-system (*SbSys*). Hence, the behavior of the two test objects, *TCa* and *Sys*, in the acceptance test cases is distributed over three test objects, *TCi*, *CUT* and *SbSys*, in the integration test cases.

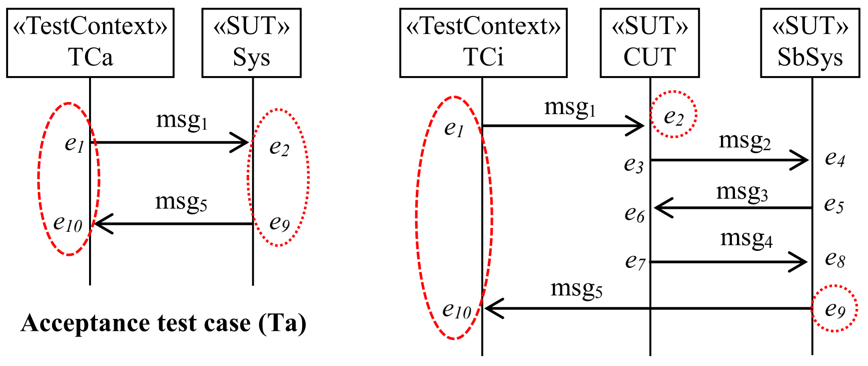
Algorithm 2. The mapping algorithm

```
read acceptance test cases: TCa[1..n]
read selected integration test cases: TCi[1..m]
for i = 1 to n do
  for j = 1 to m do
    isContained = true;
    isContained = isContained AND (TCa[i].E $\subseteq$ TCi[j].E)
    isContained = isContained AND (TCa[i].R $\subseteq$ TCi[j].R)
    if isContained = true then
      remove TCa[i]
      exit interior for loop "for j = ..."
    endif
  endfor
endfor
```

Moreover, integration test cases may have extra behaviors that reflect internal interactions between the *CUT* and *SbSys*. In other words, we should not expect the acceptance test case to be a complete fragment/block within the integration test case. To illustrate that, let us consider the test cases shown in Figure 38. Using Definition 3, the two test cases can be expressed as follow:

$Ta = (\{TCa, Sys\}, \{e1,e2,e9,e10\}, \{(e1,e10), (e2,e9), (e1,e2), (e9,e10), (e2,e10), (e1,e9)\})$

$Ti = (\{TCi, CUT, SbSys\}, \{e1,e2,e3,e4,e5,e6,e7,e8,e9,e10\}, \{(e1,e10), (e2,e3), (e3,e6), (e6,e7), (e_4,e_5), (e_5,e_8), (e_8,e_9), (e_1,e_2), (e_3,e_4), (e_5,e_6), (e_7,e_8), (e_9,e_{10}), (e_2,e_6), (e_2,e_4), (e_3,e_7), (e_6,e_8), (e_4,e_8), (e_4,e_6), (e_5,e_9), (e_8,e_{10}), (e_1,e_3), (e_3,e_5), (e_5,e_7), (e_7,e_9), (e_2,e_7), (e_2,e_5), (e_3,e_8), (e_6,e_9), (e_4,e_9), (e_4,e_7), (e_5,e_{10}), (e_1,e_6), (e_1,e_4), (e_7,e_{10}), (e_2,e_8), (e_3,e_9), (e_6,e_{10}), (e_4,e_{10}), (e_1,e_7), (e_1,e_5), (e_2,e_9), (e_3,e_{10}), (e_1,e_8), (e_2,e_{10}), (e_1,e_9)\})$



Acceptance test case (Ta)

Integration test case (Ti)

Figure 38. Scattered events

In this example, we can compare the behavior of the test controls, *TCa* and *TCi*, as a block since they have identical set of events, (e_1, e_{10}). However, the behavior, (e_2, e_9), of the system *Sys* is distributed between two test objects. The event e_2 belongs to the integrated component *CUT* while event e_9 belongs to the sub-system *SbSys*. Furthermore, the behavior of the integration test case (e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{10}) contains internal events (e_3, e_4, e_5, e_6, e_7, e_8) that are not specified in the acceptance test case and divide the behavior of the acceptance test case (e_1, e_2, e_9, e_{10}) into two fragments. The first fragment consists of e_1 and e_2, and the second fragment consists of e_9 and e_{10}. Therefore, the mapping process requires the transition closure of the event relation of the test cases to overcome this issue.

As we have abovementioned, the inclusion expression in Definition 8 is valid only for comparing test cases at the same level. It cannot be used to compare integration test cases from different integration iterations. Hence, we drive a new inclusion expression that does not depend on the instances of the test cases.

Definition 10. (Test case Inclusion)

Let $T_a = \{I_a, E_a, R_a\}$ be an acceptance test case and $T_i = \{I_i, E_i, R_i\}$ be an integration test case, then the acceptance test is included in the integration test case if and only if the following conditions are satisfied:

(1) $E_a \subseteq E_i$

(2) $R_a \subseteq R_i$

The first condition states that the events specified in the acceptance test case are all specified in the integration test case. The second condition checks that all the order relations among the events of the acceptance test case are respected in the integration test case specification. This comparison is possible as test cases have finite behaviors.

We continue to use the system specification shown in Figure 34 to illustrate our mapping process. The integration test cases are given in Figures 35-36; and the selection process selected two out of three to be mapped to the acceptance test cases. The acceptance test cases are presented in Figure 39. Using Definitions 3-4, we can express the given acceptance test cases as follows:

tcaseA1 = ({TCa, Sys}, {e1,e2,e3,e4,e5}, {(e1,e2), (e2,e3), (e4,e5), (e1,e4), (e5,e2) , (e1,e3), (e4,e2), (e1,e5), (e5,e3), (e4,e3)})

TCa = {"TCa",TestContext}

Sys ={"Sys",SUT}

e1 = (send, "e1", TCa, A1, Sys)

e2 = (receive, "e2", TCa, A4, Sys)

e3 = (UTPverdict, "e3", "pass", TCa)

e4 = (receive, "e4", Sys, A1, TCa)

e5 = (send, "e5", Sys, A4, TCa)

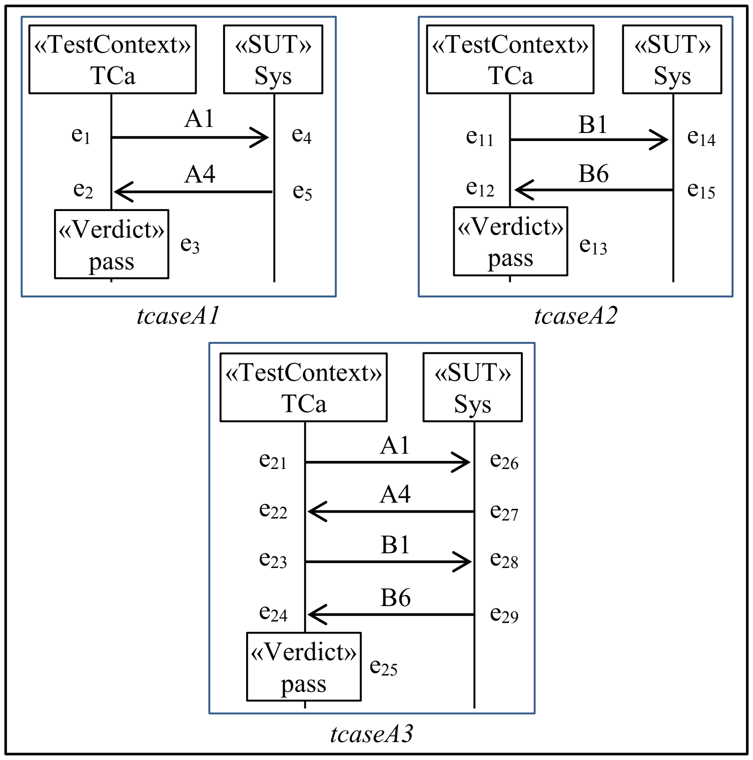


Figure 39. Acceptance test cases

tcaseA2 = ({TCa, Sys}, {e11,e12,e13,e14,e15}, {(e11,e12), (e12,e13), (e14,e15), (e11,e14), (e15,e12), (e11,e13), (e14,e12), (e11,e15), (e15,e13), (e14,e13)})

TCa = {"TCa",TestContext}

Sys ={"Sys",SUT}

e11 = (send, "e11", TCa, B1, Sys)

e12 = (receive, "e12", TCa, B6, Sys)

e13 = (UTPverdict, "e13", "pass", TCa)

e14 = (receive, "e14", Sys, B1, TCa)

e15 = (send, "e15", Sys, B6, TCa)

tcaseA3 = ({TCa, Sys}, {e21,e22,e23,e24,e25,e26,e27,e28,e29}, {(e21,e22), (e22,e23), (e23,e24), (e24,e25), (e26,e27), (e27,e28), (e28,e29), (e21,e26), (e27,e22), (e23,e28), (e29,e24), , (e21,e23), (e22,e24), (e22,e28), (e23,e25), (e26,e28), (e26,e22), (e27,e29), (e28,e24), (e21,e27), (e27,e23), (e23,e29), (e29,e25), (e21,e24), (e21,e28), (e22,e25), (e22,e29), (e26,e29), (e26,e23), (e27,e24), (e28,e25), (e21,e25), (e21,e29), (e26,e24), (e27,e25), (e26,e25)})

TCa = {"TCa",TestContext}

Sys ={"Sys",SUT}

e21 = (send, "e21", TCa, A1, Sys)

e22 = (receive, “e22”, TCa, A4, Sys)

e23 = (send, “e23”, TCa, B1, Sys)

e24 = (receive, “e24”, TCa, B6, Sys)

e25 = (UTPverdict, “e25”, “pass”, TCa)

e26 = (receive, “e26”, Sys, A1, TCa)

e27 = (send, “e27”, Sys, A4, TCa)

e28 = (receive, “e28”, Sys, B1, TCa)

e29 = (send, “e29”, Sys, B6, TCa)

The mapping process compares the acceptance test cases to the selected integration test cases using the inclusion expression Definition 10. Table 5 summarizes the outcome of the mapping process. Two acceptance test cases, *tcaseA1* and *tcaseA2*, are excluded since they are included in the selected integration test cases. The third acceptance test case, *tcaseA3*, does not satisfy the inclusion conditions. Hence, it can be transformed to a test execution code and exercised on the system. The mapping process stops the test case comparison as soon as the acceptance test case satisfies the two inclusion conditions. It also moves to the next integration test case if the first condition is unsatisfied. Furthermore, the acceptance test model has been optimized by 60%; two test cases out of three test cases had been excluded

Table 5. The results of the mapping process

Acceptance test cases	Integration test cases	First condition $E_a \subseteq E_i$	First condition $R_a \subseteq R_i$	Action
tcaseA1	*tcase11*	satisfied	satisfied	excluded
tcaseA1	*tcase21*			
tcaseA2	*tcase11*	unsatisfied		
tcaseA2	*tcase21*	satisfied	satisfied	excluded
tcaseA3	*tcase11*	unsatisfied		
tcaseA3	*tcase21*	unsatisfied		exercised

5.4 Conclusion

Test-Suite Reduction is an active research activity in the software testing. It aims to reduce the testing cost by reducing the time of the test execution [91]. The number of generated test cases is typically large for complex systems [91], as it increases rapidly as new features and/or updates are added to the system. However, most research activities focus on the reduction of the size of the test models by mapping test cases of the same test model against each other. Instead, we propose a test optimization framework that relates test cases across different testing levels: integration-level, system-level and acceptance-level testing. It maps the generated test cases to previously executed test cases and removes the redundant ones.

In this dissertation, we developed an acceptance test optimization approach. The approach optimizes the acceptance test model using the integration test models. In this approach, we investigate the given integration test cases to select the ones that are suitable to be mapped to the acceptance test cases. We have implemented a prototype demonstrated in Chapter 6.

Chapter 6
Implementation & Case Study

In this chapter, we present the implementation of the two approaches discussed in the previous chapters. We developed two prototypes to demonstrate the effectiveness of our framework. The two prototypes are integrated in one application/tool since they serve the same framework and share some of the implemented packages as explained in Section 6.3. This chapter is composed of four sections. In the first section, we discuss the development tools used to build the toll and the UTP test models. In Section 6.2, we point out some principles that should be followed to construct acceptable test models for the current release of the tool. We present the implemented application in Section 6.3; the test generation prototype is presented in Section 6.3.1and the test optimization prototype is presented in Section 6.3.2. Section 6.4 covers a case study to demonstrate the use of our application and discuss its results.

6.1 Development Tools

In order to develop our prototypes, we searched for two kinds of development tools: modelling tools and transformation tools. The transformation tool is required to build our prototypes. The modelling tool is required to build sample test models that can be used to examine our prototypes and build our case studies. During our review of such tools, we passed by a plenty of commercial and open source tools. Since we are building prototypes, we decided to use open source tools. Table 6 lists some of the development tools, which we have investigated. We used Atlas Transformation Language (ATL) [101] and Java for the transformation tool, Papyrus and Eclipse UML-Editor for the modelling tool based on the following selection criteria:

- ➢ Transformation Tools
    - ✓ Mature
    - ✓ Open source
    - ✓ Compliance with OMG QVT
- ➢ Modelling Tools

- ✓ Mature
- ✓ Open source
- ✓ Compliance with UML XMI
- ✓ Support UML profiles

Table 6. Development tools

Kind	Name	Description
Transformation	mediniQVT	Commercial [102]
Transformation	ATL	Open source under the Eclipse project [101]
Transformation	QVT Operational	Open source under the Eclipse project [103]
Transformation	QVTd	Open source under the Eclipse project [104]
Transformation	ModTransf V3	Under development [105]
Modelling	Eclipse UML Editor	Open source under the Eclipse project. Embedded in the Eclipse Modeling Tools [106]
Modelling	UMLet	Open source under the Eclipse project. Plug-in tool. Need to be installed from Eclipse Market Place [107]
Modelling	Papyrus	Open source under the Eclipse project [108]
Modelling	Visual Paradigm	Commercial [109]
Modelling	Modelio	Open source [110]

In the following sub-subsections, we present a brief introduction about the selected development tools.

6.1.1 Transformation Tool

ATL is developed to answer the OMG's "QVT Request for Proposal". The language supports model transformation for MOF's [111] and Ecore's [112] metamodels. The language is composed of declarative and imperative languages. It supports multi-input/multi-output models. The ATL language is a modular language. The ATL module consists of four sections as shown in Figure 40; two sections are mandatory, header and rules, and two sections are optional, import and helpers. The header section defines the module name, the input models and the output models. The import section allows the developer to import ATL libraries. ATL libraries define ATL helpers and enable reusability across ATL modules. The helpers section defines ATL expressions that can be called several times from the rules section. Each helper has a context related to the input models. Helpers without a context are module helpers and they are evaluated once at the initialization of the module; that means the helper returns the same value during the same execution. The rules section defines the transformation from the input model(s) to the output model(s). The transformation rules, for a specific model type, have to be associated to one ATL module, which is contained in a single file. There are two types of rules. The declarative rules, which are mandatory, have three constructs: matched, lazy and called rules. The imperative rules use the query construct. Since our approaches rely heavily on the analysis of the input models, we faced some difficulties with the implementation using ATL rules. For example, we were stuck when we found that there is no exit command from the loop, similar to *break* in Java, in the ATL imperative language. Our approaches iterate through the test cases searching for key criterion and should exit as soon as such criterion is satisfied. These difficulties lead us to depend more on Java. While the OMG QVT specification allows the execution of add-on scripts of different languages, say Java, ATL does not implement such a feature. It is not mentioned in the online documentation, and we did not get an answer about this issue at the ATL community forum. However, there is ATL APIs to be accessed from Java. Hence, we depend heavily on Java for developing our prototypes. Furthermore, we used the Eclipse UML2 project to access the test models.

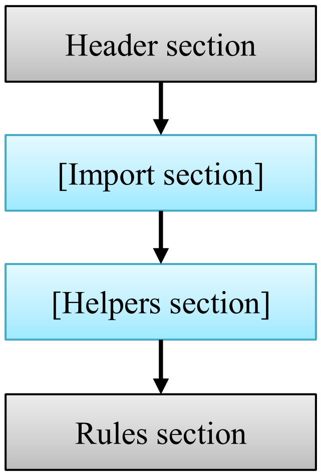


Figure 40. Structure of ATL module

6.1.2 Modelling Tool

Papyrus is a graphical editor for UML2. It is an add-on project in Eclipse modelling framework. The project intends to fully implement the OMG's UML specification. The project supports the construction of UML diagrams, SysML diagrams and UML profiles. Models are stored in two files: one file for the serialized UML and the second file for the graphical representation. The project accepts external XMI profiles. Hence, the serialized version of the OMG's UTP specification is used to build our test models. The current limitation of Papyrus is the inability to generate diagrams from serialized UML that are created by other tools. There is an ongoing work for implementing such feature but it is still immature. Hence, we use the Eclipse embedded UML editor to view such models in a tree-like syntax. We use this editor to view the output of our approaches. In this document, diagrams are created manually from the serialized UML files of the generated test models.

6.1.3 UML Testing Profile

Currently, we are using OMG's UTP specification version 1.1. The UTP specification covers only the test architecture. The test behavior is still left out for later releases except for the definition of the test case stereotype for UML sequence diagrams. Hence, we have to depend on the UML specification to link between the test structure and the test behavior of any test model, which we describe in the next section.

6.2 Test Model Settings

In order to apply our tools on the test models, there must be a clear relation between the test behavior and its corresponding test architecture. Instances in test cases must refer to test objects specified in the test architecture. In our framework, test cases are specified using UML sequence diagrams and test architectures are specified using UML class diagrams. Since the test behavior is not enriched with UTP stereotypes, we have to depend on the UML specifications to link elements in the sequence diagrams to their corresponding elements in the class diagrams. The current implementation requires the following compliance in the test model as illustrated in Figure 41:

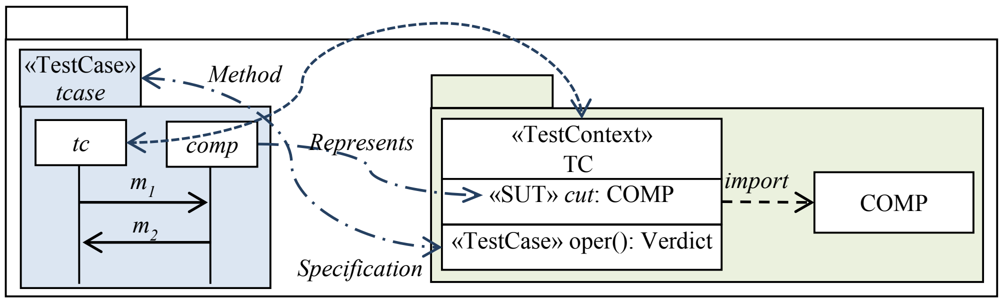


Figure 41. Test model settings

1. Test package:
    a. There must be only one test control.
    b. Test Control:
        - It must be stereotyped with UTP *TestContext*.
        - The test cases must be defined as operations and stereotyped with UTP *TestCase*. The *method* attribute of each operation must be linked to the corresponding test case.
        - Operations are typed with UTP Verdict.
    - Instances, UML lifelines, can be linked only to UML ConnectableElements, properties or association ends, and cannot be linked directly to UML Classifiers, classes. Hence, associations should be explicitly specified among test objects, test objects defined as properties in the test control or exact names donated for the instances.
2. Test cases:
    - Each active test case must be stereotyped with UTP *TestCase*.
    - They should be linked to their corresponding operation through the *specification* attribute.
    - Instances/Lifelines:
        - Each instance must be linked to its corresponding test object using the *represents* attribute, or must have the same name as its corresponding test object in the test package
        - The instances of the test control can be named after the corresponding test object in the test package, or named by the *self*-keyword as specified in the UML specification.
    - Messages: message names are unique across the test models since test cases represent execution traces. If the same message name is used by the same test object in two or more test cases that means we are specifying the same instance of that message.

6.3 TestGenO: The Test Generation and Optimization Tool

The tool integrates the two prototypes in one application. It is composed of four components as shown in Figure 42: user interface, test generation engine, test optimization engine and common packages. The user interface is responsible for handling the user interactions. The test generation engine implements the processes of the integration test generation approach. The test optimization engine implements the processes of the acceptance test optimization approach. The common package implements common libraries that are used by the two engines. We describe the two prototypes in the following subsections.

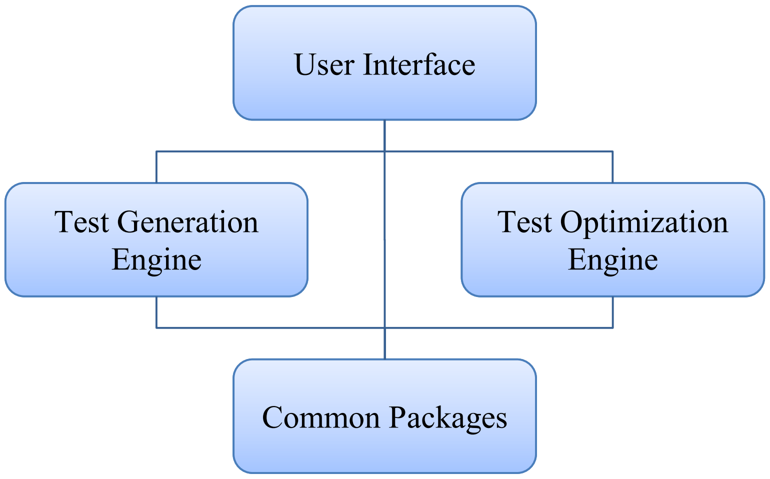


Figure 42. The architecture of the tool: *TestGenO*

6.3.1 Integration Test Model Generator

The prototype accepts multiple test models but it handles two test models at a time. It examines the test models and generates the integration test model, except if there is no interaction between the two SUTs. In that case, it sends a warning message to the user. Furthermore, the tool provides a dialog box to handle test models expressed in mathematical forms.

6.3.1.1 Architecture of the test generation prototype

The prototype is composed of five packages: Main, TMGen, UMLParser, UTPModel, and MathUtilities. The Main package, Figure 43, contains the user interface, and handles the user interactions and file I/O operations. It is the starting point of the application and manages the other packages. This package is part of the tool's user interface.

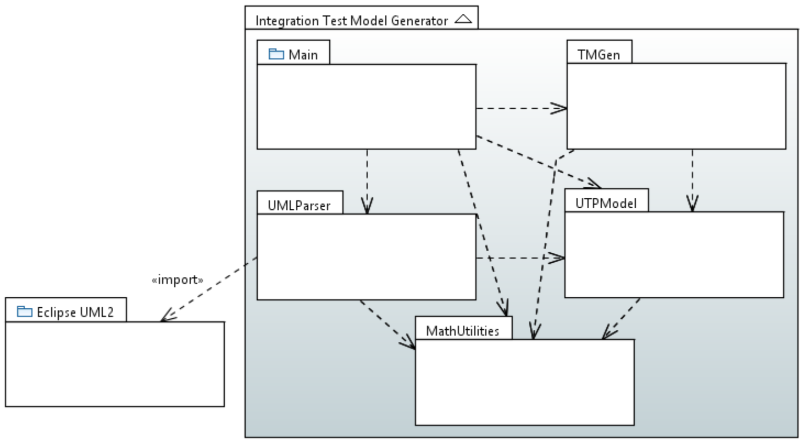


Figure 43. The integration test generation prototype

The TMGen package, Figure 44, implements the test generation approach discussed in Chapter 4. It represents the tool's test generation engine. It composes of the four processes: identification, selection, generation and optimization, as well as the essential methods required by these processes such as the event dependency tree (EDT).

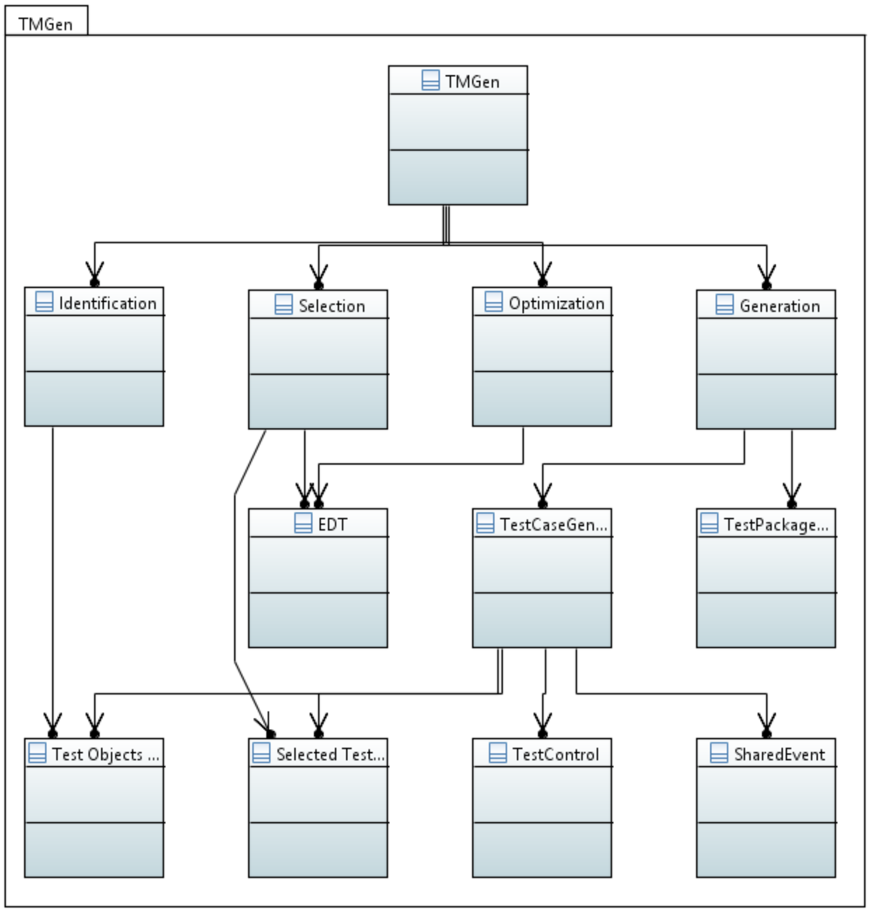


Figure 44. The generation package

The UMLParser package, Figure 45, is responsible for reading, validating and writing UML models. The input UML models have to be in XMI format. The package reads the input model to create an internal test model, *UTPModel*, and validates the structure of the test model as follows:

- It must have one UTP test package.
- It must have one UTP test context.
- There must be at least one SUT.
- There must be at least one test case.
- Instances must be linked to test objects in the test package. It can be by name or through the UML *Represents* attribute.

The package is also responsible for writing the internal test models into serialized UML models. This package is part of the tool's common packages.

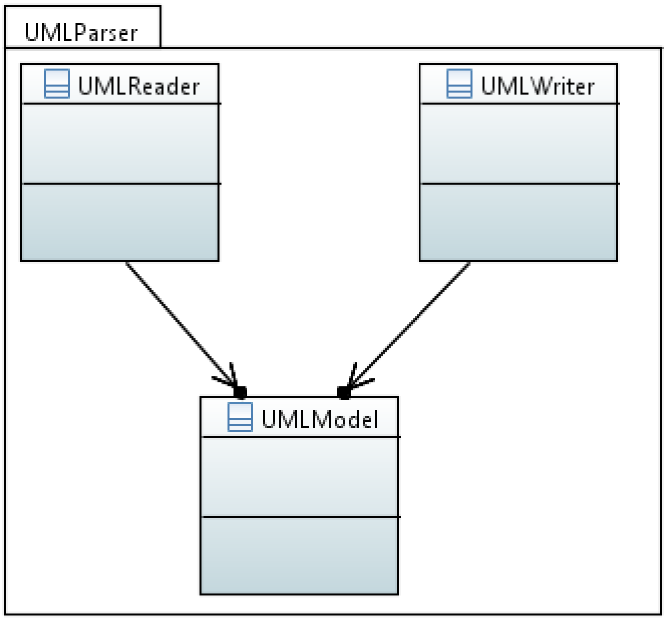


Figure 45. The UMLParser package

The UTPModel package, Figure 46, implements the structure of the test model. It is used by the processes in the TMGen to examine the input test models and to generate the integration test model. This package is part of the tool's common packages.

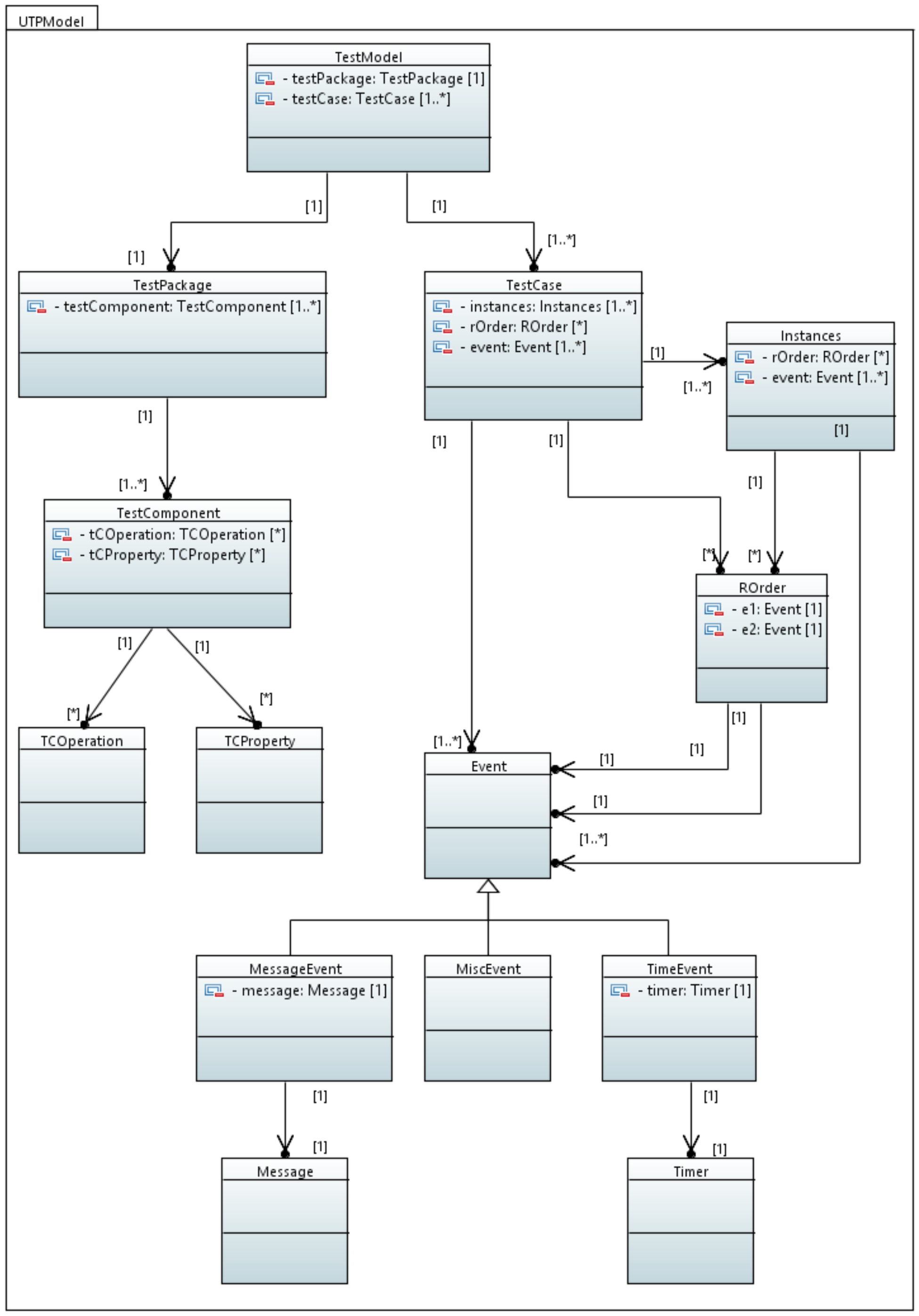


Figure 46. The test model package

The last package MathUtilities, Figure 47, implements essential structures and methods that are required by the other packages. Moreover, the package consists of the implementation of the mathematical representation of our generation approach. This package is part of the tool's common packages.

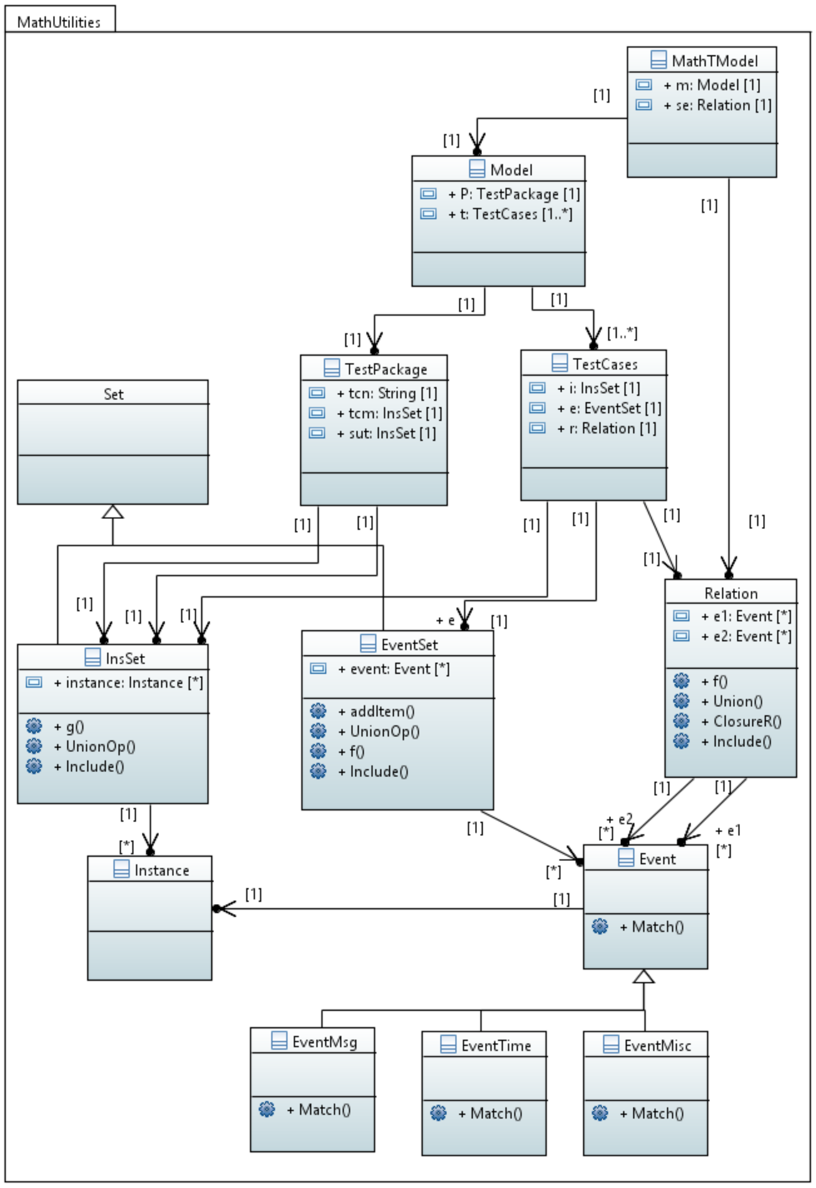


Figure 47. The math & utilities package

6.3.1.2 Limitations of the test generation prototype

The current release implements the essential functionality of the test generation approach. We list here the limitation of the prototype that needs to be implemented to increase the maturity of the tool:

- The prototype does not support UML combined fragments operators except for the sequential and the alternative operators.
- The current release does not support synchronous messages.

6.3.2 Acceptance Test Model Optimizer

The prototype takes multiple test models. Beside the acceptance test model, it accepts all corresponding integration test models. The user must select the generation order of the given integration test models. The prototype examines the integration test cases of the integration test models to select the ones that are suitable to be compared to the acceptance test cases. Subsequently, the prototype maps the acceptance test cases against the selected test cases and eliminates the redundant ones.

6.3.2.1 Architecture of the test optimization prototype

The implementation is composed of five packages: TMain, Optimization, UMLParser, UTPModel, and MathUtilities. The latter three packages are shared with the prototype of the integration test model generator with an upgrade to the MathUtilities package to handle the selection and inclusion methods. The TMain package, Figure 48, consists of the user interface and the file I/O management. It provides a dialog to order the given integration test models according to the corresponding integration strategy. This package is part of the tool's user interface.

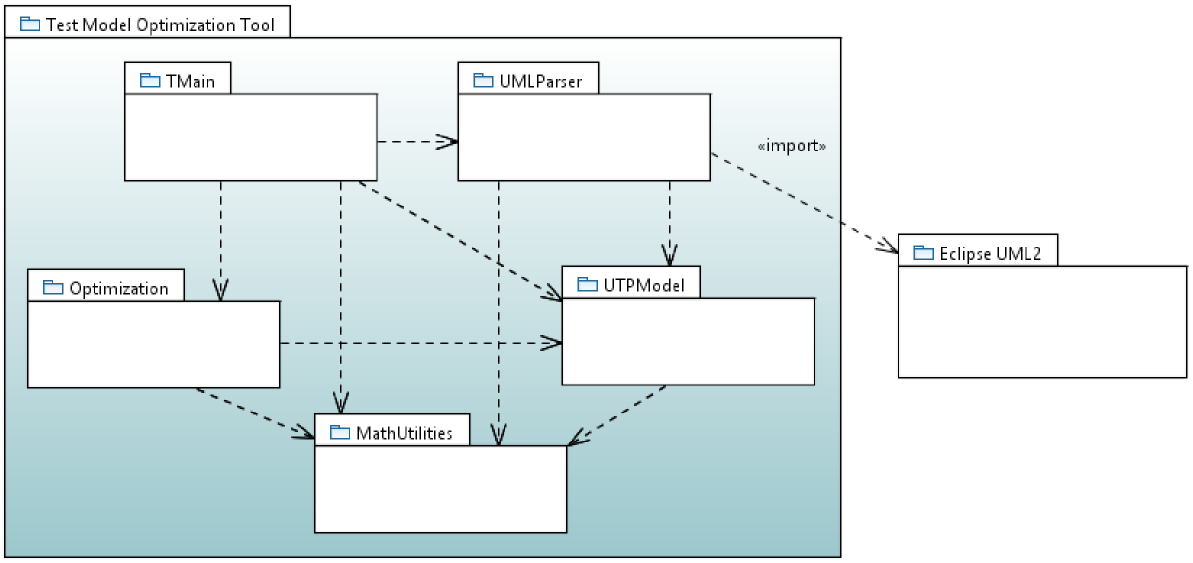


Figure 48. Packages of test optimization tool

The optimization package, Figure 49, implements the selection process and the optimization process as discussed in Chapter 5. This package represents the tool's test optimization engine.

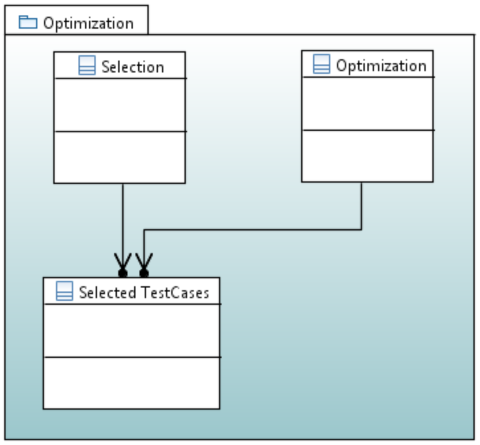


Figure 49. Optimization package

The other three packages were presented in the previous section.

6.3.2.2 Limitations of the test optimization prototype

The prototype implements the essential functionality of the test optimization approach. However, it has the following limitations:

- The prototype does not support UML combined fragments operators except for the sequential and the alternative operators.
- The current release does not support synchronous messages.

6.4 Library System - Case Study

We demonstrate our tool using the library system specified in Appendix B. The system is composed of four components to provide users with essential library services. These services are covered by test cases that have been designed to build component test models as well as the acceptance test model. In this case study, we apply our tool on these test models to generate integration test models. Furthermore, we map the generated test models to the given acceptance test model to reduce the test suite.

We present the given test models in Appendix B. Component test models are described in Section B.2 and the acceptance test model is described in Section B.3. We discuss the results of the test generation in Section 6.4.1. In Section 6.4.2, we discuss the results of the test optimization.

6.4.1 Integration Test Generation

In this section, we apply the tool on the component test models given in Section B.2. We use two different integration orders to build the integration test models. In the first one, we integrate the test models in the following integration order: ((*LibrarianTM* + *MemberTM*) + *MediaTM*) + *BookingTM*. In the second one, we integrate the test models in the following integration order: ((*LibrarianTM* + *MediaTM*) + *BookingTM*) + *MemberTM*. In this section, we focus on the last integration iteration since the intermediate results of the two integration orders are different. A complete generation with the intermediate results is presented in Appendix C.

6.4.1.1 Test Generation Using the First Integration Order

In the first integration order, we integrate the test models in the following integration order: ((*LibrarianTM* + *MemberTM*) + *MediaTM*) + *BookingTM*. The integration goes through three iterations. In the first iteration, we integrate the two component test models *LibrarianTM* and *MemberTM* to generate the integration test model *IntLibMemTM* as shown in Figure 50.

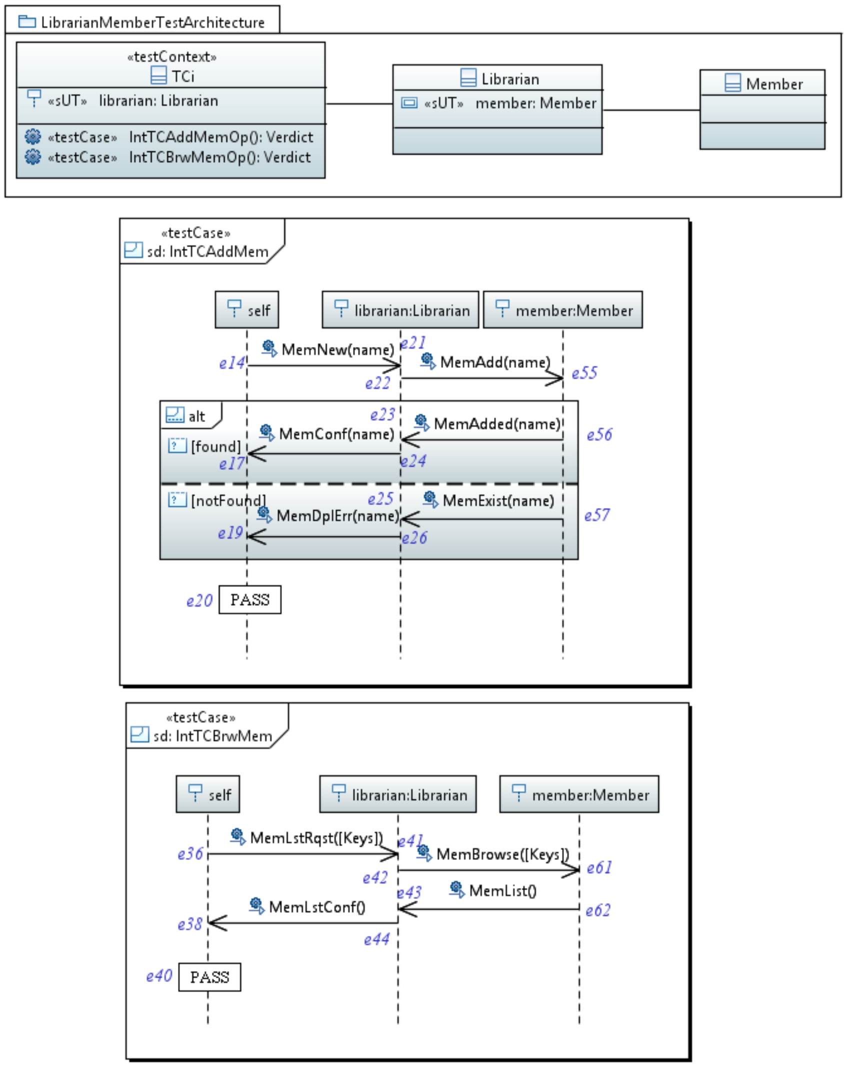


Figure 50. Generated integration test model (*IntLibMemTM*)

In the second iteration, we integrate the integration test model *IntLibMemTM* and the component test model *MediaTM* to generate the integration test model *IntLibMemMedTM* as shown in Figure 51.

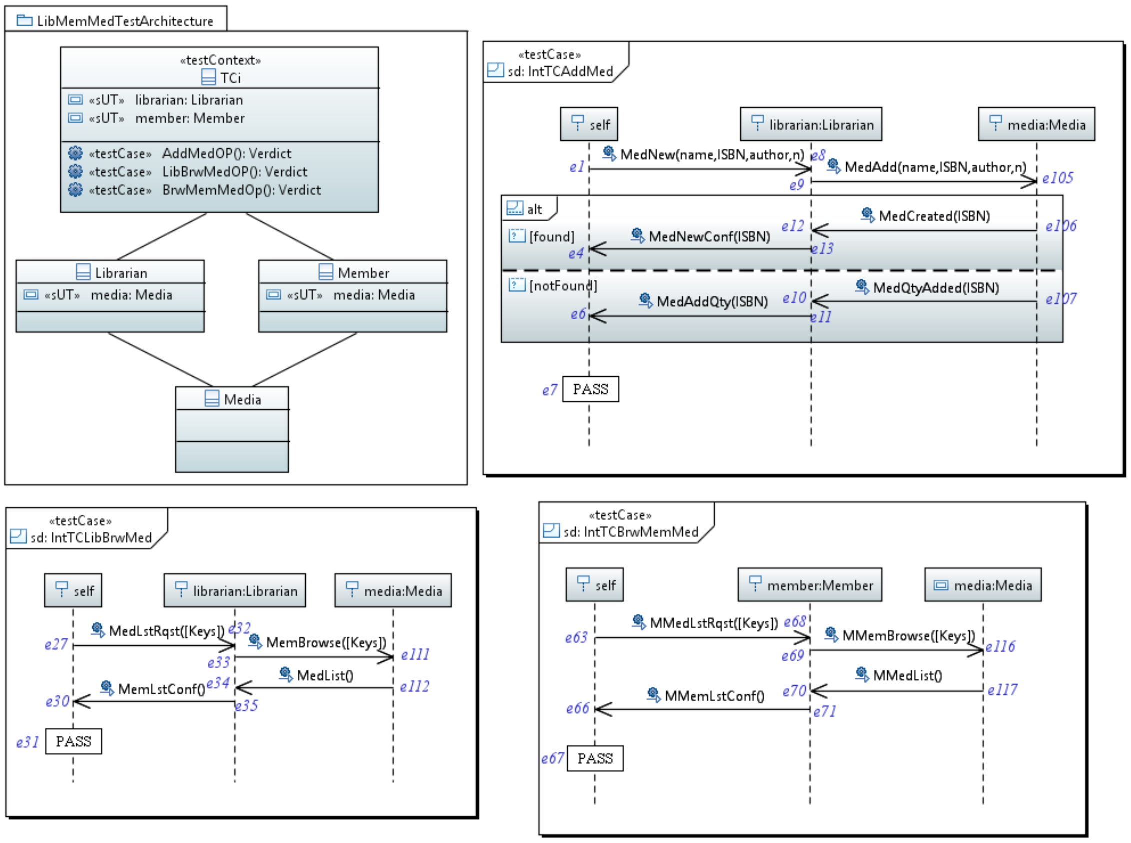

LibMemMedTestArchitecture
«testContext» TCi
«sUT» librarian: Librarian
«sUT» member: Member
«testCase» AddMedOP(): Verdict
«testCase» LibBrwMedOP(): Verdict
«testCase» BrwMemMedOp(): Verdict
Librarian
«sUT» media: Media
Member
«sUT» media: Media
Media
«testCase» sd: IntTCAddMed
self
librarian:Librarian
media:Media
MedNew(name,ISBN,author,n)
MedAdd(name,ISBN,author,n)
alt
[found]
MedCreated(ISBN)
MedNewConf(ISBN)
[notFound]
MedQtyAdded(ISBN)
MedAddQty(ISBN)
PASS
e1 e8 e9 e105 e12 e106 e13 e4 e10 e107 e11 e6 e7
«testCase» sd: IntTCLibBrwMed
self
librarian:Librarian
media:Media
MedLstRqst([Keys])
MemBrowse([Keys])
MedList()
MemLstConf()
PASS
e27 e32 e33 e111 e34 e112 e30 e35 e31
«testCase» sd: IntTCBrwMemMed
self
member:Member
media:Media
MMedLstRqst([Keys])
MMemBrowse([Keys])
MMedList()
MMemLstConf()
PASS
e63 e68 e69 e116 e70 e117 e66 e71 e67

Figure 51. Generated integration test model (*IntLibMemMedTM*)

In the last iteration, we integrate the integration test model *IntLibMemMedTM* and the component test model *BookingTM* to generate the integration test model *IntLibMemMedBkgTM* as shown in Figure 52.

6.4.1.2 Test Generation Using the Second Integration Order

In the second integration order, we integrate the test models in the following integration order: ((*LibrarianTM* + *MediaTM*) + *BookingTM*) + *MemberTM*. The integration goes through three iterations. In the first iteration, we integrate the two component test models *LibrarianTM* and *MediaTM* to generate the integration test model *IntLibMedTM* as shown in Figure 53.

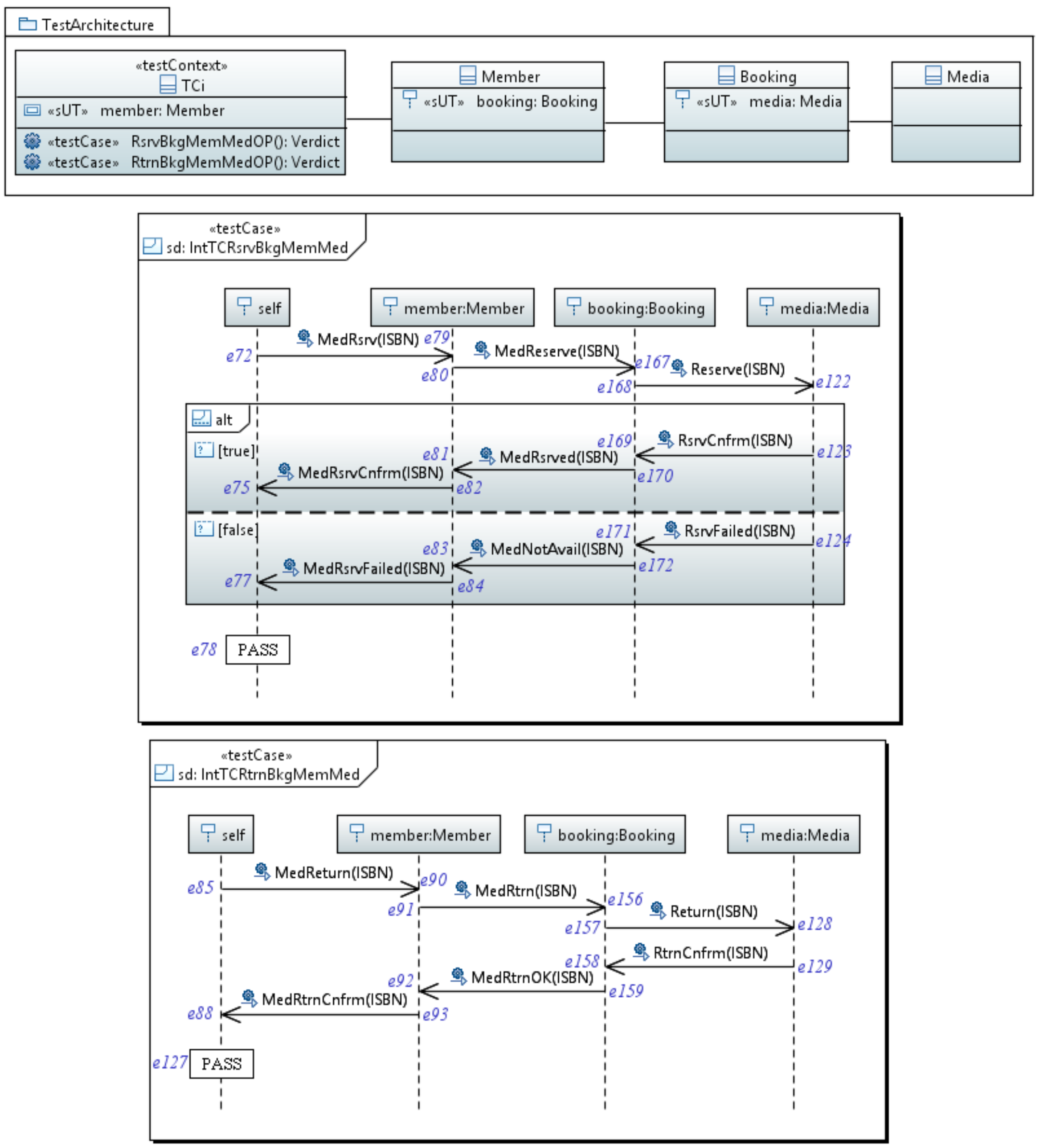


Figure 52. Generated integration test model (*IntLibMemMedBkgTM*)

In the second iteration, we integrate the integration test model *IntLibMedTM* and the component test model *BookingTM* to generate the integration test model *IntLibMedBkgTM* as shown in Figure 54. In the third iteration, we integrate the integration test model *IntLibMedBkgTM* and the component test model *MemberTM* to generate the integration test model *IntLibMedBkgMemTM* as shown in Figure 55.

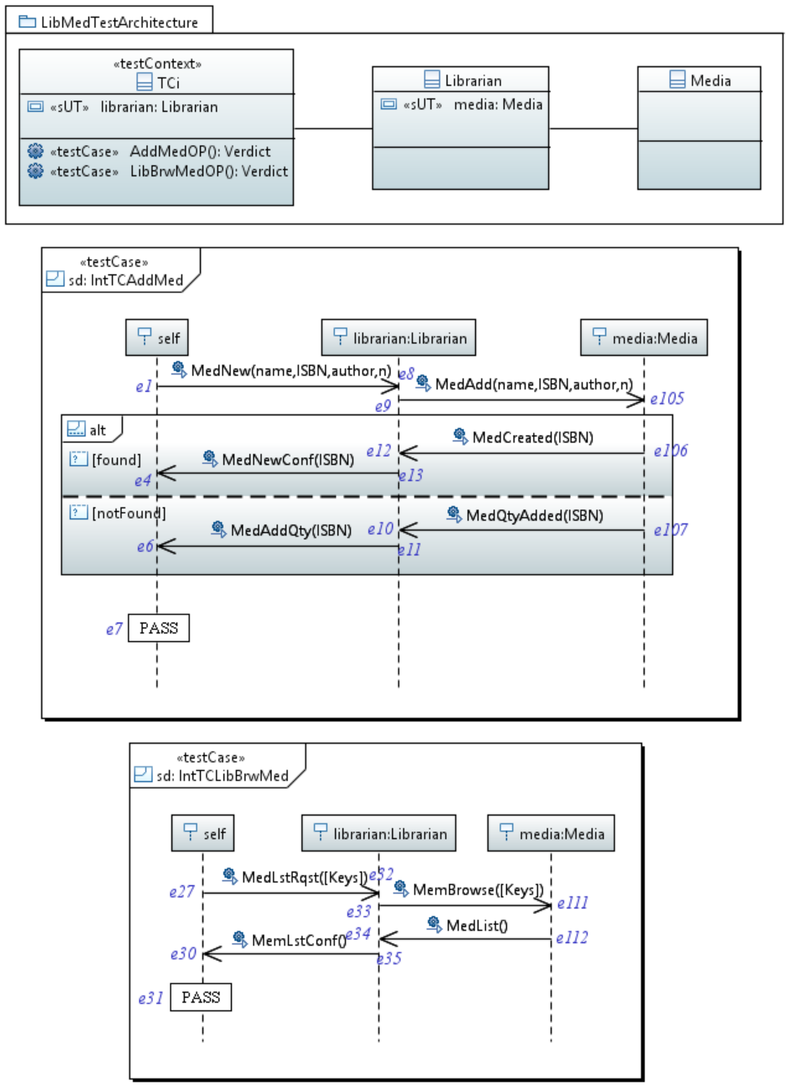


Figure 53. Generated integration test model (*IntLibMedTM*)

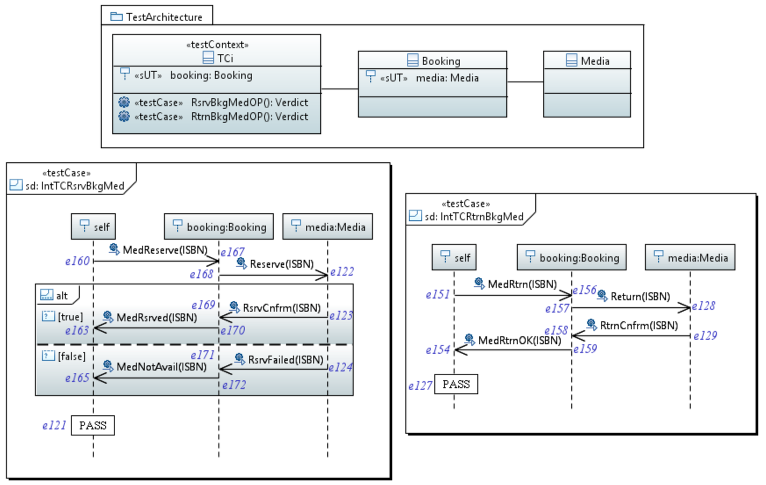


Figure 54. Generated test model (*IntLibMedBkgTM*)

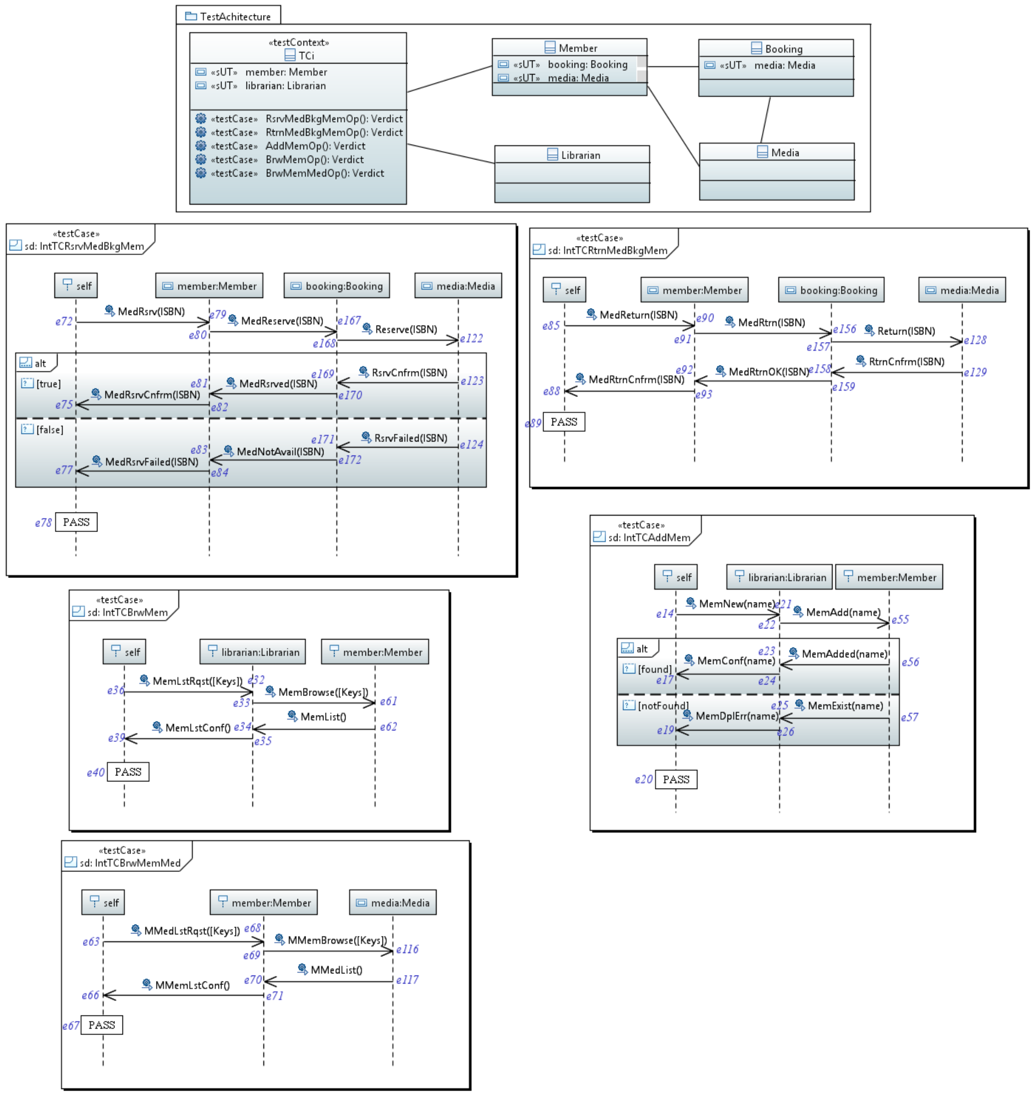


Figure 55. Generated test model (*IntLibMedBkgMemTM*)

6.4.1.3 Discussion

The summaries of the two integration orders are listed in Table 7 and Table 8. In this section, we discuss some issues that are related to the test generation. The first issue is that even though we integrated the same set of components, the tool performed a different number of steps, 10 versus 12, depending on the integration strategy. However, this issue only affects the intermediate results; it does not affect the generated test model. In this case study, the difference comes due to the transition from the second iteration to the third, steps 5 to 6. In the first integration order, the tool could not generate test cases from the previously generated integration test model, *IntLibMemMedTM*. On the other hand, the tool uses the previously generated integration test model, *IntLibMedBkgTM*, to generate two test cases: *IntTCRsrvMedBkgMem* and

IntTCRtrnMedBkgMem, in the second integration order. Therefore, we have two extra steps to examine the generated test cases against the carried-on component test cases, steps 7 & 9.

Table 7. Summary of the first integration order

#	Iteration	Test Integ.	Integrated Test Models		Generated Test Model
1	**1**	**1**	LibrarianTM	MemberTM	T = { IntTCAddMem, IntTCBrwMem } P = (TCi, {}, {Librarian, Member}) IntLibMemTM = (P, T)
2	**2**	**1**	IntLibMemTM	MediaTM	
3		**2**	LibrarianTM	MediaTM	T={IntTCAddMed, IntTCLibBrwMed} P = (TCi, {}, {Librarian, Media}) IntLibMemMedTM = (P, T)
4		**3**	IntLibMemMedTM	MemberTM	
5			MediaTM	MemberTM	T={ IntTCAddMed, IntTCLibBrwMed, IntTCBrwMemMed } P = (TCi, {}, {Librarian, Media, Member }) IntLibMemMedTM = (P, T)
6	**3**	**1**	IntLibMemMedTM	BookingTM	
7		**2**	LibrarianTM	BookingTM	
8		**3**	MemberTM	BookingTM	T={IntTCRsrvBkgMem, IntTCRtrnBkgMem } P = (TCi, {}, {Booking, Member}) IntLibMemMedBkgTM = (P, T)
9		**4**	IntLibMemMedBkgTM	MediaTM	T={IntTCRsrvBkgMemMed, IntTCRtrnBkgMemMed } P = (TCi, {}, {Booking, Member, Media}) IntLibMemMedBkgTM = (P, T)
10			BookingTM	MediaTM	

The next issue is that the generated integration test models completely cover the interfaces and services of the currently integrated component with the sub-system, steps 1, 5 and 10 in Table 7 and steps 1, 3 and 10 in Table 8. To illustrate, the generated test model IntLibMemMedTM, Table 7 step 5, is composed of three test cases: IntTCAddMed, IntTCLibBrwMed and IntTCBrwMemMed. In this iteration, we are integrating the Media component to the sub-system that is composed of Librarian and Member. The first two test cases cover the two services provided through the interface between the Librarian and the Media, and the third test case covers the service provided through the interface between the Member and the Media.

Saving test information is the next issue. We have delayed the integration of the *Member*, which has interfaces with all other components, to the last iteration in the second integration order. Successfully, the approach generated and updated test cases that cover the five services processed by this component.

Table 8. Summary of the second integration order

#	Iteration	Test Integ.	Integrated Test Models		Generated Test Model
1	1	1	LibrarianTM	MediaTM	T = { IntTCAddMed, IntTCBrwMed } P = (TCi, {}, {Librarian, Media}) IntLibMedTM = (P, T)
2	2	1	IntLibMedTM	BookingTM	
3		2	MediaTM	BookingTM	T={IntTCRsrvMedia,IntTCRtrnMedia} P = (TCi, {}, {Booking, Media}) IntLibMedBkgTM = (P, T)
4		3	IntLibMedBkgTM	LibrarianTM	
5			BookingTM	LibrarianTM	
6	3	1	IntLibMedBkgTM	MemberTM	T={IntTCRsrvMedBkgMem, IntTCRtrnMedBkgMem } P = (TCi, {}, {Booking, Media, Member}) IntLibMedBkgMemTM = (P, T)
7		2	IntLibMedBkgMemTM	LibrarianTM	
8			MemberTM	LibrarianTM	T={IntTCRsrvMedBkgMem, IntTCRtrnMedBkgMem, IntTCAddMem, IntTCBrwMem} P = (TCi, {}, {Booking, Media, Member, Librarian}) IntLibMedBkgMemTM = (P, T)
9		3	IntLibMedBkgMemTM	MediaTM	
10			MemberTM	MediaTM	T={IntTCRsrvMedBkgMem, IntTCRtrnMedBkgMem, IntTCAddMem, IntTCBrwMem, IntTCBrwMemMed } P = (TCi, {}, {Booking, Media, Member, Librarian}) IntLibMedBkgMemTM = (P, T)
11		4	IntLibMedBkgMemTM	BookingTM	
12			MemberTM	BookingTM	

The next issue is that the approach does not alter the behavior of the CUTs. The approach works on the test controls and test stubs, and on restoring events that belongs to the CUTs.

The next issue is that some test integrations do not generate test behavior: e.g. Table 7 steps 2, 6 & 7. This issue depends on the integration strategy and the tool's on-the-fly optimization. In the implementation, we embedded the redundancy removal process within the generation process to save memory space.

The last issue is that the tool produces complete sets of test cases that cover all specified services in the two integration orders. These test cases are applied on different iterations but without the use of implicit or explicit emulation of system components. Two test cases, step 3 in Table 8, were generated when the test control emulates the component *Member* but they were updated in the next iteration, step 10, when the real component was integrated.

6.4.2 Acceptance Test Optimization

In this section, we apply the tool on the acceptance test model, given in Section B.3, and the generated integration test models from the previous section. We apply the tool twice since we have two different sets of integration test models produced from two integration orders. In each integration order, we have three integration test models corresponding to the three integration iterations.

6.4.2.1 Test Optimization Using the Generated Integration Test Models in the First Integration Order

Let us start by generating the test models of the first integration order. The tool examines the test cases of the integration test models to select the ones that do not emulate system components. Table 9 presents the summary of the selection process. Test cases of the last integration test model *IntLibMemMedBkgTM* are automatically selected since they are applied on a complete system and they should not require test stubs of system components. These test cases are mapped to the test cases of preceding test models: *IntLibMemTM*, *IntLibMemMedTM*. Test cases of the first model *IntLibMemTM* are also mapped to the test cases of the second test model *IntLibMemMedTM* in order to examine if any test case emulates its CUT, *Media*.

Table 9. Selection summary of first integration order

Integration TM	Mapped to	Results
IntLibMemTM	IntLibMemMedTM	passed to the next mapping
IntLibMemTM	IntLibMemMedBkgTM	2/2 test cases are selected
IntLibMemMedTM	IntLibMemMedBkgTM	3/3 test cases are selected
IntLibMemMedBkgTM	N/A	2/2 test cases are selected

All test cases of the three integration test models are selected to be mapped to the test cases of the acceptance test model. Hence, the tool maps test cases of the acceptance test model to the selected integration test cases. Each acceptance test case is mapped to seven integration test cases as shown in Table 10. However, the mapping process, for any acceptance test case, terminates as soon as the acceptance test case is included in the specification of the currently compared integration test case.

Table 10. Mapping results of first integration order

#	Acceptance test case	Integration		Result
		Test Model	Test Case	
1	TestCaseAddMember	IntLibMemTM	IntTCAddMem	**Included**
2	TestCaseAddMedia	IntLibMemTM	IntTCAddMem	Passed
3			IntTCBrwMem	Passed
4		IntLibMemMedTM	IntTCAddMed	**Included**
5	TestCaseBrowseMembers	IntLibMemTM	IntTCAddMem	Passed
6			IntTCBrwMem	**Included**
7	TestCaseLibrarianBrowseMedia	IntLibMemTM	IntTCAddMem	Passed
8			IntTCBrwMem	Passed
9		IntLibMemMedTM	IntTCAddMed	Passed
10			IntTCLibBrwMed	**Included**
11	TestCaseMemberBrowseMedia	IntLibMemTM	IntTCAddMem	Passed
12			IntTCBrwMem	Passed
13		IntLibMemMedTM	IntTCAddMed	Passed
14			IntTCLibBrwMed	Passed
15			IntTCBrwMemMed	**Included**
16	TestCaseReserveMedia	IntLibMemTM	IntTCAddMem	Passed
17			IntTCBrwMem	Passed
18		IntLibMemMedTM	IntTCAddMed	Passed
19			IntTCLibBrwMed	Passed
20			IntTCBrwMemMed	Passed
21		IntLibMemMedBkgTM	IntTCRsrvBkgMemMed	**Included**
22	TestCaseReturnMedia	IntLibMemTM	IntTCAddMem	Passed
23			IntTCBrwMem	Passed
24		IntLibMemMedTM	IntTCAddMed	Passed
25			IntTCLibBrwMed	Passed
26			IntTCBrwMemMed	Passed
27		IntLibMemMedBkgTM	IntTCRsrvBkgMemMed	Passed
28			IntTCRtrnBkgMemMed	**Included**

Consequently, the acceptance test case is removed from the acceptance test model. Acceptance test cases that are not included in the seven integration test cases are kept in the acceptance test model. Table 10 presents the summary of the mapping process. The complete set of test cases in the acceptance test model is included in the selected test cases of the integration test models. Hence, the acceptance testing, in this case study, is skipped. We refer that to the selection of the same set of services on both integration and acceptance testing, which is not always the case in most test projects.

6.4.2.2 Test Optimization Using the Generated Integration Test Models in the Second Integration Order

Now, let us move to the generated test models of the second integration order. The tool examines the test cases of the integration test models to select the ones that do not emulate system components. Table 11 presents the summary of the selection process. Test cases of the last integration test model *IntLibMedBkgMemTM* are automatically selected since they are applied on a complete system and they should not require test stubs of system components. These test cases are used, as a reference, to be mapped to the test cases of preceding test models: *IntLibMedBkgTM*, *IntLibMedTM*. Test cases of the first model *IntLibMedTM* are also mapped to the test cases of the second test model *IntLibMedBkgTM* in order to examine if a test case emulates its CUT, *Booking*.

Table 11. Selection summary of the second integration order

Integration TM	Mapped to	Results
IntLibMedTM	IntLibMedBkgTM	passed to the next mapping
IntLibMedTM	IntLibMedBkgMemTM	2/2 test cases are selected
IntLibMedBkgTM	IntLibMedBkgMemTM	0/2 test cases are selected. The test control in both test cases emulates the component *Member*
IntLibMedBkgMemTM	N/A	5/5 test cases are selected

The two test cases of the second integration test model, *IntLibMedBkgTM*, are not selected since they emulate the system component *Member,* which is integrated in the third iteration. The rest of integration test cases are selected to be mapped to the test cases of the acceptance test model. Hence, the tool maps test cases of the acceptance test model to the selected integration test cases. Each acceptance test case is mapped to the seven selected test cases as shown in Table 12. However, the mapping process, for any acceptance test case, terminates as soon as the acceptance test case is included in the specification of the currently compared integration test case. Consequently, the acceptance test case is removed from the acceptance test model. Acceptance test cases that are not included in the seven selected test cases are left in the acceptance test model.

Table 12. Mapping results of the second integration order

#	Acceptance test case	Integration		Result
		Test Model	Test Case	
1	TestCaseAddMember	IntLibMedTM	IntTCAddMed	Passed
2			IntTCLibBrwMed	Passed
3		IntLibMedBkgMemTM	IntTCRsrvMedBkgMem	Passed
4			IntTCRtrnMedBkgMem	Passed
5			IntTCAddMem	**Included**
6	TestCaseAddMedia	IntLibMedTM	IntTCAddMed	**Included**
7	TestCaseBrowseMembers	IntLibMedTM	IntTCAddMed	Passed
8			IntTCLibBrwMed	Passed
9		IntLibMedBkgMemTM	IntTCRsrvMedBkgMem	Passed
10			IntTCRtrnMedBkgMem	Passed
11			IntTCAddMem	Passed
12			IntTCBrwMem	**Included**
13	TestCaseLibrarianBrowseMedia	IntLibMedTM	IntTCAddMed	Passed
14			IntTCLibBrwMed	**Included**
15	TestCaseMemberBrowseMedia	IntLibMedTM	IntTCAddMed	Passed
16			IntTCLibBrwMed	Passed
17		IntLibMedBkgMemTM	IntTCRsrvMedBkgMem	Passed
18			IntTCRtrnMedBkgMem	Passed
19			IntTCAddMem	Passed
20			IntTCBrwMem	Passed
21			IntTCBrwMemMed	**Included**
22	TestCaseReserveMedia	IntLibMedTM	IntTCAddMed	Passed
23			IntTCLibBrwMed	Passed
24		IntLibMedBkgMemTM	IntTCRsrvMedBkgMem	**Included**
25	TestCaseReturnMedia	IntLibMedTM	IntTCAddMed	Passed
26			IntTCLibBrwMed	Passed
27		IntLibMedBkgMemTM	IntTCRsrvMedBkgMem	Passed
28			IntTCRtrnMedBkgMem	**Included**

Table 12 presents the summary of the mapping process. The complete set of test cases in the acceptance test model is included in the selected test cases from the given test models. Hence, the acceptance testing, in this case study, is skipped. We associate that to the selection of the same set of services on both integration and acceptance testing, which is not always the case in most test projects.

6.5 Discussion

The tool generated integration test models, which cover all of the system services, for both integration orders. It also optimized the acceptance test model by removing test cases that exercised during the integration testing without emulation of system components. These results are similar to what we had experienced with other handcrafted case studies during our research. The results of the test generation are summarized in Table 13. The tool integrated four test models through three iterations. The tool generated the same test behavior in both integration orders. The generated test cases cover the seven specified services. That means, we covered 100% of the specified system functionality. Two test cases have been repeated in the second integration order since they emulated a system component in the second iteration.

Table 13. Test generation results

Iteration	Integrated Components	1st Integration Order Generated Test Cases	2nd Integration Order Generated Test Cases
1	**2**	**2**	**2**
2	**3**	**3**	**~~2~~**
3	**4**	**2**	**5**
Total		**7**	**7 + ~~2~~**

The results of test optimization are summarized in Table 14. The tools selected the seven test cases that do not emulate system components, and excluded the two test cases of the second integration order that emulate a system component. Furthermore, the complete acceptance test cases are removed since they matched the selected test cases. Hence, engineers do not need to execute the given acceptance test model during the acceptance-level testing for this particular case study.

This case study is used to demonstrate the functionalities of our tool. Further experiments using industrial case studies should be performed with our tool. While we presented sequentially the processes of the two approaches, we had merged some processes in our implementation. First, the identification process and the selection process of the test generation approach are executed together on the given test models since the detection of shared test objects in specific test cases elected such test cases to be selected too. Second, the generation process and the optimization process of the test generation approach are also combined to operate at the same time on the given test cases. This combination should prevent the construction of redundant test cases. Finally, the selection process of the optimization approach is embedded in the generation approach to detect immediately test cases that emulate CUTs.

Table 14. Test optimization results

	1st Integration Order # Test Cases	2nd Integration Order # Test Cases
Integration Test Models	**7**	**9**
Selected Test Cases	**7**	**7**
Acceptance Test Model	**7**	**7**
Excluded Test Cases	**7**	**7**
Optimized Acceptance Test Model	**0**	**0**

Chapter 7
Conclusion and Future work

7.1 Conclusion

Software testing is a critical activity in the software development process. In this dissertation, we proposed a model based testing framework that relates and links different software testing levels with enabled collaboration, automation, reusability and optimization. Two approaches have been proposed in this framework: test generation and test optimization. In order to apply these approaches, component test cases must be well-formed and must cover all component interfaces and services.

To conduct our research in a rigorous manner, we used UML sequence diagrams, which have been formally investigated [33-35], to build our test behavior. Test models are specified using UML Testing Profile, which enables systematic transformation of the test models into test code that is exercised on the IUT using well-known test execution environments, such as JUnit and TTCN-3. Using standard notations enhances the collaboration and bridges the gap between the development and testing activity. In contrary to the general software research stream, our research is dedicated to bridge the gap between the software testing levels.

The framework enables reusability across the software testing levels. Test models are systematically generated from preceding test models. We discussed the test generation approach through the generation of integration test models from component test models. We defined a test case merging operator to integrate component test cases that have a shared behavior. We have implemented a prototype and demonstrated it on a case study.

Our framework also enables systematic test optimization across the software testing levels. Test models are related to preceding test models to remove the ones that have already been exercised. Test optimization reduces the size of the test models, shortens the test's execution time and reduces the cost of the software testing. We proposed an acceptance test optimization approach that optimizes the acceptance test model by relating it to the integration test models. This approach can be applied to the system test model without any modification. A prototype has been implemented and demonstrated on a case study.

7.2 Future Work

The goal of this research is to contribute toward the reusability and optimization across the software testing levels in the software process. Several issues remain open. In this subsection, we point to several of these issues:

- Test model:
    - We have worked on a subset of the proposed UTP test model. We focused on the main parts of the UTP model: test package and test cases. The UTP test package defines the test structure in details. Test cases specify the test behavior. We have left-out two parts of the UTP model: test configuration and test architecture. UTP test configurations work as test case setups and define the initial number of instances of test objects and their connections at the start of a test case. UTP test architecture describes the test package at high-level of abstractions. Our approach can be extended to include such parts.
- Test generation approach:
    - We discussed the outlines of the generation of the system test model from the component test models. Further investigation is required. We believe that it can be embedded into our approach to generate both integration and system test models at the same time. During each integration iteration, the system test model is enriched with test cases, and at the final iteration, the system test model will be fully constructed.
- Test optimization approach:
    - We believe that the optimization of the acceptance test model by relating it to the system test model is simpler than relating it to the integration test models. The two testing levels, system and acceptance, work on complete systems. The test models have similar test architecture: test control and SUT. Further investigations for optimizing acceptance test models using system test models can be undertaken.

7.3 Publications from the Thesis

The following research papers have been generated from this thesis.

[1] Mussa, M., Khendek, F.: "Towards a Model Based Approach for Integration Testing". In Ober, I., Ober, I. (eds.) *SDL 2011: Integrating System and Software Modeling*, LNCS 7083, pp. 106-121, Springer Berlin Heidelberg, 2012.

[2] Mussa, M., Khendek, F.: "Identification and Selection of Interaction Test Scenarios for Integration Testing". In Haugen, Ø., Reed, R., Gotzhein, R. (eds.) *SAM2012: System Analysis and Modeling: Theory and Practice*, LNCS 7744, pp. 16-33, Springer Berlin Heidelberg, 2013.

[3] Mussa, M., Khendek, F.: "Merging Test Models". In 2013 18th International Conference on Engineering of Complex Computer Systems (ICECCS). pp. 167-170, IEEE, 2013.

[4] Mussa, M., Khendek, F.: "Acceptance test optimization". In Amyot, D., Fonseca, P., Casas, i., Mussbacher, G. (eds.) *System Analysis and Modeling: Models and Reusability, SAM2014*. LNCS 8769, pp. 158-173. Springer International Publishing, 2014.

References

[1] Schmidt, R.: Software Engineering: Architecture-Driven Software Development. Elsevier, Amsterdam, 2013.

[2] Tretmans, J., Brinksma, E.: “TorX: Automated Model Based Testing”. In Hartman, A., Dussa-Zieger, K. (eds.) *First European Conference on Model-Driven Software Engineering*. pp. 13. Imbuss, Möhrendorf, Germany, 2003.

[3] Myers, G. J., Sandler, C., Badgett, T.: The Art of Software Testing. John Wiley & Sons, Hoboken, N.J., 2012.

[4] Bertolino, A.: “Software Testing Research: Achievements, Challenges, Dreams”. In *2007 Future of Software Engineering*. pp. 85-103. IEEE Computer Society, Washington, DC, USA, 2007.

[5] Baker, P., Dai, Z. R., Grabowski, J., Haugen, Ø., Schieferdecker, I., Williams, C.: Model-Driven Testing : Using the UML Testing Profile. Springer Berlin Heidelberg, 2008.

[6] Aichernig, B. K., Lorber, F., Tiran, S.: “Integrating Model-Based Testing and Analysis Tools Via Test Case Exchange”. In *Theoretical Aspects of Software Engineering (TASE), 2012 Sixth International Symposium on*. pp. 119-126. IEEE, 2012.

[7] Grossmann, J., Fey, I., Krupp, A., Conrad, M., Wewetzer, C., Mueller, W.: "TestML - A test exchange language for model-based testing of embedded software". In Broy, M., Krüger, I., Meisinger, M. (eds.) *Model-Driven Development of Reliable Automotive Services*. pp. 98-117. Springer Berlin / Heidelberg, 2008.

[8] Ammann, P., Offutt, J.,: Introduction to Software Testing. Cambridge University Press, New York, 2008.

[9] Shirole, M., Kumar, R.: "UML behavioral model based test case generation: A survey". *SIGSOFT Softw.Eng.Notes*, vol. 38, pp. 1-13, july 2013.

[10] Hutchinson, J., Whittle, J., Rouncefield, M., Kristoffersen, S.: “Empirical Assessment of MDE in Industry”. In *Proceeding of the 33rd International Conference on Software Engineering*. pp. 471-480. ACM, New York, NY, USA, 2011.

[11] Schmidt, D. C.: "Guest editor's introduction: Model-driven engineering," *Computer,* vol. 39, pp. 25-31, February, 2006.

[12] Mellor, S. J.: MDA Distilled: Principles of Model-Driven Architecture. Addison-Wesley, Boston, 2004.

[13] Mellor, S. J., Balcer, M. J.: Executable UML: A Foundation for Model-Driven Architecture. Boston, 2002.

[14] Utting, M., Pretschner, A., Legeard, B.: "A taxonomy of model-based testing approaches". Softw.Test.Verif.Reliab., vol. 22, pp. 297-312. Aug, 2012.

[15] Ulrich, A.: "Introducing model-based testing techniques in industrial projects". Software Engineering (Workshops). 2007. (Available: http://subs.emis.de/LNI/Proceedings/Proceedings106/gi-proc-106-002.pdf, last accessed: 2014)

[16] OMG: Unified modeling language. 2014. (Available: http://www.uml.org, last accessed: 2014).

[17] OMG: Object management group. 2014. (Available: http://www.omg.org, last accessed: 2014).

[18] OMG: UML testing profile (UTP), version 1.2, (formal/2013-04-03). 2013. (Available: http://www.omg.org/spec/UTP/1.2, last accessed: 2014).

[19] Liang, D., Xu, K.: "Test-Driven Component Integration with UML 2.0 Testing and Monitoring Profile". In *7th International Conference on Quality Software, QSIC 2007*, pp. 32-39. IEEE Computer Society, Washington, DC, USA, 2007.

[20] Lamancha, B. P., Mateo, P. R., de Guzmán, I. R., Usaola, M. P., Velthius, M. P.: "Automated Model-Based Testing using the UML Testing Profile and QVT". In *Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation*. pp. 6:1-6:10. ACM, New York, NY, USA, 2009.

[21] Baker, P., Jervis, C.: "Testing UML2.0 models using TTCN-3 and the UML2.0 testing profile". In Gaudin, E., Najm, E., Reed, R. (eds.) *SDL 2007: Design for Dependable Systems*. pp. 86-100. Springer Berlin / Heidelberg, 2007.

[22] Busch, M., Chaparadza, R., Dai, Z. R., Hoffmann, A., Lacmene, L., Ngwangwen, T., Ndem, G. C., Ogawa, H., Serbanescu, D., Schieferdecker, I., Zander-Nowicka, J.: "Model Transformers for Test Generation from System Models". In *Proceedings of Conquest 2006, 10th International Conference on Quality Engineering in Software Technology*, September 2006. pp. 1-16. Hanser Verlag, 2006.

[23] Pietsch, S., Stanca-Kaposta, B.: "Model-based testing with UTP and TTCN-3 and its application to HL7," *Testing Technologies IST GmbH, Conquest*. Potsdam, Germany, 2008.

[24] Iyenghar, P.: "Test Framework Generation for Model-Based Testing in Embedded Systems". In *Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on*. pp. 267-274. IEEE, 2011.

[25] Iyenghar, P., Pulvermueller, E., Westerkamp, C.: "Towards Model-Based Test Automation for Embedded Systems using UML and UTP". In *Emerging Technologies Factory Automation (ETFA), 2011 IEEE 16th Conference on*. pp. 1-9. IEEE, 2011.

[26] Jingyue, Li, Slyngstad, O. P. N., Torchiano, M., Morisio, M., Bunse, C.: "A state-of-the-practice survey of risk management in development with off-the-shelf software components". *Software Engineering, IEEE Transactions on*, vol. 34, pp. 271-286, March, 2008.

[27] Budhija, N., Ahuja, S. P.: "Review of software reusability," In *International Conference on Computer Science and Information Technology (ICCSIT'2011)*. pp. 113-115. Pattaya Dec, 2011.

[28] Babu, G. N. K. S., Srivatsa, D. S. K.: "Analysis and measures of software reusability". In *IJRIC*. pp. 41-46. 2009. (Available: http://www.ijric.org/volumes/Vol1/5Vol1.pdf, last accessed: 2014).

[29] Biggerstaff, T. J., Perlis, A. J.: Software Reusability. ACM Press, New York, N.Y., 1989.

[30] JUnit. 2014. (Available: http://www.junit.org, last accessed: 2014).

[31] Willcock, C.: Introduction to TTCN-3. West Sussex, England; Hoboken, NJ, 2005.

[32] ETSI. Testing and test control notation, version 3 (TTCN-3). 2011. (Available: http://www.ttcn-3.org, last accessed: 2014).

[33] ITU-T Recommendation: Z.120, "Message Sequence Charts (MSC)". Geneva, Switzerland, 1999.

[34] Lund, M., Stølen, K.: "A fully general operational semantics for UML 2.0 sequence diagrams with potential and mandatory choice". In Misra, J., Nipkow, T., Sekerinski, E. (eds.) *FM 2006: Formal Methods*. pp. 380-395. Springer Berlin Heidelberg, 2006.

[35] Xiaoshan, Li, Zhiming, Liu, He, Jifeng: "A Formal Semantics of UML Sequence Diagram". In *Proceedings of Software Engineering Conference*. pp. 168-177. Australian, 2004.

[36] Jazequel, J. M., Meyer, B.: "Design by contract: The lessons of ariane," *Computer*, vol. 30, pp. 129-130, Jan., 1997.

[37] Takanen, A., Demott, J. D., Miller, C.: "Fuzzing for Software Security Testing and Quality Assurance". Artech House, 2008.

[38] ITU-T Recommendation: Z.100, "specification and description language (SDL)". 2007. (Available: http://www.itu.int/rec/T-REC-Z.100/en, last accessed: 2014).

[39] ISO: "ISO/IEC 13568 (2002), Information Technology -- Z Formal Specification Notation---Syntax, Type System and Semantics, ISO/IEC. First Edition." 2002.

[40] Wang, Z., Li, B., Wang, L., Li, Q.: "A Brief Survey on Automatic Integration Test Order Generation". In *SEKE 2011 - Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering*, July 7, 2011 - July 9. pp. 254-257. Knowledge Systems Institute Graduate School, Miami, FL, United states, 2011.

[41] Abdurazik, A., Offutt, J.: "Using coupling-based weights for the class integration and test order problem". *The Computer Journal*, vol. 52, pp. 557-570, Aug, 2009.

[42] Briand, L. C., Labiche, Y., Wang, Y.: "An investigation of graph-based class integration test order strategies," *IEEE Transactions on Software Engineering*, vol. 29, pp. 594-607, 2003.

[43] Agarwal, B. B., Tayal, S. P., Gupta, M.: Software Engineering & Testing :An Introduction. Jones and Bartlett Pub.. Sudbury, Mass., 2010.

[44] Valmari, A.: "The State Explosion Problem". In *Advances in Petri Nets*. pp. 429-528. Spriner-Verlag, Berlin, Germany, 1998.

[45] Yuang, M. C.: "Survey of Protocol Verification Techniques Based on Finite State Machine Models". In *Proceedings of the Computer Networking Symposium*. pp. 164-172. IEEE. 1988.

[46] OMG: OMG Model Driven Architecture (MDA). 2014. (Available: http://www.omg.org/mda, last accessed: 2014).

[47] Mens, T., Van Gorp, P.: "A taxonomy of model transformation". *Electronic Notes in Theoretical Computer Science*. vol. 152, pp. 125-142. Mar., 2006.

[48] Kurtev, I.: "State of the art of QVT: A model transformation language standard". In *Applications of graph transformations with industrial relevance*. pp. 377-393. Springer Berlin Heidelberg, 2008.

[49] Guldali, B., Mlynarski, M., Sancar, Y.: "Effort Comparison for Model-Based Testing Scenarios". In *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*. pp. 28-36. IEEE. 2010.

[50] Kleppe, A. G., Warmer, J. B., Bast, W.: MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley, Boston, MA, 2003.

[51] Stephan, M., Cordy, J. R.: "Application of Model Comparison Techniques to Model Transformation Testing". In *1st International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2013*, February 19, 2013 - February 21. pp. 307-311. INSTICC Press, Barcelona, Spain, 2013.

[52] Kolovos, D. S., Paige, R. F., Polack, F. A. C.: "Model Comparison: A Foundation for Model Composition and Model Transformation Testing". In *Proceedings of the 2006 International Workshop on Global Integrated Model Management*. pp. 13-20. ACM, New York, NY, USA, 2006.

[53] OMG: Unified Modeling Language (UML), infrastructure specification, version 2.2, (formal/2009-02-04). 2009. (Available: http://www.omg.org/spec/UML/2.2/Infrastructure, last accessed: 2014).

[54] OMG: Systems Modeling Language (SysML). 2010. (Available: http://www.omgsysml.org, last accessed: 2014).

[55] Rehman, M. J., Jabeen, F., Bertolino, A., Polini, A.: "Testing software components for integration: A survey of issues and techniques". *Software Testing Verification and Reliability*, vol. 17, pp. 95-133, 2007.

[56] Dias-Neto, A. C., Travassos, G. H.: "Evaluation of {Model-Based} Testing Techniques Selection Approaches: An External Replication". In *Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on*. pp. 269-278. IEEE Computer Society, 2009.

[57] Vidal, J., de Lamotte, F., Gogniat, G., Soulard, P., Diguet, J. P.: "A Co-Design Approach for Embedded System Modeling and Code Generation with UML and MARTE". In *2009 Design, Automation & Test in Europe Conference & Exhibition (DATE'09)*. pp. 6. IEEE, Piscataway, NJ, USA, 2009.

[58] Krishnan, P., Pari-Salas, P.: "Model-Based Testing and the UML Testing Profile". *Semantics and Algebraic Specification*, pp. 315-328. Springer Berlin Heidelberg, 2009.

[59] Hou, X., Wang, Y., Zheng, H., & Tang, G.: "Integration Testing System Scenarios Generation Based on UML". In *Computer, Mechatronics, Control and Electronic Engineering (CMCE), 2010 International Conference on*. Vol. 1, pp. 271-273. IEEE, 2010.

[60] Cherif, S., Quadri, I. R., Meftali, S., Dekeyser, J. L.: "Modeling Reconfigurable Systems-on-Chips with UML MARTE Profile: An Exploratory Analysis". In *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools (DSD)*. pp. 706-713. IEEE Computer Society, Los Alamitos, CA, USA, 2010.

[61] Faria, J. P., Paiva, A. C. R., Yang, Z.: "Test Generation from UML Sequence Diagrams". In *Quality of Information and Communications Technology (QUATIC), 2012 Eighth International Conference on the*. pp. 245-250. IEEE, 2012.

[62] Douglass, B. P.: Real Time UML Workshop for Embedded Systems. Elsevier, Burlington, MA, 2007.

[63] Le, H.: "A Collaboration-Based Testing Model for Composite Components". In *Software Engineering and Service Science (ICSESS), 2011 IEEE 2nd International Conference on*. pp. 610-613. Institute of Electrical and Electronics Engineers (IEEE), Beijing, China, 2011.

[64] Machado, P. D. L., Figueiredo, J. C. A., Lima, E. F. A., Barbosa, A. E. V., Lima, H. S.: "Component-Based Integration Testing from UML Interaction Diagrams". In *Systems, Man and Cybernetics, 2007. ISIC. IEEE International Conference on*. pp. 2679-2686. IEEE Computer Society, Montréal, Canada, 2007.

[65] OMG: Object Constraint Language (OCL), version 2.2, (formal/2010-02-01). 2010. (Available: http://www.omg.org/spec/OCL, last accessed: 2014).

[66] El-Attar, M., Miller, J.: "Developing comprehensive acceptance tests from use cases and robustness diagrams". *Requirements Engineering*, vol. 15, pp. 285-306, 2010.

[67] Rosenberg, D., Scott, K.: Use Case Driven Object Modeling with UML: A Practical Approach. Addison-Wesley, Indianapolis, IN, USA, 1999.

[68] Fit: Framework for integrated test. 2014. (Available: http://fit.c2.com, last accessed: 2014).

[69] Beck, K.: Test Driven Development: By Example. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 2002.

[70] Yuan, Q., Wu, J., Liu, C., Zhang, L.: "A Model Driven Approach Toward Business Process Test Case Generation". In *Web Site Evolution, 2008. WSE 2008. 10th International Symposium on*. pp. 41. IEEE. Beijing, China, 2008.

[71] Juric, M. B., Mathew, B., Sarang, P. G.: Business Process Execution Language for Web Services: An Architect and Developer's Guide to Orchestrating Web Services using BPEL4WS. Packt Publishing Ltd, 2006.

[72] El-Fakih, K., Petrenko, A., Yevtushenko, N.: "FSM test translation through context". In Uyar, M., Duale, A., Fecko, M. (eds.) *Testing of Communicating Systems*. pp. 245-258. Springer Berlin / Heidelberg, 2006.

[73] Berrada, I., Castanet, R., Félix, P.: "Testing communicating systems: A model, a methodology, and a tool". In Khendek, F., Dssouli, R. (eds.) *Testing of Communicating Systems*. pp. 111-128. Springer Berlin / Heidelberg, 2005.

[74] Gotzhein, R., Khendek, F.: "Compositional Testing of Communication Systems". In Uyar, M.Ü., Duale, A.Y., Fecko, M.A. (eds.) *The 18th IFIP International Conference on Testing of Communicating Systems, TestCom 2006*, may 16, 2006 - may 18. LNCS 3964, pp. 227-244. Springer Verlag, New York, NY, United states, 2006.

[75] Xie, G., Dang, Z.: "Testing systems of concurrent black-boxes—An automata-theoretic and decompositional approach". In Grieskamp, W., Weise, C. (eds.) *Formal Approaches to Software Testing*. pp. 170-186. Springer Berlin / Heidelberg, 2006.

[76] Haugset, B., Hanssen, G. K.: "Automated Acceptance Testing: A Literature Review and an Industrial Case Study". In *Agile, 2008. AGILE '08. Conference*. pp. 27-38. IEEE Computer Society, Washington, DC, USA, 2008.

[77] Fortsch, S., Westfechtel, B.: "Differencing and Merging of Software Diagrams: State of the Art and Challenges". In *ICSOFT 2007 - International Conference on Software and Data Technologies*. pp. 90-99. INSTICC Press, 2007.

[78] Roy, C. K., Cordy, J. R., Koschke, R.: "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach". *Science of Computer Programming*, vol. 74, pp. 470-495. 2009.

[79] Stephan, M., Cordy, J. R.: "A Survey of Model Comparison Approaches and Applications". In *1st International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2013*, February 19, 2013 - February 21. pp. 265-277. INSTICC Press, Barcelona, Spain, 2013.

[80] Mens, T.: "A state-of-the-art survey on software merging," *IEEE Transactions on Software Engineering*, vol. 28, pp. 449-462. IEEE, 2002.

[81] Brosch, P., Kappel, G., Langer, P., Seidl, M., Wieland, K., Wimmer, M.: "An Introduction to Model Versioning". In Proceedings of the 12th International Conference on Formal Methods for the Design of Computer, Communication, and Software Systems: Formal Methods for Model-Driven Engineering. pp. 336-398. Springer-Verlag, Berlin, Heidelberg, 2012.

[82] Störrle, H.: "Towards Clone Detection in UML Domain Models". In *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*. pp. 285-293. ACM, New York, NY, USA, 2010.

[83] Stephan, M., Cordy, J. R.: "A survey of methods and applications of model comparison". Queen's Univ., Tech.Rep, vol. 582, 2011.

[84] Xing, Z., Stroulia, E.: "UMLDiff: An Algorithm for Object-Oriented Design Differencing". In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. pp. 54-65. ACM, New York, NY, USA, 2005.

[85] Maoz, S., Ringert, J. O., Rumpe, B.: "A Manifesto for Semantic Model Differencing". In *Proceedings of the 2010 International Conference on Models in Software Engineering*. pp. 194-203. Springer-Verlag, Berlin, Heidelberg, 2011.

[86] Nejati, S., Sabetzadeh, M., Chechik, M., Easterbrook, S., Zave, P.: "Matching and Merging of Statecharts Specifications". In *29th International Conference on Software Engineering, ICSE 2007*. pp. 54-64. IEEE Computer Society, 2007.

[87] Liu, H., Niu, Z., Ma, Z., Shao, W.: "Suffix tree-based approach to detecting duplications in sequence diagrams". *IET Software*, vol. 5, pp. 385-397. 2011.

[88] Klein, J., Caillaud, B., Hélouët, L.: "Merging Scenarios". In *Proceedings of the Ninth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2004)*, June 25, 2003 - June 27. LNCS 133, pp. 193-215. Elsevier, Amsterdam, The Netherlands, 2005.

[89] Hélouët, L., Hénin, T., Chevrier, C.: "Automating Scenario Merging". In Gotzhein, R., Reed, R. (eds.) *System Analysis and Modeling: Language Profiles*. LNCS 4320, pp. 64-81. Springer Berlin Heidelberg, 2006.

[90] Yu, Y., Jones, J. A., Harrold, M. J.: "An Empirical Study of the Effects of Test-Suite Reduction on Fault Localization". In *Proceedings of the 30th International Conference on Software Engineering*. pp. 201-210. ACM, New York, NY, USA, 2008.

[91] Zhong, H., Zhang, L., Mei, H.: "An Experimental Comparison of Four Test Suite Reduction Techniques". In *Proceedings of the 28th International Conference on Software Engineering*. pp. 636-640. ACM, New York, NY, USA, 2006.

[92] Tallam, S., Gupta, N.: "A Concept Analysis Inspired Greedy Algorithm for Test Suite Minimization". In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. pp. 35-42. ACM, New York, NY, USA, 2005.

[93] Chen, W., Ying, Q., Xue, Y., Zhao, C.: "Software testing process automation based on UTP – a case study". In Li, M., Boehm, B., Osterweil, L. (eds.) *Unifying the Software Process Spectrum*. pp. 222-234. Springer Berlin / Heidelberg, 2006.

[94] Shiva, S. G., Shala, L. A.: "Software Reuse: Research and Practice". In *ITNG*. pp. 603-609. 2007.

[95] Frakes, W., Terry, C.: "Software reuse: Metrics and models". *ACM Comput.Surv.*, vol. 28, pp. 415-435, June, 1996.

[96] Frakes, W. B., Kang, K.: "Software reuse research: Status and future," *IEEE Trans. Software Eng.*, vol. 31, pp. 529-536, 2005.

[97] Szyperski, C.: "Component Technology: What, Where, and how?". In *Proceedings of the 25th International Conference on Software Engineering*. pp. 684-693. IEEE Computer Society, Washington, DC, USA, 2003.

[98] Gross, H.: Component-Based Software Testing with UML. Springer, Berlin, 2005.

[99] Lee, J., Kang, S., Lee, D.: "A Survey on Software Product Line Testing". In *Proceedings of the 16th International Software Product Line Conference* - Volume 1. pp. 31-40. ACM, New York, NY, USA, 2012.

[100] Khendek, F., Bochmann, G. V.: "Merging behavior specifications," *Formal Methods Syst. Des.,* vol. 6, pp. 259-293, 1995.

[101] ATL. 2014. (Available: http://www.eclipse.org/atl, last accessed: 2014).

[102] medini QVT, 2012. (Available: http://projects.ikv.de/qvt, last accessed: 2014).

[103] QVT Operational, 2014. (Available: http://projects.eclipse.org/projects/modeling.mmt.qvt-oml, last accessed: 2014).

[104] QVTd, 2014. (Available: http://projects.eclipse.org/projects/modeling.mmt.qvtd, last accessed: 2014).

[105] ModTransf V3, 2014. (Available: http://www.lifl.fr/~dumoulin/modTransf/, last accessed: 2014).

[106] Eclipse UML Editor, 2012. (Available: http://wiki.eclipse.org/MDT-UML2Tools, last accessed: 2014).

[107] UMLet, 2014. (Available: http://marketplace.eclipse.org/content/umlet-uml-tool-fast-uml-diagrams#.VFbzpfnF-So, last accessed: 2014).

[108] Papyrus, 2014. (Available: https://projects.eclipse.org/projects/modeling.mdt.papyrus, last accessed: 2014).

[109] Visual Paradigm, 2014. (Available: http://www.visual-paradigm.com/features/uml-and-sysml-modeling/, last accessed: 2014).

[110] Modelio, 2014. (Available: http://www.modelio.org/downloads/download-modelio.html, last accessed: 2014).

[111] OMG: Meta Object Facility (MOF) 2.0 Query/View/Transformation specification (QVT), version 1.1. 2011. (Available: http://www.omg.org/spec/QVT, last accessed: 2014).

[112] Steinberg, D.: EMF :Eclipse Modeling Framework. Addison-Wesley, Upper Saddle River, NJ, 2009.

Appendix A
Properties of the Integration Test Generation Approach

System integration may take different strategies: top-down, bottom-up, ad-hoc and big-bang, and different sequences/orders to integrate the system components. The generated test models for the same set of system components should be equivalent regardless of the adopted integration strategy and order. The intermediate results, at a given step, may not be equivalent since they integrate different sets of components.

Test cases are equivalent when they specify the same behavior. We define the equivalence between two test cases, t_1 and t_2 as follows:

Definition 11. (Test Case Equivalence)

Let $t_1 = (\ I_1 , E_1 , R_1\ )$ and $t_2 = (\ I_2 , E_2 , R_2\ )$ be two test cases, then t_1 is equivalent to t_2 if and only if the following three conditions are satisfied:

1. $I_1 = I_2$
2. $E_1 = E_2$
3. $R_1 = R_2$

Therefore, the generated test cases are equivalent if and only if our approach has two properties: commutativity and associativity. The integration expression, Definition 7, uses the union operator and two special functions, *f()* and *g()*. In mathematics, the union operator has the commutative and associative properties. Therefore, we need to investigate the commutative and associative properties of our integration expression.

A.1. System Specification

Systems are composed of a set of components. Each component has internal and/or external interfaces. Internal interfaces are used to communicate among the system components. External interfaces are used to communicate with the system environment. The general system architecture can be described as shown in Figure 56. A system with three components is adequate to investigate the commutative and associative properties.

To simplify our investigation, we assume test cases consist of two instances only: CUT and test control. The test control represents the behavior of the test environment in addition to controlling the test execution. The test environment represents the system environment as well as system components that are not yet realized during the test execution. We also assume, for simplicity, that each component has one component test case.

The system is composed of three components, *A*, *B* and *C*, and each component has one component test case: *t1*, *t2* and *t3* respectively. We assume there is an interaction between these components, and the test cases capture these interactions. The events of each component are organized into several sets to represent the corresponding component interfaces. Accordingly, sets and relations for each test case are split into several subsets to indicate such organization. The specification for each component test case is given as follows:

$t_1 = (I_1 , E_1 , R_1)$

$I_1 = \{ tc_1, a \}$

$E_1 = e_{11} \cup e_{12} \cup e_{13}$, *where*

e_{11} *a set of events specified only in* t_1

e_{12} *a set of events specified in both* t_1 *and* t_2

e_{13} *a set of events specified in both* t_1 *and* t_3

$R_1 = R_{111} \cup R_{112} \cup R_{113} \cup R_{121} \cup R_{122} \cup R_{123} \cup R_{131} \cup R_{132} \cup R_{133}$, *where*

$R_{111} \subseteq e_{11} \times e_{11}$

$R_{112} \subseteq e_{11} \times e_{12}$

$R_{113} \subseteq e_{11} \times e_{13}$

$R_{121} \subseteq e_{12} \times e_{11}$

$R_{122} \subseteq e_{12} \times e_{12}$

$R_{123} \subseteq e_{12} \times e_{13}$

$R_{131} \subseteq e_{13} \times e_{11}$

$R_{132} \subseteq e_{13} \times e_{12}$

$R_{133} \subseteq e_{13} \times e_{13}$

$t_2 = (I_2 , E_2 , R_2)$

$I_2 = \{ tc_2, b \}$

$E_2 = e_{21} \cup e_{22} \cup e_{23}$, *where*

e_{21} *a set of events specified in both* t_2 *and* t_1

e_{22} *a set of events specified only in* t_2

e_{23} a set of events specified in both t_2 and t_3

$R_2 = R_{211} \cup R_{212} \cup R_{213} \cup R_{221} \cup R_{222} \cup R_{223} \cup R_{231} \cup R_{232} \cup R_{233}$, *where*

$R_{211} \subseteq e_{21} \times e_{21}$

$R_{212} \subseteq e_{21} \times e_{22}$

$R_{213} \subseteq e_{21} \times e_{23}$

$R_{221} \subseteq e_{22} \times e_{21}$

$R_{222} \subseteq e_{22} \times e_{22}$

$R_{223} \subseteq e_{22} \times e_{23}$

$R_{231} \subseteq e_{23} \times e_{21}$

$R_{232} \subseteq e_{23} \times e_{22}$

$R_{233} \subseteq e_{23} \times e_{23}$

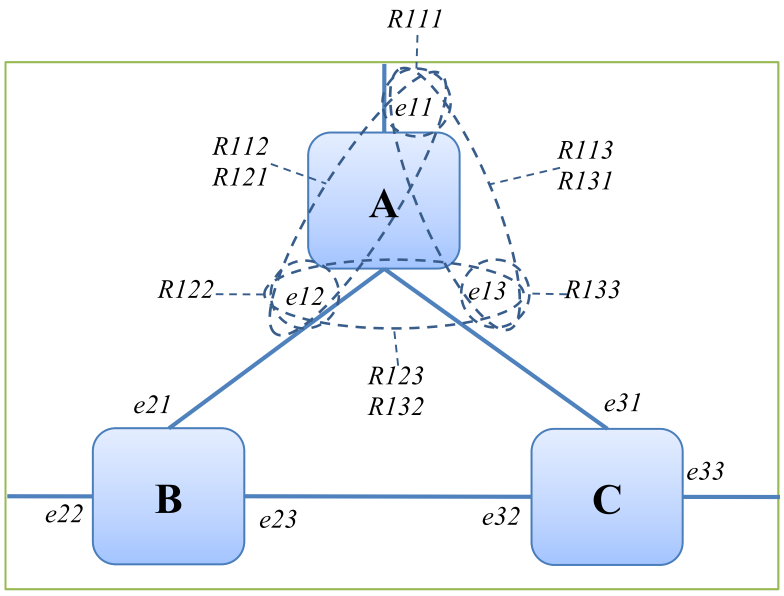


Figure 56. General system architecture

$t_3 = (I_3, E_3, R_3)$

$I_3 = \{ tc_3, c \}$

$E_3 = e_{31} \cup e_{32} \cup e_{33}$, *where*

e_{31} a set of events specified in both t_3 and t_1

e_{32} a set of events specified in both t_3 and t_2

e_{33} a set of events specified only in t_3

$R_3 = R_{311} \cup R_{312} \cup R_{313} \cup R_{321} \cup R_{322} \cup R_{323} \cup R_{331} \cup R_{332} \cup R_{333}$, *where*

$R_{311} \subseteq e_{31} \times e_{31}$

$R_{312} \subseteq e_{31} \times e_{32}$

$R_{313} \subseteq e_{31} \times e_{33}$

$R_{321} \subseteq e_{32}\, x\, e_{31}$

$R_{322} \subseteq e_{32}\, x\, e_{32}$

$R_{323} \subseteq e_{32}\, x\, e_{33}$

$R_{331} \subseteq e_{33}\, x\, e_{31}$

$R_{332} \subseteq e_{33}\, x\, e_{32}$

$R_{333} \subseteq e_{33}\, x\, e_{33}$

Please notice that

$e_{12} = e_{21}$

$e_{13} = e_{31}$

$e_{23} = e_{32}$

$R_{122} = R_{211}$

$R_{133} = R_{311}$

$R_{233} = R_{322}$

We have to bring to your attention that if there is no interaction between two components, then their corresponding variables, sets and relations will be empty; for examples, suppose there is no interaction between *A* and *C* then

$e_{13} = \{\},$

$e_{31} = \{\},$

$R_{113} = \{\},$

$R_{123} = \{\},$

$R_{131} = \{\},$

$R_{132} = \{\},$

$R_{133} = \{\},$

$R_{311} = \{\},$

$R_{312} = \{\},$

$R_{313} = \{\},$

$R_{321} = \{\}$ *and*

$R_{331} = \{\}$

The approach creates the test control for the generated test model and builds its behavior by merging the behavior of the test controls of the given test models, which we call *tci*.

A.2. Commutativity Property

To satisfy the commutative property of our approach for any two components, say A and B, we should demonstrate that the integration of their component test cases, t_1 and t_2 respectively, generates equivalent behaviors independent of the integration order: *(A + B)* or *(B + A)*. That means

$t1 + t2 = t2 + t1$ ①

By using Definition 3 and Definition 7, we get

$\mathbf{(}g(I_1)\ U\ g(I_2), f(E_1)\ U\ f(E_2), f(R_1)\ U f(R_2)\mathbf{)} = \mathbf{(}\ g(I_2)\ U\ g(I_1), f(E_2)\ U\ f(E_1), f(R_2)\ U f(R_1)\mathbf{)}$

Hence, to validate eq. ①, we need to show that

$g(I_1)\ U\ g(I_2) = g(I_2)\ U\ g(I_1)$ ②

$f(E_1)\ U\ f(E_2) = f(E_2)\ U\ f(E_1)$ ③

$f(R_1)\ U f(R_2) = f(R_2)\ U f(R_1)$ ④

Let us evaluate the left side of eq. ② first by substituting the values of I_1 and I_2 and using our equivalent definition, Definition 11.

$g(I_1)\ U\ g(I_2) = g(\{\ tc_1, a\ \})\ U\ g(\{\ tc_2, b\ \})$

Then, we apply the *g()* function

$g(I_1)\ U\ g(I_2) = \{\ \mathbf{tc_i}, a\ \}\ U\ \{\ \mathbf{tc_i}, b\ \}$

Then, we apply the union operator

$g(I_1)\ U\ g(I_2) = \{\ \mathbf{tc_i}, a, b\ \}$

Next, we perform the same sequence on the right side of eq. ②

$g(I_2)\ U\ g(I_1) = g(\{\ tc_2, b\ \})\ U\ g(\{\ tc_1, a\ \})$

$= \{\ \mathbf{tc_i}, b\ \}\ U\ \{\ \mathbf{tc_i}, a\ \}$

$= \{\ \mathbf{tc_i}, b, a\ \}$

The two sides are equivalent. Thus, we say eq. ② holds to be correct. We are going to take the same evaluation approach with eq. ③. First, we evaluate the left side of eq. ③.

$f(E_1)\ U\ f(E_2) = f(e_{11}\ U\ e_{12}\ U\ e_{13})\ U\ f(e_{21}\ U\ e_{22}\ U\ e_{23})$

Since $e_{12} = e_{21}$, the *f()* function replaces e_{21} with e_{12}

$f(E_1)\ U\ f(E_2) = e_{11}\ U\ e_{12}\ U\ e_{13}\ U\ \mathbf{e_{12}}\ U\ e_{22}\ U\ e_{23}$

$= e_{11}\ U\ e_{12}\ U\ e_{13}\ U\ e_{22}\ U\ e_{23}$

Then, we evaluate the right side of eq. ③.

$f(E_2)\ U\ f(E_1) = f(e_{21}\ U\ e_{22}\ U\ e_{23})\ U\ f(e_{11}\ U\ e_{12}\ U\ e_{13})$

Since $e_{12} = e_{21}$, the *f()* function replaces e_{21} with e_{12}

$f(E_2)\ U\ f(E_1) = \mathbf{e_{12}}\ U\ e_{22}\ U\ e_{23}\ U\ e_{11}\ U\ e_{12}\ U\ e_{13}$

$= e_{12}\ U\ e_{22}\ U\ e_{23}\ U\ e_{11}\ U\ e_{13}$

Hence, the two sides are equivalent, and this proves that eq. ③ holds true. The same evaluation approach will be applied on eq. ④. We take the left side of the equation first.

$f(R_1)\ U\ f(R_2)\ = f(R_{111}\ U\ R_{112}\ U\ R_{113}\ U\ R_{121}\ U\ R_{122}\ U\ R_{123}\ U\ R_{131}\ U\ R_{132}\ U\ R_{133})\ U\ f(R_{211}\ U\ R_{212}\ U\ R_{213}\ U\ R_{221}\ U\ R_{222}\ U\ R_{223}\ U\ R_{231}\ U\ R_{232}\ U\ R_{233})$

Since $R_{122} = R_{211}$, the *f()* function replaces R_{211} with R_{122}

$f(R_1)\ U\ f(R_2)\ = R_{111}\ U\ R_{112}\ U\ R_{113}\ U\ R_{121}\ U\ R_{122}\ U\ R_{123}\ U\ R_{131}\ U\ R_{132}\ U\ R_{133}\ U\ \mathbf{R_{122}}\ U\ R_{212}\ U\ R_{213}\ U\ R_{221}\ U\ R_{222}\ U\ R_{223}\ U\ R_{231}\ U\ R_{232}\ U\ R_{233}$

$= R_{111}\ U\ R_{112}\ U\ R_{113}\ U\ R_{121}\ U\ R_{122}\ U\ R_{123}\ U\ R_{131}\ U\ R_{132}\ U\ R_{133}\ U\ R_{212}\ U\ R_{213}\ U\ R_{221}\ U\ R_{222}\ U\ R_{223}\ U\ R_{231}\ U\ R_{232}\ U\ R_{233}$

The next step is to evaluate the right side of eq. ④.

$f(R_2)\ U\ f(R_1)\ = f(R_{211}\ U\ R_{212}\ U\ R_{213}\ U\ R_{221}\ U\ R_{222}\ U\ R_{223}\ U\ R_{231}\ U\ R_{232}\ U\ R_{233})\ U\ f(R_{111}\ U\ R_{112}\ U\ R_{113}\ U\ R_{121}\ U\ R_{122}\ U\ R_{123}\ U\ R_{131}\ U\ R_{132}\ U\ R_{133})$

$= \mathbf{R_{122}}\ U\ R_{212}\ U\ R_{213}\ U\ R_{221}\ U\ R_{222}\ U\ R_{223}\ U\ R_{231}\ U\ R_{232}\ U\ R_{233}\ U\ R_{111}\ U\ R_{112}\ U\ R_{113}\ U\ R_{121}\ U\ R_{122}\ U\ R_{123}\ U\ R_{131}\ U\ R_{132}\ U\ R_{133}$

$= R_{122}\ U\ R_{212}\ U\ R_{213}\ U\ R_{221}\ U\ R_{222}\ U\ R_{223}\ U\ R_{231}\ U\ R_{232}\ U\ R_{233}\ U\ R_{111}\ U\ R_{112}\ U\ R_{113}\ U\ R_{121}\ U\ R_{123}\ U\ R_{131}\ U\ R_{132}\ U\ R_{133}$

The results of both sides of ④ are equivalent. Since equations ②, ③ and ④ are passed; then equation ① holds true too. Hence, the commutative property holds true in the integration approach.

A.3. Associativity Property

To satisfy the associative property of the integration approach for any three components, *A*, *B* and *C*, we should demonstrate that the integration of their component test cases, t_1, t_2 and t_3 respectively, generate the same behavior in any integration order. In other words, we should satisfy the following expression.

$t1 + (t2 + t3) = (t1 + t2) + t3$ ⑤

By using Definition 3 and Definition 7, we can refactor eq. ⑤ as follows:

$g(I_1)\ U\ \mathbf{(}\ g(I_2)\ U\ g(I_3)\ \mathbf{)} = \mathbf{(}\ g(I_1)\ U\ g(I_2)\ \mathbf{)}\ U\ g(I_3)$ ⑥

$f(E_1)\ U\ \mathbf{(}\ f(E_2)\ U\ f(E_3)\ \mathbf{)} = \mathbf{(}\ f(E_1)\ U\ f(E_2)\ \mathbf{)}\ U\ f(E_3)$ ⑦

$f(R_1)\ U\ \mathbf{(}\ f(R_2)\ U\ f(R_3)\ \mathbf{)} = \mathbf{(}\ f(R_1)\ U\ f(R_2)\ \mathbf{)}\ U\ f(R_3)$ ⑧

Hence, we have to prove the correctness of eq. ⑥, ⑦ and ⑧, so eq. ⑤ will hold true. Let us start by examining eq. ⑥. First, we evaluate the left side of the equation.

$g(I_1)\ \ U\ \ \mathbf{(}\ g(I_2)\ U\ \ g(I_3)\ \mathbf{)} = g(\{tc_1, a\})\ U\ \mathbf{(}\ g(\{tc_2, b\})\ U\ g(\{tc_3, c\})\ \mathbf{)}$

Then, we apply *g()*

$= \{\mathbf{tc_i}, a\}\ U\ \mathbf{(}\ \{\mathbf{tc_i}, b\}\ U\ \{\mathbf{tc_i}, c\}\ \mathbf{)}$

$= \{\mathbf{tc_i}, a\}\ U\ \{\mathbf{tc_i}, b,\ c\}$

$= \{\mathbf{tc_i}, a, b,\ c\}$

Then, take the right side of eq. ⑥

$(g(I_1) \cup g(I_2)) \cup g(I_3) = (g(\{tc_1, a\}) \cup g(\{tc_2, b\})) \cup g(\{tc_3, c\})$

$= (\{\mathbf{tc_i}, a\} \cup \{\mathbf{tc_i}, b\}) \cup \{\mathbf{tc_i}, c\}$

$= \{\mathbf{tc_i}, a, b\} \cup \{\mathbf{tc_i}, c\}$

$= \{\mathbf{tc_i}, a, b, c\}$

The two sides are equivalent. Thus, we can say eq. ⑥ is correct. We are going to take the same evaluation approach with eq. ⑦. First, we evaluate the left side of eq. ⑦.

$f(E_1) \cup (f(E_2) \cup f(E_3)) = f(e_{11} \cup e_{12} \cup e_{13}) \cup (f(e_{21} \cup e_{22} \cup e_{23}) \cup f(e_{31} \cup e_{32} \cup e_{33}))$

Then, we apply *f()*, which replaces the following sets

$e_{12} = e_{21}$,

$e_{13} = e_{31}$ *and*

$e_{23} = e_{32}$.

$f(E_1) \cup (f(E_2) \cup f(E_3)) = (e_{11} \cup e_{12} \cup e_{13}) \cup ((\mathbf{e_{12}} \cup e_{22} \cup e_{23}) \cup (\mathbf{e_{13}} \cup \mathbf{e_{23}} \cup e_{33}))$

$= (e_{11} \cup e_{12} \cup e_{13}) \cup (\mathbf{e_{12}} \cup e_{22} \cup e_{23} \cup \mathbf{e_{13}} \cup e_{33})$

$= e_{11} \cup e_{12} \cup e_{13} \cup e_{22} \cup e_{23} \cup e_{33}$

Then, we evaluate the right side of eq. ⑦.

$(f(E_1) \cup f(E_2)) \cup f(E_3) = (f(e_{11} \cup e_{12} \cup e_{13}) \cup f(e_{21} \cup e_{22} \cup e_{23})) \cup f(e_{31} \cup e_{32} \cup e_{33})$

$= ((e_{11} \cup e_{12} \cup e_{13}) \cup (\mathbf{e_{12}} \cup e_{22} \cup e_{23})) \cup (\mathbf{e_{13}} \cup \mathbf{e_{23}} \cup e_{33})$

$= (e_{11} \cup e_{12} \cup e_{13} \cup e_{22} \cup e_{23}) \cup (\mathbf{e_{13}} \cup \mathbf{e_{23}} \cup e_{33})$

$= e_{11} \cup e_{12} \cup e_{13} \cup e_{22} \cup e_{23} \cup e_{33}$

Therefore, the two sides are equivalent, and that proves that eq. ⑦ holds true. The same evaluation approach will be applied on eq. ⑧. We take the left side of the equation first.

$f(R_1) \cup (f(R_2) \cup f(R_3)) = f(R_{111} \cup R_{112} \cup R_{113} \cup R_{121} \cup R_{122} \cup R_{123} \cup R_{131} \cup R_{132} \cup R_{133}) \cup (f(R_{211} \cup R_{212} \cup R_{213} \cup R_{221} \cup R_{222} \cup R_{223} \cup R_{231} \cup R_{232} \cup R_{233}) \cup f(R_{311} \cup R_{312} \cup R_{313} \cup R_{321} \cup R_{322} \cup R_{323} \cup R_{331} \cup R_{332} \cup R_{333}))$

Then, we apply *f()*, which replaces the following relations

$R_{122} = R_{211}$,

$R_{133} = R_{311}$ *and*

$R_{233} = R_{322}$

$f(R_1) \cup (f(R_2) \cup f(R_3)) = (R_{111} \cup R_{112} \cup R_{113} \cup R_{121} \cup R_{122} \cup R_{123} \cup R_{131} \cup R_{132} \cup R_{133}) \cup ((\mathbf{R_{122}} \cup R_{212} \cup R_{213} \cup R_{221} \cup R_{222} \cup R_{223} \cup R_{231} \cup R_{232} \cup R_{233}) \cup (\mathbf{R_{133}} \cup R_{312} \cup R_{313} \cup R_{321} \cup \mathbf{R_{233}} \cup R_{323} \cup R_{331} \cup R_{332} \cup R_{333}))$

$= (R_{111} \cup R_{112} \cup R_{113} \cup R_{121} \cup R_{122} \cup R_{123} \cup R_{131} \cup R_{132} \cup R_{133}) \cup (\mathbf{R_{122}} \cup R_{212} \cup R_{213} \cup R_{221} \cup R_{222} \cup R_{223} \cup R_{231} \cup R_{232} \cup R_{233} \cup \mathbf{R_{133}} \cup R_{312} \cup R_{313} \cup R_{321} \cup R_{323} \cup R_{331} \cup R_{332} \cup R_{333})$

$= R_{111} \cup R_{112} \cup R_{113} \cup R_{121} \cup R_{122} \cup R_{123} \cup R_{131} \cup R_{132} \cup R_{133} \cup R_{212} \cup R_{213} \cup R_{221} \cup R_{222} \cup R_{223} \cup R_{231} \cup R_{232} \cup R_{233} \cup R_{312} \cup R_{313} \cup R_{321} \cup R_{323} \cup R_{331} \cup R_{332} \cup R_{333}$

The next step is to evaluate the right side of eq. ⑧.

$$\boldsymbol{(}\, f(R_1)\ U\ f(R_2)\, \boldsymbol{)}\ U\ f(R_3) = \boldsymbol{(}\, f(R_{111}\ U\ R_{112}\ U\ R_{113}\ U\ R_{121}\ U\ R_{122}\ U\ R_{123}\ U\ R_{131}\ U\ R_{132}\ U\ R_{133})\ U\ f(R_{211}\ U\ R_{212}\ U\ R_{213}\ U\ R_{221}\ U\ R_{222}\ U\ R_{223}\ U\ R_{231}\ U\ R_{232}\ U\ R_{233})\, \boldsymbol{)}\ U\ f(R_{311}\ U\ R_{312}\ U\ R_{313}\ U\ R_{321}\ U\ R_{322}\ U\ R_{323}\ U\ R_{331}\ U\ R_{332}\ U\ R_{333})$$

Then, we apply *f()*

$$= \boldsymbol{(}\, (R_{111}\ U\ R_{112}\ U\ R_{113}\ U\ R_{121}\ U\ R_{122}\ U\ R_{123}\ U\ R_{131}\ U\ R_{132}\ U\ R_{133})\ U\ (\boldsymbol{R_{122}}\ U\ R_{212}\ U\ R_{213}\ U\ R_{221}\ U\ R_{222}\ U\ R_{223}\ U\ R_{231}\ U\ R_{232}\ U\ R_{233})\, \boldsymbol{)}\ U\ (\boldsymbol{R_{133}}\ U\ R_{312}\ U\ R_{313}\ U\ R_{321}\ U\ R_{233}\ U\ \boldsymbol{R_{323}}\ U\ R_{331}\ U\ R_{332}\ U\ R_{333})$$

$$= \boldsymbol{(}\, R_{111}\ U\ R_{112}\ U\ R_{113}\ U\ R_{121}\ U\ R_{122}\ U\ R_{123}\ U\ R_{131}\ U\ R_{132}\ U\ R_{133}\ U\ R_{212}\ U\ R_{213}\ U\ R_{221}\ U\ R_{222}\ U\ R_{223}\ U\ R_{231}\ U\ R_{232}\ U\ R_{233}\, \boldsymbol{)}\ U\ (\boldsymbol{R_{133}}\ U\ R_{312}\ U\ R_{313}\ U\ R_{321}\ U\ \boldsymbol{R_{233}}\ U\ R_{323}\ U\ R_{331}\ U\ R_{332}\ U\ R_{333})$$

$$= R_{111}\ U\ R_{112}\ U\ R_{113}\ U\ R_{121}\ U\ R_{122}\ U\ R_{123}\ U\ R_{131}\ U\ R_{132}\ U\ R_{133}\ U\ R_{212}\ U\ R_{213}\ U\ R_{221}\ U\ R_{222}\ U\ R_{223}\ U\ R_{231}\ U\ R_{232}\ U\ R_{233}\ U\ R_{312}\ U\ R_{313}\ U\ R_{321}\ U\ R_{323}\ U\ R_{331}\ U\ R_{332}\ U\ R_{333}$$

The results of both sides of ⑧ are equivalent. Since equations ⑥, ⑦ and ⑧ are passed; then equation ⑤ holds true too. Hence, the associative property holds true for the integration approach.

Appendix B

Case Study: Specifications

We built the specification of a simple library system to demonstrate our framework. The system architecture is described in Section B.1. Component test models are described in Section B.2. The acceptance test model is described in Section B.3.

B.1. System Specification

The system is composed of four components: *Librarian*, *Member*, *Media* and *Booking* as shown in Figure 57. The *Librarian* component provides the necessary services for the librarians, while the *Member* component provides the necessary services for the subscribers. The *Media* component manages the records of different media that hold in the library such as books, DVDs, maps, etc. The *Booking* component manages the reservation of the library media by subscribers.

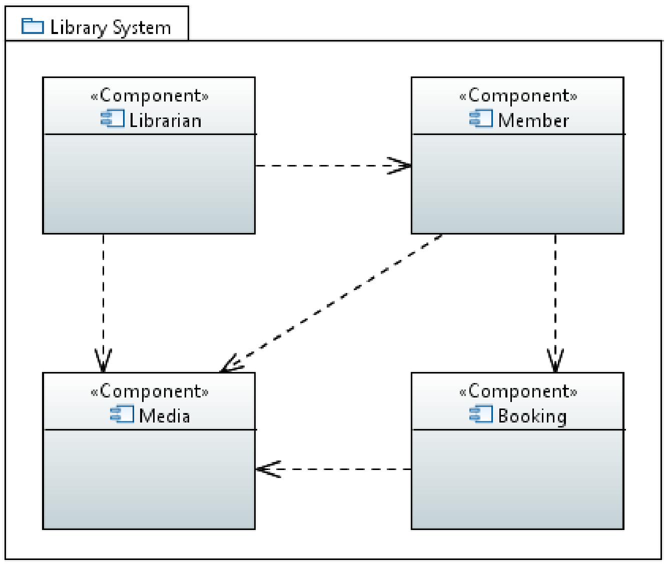


Figure 57. Library system architecture

B.2. Component Test Models

Four component test models are developed. They cover the basic services provided by the library system, which are:

- For the librarians

1 Add new media
2 Add new member
3 Browse media
4 Browse members

- For the subscribers
  1 Browse media
  2 Reserve media
  3 Return media

The Librarian test model is illustrated in Figure 58. Using Definitions 1-4, we express the given test model as

LibrarianTM = (P, T)

P = (tcn, tcm, sut)

tcn = LibTstCntrl

tcm = { }

sut = { Librarian }

T = { TestCaseAddMedia, TestCaseAddMember, TestCaseBrowseMedia, TestCaseBrowseMembers }

TestCaseAddMedia = ({libTstCntrl, librarian}, {e1,e2,e3,e4,e5,e6,e7,e8,e9,e10,e11, e12,e13}, {(e1,e2), (e2,e3), (e3,e4), (e4,e7), (e2,e5), (e5,e6), (e6,e7), (e8,e9), (e9,e10), (e10,e11), (e9,e12), (e12,e13), (e1,e8), (e9,e2), (e3,e10), (e11,e4), (e5,e12), (e13,e6), (e1,e3), (e1,e5), (e2,e4), (e2,e10), (e3,e7), (e2,e6), (e2,e12), (e5,e7), (e8,e10), (e8,e12), (e8,e2), (e9,e11), (e10,e4), (e9,e13), (e12,e6), (e1,e9), (e9,e3), (e9,e5), (e3,e11), (e11,e7), (e5,e13), (e13,e7), (e1,e4), (e1,e10), (e1,e6), (e1,e12), (e2,e7), (e2,e11), (e2,e13), (e8,e11), (e8,e13), (e8,e3), (e8,e5), (e9,e4), (e10,e7), (e9,e6), (e12,e7), (e1,e7), (e1,e11), (e1,e13), (e8,e4), (e8,e6), (e9,e7), (e8,e7)})

TestCaseAddMember = ({libTstCntrl, librarian}, {e14,e15,e16,e17,e18,e19,e20,e21, e22,e23,e24,e25,e26}, {(e14,e15), (e15,e16), (e16,e17), (e17,e20), (e15,e18), (e18,e19), (e19,e20), (e21,e22), (e22,e23), (e23,e24), (e22,e25), (e25,e26), (e14,e21), (e22,e15), (e16,e23), (e24,e17), (e18,e25), (e26,e19), (e14,e16), (e14,e18), (e15,e17), (e15,e23), (e16,e20), (e15,e19), (e15,e25), (e18,e20), (e21,e23), (e21,e25), (e21,e15), (e22,e24), (e23,e17), (e22,e26), (e25,e19), (e14,e22), (e22,e16), (e22,e18), (e16,e24), (e24,e20), (e18,e26), (e26,e20), (e14,e17), (e14,e23), (e14,e19), (e14,e25), (e15,e20), (e15,e24), (e15,e26), (e21,e24), (e21,e26), (e21,e16), (e21,e18), (e22,e17), (e23,e20), (e22,e19), (e25,e20), (e14,e20), (e14,e24), (e14,e26), (e21,e17), (e21,e19), (e22,e20), (e21,e20)})

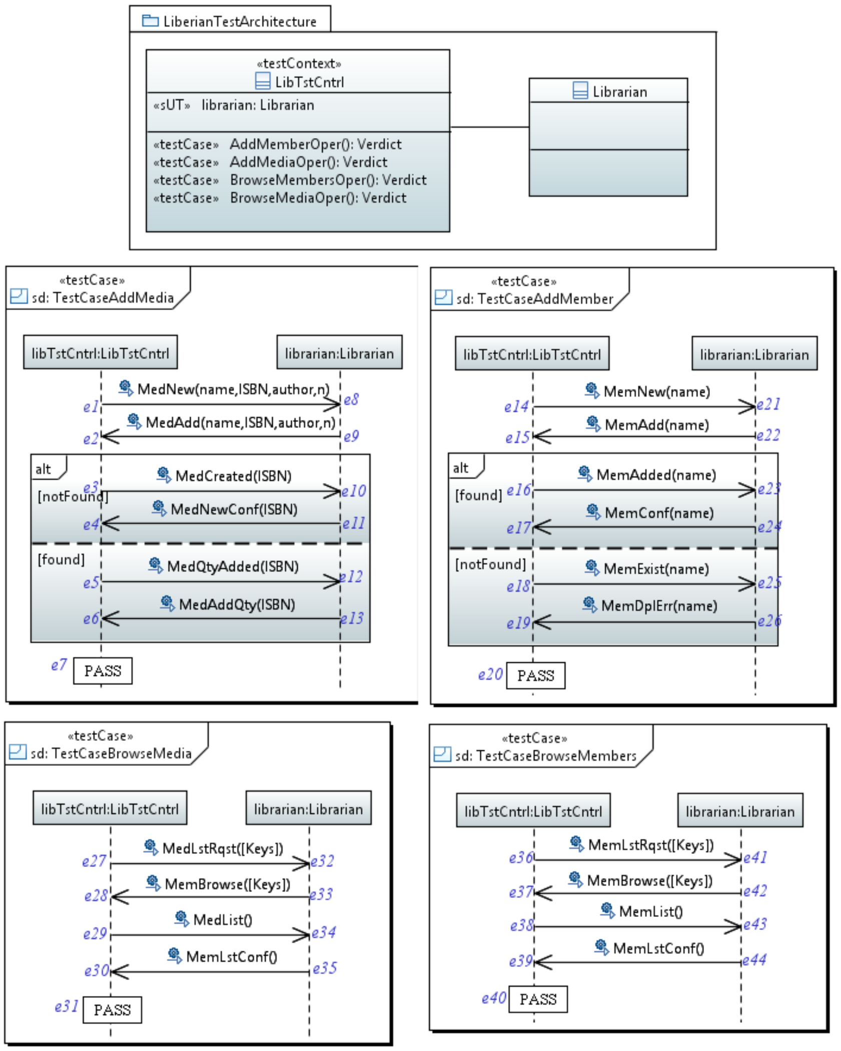

LiberianTestArchitecture
«testContext» LibTstCntrl
«sUT» librarian: Librarian
«testCase» AddMemberOper(): Verdict
«testCase» AddMediaOper(): Verdict
«testCase» BrowseMembersOper(): Verdict
«testCase» BrowseMediaOper(): Verdict
Librarian
«testCase» sd: TestCaseAddMedia
libTstCntrl:LibTstCntrl
librarian:Librarian
MedNew(name,ISBN,author,n)
MedAdd(name,ISBN,author,n)
alt
[notFound]
MedCreated(ISBN)
MedNewConf(ISBN)
[found]
MedQtyAdded(ISBN)
MedAddQty(ISBN)
e1 e2 e3 e4 e5 e6 e7 e8 e9 e10 e11 e12 e13
PASS
«testCase» sd: TestCaseAddMember
MemNew(name)
MemAdd(name)
[found]
MemAdded(name)
MemConf(name)
[notFound]
MemExist(name)
MemDplErr(name)
e14 e15 e16 e17 e18 e19 e20 e21 e22 e23 e24 e25 e26
«testCase» sd: TestCaseBrowseMedia
MedLstRqst([Keys])
MemBrowse([Keys])
MedList()
MemLstConf()
e27 e28 e29 e30 e31 e32 e33 e34 e35
«testCase» sd: TestCaseBrowseMembers
MemLstRqst([Keys])
MemBrowse([Keys])
MemList()
MemLstConf()
e36 e37 e38 e39 e40 e41 e42 e43 e44

Figure 58. Librarian test model (*LibrarianTM*)

TestCaseBrowseMedia = ({libTstCntrl, librarian}, {e27,e28,e29,e30,e31,e32,e33,e34, e35}, {(e27,e28), (e28,e29), (e29,e30), (e30,e31), (e32,e33), (e33,e34), (e34,e35), (e27,e32), (e33,e28), (e29,e34), (e35,e30), (e27,e29), (e28,e30), (e28,e34), (e29,e31), (e32,e34), (e32,e28), (e33,e35), (e34,e30), (e27,e33), (e33,e29), (e29,e35), (e35,e31), (e27,e30), (e27,e34), (e28,e31), (e28,e35), (e32,e35), (e32,e29), (e33,e30), (e34,e31), (e27,e31), (e27,e35), (e32,e30), (e33,e31), (e32,e31)})

TestCaseBrowseMembers = ({libTstCntrl, librarian}, {e36,e37,e38,e39,e40,e41, e42,e43,e44}, {(e36,e37), (e37,e38), (e38,e39), (e39,e40), (e41,e42), (e42,e43), (e43,e44), (e36,e41), (e42,e37), (e38,e43), (e44,e39), (e36,e38), (e37,e39), (e37,e43), (e38,e40), (e41,e43), (e41,e37), (e42,e44), (e43,e39), (e36,e42), (e42,e38), (e38,e44), (e44,e40), (e36,e39), (e36,e43), (e37,e40), (e37,e44), (e41,e44), (e41,e38), (e42,e39), (e43,e40), (e36,e40), (e36,e44), (e41,e39), (e42,e40), (e41,e40)})

The Member test model is illustrated in Figure 59. Using Definitions 1-4, we express the given test model as

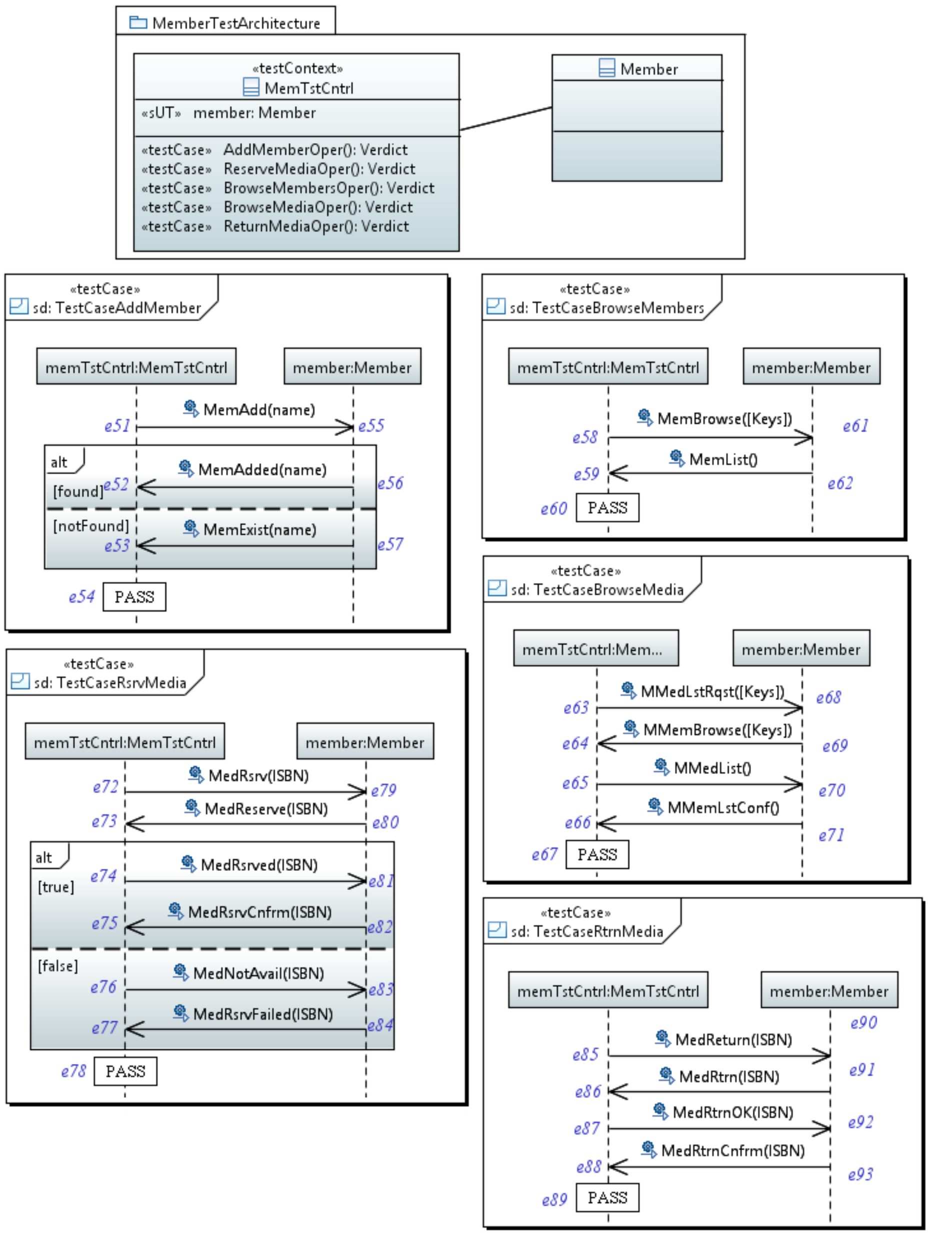


Figure 59. Member test model (*MemberTM*)

MemberTM = (*P*, *T*)

P = (*tcn*, *tcm*, *sut*)

tcn = *MemTstCntrl*

tcm = { }

sut = { *Member* }

T = { *TestCaseAddMember*, *TestCaseBrowseMembers*, *TestCaseBrowseMedia*, *TestCaseRsrvMedia*, *TestCaseRtrnMedia*}

TestCaseAddMember = ({memTstCntrl, member}, {e51,e52,e53,e54,e55,e56,e57}, {(e51,e52), (e51,e53), (e52,e54), (e53,e54), (e55,e56), (e55,e57), (e51,e55), (e56,e52), (e57,e53), (e51,e54), (e55,e52), (e55,e53), (e51,e56), (e51,e57), (e56,e54), (e57,e54), (e55,e54)})

TestCaseBrowseMembers = ({memTstCntrl, member}, {e58,e59,e60,e61,e62}, {(e58,e59), (e59,e60), (e61,e62), (e58,e61), (e62,e59), (e58,e60), (e61,e59), (e58,e62), (e62,e60), (e61,e60)})

TestCaseBrowseMedia = ({memTstCntrl, member}, {e63,e64,e65,e66,e67,e68,e69, e70,e71}, {(e63,e64), (e64,e65), (e65,e66), (e66,e67), (e68,e69), (e69,e70), (e70,e71), (e63,e68), (e69,e64), (e65,e70), (e71,e66), (e63,e65), (e64,e66), (e64,e70), (e65,e67), (e68,e70), (e68,e64), (e69,e71), (e70,e66), (e63,e69), (e69,e65), (e65,e71), (e71,e67), (e63,e66), (e63,e70), (e64,e67), (e64,e71), (e68,e71), (e68,e65), (e69,e66), (e70,e67), (e63,e67), (e63,e71), (e68,e66), (e69,e67), (e68,e67)})

TestCaseRsrvMedia = ({memTstCntrl, member}, {e72,e73,e74,e75,e76,e77,e78, e79,e80,e81,e82,e83,e84}, {(e72,e73), (e73,e74), (e74,e75), (e75,e78), (e73,e76), (e76,e77), (e77,e78), (e79,e80), (e80,e81), (e81,e82), (e80,e83), (e83,e84), (e72,e79), (e80,e73), (e74,e81), (e82,e75), (e76,e83), (e84,e77), (e72,e74), (e72,e76), (e73,e75), (e73,e81), (e74,e78), (e73,e77), (e73,e83), (e76,e78), (e79,e81), (e79,e83), (e79,e73), (e80,e82), (e81,e75), (e80,e84), (e83,e77), (e72,e80), (e80,e74), (e80,e76), (e74,e82), (e82,e78), (e76,e84), (e84,e78), (e72,e75), (e72,e81), (e72,e77), (e72,e83), (e73,e78), (e73,e82), (e73,e84), (e79,e82), (e79,e84), (e79,e74), (e79,e76), (e80,e75), (e81,e78), (e80,e77), (e83,e78), (e72,e78), (e72,e82), (e72,e84), (e79,e75), (e79,e77), (e80,e78), (e79,e78)})

TestCaseRtrnMedia = ({memTstCntrl, member}, {e85,e86,e87,e88,e89,e90,e91, e92,e93}, {(e85,e86), (e86,e87), (e87,e88), (e88,e89), (e90,e91), (e91,e92), (e92,e93), (e85,e90), (e91,e86), (e87,e92), (e93,e88), (e85,e87), (e86,e88), (e86,e92), (e87,e89), (e90,e92), (e90,e86), (e91,e93), (e92,e88), (e85,e91), (e91,e87), (e87,e93), (e93,e89), (e85,e88), (e85,e92), (e86,e89), (e86,e93), (e90,e93), (e90,e87), (e91,e88), (e92,e89), (e85,e89), (e85,e93), (e90,e88), (e91,e89), (e90,e89)})

The Media test model is illustrated in Figure 60. Using Definitions 1-4, we express the given test model as

MediaTM = (P, T)

P = (tcn, tcm, sut)

tcn = MedTstCntrl

tcm = { }

sut = { Media }

T = { TestCaseAddMedia, TestCaseLibBrowseMedia, TestCaseMemBrowseMedia, TestCaseRsrvMedia, TestCaseRtrnMedia}

TestCaseAddMedia = ({medTstCntrl, media}, {e101,e102,e103,e104,e105,e106, e107}, {(e101,e102), (e101,e103), (e102,e104), (e103,e104), (e105,e106), (e105,e107), (e101,e105), (e106,e102), *(e107,e103), (e101,e104), (e105,e102), (e105,e103), (e101,e106), (e101,e107), (e106,e104), (e107,e104), (e105,e104)})*

TestCaseLibBrowseMedia = ({medTstCntrl, media}, {e108,e109,e110,e111,e112}, {(e108,e109), (e109,e110), (e111,e112), (e108,e111), (e112,e109), (e108,e110), (e111,e109), (e108,e112), (e112,e110), (e111,e110)})

TestCaseMemBrowseMedia = ({medTstCntrl, media}, {e113,e114,e115,e116,e117}, {(e113,e114), (e114,e115), (e116,e117), (e113,e116), (e117,e114), (e113,e115), (e116,e114), (e113,e117), (e117,e115), (e116,e115)})

TestCaseRsrvMedia = ({medTstCntrl, media}, {e118,e119,e120,e121,e122,e123, e124}, {(e118,e119), (e119,e121), (e118,e120), (e120,e121), (e122,e123), (e122,e124), (e118,e122), (e123,e119), (e124,e120), (e118,e121), (e122,e119), (e122,e120), (e118,e123), (e118,e124), (e123,e121), (e124,e121), (e122,e121)})

TestCaseRtrnMedia = ({medTstCntrl, media}, {e125,e126,e127,e128,e129}, {(e125,e126), (e126,e127), (e128,e129), (e125,e128), (e129,e126), (e125,e127), (e128,e126), (e125,e129), (e129,e127), (e128,e127)})

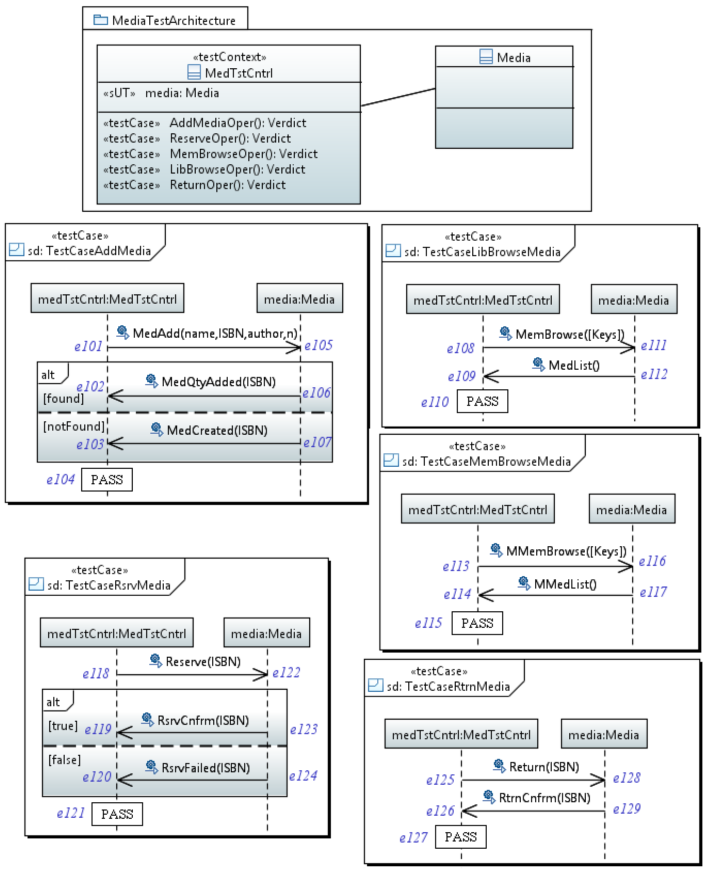


Figure 60. Media test model (*MediaTM*)

The Booking test model is illustrated in Figure 61. Using Definitions 1-4, we express the given test model as

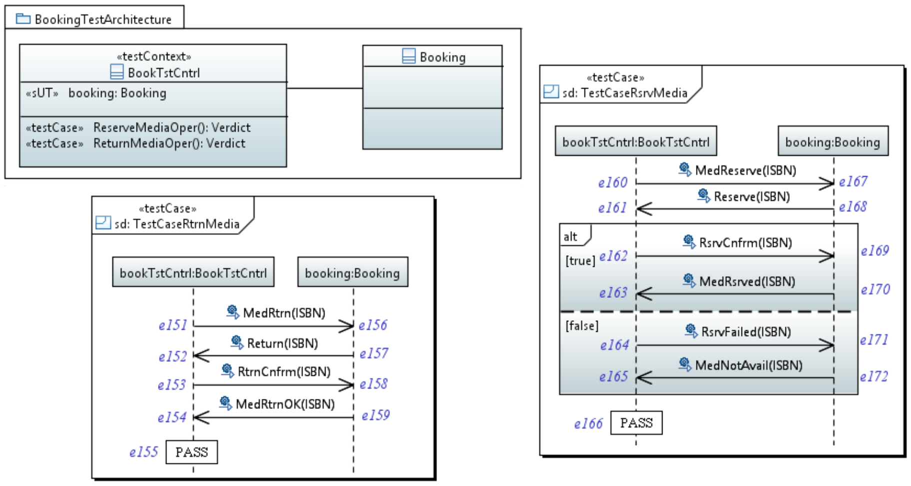


Figure 61. Booking test model (*BookingTM*)

BookingTM = (P, T)

P = (tcn, tcm, sut)

tcn = BookTstCntrl

tcm = { }

sut = { Booking }

T = { TestCaseRsrvMedia, TestCaseRtrnMedia }

TestCaseRsrvMedia = ({bookTstCntrl, booking}, {e160,e161,e162,e163,e164,e165, e166,e167,e168,e169,e170,e171,e172}, {(e160,e161), (e161,e162), (e162,e163), (e163,e166), (e161,e164), (e164,e165), (e165,e166), (e167,e168), (e168,e169), (e169,e170), (e168,e171), (e171,e172), (e160,e167), (e168,e161), (e162,e169), (e170,e163), (e164,e171), (e172,e165), (e160,e162), (e160,e164), (e161,e163), (e161,e169), (e162,e166), (e161,e165), (e161,e171), (e164,e166), (e167,e169), (e167,e171), (e167,e161), (e168,e170), (e169,e163), (e168,e172), (e171,e165), (e160,e168), (e168,e162), (e168,e164), (e162,e170), (e170,e166), (e164,e172), (e172,e166), (e160,e163), (e160,e169), (e160,e165), (e160,e171), (e161,e166), (e161,e170), (e161,e172), (e167,e170), (e167,e172), (e167,e162), (e167,e164), (e168,e163), (e169,e166), (e168,e165), (e171,e166), (e160,e166), (e160,e170), (e160,e172), (e167,e163), (e167,e165), (e168,e166), (e167,e166)})

TestCaseRtrnMedia = ({bookTstCntrl, booking}, {e151,e152,e153,e154,e155,e156, e157,e158,e159}, {(e151,e152), (e152,e153), (e153,e154), (e154,e155), (e156,e157), (e157,e158), (e158,e159), (e151,e156), (e157,e152), (e153,e158), (e159,e154), (e151,e153), (e152,e154), (e152,e158), (e153,e155), (e156,e158), (e156,e152), (e157,e159), (e158,e154), (e151,e157), (e157,e153), (e153,e159), (e159,e155), (e151,e154), (e151,e158), (e152,e155), (e152,e159), (e156,e159), *(e156,e153), (e157,e154), (e158,e155), (e151,e155), (e151,e159), (e156,e154), (e157,e155), (e156,e155)})*

B.3. Acceptance Test Model

We have developed test cases that cover the same services targeted in the component testing, Section B.2. The acceptance test model is illustrated in Figure 62. Using Definitions 1-4, we express the given test model as

AcceptanceTM = (P, T)

P = (tcn, tcm, sut)

tcn = AccSysTstCntrl

tcm = { }

sut = { LibrarySystem }

T = { TestCaseAddMember, TestCaseAddMedia, TestCaseBrowseMembers, TestCaseLibrarianBrowseMedia, TestCaseMemberBrowseMedia, TestCaseReserveMedia, TestCaseReturnMedia}

TestCaseAddMember = ({accSysTstCntrl, librarySystem }, {e218,e219,e220,e221, e222,e223,e224}, {(e218,e219), (e219,e221), (e218,e220), (e220,e221), (e222,e223), (e222,e224), (e218,e222), (e223,e219), (e224,e220), (e218,e221), (e222,e219), (e222,e220), (e218,e223), (e218,e224), (e223,e221), (e224,e221), (e222,e221)})

TestCaseAddMedia = ({accSysTstCntrl, librarySystem }, {e211,e212,e213,e214, e215,e216,e217}, {(e211,e212), (e212,e214), (e211,e213), (e213,e214), (e215,e216), (e215,e217), (e211,e215), (e216,e212), (e217,e213), (e211,e214), (e215,e212), (e215,e213), (e211,e216), (e211,e217), (e216,e214), (e217,e214), (e215,e214)})

TestCaseBrowseMembers = ({accSysTstCntrl, librarySystem }, {e201,e202,e203, e204,e205}, {(e201,e202), (e202,e203), (e204,e205), (e201,e204), (e205,e202), (e201,e203), (e204,e202), (e201,e205), (e205,e203), (e204,e203)})

TestCaseLibrarianBrowseMedia = ({accSysTstCntrl, librarySystem }, {e206,e207, e208,e209,e210}, {(e206,e207), (e207,e208), (e209,e210), (e206,e209), (e210,e207), (e206,e208), (e209,e207), (e206,e210), (e210,e208), (e209,e208)})

TestCaseMemberBrowseMedia = ({accSysTstCntrl, librarySystem }, {e225,e226, e227,e228,e229}, {(e225,e226), (e226,e227), (e228,e229), (e225,e228), (e229,e226), (e225,e227), (e228,e226), (e225,e229), (e229,e227), (e228,e227)})

TestCaseReserveMedia = ({accSysTstCntrl, librarySystem }, {e230,e231,e232,e233, e234,e235,e236}, {(e230,e231), (e231,e233), (e230,e232), (e232,e233), (e234,e235), (e234,e236), (e230,e234), (e235,e231), (e236,e232), (e230,e233), (e234,e231), (e234,e232), (e230,e235), (e230,e236), (e235,e233), (e236,e233), (e234,e233)})

TestCaseReturnMedia = ({accSysTstCntrl, librarySystem }, {e237,e238,e239, e240,e241}, {(e237,e238), (e238,e239), (e240,e241), (e237,e240), (e241,e238), (e237,e239), (e240,e238), (e237,e241), (e241,e239), (e240,e239)})

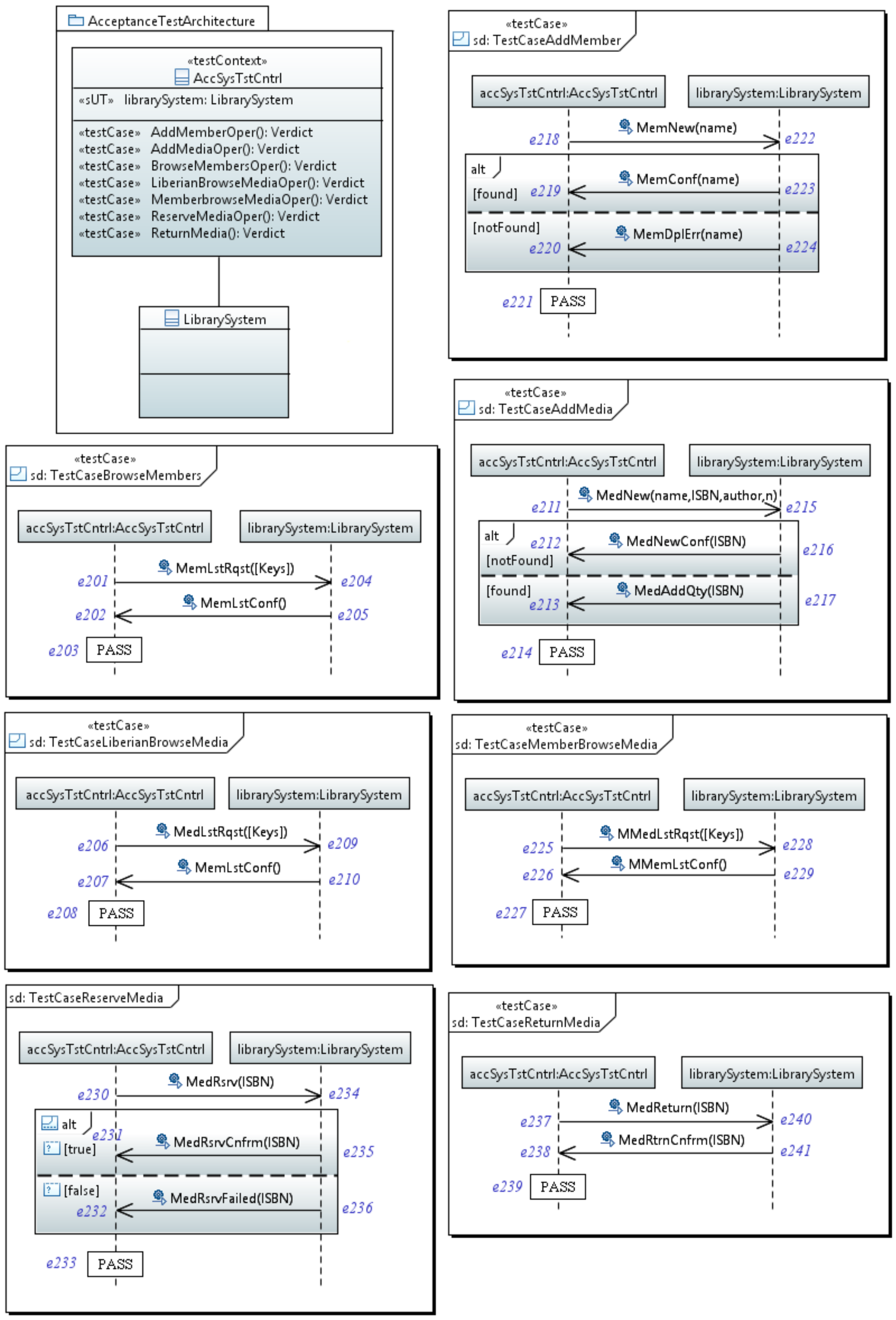

AcceptanceTestArchitecture
«testContext» AccSysTstCntrl
«sUT» librarySystem: LibrarySystem
«testCase» AddMemberOper(): Verdict
«testCase» AddMediaOper(): Verdict
«testCase» BrowseMembersOper(): Verdict
«testCase» LiberianBrowseMediaOper(): Verdict
«testCase» MemberbrowseMediaOper(): Verdict
«testCase» ReserveMediaOper(): Verdict
«testCase» ReturnMedia(): Verdict
LibrarySystem
«testCase» sd: TestCaseAddMember
accSysTstCntrl:AccSysTstCntrl
librarySystem:LibrarySystem
MemNew(name)
e218
e222
alt
[found]
e219
MemConf(name)
e223
[notFound]
e220
MemDplErr(name)
e224
e221
PASS
«testCase» sd: TestCaseAddMedia
MedNew(name,ISBN,author,n)
e211
e215
alt
e212
MedNewConf(ISBN)
[notFound]
e216
[found]
e213
MedAddQty(ISBN)
e217
e214
PASS
«testCase» sd: TestCaseBrowseMembers
MemLstRqst([Keys])
e201
e204
MemLstConf()
e202
e205
e203
PASS
«testCase» sd: TestCaseLiberianBrowseMedia
MedLstRqst([Keys])
e206
e209
MemLstConf()
e207
e210
e208
PASS
«testCase» sd: TestCaseMemberBrowseMedia
MMedLstRqst([Keys])
e225
e228
MMemLstConf()
e226
e229
e227
PASS
sd: TestCaseReserveMedia
MedRsrv(ISBN)
e230
e234
alt
e231
[true]
MedRsrvCnfrm(ISBN)
e235
[false]
MedRsrvFailed(ISBN)
e232
e236
e233
PASS
«testCase» sd: TestCaseReturnMedia
MedReturn(ISBN)
e237
e240
MedRtrnCnfrm(ISBN)
e238
e241
e239
PASS

Figure 62. Acceptance test model (*AcceptanceTM*)

Appendix C

Case Study: Integration Test Generation

In this subsection, we apply the tool, Section 6.3, on the component test models given in Section B.2. We use two different integration orders to build the integration test models. In the first, we integrate the test models in the following integration order: ((*LibrarianTM* + *MemberTM*) + *MediaTM*) + *BookingTM*. In the second, we integrate the test models in the following integration order: ((*LibrarianTM* + *MediaTM*) + *BookingTM*) + *MemberTM*. In the second integration, we integrate the *MemberTM* at the last iteration, while it has interfaces with the other components, to demonstrate the recovery of the test behavior of such interfaces through the carried-on component test cases.

C.1. First Integration Order

We generate integration test models by integrating the component test models in the following order: ((*LibrarianTM* + *MemberTM*) + *MediaTM*) + *BookingTM*. The integration goes through three iterations: (*LibrarianTM* + *MemberTM*), ((*LibrarianTM* + *MemberTM*) + *MediaTM*) then ((*LibrarianTM* + *MemberTM*) + *MediaTM*) + *BookingTM*.

C.1.1. First Iteration: *LibrarianTM+MemberTM*

In the first iteration, we integrate component test models of *Librarian* and *Member* to generate the first integration test model; let us call it *IntLibMemTM*. The tool starts by applying the identification process on the given test models. In the second phase of the identification process, the tool detects that the test control *LibTstCntrl* emulates the CUT *Member* through the following events: (*e15*,*e55*), (*e16*,*e56*), (*e18*,*e57*), (*e37*,*e61*) and (*e38*,*e62*). It also detects that the test control *MemTstCntrl* emulates the CUT *Librarian* through the following events: (*e51*,*e22*), (*e52*,*e23*), (*e53*,*e25*), (*e58*,*e42*) and (*e59*,*e43*). In the selection process, the tool selects four test cases as complete integration test cases: *LibrarianTM:TestCaseAddMember*, *LibrarianTM:TestCaseBrowseMembers*, *MemberTM:TestCaseAddMember* and *MemberTM:TestCaseBrowseMembers*. Next, the tool builds EDTs for the test cases to detect complement integration test cases. Figure 63 shows only two EDTs for test cases that have integration interactions. Hence, we have two pairs of complement integration test cases: *(LibrarianTM:TestCaseAddMember*, *MemberTM:TestCaseAddMember)* and *(LibrarianTM:TestCaseBrowseMembers*, *MemberTM:TestCaseBrowseMembers)*. The tool excludes the complete integration test cases since they are involved in the complement integration test cases. The next step is to generate the test behavior from the given complement integration test cases by merging each pair to generate integration test cases. To merge the first pair, the tool creates the shared events set, Definition 6, using the event matching expression, Definition 5, and creates the integration test control *TCi*.

se = { (e51,e22), (e52,e23), (e53,e25), (e15,e55), (e16,e56), (e18,e57), (e54, e20) }

Then, the tool generates the first integration test case by applying Definition 7:

IntTCAddMem = t1 + t2

= LibrarianTM:TestCaseAddMember + MemberTM:TestCaseAddMember

= (g(I1) U g(I2) , f(E1) U f(E2), f(R1) U f(R2))

g(I1) U g(I2) = g({libTstCntrl, librarian}) U g({memTstCntrl, member}) = {tci, librarian} U {tci, member} = {librarian, member, tci}

f(E1) U f(E2) = f({e14,e15,e16,e17,e18,e19,e20,e21,e22,e23,e24,e25,e26}) U f({e51,e52,e53,e54,e55,e56,e57}) = {e14,e55,e56,e17,e57,e19,e20,e21,e22,e23, e24,e25,e26} U {e22,e23,e25,e20,e55,e56,e57} = {e14,e17,e19,e20,e21,e22,e23, e24,e25,e26,e55,e56,e57}

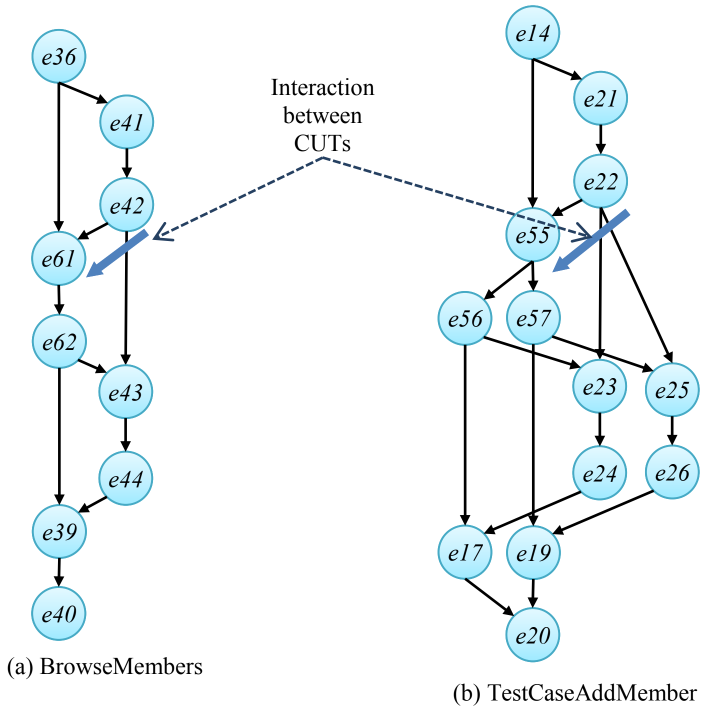


Figure 63. EDTs for Librarian Member integration

f(R1) U f(R2) = f({(e14,e15), (e15,e16), (e16,e17), (e17,e20), (e15,e18), (e18,e19), (e19,e20), (e21,e22), (e22,e23), (e23,e24), (e22,e25), (e25,e26), (e14,e21), (e22,e15), (e16,e23), (e24,e17), (e18,e25), (e26,e19), (e14,e16), (e14,e18), (e15,e17), (e15,e23), (e16,e20), (e15,e19), (e15,e25), (e18,e20), (e21,e23), (e21,e25), (e21,e15), (e22,e24), (e23,e17), (e22,e26), (e25,e19), (e14,e22), (e22,e16), (e22,e18), (e16,e24), (e24,e20), (e18,e26), (e26,e20), (e14,e17), (e14,e23), (e14,e19), (e14,e25), (e15,e20), (e15,e24), (e15,e26), (e21,e24), (e21,e26), (e21,e16), (e21,e18), (e22,e17), (e23,e20), (e22,e19), (e25,e20), (e14,e20), (e14,e24), (e14,e26), (e21,e17), (e21,e19), (e22,e20), (e21,e20)}) U f({(e51,e52), (e51,e53), (e52,e54), (e53,e54), (e55,e56), (e55,e57), (e51,e55), (e56,e52), (e57,e53), (e51,e54), (e55,e52), (e55,e53), (e51,e56), (e51,e57), (e56,e54), (e57,e54), (e55,e54)}) = {(e14,e55), (e55,e56), (e56,e17), (e17,e20), (e55,e57), (e57,e19), (e19,e20), (e21,e22), (e22,e23), (e23,e24), (e22,e25), (e25,e26), (e14,e21), (e22,e55), (e56,e23), (e24,e17), (e57,e25), (e26,e19), (e14,e56), (e14,e57), (e55,e17), (e55,e23), (e56,e20), (e55,e19), (e55,e25), (e57,e20), (e21,e23), (e21,e25), (e21,e55), (e22,e24), (e23,e17), (e22,e26), (e25,e19), (e14,e22), (e22,e56), (e22,e57), (e56,e24), (e24,e20), (e57,e26), (e26,e20), (e14,e17), (e14,e23), (e14,e19), (e14,e25), (e55,e20), (e55,e24), (e55,e26), (e21,e24), (e21,e26), (e21,e56), (e21,e57), (e22,e17), (e23,e20), (e22,e19), (e25,e20), (e14,e20), (e14,e24), (e14,e26), (e21,e17), (e21,e19), (e22,e20), (e21,e20)} U {(e22,e23), (e22,e25), (e23,e20), (e25,e20), (e55,e56), (e55,e57), (e22,e55), (e56,e23), (e57,e25), (e22,e20), (e55,e23), (e55,e25), (e22,e56), (e22,e57), (e56,e20), (e57,e20), (e55,e20)} = { (e14, e55), (e55, e56), (e56, e17), (e17, e20), (e55, e57), (e57, e19), (e19, e20), (e21, e22), (e22, e23), (e23, e24), (e22, e25), (e25, e26), (e14, e21), (e22, e55), (e56, e23), (e24, e17), (e57, e25), (e26, e19), (e14, e56), (e14, e57), (e55, e17), (e55, e23), (e56, e20), (e55, e19), (e55, e25), (e57, e20), (e21, e23), (e21, e25), (e21, e55), (e22, e24), (e23, e17), (e22, e26), (e25, e19), (e14, e22), (e22, e56), (e22, e57), (e56, e24), (e24, e20), (e57, e26), (e26, e20), (e14, e17), (e14, e23), (e14, e19), (e14, e25), (e55, e20), (e55, e24), (e55, e26), (e21, e24), (e21, e26), (e21, e56), (e21, e57), (e22, e17), (e23, e20), (e22, e19), (e25, e20), (e14, e20), (e14, e24), (e14, e26), (e21, e17), (e21, e19), (e22, e20), (e21, e20), (e14, e55) }

Next, the tool generates the second integration test case by merging the second pair. The tool starts by creating the shared events set.

se = { (e37,e61), (e38,e62), (e58,e42), (e59,e43), (e60,e40) }

Then, the tool generates the second integration test case by applying Definition 7:

IntTCBrwMem = $t_1 + t_2$

= *LibrarianTM:TestCaseBrowseMembers + MemberTM:TestCaseBrowseMembers*

= ($g(I_1)$ U $g(I_2)$, $f(E_1)$ U $f(E_2)$, $f(R_1)$ U $f(R_2)$))

$g(I_1)$ U $g(I_2)$ = *g({libTstCntrl, librarian}) U g({memTstCntrl, member}) = { tci, librarian } U { tci, member } = { librarian, member, tci }*

$f(E_1)$ U $f(E_2)$ = *f({e36,e37,e38,e39,e40,e41,e42,e43,e44}) U f({e58,e59,e60,e61, e62}) = {e36,e61,e62,e39,e40,e41,e42,e43,e44} U {e42,e43,e40,e61,e62} = {e36, e39,e40,e41,e42,e43,e44,e61,e62}* *f(R_1) U f(R_2) = f({ (e36,e37), (e37,e38), (e38,e39), (e39,e40), (e41,e42), (e42,e43), (e43,e44), (e36,e41), (e42,e37), (e38,e43), (e44,e39), (e36,e38), (e37,e39), (e37,e43), (e38,e40), (e41,e43), (e41,e37), (e42,e44), (e43,e39), (e36,e42), (e42,e38), (e38,e44), (e44,e40), (e36,e39), (e36,e43), (e37,e40), (e37,e44), (e41,e44), (e41,e38), (e42,e39), (e43,e40), (e36,e40), (e36,e44), (e41,e39), (e42,e40), (e41,e40) }) U f({ (e58,e59), (e59,e60), (e61,e62), (e58,e61), (e62,e59), (e58,e60), (e61,e59), (e58,e62), (e62,e60), (e61,e60) }) = {(e36,e61), (e61,e62), (e62,e39), (e39,e40), (e41,e42), (e42,e43), (e43,e44), (e36,e41), (e42,e61), (e62,e43), (e44,e39), (e36,e62), (e61,e39), (e61,e43), (e62,e40), (e41,e43), (e41,e61), (e42,e44), (e43,e39), (e36,e42), (e42,e62), (e62,e44), (e44,e40), (e36,e39), (e36,e43), (e61,e40), (e61,e44), (e41,e44), (e41,e62), (e42,e39), (e43,e40), (e36,e40), (e36,e44), (e41,e39), (e42,e40), (e41,e40) } U { (e42,e43), (e43,e40), (e61,e62), (e42,e61), (e62,e43), (e42,e40), (e61,e43), (e42,e62), (e62,e40), (e61,e40) } = { (e36, e61), (e61, e62), (e62, e39), (e39, e40), (e41, e42), (e42, e43), (e43, e44), (e36, e41), (e42, e61), (e62, e43), (e44, e39), (e36, e62), (e61, e39), (e61, e43), (e62, e40), (e41, e43), (e41, e61), (e42, e44), (e43, e39), (e36, e42), (e42, e62), (e62, e44), (e44, e40), (e36, e39), (e36, e43), (e61, e40), (e61, e44), (e41, e44), (e41, e62), (e42, e39), (e43, e40), (e36, e40), (e36, e44), (e41, e39), (e42, e40), (e41, e40), (e36, e61) }*

After generating the test behavior, the tool generates the test structure as follows:

T = { IntTCAddMem, IntTCBrwMem }

P = (TCi, {}, {Librarian, Member})

IntLibMemTM = (P, T)

The generated integration test model *IntLibMemTM*, shown in Figure 64, is exercised on the sub-system, and upon successful testing results, we move to the next integration iteration.

C.1.2. Second Iteration: *(LibrarianTM+MemberTM)+MediaTM*

In the second integration iteration, the tool generates the second integration test model, let us say *IntLibMemMedTM*, to examine the integration of ((*Librarian* + *Member*) + *Media*. The tool performs three test integrations. In the first test integration, the tool integrates the previously generated test model *IntLibMemTM* and the component test model *MediaTM*. The tool starts by applying on the given test models the identification process, which does not detect any shared test objects between the two test models. Hence, the tool stops the current test integration and proceeds to the next test integration.

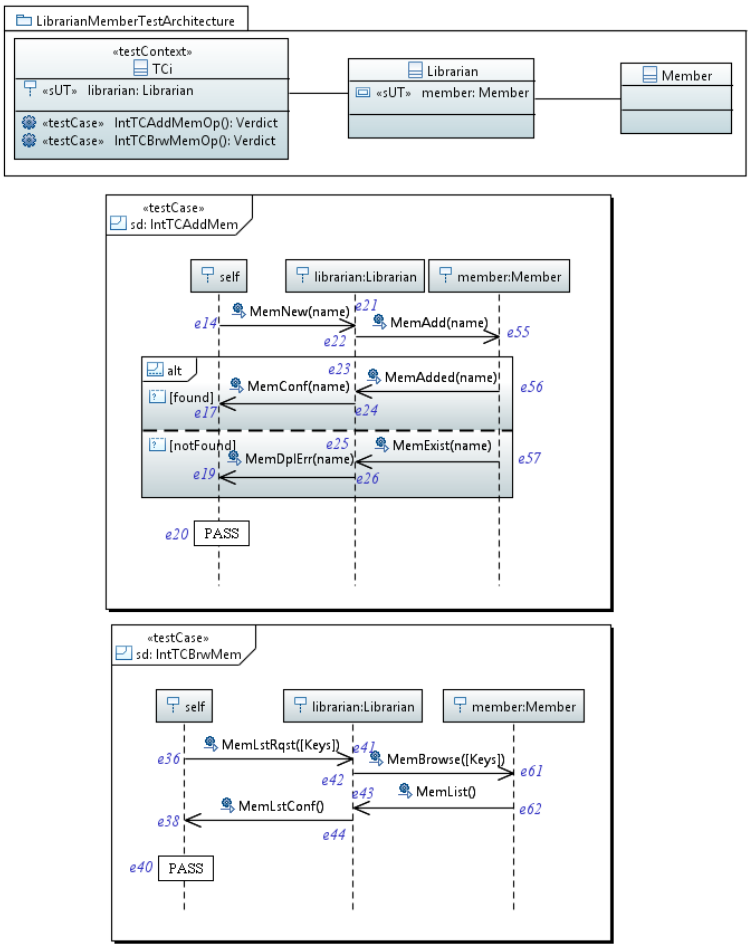


Figure 64. Generated integration test model (*IntLibMemTM*)

In the second test integration, the tool integrates the component test model *MediaTM* and *LibrarianTM*. The identification process detects that the test control *LibTstCntrl* emulates the CUT *Media* through the following events: *(e2,e105), (e3,e107), (e5,e106), (e28,e111)* and *(e29,e112)*, and the test control *MedTstCntrl* emulates the CUT *Librarian* through the following events: *(e101,e9), (e102,e12), (e103,e10), (e108,e33)* and *(e109,e34)*. The selection process selects four test cases as complete integration test cases: *MediaTM:TestCaseAddMedia, MediaTM:TestCaseLibBrowseMedia, LibrarianTM:TestCaseAddMedia* and *LibrarianTM:TestCaseBrowseMedia*. It also creates the EDTs and selects two pairs as complement integration test cases: *(MediaTM:TestCaseAddMedia, LibrarianTM:TestCaseAddMedia)* and *(MediaTM:TestCaseLibBrowseMedia, LibrarianTM:TestCaseBrowseMedia)*. The tool excludes the complete integration test cases since they are included in the second list. The next step is that the tool generates the first integration test case by merging the first pair of complement integration test cases. It examines the two test cases to create the shared events set.

se = { (e2,e105), (e3,e107), (e4,e106), (e101,e9), (e102,e12), (e103,e10), (e104,e7) }

Following that, the tool merges the two test cases

IntTCAddMed = $t_1 + t_2$

= *MediaTM:TestCaseAddMedia + LibrarianTM:TestCaseAddMedia*

= ($g(I_1)$ U $g(I_2)$, $f(E_1)$ U $f(E_2)$, $f(R_1)$ U $f(R_2)$))

$g(I_1)$ U $g(I_2)$ = *g({medTstCntrl, media}) U g({libTstCntrl, librarian}) = {tci, media} U {tci, librarian} = {librarian, media, tci}*

$f(E_1)$ U $f(E_2)$ = *f({e101,e102,e103,e104,e105,e106,e107}) U f({e1,e2,e3,e4,e5,e6,e7, e8,e9,e10,e11,e12,e13}) = {e9,e12,e10,e7,e105,e106,e107} U {e1,e105,e107,e4, e106,e6,e7,e8,e9,e10,e11,e12,e13} = {e1,e10,e105,e106,e107,e11,e12,e13,e4,e6, e7,e8,e9}*

$f(R_1)$ U $f(R_2)$ = *f({(e101,e102), (e101,e103), (e102,e104), (e103,e104), (e105,e106), (e105,e107), (e101,e105), (e106,e102), (e107,e103), (e101,e104), (e105,e102), (e105,e103), (e101,e106), (e101,e107), (e106,e104), (e107,e104), (e105,e104)}) U f({(e1,e2), (e2,e3), (e3,e4), (e4,e7), (e2,e5), (e5,e6), (e6,e7), (e8,e9), (e9,e10), (e10,e11), (e9,e12), (e12,e13), (e1,e8), (e9,e2), (e3,e10), (e11,e4), (e5,e12), (e13,e6), (e1,e3), (e1,e5), (e2,e4), (e2,e10), (e3,e7), (e2,e6), (e2,e12), (e5,e7), (e8,e10), (e8,e12), (e8,e2), (e9,e11), (e10,e4), (e9,e13), (e12,e6), (e1,e9), (e9,e3), (e9,e5), (e3,e11), (e11,e7), (e5,e13), (e13,e7), (e1,e4), (e1,e10), (e1,e6), (e1,e12), (e2,e7), (e2,e11), (e2,e13), (e8,e11), (e8,e13), (e8,e3), (e8,e5), (e9,e4), (e10,e7), (e9,e6), (e12,e7), (e1,e7), (e1,e11), (e1,e13), (e8,e4), (e8,e6), (e9,e7), (e8,e7)}) = {(e9,e12), (e9,e10), (e12,e7), (e10,e7), (e105,e106), (e105,e107), (e9,e105), (e106,e12), (e107,e10), (e9,e7), (e105,e12), (e105,e10), (e9,e106), (e9,e107), (e106,e7), (e107,e7), (e105,e7)} U {(e1,e105), (e105,e107), (e107,e4), (e4,e7), (e105,e106), (e106,e6), (e6,e7), (e8,e9), (e9,e10), (e10,e11), (e9,e12), (e12,e13), (e1,e8), (e9,e105), (e107,e10), (e11,e4), (e106,e12), (e13,e6), (e1,e107), (e1,e106), (e105,e4), (e105,e10), (e107,e7), (e105,e6), (e105,e12), (e106,e7), (e8,e10), (e8,e12), (e8,e105), (e9,e11), (e10,e4), (e9,e13), (e12,e6), (e1,e9), (e9,e107), (e9,e106), (e107,e11), (e11,e7), (e106,e13), (e13,e7), (e1,e4), (e1,e10), (e1,e6), (e1,e12), (e105,e7), (e105,e11), (e105,e13), (e8,e11), (e8,e13), (e8,e107), (e8,e106), (e9,e4), (e10,e7), (e9,e6), (e12,e7), (e1,e7), (e1,e11), (e1,e13), (e8,e4), (e8,e6), (e9,e7), (e8,e7)} = { (e33, e34), (e34, e31), (e111, e112), (e33, e111), (e112, e34), (e33, e31), (e111, e34), (e33, e112), (e112, e31), (e111, e31), (e27, e111), (e112, e30), (e30, e31), (e32, e33), (e34, e35), (e27, e32), (e35, e30), (e27, e112), (e111, e30), (e32, e34), (e32, e111), (e33, e35), (e34, e30), (e27, e33), (e112, e35), (e35, e31), (e27, e30), (e27, e34), (e111, e35), (e32, e35), (e32, e112), (e33, e30), (e27, e31), (e27, e35), (e32, e30), (e32, e31), (e33, e34), (e32, e34), (e27, e34) }*

Next, the tool generates the second integration test case by merging the second pair of complement integration test cases. It examines the two test cases to create the shared events set.

se = { (e28,e111), (e29,e112), (e108,e33), (e109,e34), (e110,e31) }

Following that, the tool merges the two test cases

IntTCLibBrwMed = $t_1 + t_2$

= MediaTM:TestCaseLibBrowseMedia, LibrarianTM:TestCaseBrowseMedia

$= (g(I_1)\ U\ g(I_2)\ ,\ f(E_1)\ U\ f(E_2),\ f(R_1)\ U f(R_2)\ )$

$g(I_1)\ U\ g(I_2)$ *= g({medTstCntrl, media}) U g({libTstCntrl, librarian}) = {tci, media} U {tci, librarian} = {librarian, media, tci}*

$f(E_1)\ U\ f(E_2)$ *= f({e108,e109,e110,e111,e112}) U f({e27,e28,e29,e30,e31,e32,e33, e34,e35}) = {e33,e34,e31,e111,e112} U {e27,e111,e112,e30,e31,e32,e33,e34,e35} = {e111,e112,e27,e30,e31,e32,e33,e34,e35}*

$f(R_1)\ U\ f(R_2)$ *= f({(e108,e109), (e109,e110), (e111,e112), (e108,e111), (e112,e109), (e108,e110), (e111,e109), (e108,e112), (e112,e110), (e111,e110)}) U f({(e27,e28), (e28,e29), (e29,e30), (e30,e31), (e32,e33), (e33,e34), (e34,e35), (e27,e32), (e33,e28), (e29,e34), (e35,e30), (e27,e29), (e28,e30), (e28,e34), (e29,e31), (e32,e34), (e32,e28), (e33,e35), (e34,e30), (e27,e33), (e33,e29), (e29,e35), (e35,e31), (e27,e30), (e27,e34), (e28,e31), (e28,e35), (e32,e35), (e32,e29), (e33,e30), (e34,e31), (e27,e31), (e27,e35), (e32,e30), (e33,e31), (e32,e31)}) = {(e33,e34), (e34,e31), (e111,e112), (e33,e111), (e112,e34), (e33,e31), (e111,e34), (e33,e112), (e112,e31), (e111,e31)} U {(e27,e111), (e111,e112), (e112,e30), (e30,e31), (e32,e33), (e33,e34), (e34,e35), (e27,e32), (e33,e111), (e112,e34), (e35,e30), (e27,e112), (e111,e30), (e111,e34), (e112,e31), (e32,e34), (e32,e111), (e33,e35), (e34,e30), (e27,e33), (e33,e112), (e112,e35), (e35,e31), (e27,e30), (e27,e34), (e111,e31), (e111,e35), (e32,e35), (e32,e112), (e33,e30), (e34,e31), (e27,e31), (e27,e35), (e32,e30), (e33,e31), (e32,e31)} = {(e33,e34), (e34,e31), (e111,e112), (e33,e111), (e112,e34), (e33,e31), (e111,e34), (e33,e112), (e112,e31), (e111,e31), (e27,e111), (e112,e30), (e30,e31), (e32,e33), (e34,e35), (e27,e32), (e35,e30), (e27,e112), (e111,e30), (e32,e34), (e32,e111), (e33,e35), (e34,e30), (e27,e33), (e112,e35), (e35,e31), (e27,e30), (e27,e34), (e111,e35), (e32,e35), (e32,e112), (e33,e30), (e27,e31), (e27,e35), (e32,e30), (e32,e31)}*

After generating the test behavior, the tool generates the test structure as follows:

T = { IntTCAddMed, IntTCLibBrwMed }

P = (TCi, {}, {Librarian, Media})

IntLibMemMedTM = (P, T)

The intermediate generated integration test model *IntLibMemMedTM*, from the second test integration, is shown in Figure 65. Following that, the tool proceeds to the next test integration.

In the third test integration, the tool examines the test cases of *MemberTM* against the currently generated test cases of *IntLibMemMedTM* and the test cases of *MediaTM*. The identification process does not detect shared test objects between *MemberTM* and *IntLibMemMedTM*, but it detects shared test objects between *MemberTM* and *MediaTM*. It detects that the test control

MedTstCntrl emulates the CUT *Member* through the following events: *(e113,e69) and (e114,e70)*, and the test control *MemTstCntrl* emulates the CUT *Media* through the following events: *(e64,e116) and (e65,e117)*. The selection process selects two test cases as complete integration test cases: *MediaTM:TestCaseMemBrowseMedia and MemberTM:TestCaseBrowseMedia*. It also creates the EDTs and selects one pair as complement integration test cases: *(MediaTM:TestCaseMemBrowseMedia, MemberTM:TestCaseBrowseMedia)*. The tool excludes the complete integration test cases since they are included in the second list.

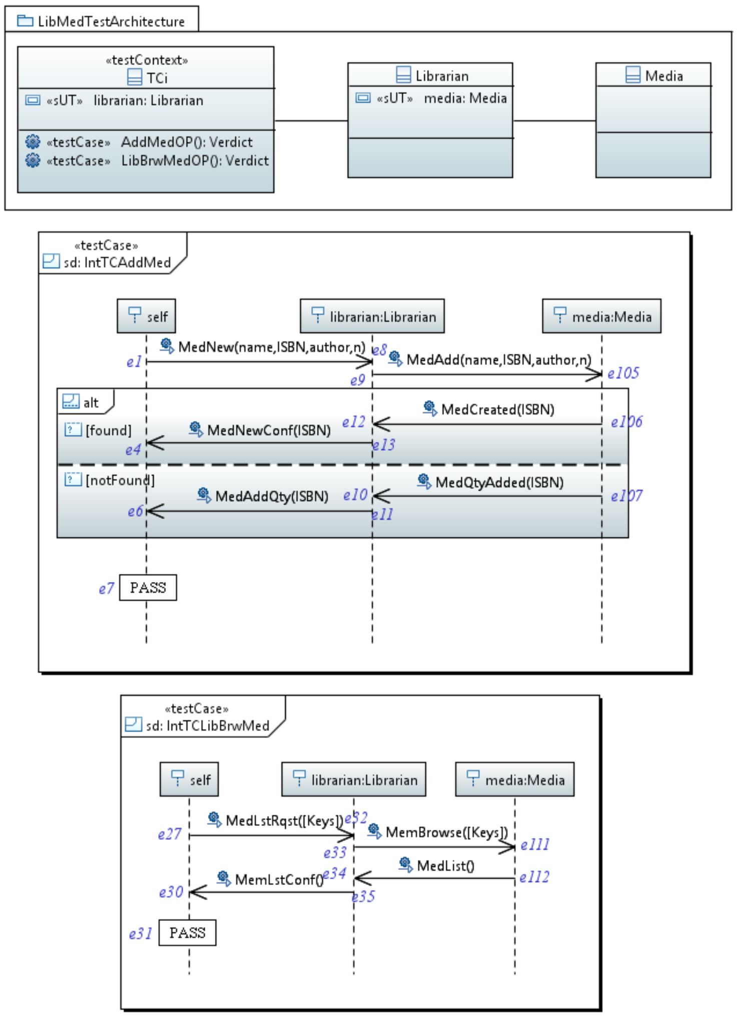


Figure 65. Intermediate generated test model (*IntLibMemMedTM*)

In the next step, the tool generates the third integration test case by merging the pair of complement integration test cases. It examines the two test cases to create the shared events set.

se = { (e113,e69), (e114,e70), (e64,e116), (e65,e117), (e115,e67) }

Following that, the tool merges the two test cases

$IntTCBrwMemMed = t_1 + t_2$

= MediaTM:TestCaseMemBrowseMedia + MemberTM:TestCaseBrowseMedia

= ($g(I_1)$ U $g(I_2)$, $f(E_1)$ U $f(E_2)$, $f(R_1)$ U $f(R_2)$)

$g(I_1)$ U $g(I_2)$ = g({medTstCntrl, media}) U g({memTstCntrl, member}) = {tci, media} U {tci, member} = {media, member, tci}

$f(E_1)$ U $f(E_2)$ = f({e113,e114,e115,e116,e117}) U f({e63,e64,e65,e66,e67,e68,e69, e70,e71}) = {e69,e70,e67,e116,e117} U {e63,e116,e117,e66,e67,e68,e69, e70,e71} = {e116,e117,e63,e66,e67,e68,e69,e70,e71}

$f(R_1)$ U $f(R_2)$ = f({(e113,e114), (e114,e115), (e116,e117), (e113,e116), (e117,e114), (e113,e115), (e116,e114), (e113,e117), (e117,e115), (e116,e115)}) U f({(e63,e64), (e64,e65), (e65,e66), (e66,e67), (e68,e69), (e69,e70), (e70,e71), (e63,e68), (e69,e64), (e65,e70), (e71,e66), (e63,e65), (e64,e66), (e64,e70), (e65,e67), (e68,e70), (e68,e64), (e69,e71), (e70,e66), (e63,e69), (e69,e65), (e65,e71), (e71,e67), (e63,e66), (e63,e70), (e64,e67), (e64,e71), (e68,e71), (e68,e65), (e69,e66), (e70,e67), (e63,e67), (e63,e71), (e68,e66), (e69,e67), (e68,e67)}) = {(e69,e70), (e70,e67), (e116,e117), (e69,e116), (e117,e70), (e69,e67), (e116,e70), (e69,e117), (e117,e67), (e116,e67)} U {(e63,e116), (e116,e117), (e117,e66), (e66,e67), (e68,e69), (e69,e70), (e70,e71), (e63,e68), (e69,e116), (e117,e70), (e71,e66), (e63,e117), (e116,e66), (e116,e70), (e117,e67), (e68,e70), (e68,e116), (e69,e71), (e70,e66), (e63,e69), (e69,e117), (e117,e71), (e71,e67), (e63,e66), (e63,e70), (e116,e67), (e116,e71), (e68,e71), (e68,e117), (e69,e66), (e70,e67), (e63,e67), (e63,e71), (e68,e66), (e69,e67), (e68,e67)} = { (e69, e70), (e70, e67), (e116, e117), (e69, e116), (e117, e70), (e69, e67), (e116, e70), (e69, e117), (e117, e67), (e116, e67), (e63, e116), (e117, e66), (e66, e67), (e68, e69), (e70, e71), (e63, e68), (e71, e66), (e63, e117), (e116, e66), (e68, e70), (e68, e116), (e69, e71), (e70, e66), (e63, e69), (e117, e71), (e71, e67), (e63, e66), (e63, e70), (e116, e71), (e68, e71), (e68, e117), (e69, e66), (e63, e67), (e63, e71), (e68, e66), (e68, e67), (e69, e70), (e68, e70), (e63, e70) }

After generating the test case, the tool updates the test structure as follows:

T = { IntTCAddMed, IntTCLibBrwMed, IntTCBrwMemMed }

P = (TCi, {}, {Librarian, Media, Member })

IntLibMemMedTM = (P, T)

The generated integration test model *IntLibMemMedTM*, for the second integration iteration, is shown in Figure 66. The test model is exercised on the integrated sub-system and upon a successful test, we move to the third and last integration iteration.

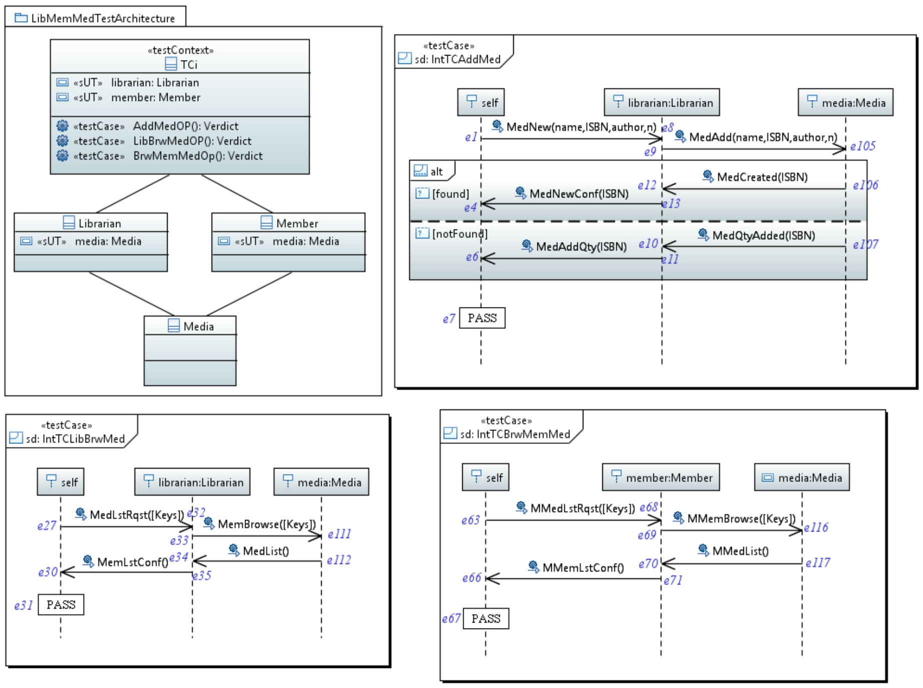


Figure 66. Generated integration test model (*IntLibMemMedTM*)

C.1.3. Third Iteration: *((LibrarianTM+MemberTM)+MediaTM)+BookingTM*

In the third integration iteration, the tool generates the third integration test model, let us call it *IntLibMemMedBkgTM*, to examine the integration of (((*Librarian* + *Member*) + *Media)* + *Booking)*. The tool performs four test integrations. In the first test integration, the tool integrates the previously generated test model *IntLibMemMedTM* and the component test model *BookingTM*. The tool begins by applying on the given test models the identification process, which does not detect any shared test objects between the two test models. Hence, the tool stops the test integration and proceeds to the next test integration. In the second test integration, the tool integrates the component test model *BookingTM* and *LibrarianTM*. The identification process does not detect any shared test objects between the two test models. Hence, the tool stops the test integration and proceeds to the next test integration.

In the third test integration, the tool integrates the component test model *BookingTM* and *MemberTM*. The identification process detects that the test control *BookTstCntrl* emulates the CUT *Member* through the following events: *(e151,e91), (e154,e92), (e160,e80), (e163,e81)* and *(e165,e83)*, and the test control *MemTstCntrl* emulates the CUT *Booking* through the following events: *(e73,e167), (e74,e170), (e76,e172), (e86,e156)* and *(e87,e159).* The selection process selects four test cases as complete integration test cases: *BookingTM:TestCaseRsrvMedia,* *BookingTM:TestCaseRtrnMedia*, *MemberTM:TestCaseRsrvMedia* and *MemberTM:TestCaseRtrnMedia*. It also creates the EDTs and selects two pairs as complement integration test cases: *(BookingTM:TestCaseRsrvMedia, MemberTM:TestCaseRsrvMedia) and (BookingTM:TestCaseRtrnMedia, MemberTM:TestCaseRtrnMedia)*. The tool excludes the complete integration test cases since they are included in the second list. For the next step, the tool generates the first integration test case by merging the first pair of complement integration test cases. It examines the two test cases to create the shared events set.

se = {(e160,e80), (e163,e81), (e165,e83), (e73,e167), (e74,e170), (e76,e172), (e166,e78)}

Following that, the tool merges the two test cases

IntTCRsrvBkgMem $= t_1 + t_2$

= BookingTM:TestCaseRsrvMedia + MemberTM:TestCaseRsrvMedia

$= (g(I_1)\ U\ g(I_2) , f(E_1)\ U\ f(E_2), f(R_1)\ U f(R_2))$

$g(I_1)\ U\ g(I_2)$ *= g({bookTstCntrl, booking}) U g({memTstCntrl, member}) = {tci, booking} U {tci, member} = {booking, member, tci}*

$f(E_1)\ U\ f(E_2)$ *= f({e160,e161,e162,e163,e164,e165,e166,e167,e168,e169,e170, e171,e172}) U f({e72,e73,e74,e75,e76,e77,e78,e79,e80,e81,e82,e83,e84}) = {e80, e161,e162,e81,e164,e83,e78,e167,e168,e169,e170,e171,e172} U {e72,e167,e170, e75,e172,e77,e78,e79,e80,e81,e82,e83,e84} = {e161,e162,e164,e167,e168,e169, e170,e171,e172,e72,e75,e77,e78,e79,e80,e81,e82,e83,e84}*

$f(R_1)\ U f(R_2)$ *= f({(e160,e161), (e161,e162), (e162,e163), (e163,e166), (e161,e164), (e164,e165), (e165,e166), (e167,e168), (e168,e169), (e169,e170), (e168,e171), (e171,e172), (e160,e167), (e168,e161), (e162,e169), (e170,e163), (e164,e171), (e172,e165), (e160,e162), (e160,e164), (e161,e163), (e161,e169), (e162,e166), (e161,e165), (e161,e171), (e164,e166), (e167,e169), (e167,e171), (e167,e161), (e168,e170), (e169,e163), (e168,e172), (e171,e165), (e160,e168), (e168,e162), (e168,e164), (e162,e170), (e170,e166), (e164,e172), (e172,e166), (e160,e163), (e160,e169), (e160,e165), (e160,e171), (e161,e166), (e161,e170), (e161,e172), (e167,e170), (e167,e172), (e167,e162), (e167,e164), (e168,e163), (e169,e166), (e168,e165), (e171,e166), (e160,e166), (e160,e170), (e160,e172), (e167,e163), (e167,e165), (e168,e166), (e167,e166)}) U f({(e72,e73), (e73,e74), (e74,e75), (e75,e78), (e73,e76), (e76,e77), (e77,e78), (e79,e80), (e80,e81), (e81,e82), (e80,e83), (e83,e84), (e72,e79), (e80,e73), (e74,e81), (e82,e75), (e76,e83), (e84,e77), (e72,e74), (e72,e76), (e73,e75), (e73,e81), (e74,e78), (e73,e77), (e73,e83), (e76,e78), (e79,e81), (e79,e83), (e79,e73), (e80,e82), (e81,e75), (e80,e84), (e83,e77), (e72,e80), (e80,e74), (e80,e76), (e74,e82), (e82,e78), (e76,e84), (e84,e78), (e72,e75), (e72,e81), (e72,e77), (e72,e83), (e73,e78), (e73,e82), (e73,e84), (e79,e82), (e79,e84), (e79,e74), (e79,e76), (e80,e75), (e81,e78), (e80,e77), (e83,e78), (e72,e78), (e72,e82), (e72,e84), (e79,e75), (e79,e77), (e80,e78), (e79,e78)}) = { (e80,e161), (e161,e162), (e162,e81), (e81,e78), (e161,e164), (e164,e83), (e83,e78), (e167,e168), (e168,e169),* *(e169,e170), (e168,e171), (e171,e172), (e80,e167), (e168,e161), (e162,e169), (e170,e81), (e164,e171), (e172,e83), (e80,e162), (e80,e164), (e161,e81), (e161,e169), (e162,e78), (e161,e83), (e161,e171), (e164,e78), (e167,e169), (e167,e171), (e167,e161), (e168,e170), (e169,e81), (e168,e172), (e171,e83), (e80,e168), (e168,e162), (e168,e164), (e162,e170), (e170,e78), (e164,e172), (e172,e78), (e80,e81), (e80,e169), (e80,e83), (e80,e171), (e161,e78), (e161,e170), (e161,e172), (e167,e170), (e167,e172), (e167,e162), (e167,e164), (e168,e81), (e169,e78), (e168,e83), (e171,e78), (e80,e78), (e80,e170), (e80,e172), (e167,e81), (e167,e83), (e168,e78), (e167,e78)} U { (e72,e167), (e167,e170), (e170,e75), (e75,e78), (e167,e172), (e172,e77), (e77,e78), (e79,e80), (e80,e81), (e81,e82), (e80,e83), (e83,e84), (e72,e79), (e80,e167), (e170,e81), (e82,e75), (e172,e83), (e84,e77), (e72,e170), (e72,e172), (e167,e75), (e167,e81), (e170,e78), (e167,e77), (e167,e83), (e172,e78), (e79,e81), (e79,e83), (e79,e167), (e80,e82), (e81,e75), (e80,e84), (e83,e77), (e72,e80), (e80,e170), (e80,e172), (e170,e82), (e82,e78), (e172,e84), (e84,e78), (e72,e75), (e72,e81), (e72,e77), (e72,e83), (e167,e78), (e167,e82), (e167,e84), (e79,e82), (e79,e84), (e79,e170), (e79,e172), (e80,e75), (e81,e78), (e80,e77), (e83,e78), (e72,e78), (e72,e82), (e72,e84), (e79,e75), (e79,e77), (e80,e78), (e79,e78)} = { (e80, e161), (e161, e162), (e162, e81), (e81, e78), (e161, e164), (e164, e83), (e83, e78), (e167, e168), (e168, e169), (e169, e170), (e168, e171), (e171, e172), (e80, e167), (e168, e161), (e162, e169), (e170, e81), (e164, e171), (e172, e83), (e80, e162), (e80, e164), (e161, e81), (e161, e169), (e162, e78), (e161, e83), (e161, e171), (e164, e78), (e167, e169), (e167, e171), (e167, e161), (e168, e170), (e169, e81), (e168, e172), (e171, e83), (e80, e168), (e168, e162), (e168, e164), (e162, e170), (e170, e78), (e164, e172), (e172, e78), (e80, e81), (e80, e169), (e80, e83), (e80, e171), (e161, e78), (e161, e170), (e161, e172), (e167, e170), (e167, e172), (e167, e162), (e167, e164), (e168, e81), (e169, e78), (e168, e83), (e171, e78), (e80, e78), (e80, e170), (e80, e172), (e167, e81), (e167, e83), (e168, e78), (e167, e78), (e72, e167), (e170, e75), (e75, e78), (e172, e77), (e77, e78), (e79, e80), (e81, e82), (e83, e84), (e72, e79), (e82, e75), (e84, e77), (e72, e170), (e72, e172), (e167, e75), (e167, e77), (e79, e81), (e79, e83), (e79, e167), (e80, e82), (e81, e75), (e80, e84), (e83, e77), (e72, e80), (e170, e82), (e82, e78), (e172, e84), (e84, e78), (e72, e75), (e72, e81), (e72, e77), (e72, e83), (e167, e82), (e167, e84), (e79, e82), (e79, e84), (e79, e170), (e79, e172), (e80, e75), (e80, e77), (e72, e78), (e72, e82), (e72, e84), (e79, e75), (e79, e77), (e79, e78), (e162, e82), (e162, e75), (e164, e84), (e164, e77), (e169, e75), (e169, e82), (e171, e77), (e171, e84), (e80, e161), (e161, e82), (e161, e75), (e161, e84), (e161, e77), (e168, e75), (e168, e82), (e168, e77), (e168, e84), (e72, e168), (e72, e169), (e72, e171), (e72, e161), (e72, e162), (e72, e164), (e79, e161), (e79, e162), (e79, e164), (e79, e168), (e79, e169), (e79, e171), (e79, e161), (e72, e161) }*

Next, the tool generates the second integration test case by merging the second pair of complement integration test cases. It examines the two test cases to create the shared events set.

se = { (e151,e91), (e154,e92), (e86,e156), (e87,e159), (e155,e89) }

Following that, the tool merges the two test cases

IntTCRtrnBkgMem = $t_1 + t_2$

= BookingTM:TestCaseRtrnMedia + MemberTM:TestCaseRtrnMedia

= (g(I_1) U g(I_2) , f(E_1) U f(E_2), f(R_1) U f(R_2))

g(I_1) U g(I_2) = g({bookTstCntrl, booking}) U g({memTstCntrl, member}) = {tci, booking} U {tci, member} = {booking, member, tci}

f(E_1) U f(E_2) = f({e151,e152,e153,e154,e155,e156,e157,e158,e159}) U f({e85,e86, e87,e88,e89,e90,e91,e92,e93}) = {e91,e152,e153,e92,e89,e156,e157,e158,e159} U {e85,e156,e159,e88,e89,e90,e91,e92,e93} = {e152,e153,e156,e157,e158,e159, e85,e88,e89,e90,e91,e92,e93}

f(R_1) U f(R_2) = f({(e151,e152), (e152,e153), (e153,e154), (e154,e155), (e156,e157), (e157,e158), (e158,e159), (e151,e156), (e157,e152), (e153,e158), (e159,e154), (e151,e153), (e152,e154), (e152,e158), (e153,e155), (e156,e158), (e156,e152), (e157,e159), (e158,e154), (e151,e157), (e157,e153), (e153,e159), (e159,e155), (e151,e154), (e151,e158), (e152,e155), (e152,e159), (e156,e159), (e156,e153), (e157,e154), (e158,e155), (e151,e155), (e151,e159), (e156,e154), (e157,e155), (e156,e155)}) U f({(e85,e86), (e86,e87), (e87,e88), (e88,e89), (e90,e91), (e91,e92), (e92,e93), (e85,e90), (e91,e86), (e87,e92), (e93,e88), (e85,e87), (e86,e88), (e86,e92), (e87,e89), (e90,e92), (e90,e86), (e91,e93), (e92,e88), (e85,e91), (e91,e87), (e87,e93), (e93,e89), (e85,e88), (e85,e92), (e86,e89), (e86,e93), (e90,e93), (e90,e87), (e91,e88), (e92,e89), (e85,e89), (e85,e93), (e90,e88), (e91,e89), (e90,e89)}) = {(e91,e152), (e152,e153), (e153,e92), (e92,e89), (e156,e157), (e157,e158), (e158,e159), (e91,e156), (e157,e152), (e153,e158), (e159,e92), (e91,e153), (e152,e92), (e152,e158), (e153,e89), (e156,e158), (e156,e152), (e157,e159), (e158,e92), (e91,e157), (e157,e153), (e153,e159), (e159,e89), (e91,e92), (e91,e158), (e152,e89), (e152,e159), (e156,e159), (e156,e153), (e157,e92), (e158,e89), (e91,e89), (e91,e159), (e156,e92), (e157,e89), (e156,e89)} U {(e85,e156), (e156,e159), (e159,e88), (e88,e89), (e90,e91), (e91,e92), (e92,e93), (e85,e90), (e91,e156), (e159,e92), (e93,e88), (e85,e159), (e156,e88), (e156,e92), (e159,e89), (e90,e92), (e90,e156), (e91,e93), (e92,e88), (e85,e91), (e91,e159), (e159,e93), (e93,e89), (e85,e88), (e85,e92), (e156,e89), (e156,e93), (e90,e93), (e90,e159), (e91,e88), (e92,e89), (e85,e89), (e85,e93), (e90,e88), (e91,e89), (e90,e89)} = { (e91, e152), (e152, e153), (e153, e92), (e92, e89), (e156, e157), (e157, e158), (e158, e159), (e91, e156), (e157, e152), (e153, e158), (e159, e92), (e91, e153), (e152, e92), (e152, e158), (e153, e89), (e156, e158), (e156, e152), (e157, e159), (e158, e92), (e91, e157), (e157, e153), (e153, e159), (e159, e89), (e91, e92), (e91, e158), (e152, e89), (e152, e159), (e156, e159), (e156, e153), (e157, e92), (e158, e89), (e91, e89), (e91, e159), (e156, e92), (e157, e89), (e156, e89), (e85, e156), (e159, e88), (e88, e89), (e90, e91), (e92, e93), (e85, e90), (e93, e88), (e85, e159), (e156, e88), (e90, e92), (e90, e156), (e91, e93), (e92, e88), (e85, e91), (e159, e93), (e93, e89), (e85, e88), (e85, e92), (e156, e93), (e90, e93), (e90, e159), (e91, e88), (e85, e89), (e85, e93), (e90, e88), (e90, e89), (e153, e93), (e153, e88), (e158, e88), (e158, e93), (e91, e152), (e152, e93), (e152, e88), (e157, e88), (e157, e93), (e85, e157), (e85, e158), (e85, e152), (e85, e153), (e90, e152), (e90, e153), (e90, e157), (e90, e158), (e90, e152), (e85, e152) }

After generating the test behavior, the tool generates the test structure as follows:

T = { IntTCRsrvBkgMem, IntTCRtrnBkgMem }

P = (TCi, {}, {Booking, Member})

IntLibMemMedBkgTM = (P, T)

The intermediate generated integration test model *IntLibMemMedBkgTM* is shown in Figure 67. Next, we move to the fourth test integration.

In the fourth test integration, the tool integrates the component test model *MediaTM* to *IntLibMemMedBkgTM* and *BookingTM*. Test cases of *MediaTM*, which are integrated with *IntLibMemMedBkgTM*, are not used to integrate with *BookingTM* test cases. The identification process detects that the test control *TCi* emulates the CUT *Media* through the following events: *(e161,e122), (e162,e123), (e164,e124), (e152,e128)* and *(e153,e129)*, and the test control *MedTstCntrl* emulates the CUT *Booking* through the following events: *(e118,e168), (e119,e169), (e120,e171), (e125,e157)* and *(e126,e158).* The selection process selects four test cases as complete integration test cases: *MediaTM:TestCaseRsrvMedia, MediaTM:TestCaseRtrnMedia, IntLibMemMedBkgTM:IntTCRsrvBkgMem* and *IntLibMemMedBkgTM:IntTCRtrnBkgMem*. It also creates the EDTs and selects two pairs as complement integration test cases: *(MediaTM:TestCaseRsrvMedia, IntLibMemMedBkgTM:IntTCRsrvBkgMem) and (MediaTM:TestCaseRtrnMedia, IntLibMemMedBkgTM:IntTCRtrnBkgMem)*. The tool excludes the complete integration test cases since they are included in the second pattern.

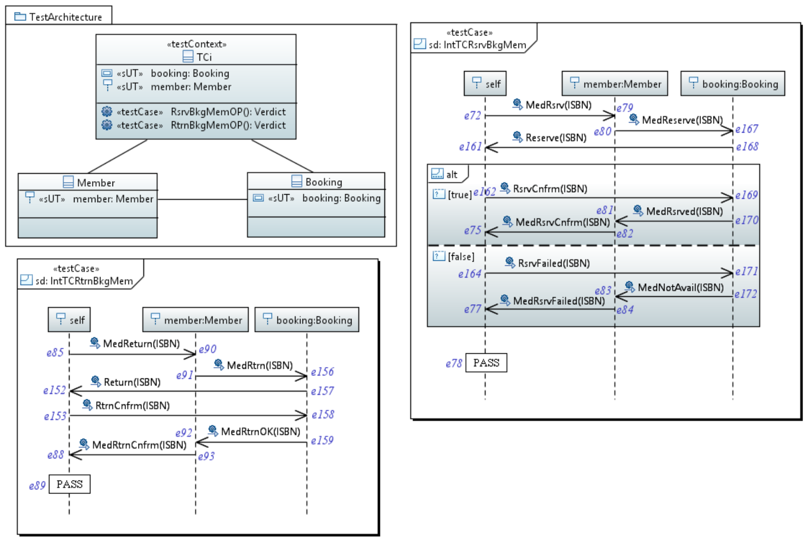


Figure 67. Intermediate generated test model (*IntLibMemMedBkgTM*)

For the next step, the tool generates the third integration test case by merging the first pair of complement integration test cases. It examines the two test cases to create the shared events set.

se = {(e161,e122), (e162,e123), (e164,e124), (e118,e168), (e119,e169), (e120,e171), (e121,e78)}

Following that, the tool merges the two test cases

IntTCRsrvBkgMemMed = $t_1 + t_2$

= *MediaTM:TestCaseRsrvMedia + IntLibMemMedBkgTM:IntTCRsrvBkgMem*

= $(g(I_1)\ U\ g(I_2)\ ,\ f(E_1)\ U\ f(E_2),\ f(R_1)\ U f(R_2)\ )$

$g(I_1)\ U\ g(I_2)$ = *g({medTstCntrl, media}) U g({booking, member, tci}) = {TCi, media} U {booking, member, TCi} = {booking, media, member, TCi}*

$f(E_1)\ U\ f(E_2)$ = *f({e118,e119,e120,e121,e122,e123,e124}) U f({e161,e162,e164, e167,e168,e169,e170,e171,e172,e72,e75,e77,e78,e79,e80,e81,e82,e83,e84}) = {e168,e169,e171,e78,e122,e123,e124} U {e122,e123,e124,e167,e168,e169,e170, e171,e172,e72,e75,e77,e78,e79,e80,e81,e82,e83,e84} = {e122,e123,e124,e167, e168,e169,e170,e171,e172,e72,e75,e77,e78,e79,e80,e81,e82,e83,e84}*

$f(R_1)\ U\ f(R_2)$ = *f({(e118,e119), (e119,e121), (e118,e120), (e120,e121), (e122,e123), (e122,e124), (e118,e122), (e123,e119), (e124,e120), (e118,e121), (e122,e119), (e122,e120), (e118,e123), (e118,e124), (e123,e121), (e124,e121), (e122,e121)}) U f({(e80,e161), (e161,e162), (e162,e81), (e81,e78), (e161,e164), (e164,e83), (e83,e78), (e167,e168), (e168,e169), (e169,e170), (e168,e171), (e171,e172), (e80,e167), (e168,e161), (e162,e169), (e170,e81), (e164,e171), (e172,e83), (e80,e162), (e80,e164), (e161,e81), (e161,e169), (e162,e78), (e161,e83), (e161,e171), (e164,e78), (e167,e169), (e167,e171), (e167,e161), (e168,e170), (e169,e81), (e168,e172), (e171,e83), (e80,e168), (e168,e162), (e168,e164), (e162,e170), (e170,e78), (e164,e172), (e172,e78), (e80,e81), (e80,e169), (e80,e83), (e80,e171), (e161,e78), (e161,e170), (e161,e172), (e167,e170), (e167,e172), (e167,e162), (e167,e164), (e168,e81), (e169,e78), (e168,e83), (e171,e78), (e80,e78), (e80,e170), (e80,e172), (e167,e81), (e167,e83), (e168,e78), (e167,e78), (e72,e167), (e170,e75), (e75,e78), (e172,e77), (e77,e78), (e79,e80), (e81,e82), (e83,e84), (e72,e79), (e82,e75), (e84,e77), (e72,e170), (e72,e172), (e167,e75), (e167,e77), (e79,e81), (e79,e83), (e79,e167), (e80,e82), (e81,e75), (e80,e84), (e83,e77), (e72,e80), (e170,e82), (e82,e78), (e172,e84), (e84,e78), (e72,e75), (e72,e81), (e72,e77), (e72,e83), (e167,e82), (e167,e84), (e79,e82), (e79,e84), (e79,e170), (e79,e172), (e80,e75), (e80,e77), (e72,e78), (e72,e82), (e72,e84), (e79,e75), (e79,e77), (e79,e78), (e162,e82), (e162,e75), (e164,e84), (e164,e77), (e169,e75), (e169,e82), (e171,e77), (e171,e84), (e80,e161), (e161,e82), (e161,e75), (e161,e84), (e161,e77), (e168,e75), (e168,e82), (e168,e77), (e168,e84), (e72,e168), (e72,e169), (e72,e171), (e72,e161), (e72,e162), (e72,e164), (e79,e161), (e79,e162), (e79,e164), (e79,e168), (e79,e169), (e79,e171), (e79,e161), (e72,e161)}) = {(e168,e169), (e169,e78), (e168,e171), (e171,e78), (e122,e123), (e122,e124), (e168,e122), (e123,e169), (e124,e171), (e168,e78), (e122,e169), (e122,e171), (e168,e123), (e168,e124), (e123,e78), (e124,e78), (e122,e78)} U {(e80,e122), (e122,e123), (e123,e81), (e81,e78), (e122,e124), (e124,e83), (e83,e78), (e167,e168), (e168,e169),* *(e169,e170), (e168,e171), (e171,e172), (e80,e167), (e168,e122), (e123,e169), (e170,e81), (e124,e171), (e172,e83), (e80,e123), (e80,e124), (e122,e81), (e122,e169), (e123,e78), (e122,e83), (e122,e171), (e124,e78), (e167,e169), (e167,e171), (e167,e122), (e168,e170), (e169,e81), (e168,e172), (e171,e83), (e80,e168), (e168,e123), (e168,e124), (e123,e170), (e170,e78), (e124,e172), (e172,e78), (e80,e81), (e80,e169), (e80,e83), (e80,e171), (e122,e78), (e122,e170), (e122,e172), (e167,e170), (e167,e172), (e167,e123), (e167,e124), (e168,e81), (e169,e78), (e168,e83), (e171,e78), (e80,e78), (e80,e170), (e80,e172), (e167,e81), (e167,e83), (e168,e78), (e167,e78), (e72,e167), (e170,e75), (e75,e78), (e172,e77), (e77,e78), (e79,e80), (e81,e82), (e83,e84), (e72,e79), (e82,e75), (e84,e77), (e72,e170), (e72,e172), (e167,e75), (e167,e77), (e79,e81), (e79,e83), (e79,e167), (e80,e82), (e81,e75), (e80,e84), (e83,e77), (e72,e80), (e170,e82), (e82,e78), (e172,e84), (e84,e78), (e72,e75), (e72,e81), (e72,e77), (e72,e83), (e167,e82), (e167,e84), (e79,e82), (e79,e84), (e79,e170), (e79,e172), (e80,e75), (e80,e77), (e72,e78), (e72,e82), (e72,e84), (e79,e75), (e79,e77), (e79,e78), (e123,e82), (e123,e75), (e124,e84), (e124,e77), (e169,e75), (e169,e82), (e171,e77), (e171,e84), (e80,e122), (e122,e82), (e122,e75), (e122,e84), (e122,e77), (e168,e75), (e168,e82), (e168,e77), (e168,e84), (e72,e168), (e72,e169), (e72,e171), (e72,e122), (e72,e123), (e72,e124), (e79,e122), (e79,e123), (e79,e124), (e79,e168), (e79,e169), (e79,e171), (e79,e122), (e72,e122)} = {(e168,e169), (e169,e78), (e168,e171), (e171,e78), (e122,e123), (e122,e124), (e168,e122), (e123,e169), (e124,e171), (e168,e78), (e122,e169), (e122,e171), (e168,e123), (e168,e124), (e123,e78), (e124,e78), (e122,e78), (e80,e122), (e123,e81), (e81,e78), (e124,e83), (e83,e78), (e167,e168), (e169,e170), (e171,e172), (e80,e167), (e170,e81), (e172,e83), (e80,e123), (e80,e124), (e122,e81), (e122,e83), (e167,e169), (e167,e171), (e167,e122), (e168,e170), (e169,e81), (e168,e172), (e171,e83), (e80,e168), (e123,e170), (e170,e78), (e124,e172), (e172,e78), (e80,e81), (e80,e169), (e80,e83), (e80,e171), (e122,e170), (e122,e172), (e167,e170), (e167,e172), (e167,e123), (e167,e124), (e168,e81), (e168,e83), (e80,e78), (e80,e170), (e80,e172), (e167,e81), (e167,e83), (e167,e78), (e72,e167), (e170,e75), (e75,e78), (e172,e77), (e77,e78), (e79,e80), (e81,e82), (e83,e84), (e72,e79), (e82,e75), (e84,e77), (e72,e170), (e72,e172), (e167,e75), (e167,e77), (e79,e81), (e79,e83), (e79,e167), (e80,e82), (e81,e75), (e80,e84), (e83,e77), (e72,e80), (e170,e82), (e82,e78), (e172,e84), (e84,e78), (e72,e75), (e72,e81), (e72,e77), (e72,e83), (e167,e82), (e167,e84), (e79,e82), (e79,e84), (e79,e170), (e79,e172), (e80,e75), (e80,e77), (e72,e78), (e72,e82), (e72,e84), (e79,e75), (e79,e77), (e79,e78), (e123,e82), (e123,e75), (e124,e84), (e124,e77), (e169,e75), (e169,e82), (e171,e77), (e171,e84), (e122,e82), (e122,e75), (e122,e84), (e122,e77), (e168,e75), (e168,e82), (e168,e77), (e168,e84), (e72,e168), (e72,e169), (e72,e171), (e72,e122), (e72,e123), (e72,e124), (e79,e122), (e79,e123), (e79,e124), (e79,e168), (e79,e169), (e79,e171)}*

Next, the tool generates the fourth integration test case by merging the second pair of complement integration test cases. It examines the two test cases to create the shared events set.

se = { (e152,e128), (e153,e129), (e125,e157), (e126,e158), (e89,e127) }

Following that, the tool merges the two test cases

IntTCRtrnBkgMemMed = $t_1 + t_2$

= MediaTM:TestCaseRtrnMedia + IntLibMemMedBkgTM:IntTCRtrnBkgMem

= ($g(I_1)$ U $g(I_2)$, $f(E_1)$ U $f(E_2)$, $f(R_1)$ U $f(R_2)$))

$g(I_1)$ U $g(I_2)$ = *g({medTstCntrl, media}) U g({booking, member, tci}) = {TCi, media} U {booking, member, TCi} = {booking, media, member, TCi}*

$f(E_1)$ U $f(E_2)$ = *f({e125,e126,e127,e128,e129}) U f({e152,e153,e156,e157,e158, e159,e85,e88,e89,e90,e91,e92,e93}) = {e157,e158,e127,e128,e129} U {e128, e129,e156,e157,e158,e159,e85,e88,e127,e90,e91,e92,e93} = {e127,e128,e129, e156,e157,e158,e159,e85,e88,e90,e91,e92,e93}*

$f(R_1)$ U $f(R_2)$ = *f({(e125,e126), (e126,e127), (e128,e129), (e125,e128), (e129,e126), (e125,e127), (e128,e126), (e125,e129), (e129,e127), (e128,e127)}) U f({(e91,e152), (e152,e153), (e153,e92), (e92,e89), (e156,e157), (e157,e158), (e158,e159), (e91,e156), (e157,e152), (e153,e158), (e159,e92), (e91,e153), (e152,e92), (e152,e158), (e153,e89), (e156,e158), (e156,e152), (e157,e159), (e158,e92), (e91,e157), (e157,e153), (e153,e159), (e159,e89), (e91,e92), (e91,e158), (e152,e89), (e152,e159), (e156,e159), (e156,e153), (e157,e92), (e158,e89), (e91,e89), (e91,e159), (e156,e92), (e157,e89), (e156,e89), (e85,e156), (e159,e88), (e88,e89), (e90,e91), (e92,e93), (e85,e90), (e93,e88), (e85,e159), (e156,e88), (e90,e92), (e90,e156), (e91,e93), (e92,e88), (e85,e91), (e159,e93), (e93,e89), (e85,e88), (e85,e92), (e156,e93), (e90,e93), (e90,e159), (e91,e88), (e85,e89), (e85,e93), (e90,e88), (e90,e89), (e153,e93), (e153,e88), (e158,e88), (e158,e93), (e91,e152), (e152,e93), (e152,e88), (e157,e88), (e157,e93), (e85,e157), (e85,e158), (e85,e152), (e85,e153), (e90,e152), (e90,e153), (e90,e157), (e90,e158), (e90,e152), (e85,e152)}) = {(e157,e158), (e158,e127), (e128,e129), (e157,e128), (e129,e158), (e157,e127), (e128,e158), (e157,e129), (e129,e127), (e128,e127)} U {(e91,e128), (e128,e129), (e129,e92), (e92,e127), (e156,e157), (e157,e158), (e158,e159), (e91,e156), (e157,e128), (e129,e158), (e159,e92), (e91,e129), (e128,e92), (e128,e158), (e129,e127), (e156,e158), (e156,e128), (e157,e159), (e158,e92), (e91,e157), (e157,e129), (e129,e159), (e159,e127), (e91,e92), (e91,e158), (e128,e127), (e128,e159), (e156,e159), (e156,e129), (e157,e92), (e158,e127), (e91,e127), (e91,e159), (e156,e92), (e157,e127), (e156,e127), (e85,e156), (e159,e88), (e88,e127), (e90,e91), (e92,e93), (e85,e90), (e93,e88), (e85,e159), (e156,e88), (e90,e92), (e90,e156), (e91,e93), (e92,e88), (e85,e91), (e159,e93), (e93,e127), (e85,e88), (e85,e92), (e156,e93), (e90,e93), (e90,e159), (e91,e88), (e85,e127), (e85,e93), (e90,e88), (e90,e127), (e129,e93), (e129,e88), (e158,e88), (e158,e93), (e91,e128), (e128,e93), (e128,e88), (e157,e88), (e157,e93), (e85,e157), (e85,e158), (e85,e128), (e85,e129), (e90,e128), (e90,e129), (e90,e157), (e90,e158), (e90,e128), (e85,e128)} = {(e157,e158), (e158,e127), (e128,e129), (e157,e128), (e129,e158), (e157,e127), (e128,e158), (e157,e129), (e129,e127), (e128,e127), (e91,e128), (e129,e92), (e92,e127), (e156,e157), (e158,e159), (e91,e156), (e159,e92), (e91,e129), (e128,e92), (e156,e158), (e156,e128), (e157,e159), (e158,e92), (e91,e157), (e129,e159), (e159,e127), (e91,e92), (e91,e158), (e128,e159), (e156,e159), (e156,e129), (e157,e92), (e91,e127), (e91,e159), (e156,e92), (e156,e127), (e85,e156), (e159,e88), (e88,e127), (e90,e91),* *(e92,e93), (e85,e90), (e93,e88), (e85,e159), (e156,e88), (e90,e92), (e90,e156), (e91,e93), (e92,e88), (e85,e91), (e159,e93), (e93,e127), (e85,e88), (e85,e92), (e156,e93), (e90,e93), (e90,e159), (e91,e88), (e85,e127), (e85,e93), (e90,e88), (e90,e127), (e129,e93), (e129,e88), (e158,e88), (e158,e93), (e128,e93), (e128,e88), (e157,e88), (e157,e93), (e85,e157), (e85,e158), (e85,e128), (e85,e129), (e90,e128), (e90,e129), (e90,e157), (e90,e158)}*

After generating the test behavior, the tool updates the test structure as follows:

T = { IntTCRsrvBkgMemMed, IntTCRtrnBkgMemMed }

P = (TCi, {}, {Booking, Member, Media})

IntLibMemMedBkgTM = (P, T)

Figure 68 shows the generated integration test model for the third integration iteration.

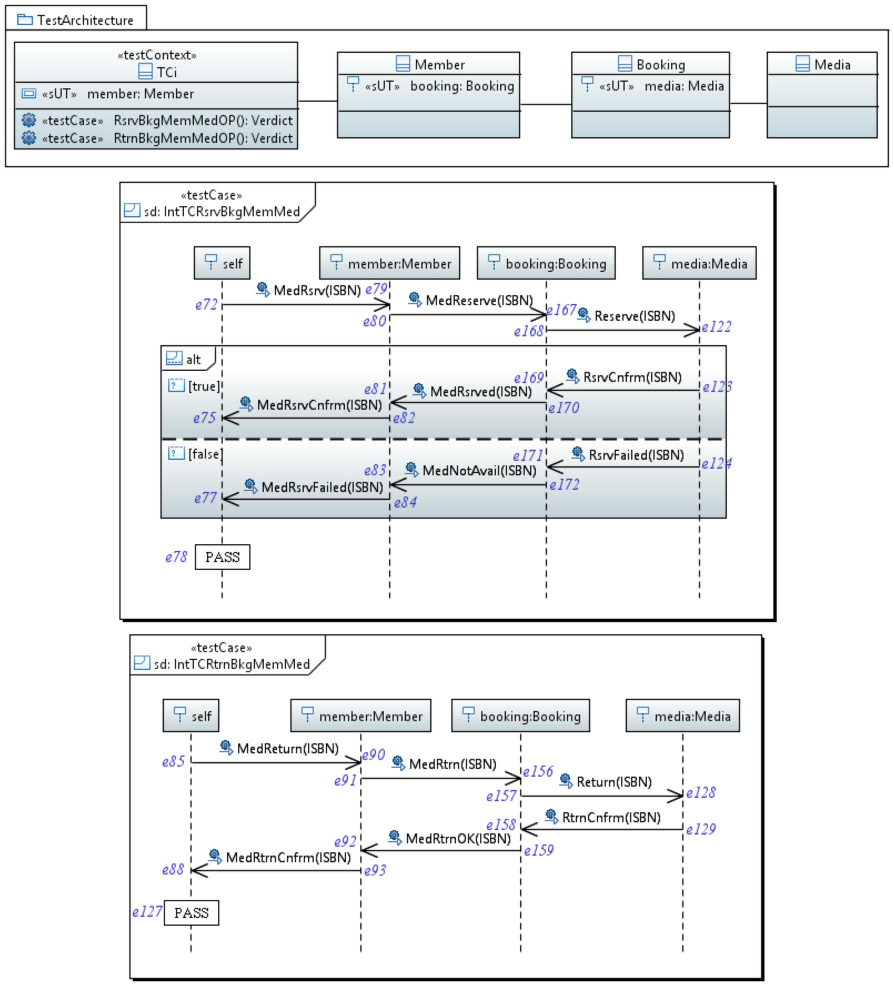


Figure 68. Generated integration test model (*IntLibMemMedBkgTM*)

C.2. Second Integration Order

We generate integration test models by integrating the component test models in the following order: ((*LibrarianTM* + *MediaTM*) + *BookingTM*) + *MemberTM*. The integration goes through three iterations as follows:

1. (*LibrarianTM* + *MediaTM*),
2. ((*LibrarianTM* + *MediaTM*) + *BookingTM*) then
3. ((*LibrarianTM* + *MediaTM*) + *BookingTM*) + *MemberTM*

C.2.1. First Iteration: *LibrarianTM+MediaTM*

In the first iteration, we integrate component test models of *Librarian* and *Media* to generate the first integration test model; let us call it *IntLibMedTM*. The identification process does not detect shared test objects in the first phase. In the second phase of the identification process, the tool detects that the test control *LibTstCntrl* emulates the CUT *Media* through the following events: *(e2,e105), (e3,e107), (e5,e106), (e28,e111)* and *(e29,e112)*. It also detects that the test control *MedTstCntrl* emulates the CUT *Librarian* through the following events: *(e108,e33), (e109,e34), (e101,e9), (e102,e12)* and *(e103,e10)*. In the selection process, the tool selects four test cases as complete integration test cases: *LibrarianTM:TestCaseAddMedia*, *LibrarianTM:TestCaseBrowseMedia*, *MediaTM:TestCaseAddMedia* and *MediaTM:TestCaseLibBrowseMedia*. Next, the tool builds EDTs for the test cases to detect complement integration test cases. Figure 69 shows two EDTs for test cases that have integration interactions. Hence, we have two pairs of complement integration test cases: *(LibrarianTM:TestCaseAddMedia*, *MediaTM:TestCaseAddMedia)* and *(LibrarianTM:TestCaseBrowseMedia*, *MediaTM:TestCaseLibBrowseMedia)*. The tool excludes the complete integration test cases since they are involved in the complement integration test cases. The next step is to generate the test behavior from the given complement integration test cases by merging each pair to generate an integration test case. To merge the first pair, the tool creates the integration test control *TCi* and creates the shared events set, Definition 6, using the event matching expression, Definition 5.

se = { (e2,e105), (e3,e107), (e5,e106), (e101,e9), (e102,e12), (e103,e10), (e104, e7) }

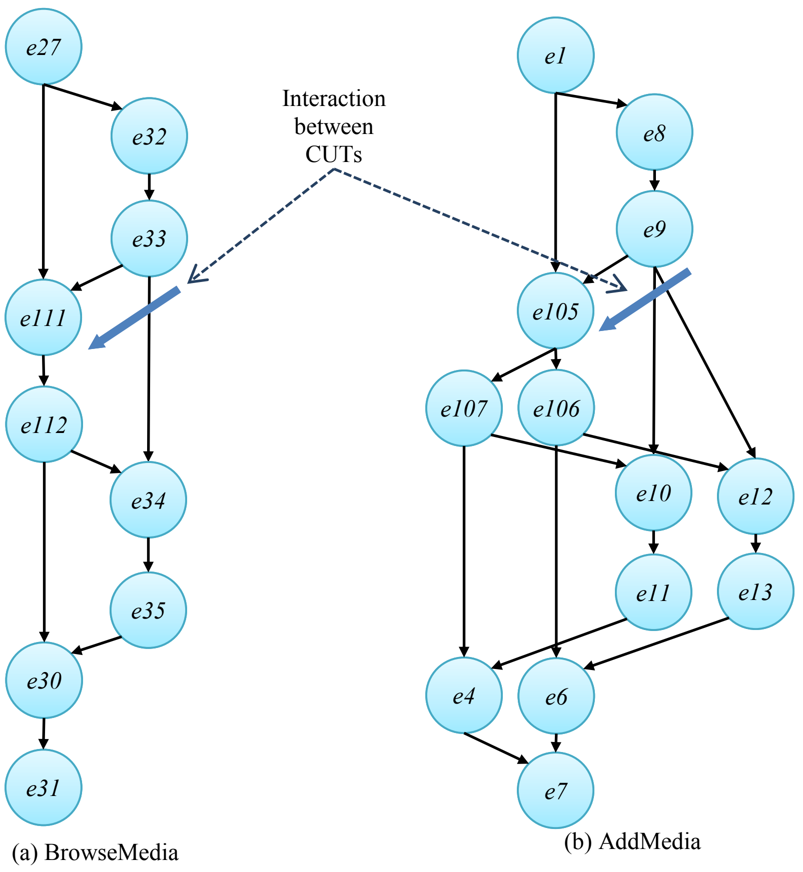


Figure 69. EDTs for Librarian Media integration

Then, the tool generates the first integration test case by applying Definition 7:

$IntTCAddMed = t_1 + t_2$

$= LibrarianTM{:}TestCaseAddMedia + MediaTM{:}TestCaseAddMedia$

$= (\ g(I_1)\ \cup\ g(I_2)\ ,\ f(E_1)\ \cup\ f(E_2),\ f(R_1) \cup f(R_2)\ )$

$g(I_1)\ \cup\ g(I_2) = g(\{libTstCntrl, librarian\}) \cup g(\{medTstCntrl, media\}) = \{tci, librarian\} \cup \{tci, media\}$

$= \{librarian, media, tci\}$

$f(E_1)\ \cup\ f(E_2) = f(\{e1,e2,e3,e4,e5,e6,e7,e8,e9,e10,e11,e12,e13\})\ \cup\ f(\{e101,e102, e103,e104,e105,e106,e107\}) = \{e1,e105,e107,e4,e106,e6,e7,e8,e9,e10,e11,e12, e13\}\ \cup \{e9,e12,e10,e7,e105,e106,e107\} = \{e1,e10,e105,e106,e107,e11,e12,e13, e4,e6,e7,e8,e9\}$

$f(R_1) \cup f(R_2) = f(\{(e1,e2), (e2,e3), (e3,e4), (e4,e7), (e2,e5), (e5,e6), (e6,e7), (e8,e9), (e9,e10), (e10,e11), (e9,e12), (e12,e13), (e1,e8), (e9,e2), (e3,e10), (e11,e4), (e5,e12), (e13,e6), (e1,e3), (e1,e5), (e2,e4), (e2,e10), (e3,e7), (e2,e6), (e2,e12), (e5,e7), (e8,e10), (e8,e12), (e8,e2), (e9,e11), (e10,e4), (e9,e13), (e12,e6), (e1,e9), (e9,e3), (e9,e5), (e3,e11), (e11,e7), (e5,e13), (e13,e7), (e1,e4), (e1,e10), (e1,e6), (e1,e12), (e2,e7), (e2,e11), (e2,e13), (e8,e11), (e8,e13), (e8,e3), (e8,e5), (e9,e4), (e10,e7), (e9,e6), (e12,e7), (e1,e7), (e1,e11), (e1,e13), (e8,e4), (e8,e6), (e9,e7), (e8,e7)\}) \cup f(\{(e101,e102), (e101,e103), (e102,e104), (e103,e104), (e105,e106), (e105,e107), (e101,e105), (e106,e102), (e107,e103),$

(e101,e104), (e105,e102), (e105,e103), (e101,e106), (e101,e107), (e106,e104), (e107,e104), (e105,e104)}) = {(e1,e105), (e105,e107), (e107,e4), (e4,e7), (e105,e106), (e106,e6), (e6,e7), (e8,e9), (e9,e10), (e10,e11), (e9,e12), (e12,e13), (e1,e8), (e9,e105), (e107,e10), (e11,e4), (e106,e12), (e13,e6), (e1,e107), (e1,e106), (e105,e4), (e105,e10), (e107,e7), (e105,e6), (e105,e12), (e106,e7), (e8,e10), (e8,e12), (e8,e105), (e9,e11), (e10,e4), (e9,e13), (e12,e6), (e1,e9), (e9,e107), (e9,e106), (e107,e11), (e11,e7), (e106,e13), (e13,e7), (e1,e4), (e1,e10), (e1,e6), (e1,e12), (e105,e7), (e105,e11), (e105,e13), (e8,e11), (e8,e13), (e8,e107), (e8,e106), (e9,e4), (e10,e7), (e9,e6), (e12,e7), (e1,e7), (e1,e11), (e1,e13), (e8,e4), (e8,e6), (e9,e7), (e8,e7)} U {(e9,e12), (e9,e10), (e12,e7), (e10,e7), (e105,e106), (e105,e107), (e9,e105), (e106,e12), (e107,e10), (e9,e7), (e105,e12), (e105,e10), (e9,e106), (e9,e107), (e106,e7), (e107,e7), (e105,e7)} = {(e1,e105), (e105,e107), (e107,e4), (e4,e7), (e105,e106), (e106,e6), (e6,e7), (e8,e9), (e9,e10), (e10,e11), (e9,e12), (e12,e13), (e1,e8), (e9,e105), (e107,e10), (e11,e4), (e106,e12), (e13,e6), (e1,e107), (e1,e106), (e105,e4), (e105,e10), (e107,e7), (e105,e6), (e105,e12), (e106,e7), (e8,e10), (e8,e12), (e8,e105), (e9,e11), (e10,e4), (e9,e13), (e12,e6), (e1,e9), (e9,e107), (e9,e106), (e107,e11), (e11,e7), (e106,e13), (e13,e7), (e1,e4), (e1,e10), (e1,e6), (e1,e12), (e105,e7), (e105,e11), (e105,e13), (e8,e11), (e8,e13), (e8,e107), (e8,e106), (e9,e4), (e10,e7), (e9,e6), (e12,e7), (e1,e7), (e1,e11), (e1,e13), (e8,e4), (e8,e6), (e9,e7), (e8,e7)}

After that, the tool generates the second integration test case by merging the second pair. The tool starts by creating the shared events set.

se = { (e28,e111), (e29,e112), (e108,e33), (e109,e34), (e110,e31) }

Then, the tool generates the second integration test case by applying Definition 7:

IntTCBrwMed = $t_1 + t_2$

= *LibrarianTM:TestCaseBrowseMedia + MediaTM:TestCaseLibBrowseMedia*

= $(g(I_1)\ U\ g(I_2)\ ,\ f(E_1)\ U\ f(E_2),\ f(R_1)\ U f(R_2)\ )$

$g(I_1)\ U\ g(I_2)$ = *g({libTstCntrl, librarian}) U g({medTstCntrl, media}) = {tci, librarian} U {tci, media} = {librarian, media, tci}*

$f(E_1)\ U\ f(E_2)$ = *f({e27,e28,e29,e30,e31,e32,e33,e34,e35}) U f({e108,e109,e110, e111,e112}) = {e27,e111,e112,e30,e31,e32,e33,e34,e35} U {e33,e34,e31,e111, e112} = {e111,e112,e27,e30,e31,e32,e33,e34,e35}*

$f(R_1)\ U f(R_2)$ = *f({(e27,e28), (e28,e29), (e29,e30), (e30,e31), (e32,e33), (e33,e34), (e34,e35), (e27,e32), (e33,e28), (e29,e34), (e35,e30), (e27,e29), (e28,e30), (e28,e34), (e29,e31), (e32,e34), (e32,e28), (e33,e35), (e34,e30), (e27,e33), (e33,e29), (e29,e35), (e35,e31), (e27,e30), (e27,e34), (e28,e31), (e28,e35), (e32,e35), (e32,e29), (e33,e30), (e34,e31), (e27,e31), (e27,e35), (e32,e30), (e33,e31), (e32,e31)}) U f({(e108,e109), (e109,e110), (e111,e112), (e108,e111), (e112,e109), (e108,e110), (e111,e109), (e108,e112), (e112,e110), (e111,e110)}) = {(e27,e111), (e111,e112), (e112,e30), (e30,e31), (e32,e33), (e33,e34), (e34,e35), (e27,e32), (e33,e111), (e112,e34), (e35,e30), (e27,e112), (e111,e30), (e111,e34), (e112,e31), (e32,e34), (e32,e111), (e33,e35), (e34,e30), (e27,e33), (e33,e112), (e112,e35), (e35,e31), (e27,e30), (e27,e34), (e111,e31), (e111,e35), (e32,e35),* *(e32,e112), (e33,e30), (e34,e31), (e27,e31), (e27,e35), (e32,e30), (e33,e31), (e32,e31)} U {(e33,e34), (e34,e31), (e111,e112), (e33,e111), (e112,e34), (e33,e31), (e111,e34), (e33,e112), (e112,e31), (e111,e31)} = {(e27,e111), (e111,e112), (e112,e30), (e30,e31), (e32,e33), (e33,e34), (e34,e35), (e27,e32), (e33,e111), (e112,e34), (e35,e30), (e27,e112), (e111,e30), (e111,e34), (e112,e31), (e32,e34), (e32,e111), (e33,e35), (e34,e30), (e27,e33), (e33,e112), (e112,e35), (e35,e31), (e27,e30), (e27,e34), (e111,e31), (e111,e35), (e32,e35), (e32,e112), (e33,e30), (e34,e31), (e27,e31), (e27,e35), (e32,e30), (e33,e31), (e32,e31)}*

After generating the test behavior, the tool generates the test structure as follows:

T = { IntTCAddMed, IntTCBrwMed }

P = (TCi, {}, {Librarian, Media})

IntLibMedTM = (P, T)

The generated integration test model *IntLibMedTM*, shown in Figure 70, is exercised on the sub-system, and upon a successful test, we move to the next integration iteration.

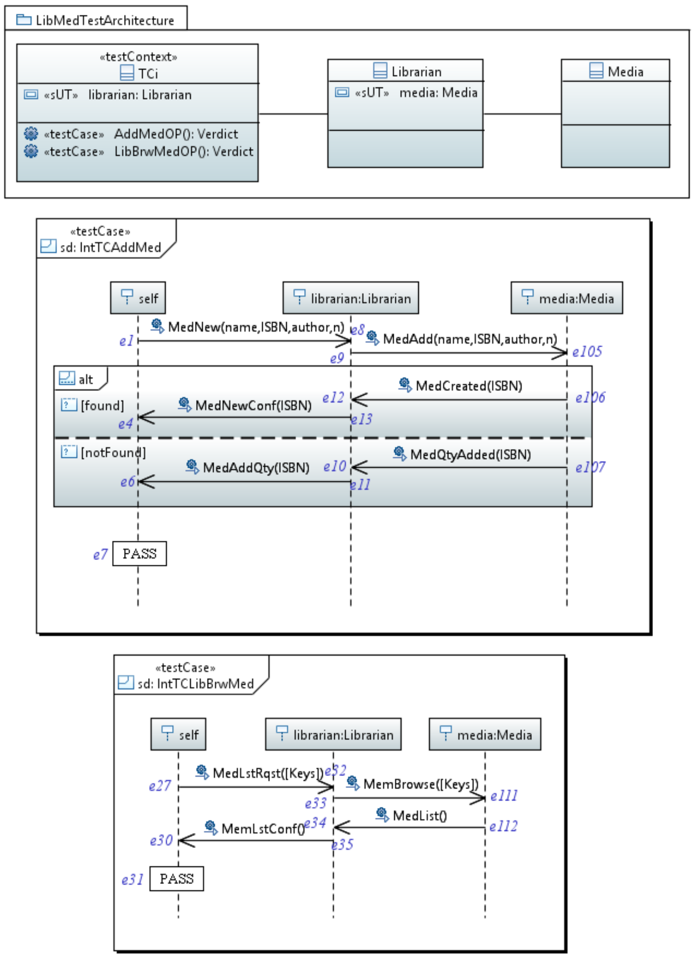


Figure 70. Generated integration test model (*IntLibMedTM*)

C.2.2. Second Iteration: *(LibrarianTM+MediaTM)+BookingTM*

In the second integration iteration, the tool generates the second integration test model, let us call it *IntLibMedBkgTM*, to examine the integration of ((*Librarian* + *Media*) + *Booking*. The tool performs three test integrations. In the first test integration, the tool integrates the previously generated test model *IntLibMedTM* and the component test model *BookingTM*. The tool starts by applying on the given test models the identification process, which does not detect any shared test objects between the two test models. Hence, the tool stops the current test integration and proceeds to the next test integration. In the second test integration, the tool integrates the component test model *BookingTM* and *MediaTM*. The identification process detects that the test control *BookLibTstCntrl* emulates the CUT *Media* through the following events: *(e152,e128), (e153,e129), (e161,e122), (e162,e123)* and *(e164,e124)*, and the test control *MedTstCntrl* emulates the CUT *Booking* through the following events: *(e118,e168), (e119,e169), (e120,e171), (e125,e157)* and *(e126,e158).* The selection process selects four test cases as complete integration test cases: *MediaTM:TestCaseRsrvMedia, MediaTM:TestCaseRtrnMedia, BookingTM:TestCaseRtrnMedia* and *BookingTM:TestCaseRsrvMedia*. It also creates the EDTs and selects two pairs as complement integration test cases: *(BookingTM:TestCaseRsrvMedia, MediaTM:TestCaseRsrvMedia)* and *(BookingTM:TestCaseRtrnMedia, MediaTM:TestCaseRtrnMedia)*. The tool excludes the complete integration test cases since they are included in the second list. The next step is that the tool generates the first integration test case by merging the first pair of complement integration test cases. It examines the two test cases to create the shared events set.

$$se = \{ (e161,e122), (e162,e123), (e164,e124), (e118,e168), (e119,e169), (e120,e171), (e166,e121) \}$$

Following that, the tool merges the two test cases

$IntTCRsrvMedia = t_1 + t_2$

$= BookingTM{:}TestCaseRsrvMedia + MediaTM{:}TestCaseRsrvMedia$

$= (g(I_1) \cup g(I_2) ,\ f(E_1) \cup f(E_2),\ f(R_1) \cup f(R_2))$

$g(I_1) \cup g(I_2) = g(\{bookTstCntrl, booking\}) \cup g(\{medTstCntrl, media\}) = \{tci, booking\} \cup \{tci, media\}$

$= \{booking, media, tci\}$

$f(E_1) \cup f(E_2) = f(\{e160,e161,e162,e163,e164,e165,e166,e167,e168,e169,e170,e171, e172\}) \cup f(\{e118,e119,e120,e121,e122,e123,e124\}) = \{e160,e122,e123,e163, e124,e165,e121,e167,e168,e169,e170,e171,e172\} \cup \{e168,e169,e171,e121,e122, e123,e124\} = \{e121,e122,e123,e124,e160,e163,e165,e167,e168,e169,e170,e171, e172\}$

f(R_1) U f(R_2) = f({(e160,e161), (e161,e162), (e162,e163), (e163,e166), (e161,e164), (e164,e165), (e165,e166), (e167,e168), (e168,e169), (e169,e170), (e168,e171), (e171,e172), (e160,e167), (e168,e161), (e162,e169), (e170,e163), (e164,e171), (e172,e165), (e160,e162), (e160,e164), (e161,e163), (e161,e169), (e162,e166), (e161,e165), (e161,e171), (e164,e166), (e167,e169), (e167,e171), (e167,e161), (e168,e170), (e169,e163), (e168,e172), (e171,e165), (e160,e168), (e168,e162), (e168,e164), (e162,e170), (e170,e166), (e164,e172), (e172,e166), (e160,e163), (e160,e169), (e160,e165), (e160,e171), (e161,e166), (e161,e170), (e161,e172), (e167,e170), (e167,e172), (e167,e162), (e167,e164), (e168,e163), (e169,e166), (e168,e165), (e171,e166), (e160,e166), (e160,e170), (e160,e172), (e167,e163), (e167,e165), (e168,e166), (e167,e166)}) U f({(e118,e119), (e119,e121), (e118,e120), (e120,e121), (e122,e123), (e122,e124), (e118,e122), (e123,e119), (e124,e120), (e118,e121), (e122,e119), (e122,e120), (e118,e123), (e118,e124), (e123,e121), (e124,e121), (e122,e121)}) = {(e160,e122), (e122,e123), (e123,e163), (e163,e121), (e122,e124), (e124,e165), (e165,e121), (e167,e168), (e168,e169), (e169,e170), (e168,e171), (e171,e172), (e160,e167), (e168,e122), (e123,e169), (e170,e163), (e124,e171), (e172,e165), (e160,e123), (e160,e124), (e122,e163), (e122,e169), (e123,e121), (e122,e165), (e122,e171), (e124,e121), (e167,e169), (e167,e171), (e167,e122), (e168,e170), (e169,e163), (e168,e172), (e171,e165), (e160,e168), (e168,e123), (e168,e124), (e123,e170), (e170,e121), (e124,e172), (e172,e121), (e160,e163), (e160,e169), (e160,e165), (e160,e171), (e122,e121), (e122,e170), (e122,e172), (e167,e170), (e167,e172), (e167,e123), (e167,e124), (e168,e163), (e169,e121), (e168,e165), (e171,e121), (e160,e121), (e160,e170), (e160,e172), (e167,e163), (e167,e165), (e168,e121), (e167,e121)} U {(e168,e169), (e169,e121), (e168,e171), (e171,e121), (e122,e123), (e122,e124), (e168,e122), (e123,e169), (e124,e171), (e168,e121), (e122,e169), (e122,e171), (e168,e123), (e168,e124), (e123,e121), (e124,e121), (e122,e121)} = {(e160,e122), (e122,e123), (e123,e163), (e163,e121), (e122,e124), (e124,e165), (e165,e121), (e167,e168), (e168,e169), (e169,e170), (e168,e171), (e171,e172), (e160,e167), (e168,e122), (e123,e169), (e170,e163), (e124,e171), (e172,e165), (e160,e123), (e160,e124), (e122,e163), (e122,e169), (e123,e121), (e122,e165), (e122,e171), (e124,e121), (e167,e169), (e167,e171), (e167,e122), (e168,e170), (e169,e163), (e168,e172), (e171,e165), (e160,e168), (e168,e123), (e168,e124), (e123,e170), (e170,e121), (e124,e172), (e172,e121), (e160,e163), (e160,e169), (e160,e165), (e160,e171), (e122,e121), (e122,e170), (e122,e172), (e167,e170), (e167,e172), (e167,e123), (e167,e124), (e168,e163), (e169,e121), (e168,e165), (e171,e121), (e160,e121), (e160,e170), (e160,e172), (e167,e163), (e167,e165), (e168,e121), (e167,e121)}

Next, the tool generates the second integration test case by merging the second pair of complement integration test cases. It examines the two test cases to create the shared events set.

se = { (e152,e128), (e153,e129), (e125,e157), (e126,e158), (e155,e127) }

Following that, the tool merges the two test cases

IntTCRtrnMedia = t_1 + t_2

= BookingTM:TestCaseRtrnMedia + MediaTM:TestCaseRtrnMedia

= (g(I_1 *) U g(* I_2 *) , f(* E_1 *) U f(* E_2 *), f(* R_1 *) U f(* R_2 *))*

g(I_1 *) U g(* I_2 *) = g({bookTstCntrl, booking}) U g({medTstCntrl, media}) = {tci, booking} U {tci, media} = {booking, media, tci}*

f(E_1 *) U f(* E_2 *) = f({e151,e152,e153,e154,e155,e156,e157,e158,e159}) U f({e125, e126,e127,e128,e129}) = {e151,e128,e129,e154,e127,e156,e157,e158,e159} U {e157,e158,e127,e128,e129} = {e127,e128,e129,e151,e154,e156,e157,e158,e159}*

f(R_1 *) U f(* R_2 *) = f({(e151,e152), (e152,e153), (e153,e154), (e154,e155), (e156,e157), (e157,e158), (e158,e159), (e151,e156), (e157,e152), (e153,e158), (e159,e154), (e151,e153), (e152,e154), (e152,e158), (e153,e155), (e156,e158), (e156,e152), (e157,e159), (e158,e154), (e151,e157), (e157,e153), (e153,e159), (e159,e155), (e151,e154), (e151,e158), (e152,e155), (e152,e159), (e156,e159), (e156,e153), (e157,e154), (e158,e155), (e151,e155), (e151,e159), (e156,e154), (e157,e155), (e156,e155)}) U f({(e125,e126), (e126,e127), (e128,e129), (e125,e128), (e129,e126), (e125,e127), (e128,e126), (e125,e129), (e129,e127), (e128,e127)}) = {(e151,e128), (e128,e129), (e129,e154), (e154,e127), (e156,e157), (e157,e158), (e158,e159), (e151,e156), (e157,e128), (e129,e158), (e159,e154), (e151,e129), (e128,e154), (e128,e158), (e129,e127), (e156,e158), (e156,e128), (e157,e159), (e158,e154), (e151,e157), (e157,e129), (e129,e159), (e159,e127), (e151,e154), (e151,e158), (e128,e127), (e128,e159), (e156,e159), (e156,e129), (e157,e154), (e158,e127), (e151,e127), (e151,e159), (e156,e154), (e157,e127), (e156,e127)} U {(e157,e158), (e158,e127), (e128,e129), (e157,e128), (e129,e158), (e157,e127), (e128,e158), (e157,e129), (e129,e127), (e128,e127)} = {(e151,e128), (e128,e129), (e129,e154), (e154,e127), (e156,e157), (e157,e158), (e158,e159), (e151,e156), (e157,e128), (e129,e158), (e159,e154), (e151,e129), (e128,e154), (e128,e158), (e129,e127), (e156,e158), (e156,e128), (e157,e159), (e158,e154), (e151,e157), (e157,e129), (e129,e159), (e159,e127), (e151,e154), (e151,e158), (e128,e127), (e128,e159), (e156,e159), (e156,e129), (e157,e154), (e158,e127), (e151,e127), (e151,e159), (e156,e154), (e157,e127), (e156,e127)}*

Following the generation of the test behavior, the tool generates the test structure as follows:

T = { IntTCRsrvMedia, IntTCRtrnMedia }

P = (TCi, {}, {Booking, Media})

IntLibMedBkgTM = (P, T)

The intermediate generated integration test model *IntLibMedBkgTM*, from the second test integration, is shown in Figure 71. Following that, the tool proceeds to the next test integration.

In the third test integration, the tool examines the test cases of *LibrarianTM* against the currently generated test cases of *IntLibMedBkgTM* and the test cases of *BookingTM*. *LibrarianTM's* test cases that are integrated with *IntLibMedBkgTM*'s test cases are not examined against the test cases of *BookingTM*. The identification process does not detect shared test objects between *LibrarianTM* and *IntLibMedBkgTM* nor between *LibrarianTM* and *BookingTM*. Hence, the intermediate generated test model is the final generated integration test model *IntLibMedBkgTM* for the second integration iteration as shown in Figure 71. The test model is exercised on the integrated sub-system, and upon successful test results, we move to the third and last integration iteration.

C.2.3. Third Iteration: *((LibrarianTM+MediaTM)+BookingTM)+MemberTM*

In the third integration iteration, the tool generates the third integration test model, let us call it *IntLibMedBkgMemTM*, to examine the integration of *(((Librarian + Media) + Booking) + Member*). The tool performs four test model integrations.

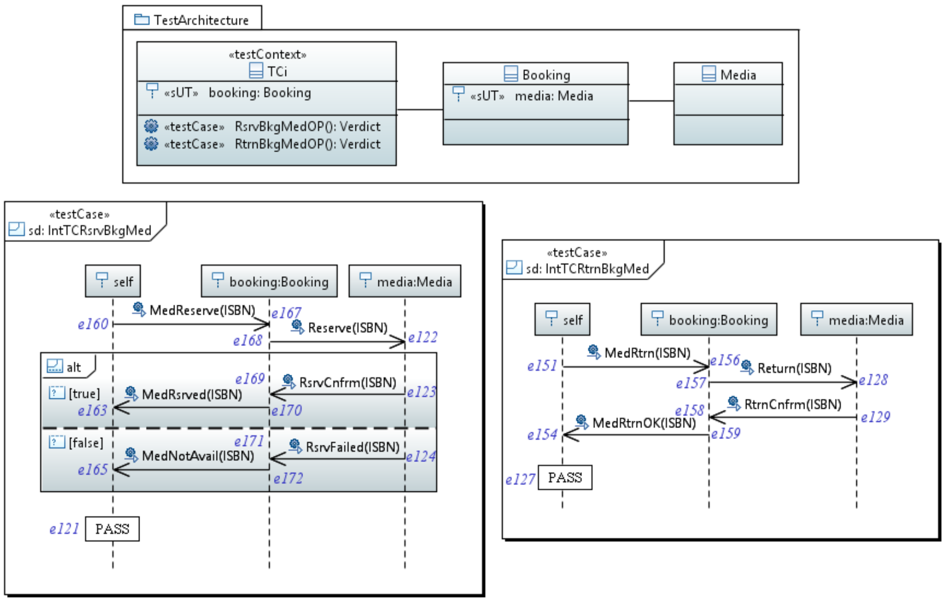


Figure 71. Generated test model (*IntLibMedBkgTM*)

In the first test integration, the tool integrates the previously generated test model *IntLibMedBkgTM* and the component test model *MemberTM*. The tool begins by applying the identification process on the given test models. The identification process detects that the test control *TCi* emulates the CUT *Member* through the following events: *(e151,e91), (e154,e92), (e160,e80), (e163,e81)* and *(e165,e83)*, and the test control *MemTstCntrl* emulates the CUT *Booking* through the following events: *(e73,e167), (e74,e170), (e76,e172), (e86,e156)* and *(e87,e159)*. The selection process selects four test cases as complete integration test cases: *IntLibMedBkgTM:TestCaseRsrvMedia*, *IntLibMedBkgTM:TestCaseRtrnMedia*, *MemberTM:TestCaseRsrvMedia* and *MemberTM:TestCaseRtrnMedia*. It also creates the EDTs and selects two pairs as complement integration test cases: *(IntLibMedBkgTM:TestCaseRsrvMedia, MemberTM:TestCaseRsrvMedia) and (IntLibMedBkgTM:TestCaseRtrnMedia, MemberTM:TestCaseRtrnMedia)*. The tool excludes the complete integration test cases since they are included in the second list. For the next step, the tool generates the first integration test case by merging the first pair of complement integration test cases. It examines the two test cases to create the shared events set.

se = {(e160,e80), (e163,e81), (e165,e83), (e73,e167), (e74,e170), (e76,e172), (e121,e78)}

Following that, the tool merges the two test cases

IntTCRsrvMedBkgMem = $t_1 + t_2$

= *IntLibMedBkgTM:TestCaseRsrvMedia + MemberTM:TestCaseRsrvMedia*

= $(g(I_1) \cup g(I_2), f(E_1) \cup f(E_2), f(R_1) \cup f(R_2))$

$g(I_1) \cup g(I_2)$ = *g({booking, media, tci}) U g({memTstCntrl, member}) = {booking, media, tci} U {tci, member} = {booking, media, member, tci}*

$f(E_1) \cup f(E_2)$ = *f({e121,e122,e123,e124,e160,e163,e165,e167,e168,e169,e170,e171, e172}) U f({e72,e73,e74,e75,e76,e77,e78,e79,e80,e81,e82,e83,e84}) = {e78,e122, e123,e124,e80,e81,e83,e167,e168,e169,e170,e171,e172} U {e72,e167,e170,e75, e172,e77,e78,e79,e80,e81,e82,e83,e84}*

$f(R_1) \cup f(R_2)$ = *f({(e160,e122); (e122,e123); (e123,e163); (e163,e121); (e122,e124); (e124,e165); (e165,e121); (e167,e168); (e168,e169); (e169,e170); (e168,e171); (e171,e172); (e160,e167); (e168,e122); (e123,e169); (e170,e163); (e124,e171); (e172,e165); (e160,e123); (e160,e124); (e122,e163); (e122,e169); (e123,e121); (e122,e165); (e122,e171); (e124,e121); (e167,e169); (e167,e171); (e167,e122); (e168,e170); (e169,e163); (e168,e172); (e171,e165); (e160,e168); (e168,e123); (e168,e124); (e123,e170); (e170,e121); (e124,e172); (e172,e121); (e160,e163); (e160,e169); (e160,e165); (e160,e171); (e122,e121); (e122,e170); (e122,e172); (e167,e170); (e167,e172); (e167,e123); (e167,e124); (e168,e163); (e169,e121); (e168,e165); (e171,e121); (e160,e121); (e160,e170); (e160,e172); (e167,e163); (e167,e165); (e168,e121); (e167,e121)}) U f({(e72,e73); (e73,e74); (e74,e75); (e75,e78); (e73,e76); (e76,e77); (e77,e78); (e79,e80); (e80,e81); (e81,e82); (e80,e83); (e83,e84); (e72,e79); (e80,e73); (e74,e81); (e82,e75); (e76,e83); (e84,e77); (e72,e74); (e72,e76); (e73,e75); (e73,e81); (e74,e78); (e73,e77); (e73,e83); (e76,e78); (e79,e81); (e79,e83); (e79,e73); (e80,e82); (e81,e75); (e80,e84); (e83,e77); (e72,e80); (e80,e74); (e80,e76); (e74,e82); (e82,e78); (e76,e84); (e84,e78); (e72,e75); (e72,e81); (e72,e77); (e72,e83); (e73,e78); (e73,e82); (e73,e84); (e79,e82); (e79,e84); (e79,e74); (e79,e76); (e80,e75); (e81,e78); (e80,e77); (e83,e78); (e72,e78); (e72,e82); (e72,e84); (e79,e75); (e79,e77); (e80,e78); (e79,e78)}) = {(e80,e122); (e122,e123); (e123,e81); (e81,e78); (e122,e124); (e124,e83); (e83,e78); (e167,e168); (e168,e169); (e169,e170); (e168,e171); (e171,e172); (e80,e167); (e168,e122); (e123,e169); (e170,e81); (e124,e171); (e172,e83); (e80,e123); (e80,e124); (e122,e81); (e122,e169); (e123,e78);*

(e122,e83); (e122,e171); (e124,e78); (e167,e169); (e167,e171); (e167,e122); (e168,e170); (e169,e81); (e168,e172); (e171,e83); (e80,e168); (e168,e123); (e168,e124); (e123,e170); (e170,e78); (e124,e172); (e172,e78); (e80,e81); (e80,e169); (e80,e83); (e80,e171); (e122,e78); (e122,e170); (e122,e172); (e167,e170); (e167,e172); (e167,e123); (e167,e124); (e168,e81); (e169,e78); (e168,e83); (e171,e78); (e80,e78); (e80,e170); (e80,e172); (e167,e81); (e167,e83); (e168,e78); (e167,e78)} U {(e72,e167); (e167,e170); (e170,e75); (e75,e78); (e167,e172); (e172,e77); (e77,e78); (e79,e80); (e80,e81); (e81,e82); (e80,e83); (e83,e84); (e72,e79); (e80,e167); (e170,e81); (e82,e75); (e172,e83); (e84,e77); (e72,e170); (e72,e172); (e167,e75); (e167,e81); (e170,e78); (e167,e77); (e167,e83); (e172,e78); (e79,e81); (e79,e83); (e79,e167); (e80,e82); (e81,e75); (e80,e84); (e83,e77); (e72,e80); (e80,e170); (e80,e172); (e170,e82); (e82,e78); (e172,e84); (e84,e78); (e72,e75); (e72,e81); (e72,e77); (e72,e83); (e167,e78); (e167,e82); (e167,e84); (e79,e82); (e79,e84); (e79,e170); (e79,e172); (e80,e75); (e81,e78); (e80,e77); (e83,e78); (e72,e78); (e72,e82); (e72,e84); (e79,e75); (e79,e77); (e80,e78); (e79,e78)} = {(e80,e122); (e122,e123); (e123,e81); (e81,e78); (e122,e124); (e124,e83); (e83,e78); (e167,e168); (e168,e169); (e169,e170); (e168,e171); (e171,e172); (e80,e167); (e168,e122); (e123,e169); (e170,e81); (e124,e171); (e172,e83); (e80,e123); (e80,e124); (e122,e81); (e122,e169); (e123,e78); (e122,e83); (e122,e171); (e124,e78); (e167,e169); (e167,e171); (e167,e122); (e168,e170); (e169,e81); (e168,e172); (e171,e83); (e80,e168); (e168,e123); (e168,e124); (e123,e170); (e170,e78); (e124,e172); (e172,e78); (e80,e81); (e80,e169); (e80,e83); (e80,e171); (e122,e78); (e122,e170); (e122,e172); (e167,e170); (e167,e172); (e167,e123); (e167,e124); (e168,e81); (e169,e78); (e168,e83); (e171,e78); (e80,e78); (e80,e170); (e80,e172); (e167,e81); (e167,e83); (e168,e78); (e167,e78); (e72,e167); (e170,e75); (e75,e78); (e172,e77); (e77,e78); (e79,e80); (e81,e82); (e83,e84); (e72,e79); (e82,e75); (e84,e77); (e72,e170); (e72,e172); (e167,e75); (e167,e77); (e79,e81); (e79,e83); (e79,e167); (e80,e82); (e81,e75); (e80,e84); (e83,e77); (e72,e80); (e170,e82); (e82,e78); (e172,e84); (e84,e78); (e72,e75); (e72,e81); (e72,e77); (e72,e83); (e167,e82); (e167,e84); (e79,e82); (e79,e84); (e79,e170); (e79,e172); (e80,e75); (e80,e77); (e72,e78); (e72,e82); (e72,e84); (e79,e75); (e79,e77); (e79,e78); (e123,e82); (e123,e75); (e124,e84); (e124,e77); (e169,e75); (e169,e82); (e171,e77); (e171,e84); (e122,e82); (e122,e75); (e122,e84); (e122,e77); (e168,e75); (e168,e82); (e168,e77); (e168,e84); (e72,e168); (e72,e169); (e72,e171); (e72,e122); (e72,e123); (e72,e124); (e79,e122); (e79,e123); (e79,e124); (e79,e168); (e79,e169); (e79,e171)}

Next, the tool generates the second integration test case by merging the second pair of complement integration test cases. It examines the two test cases to create the shared events set.

se = { (e151,e91), (e154,e92), (e86,e156), (e87,e159), (e127,e89) }

Following that, the tool merges the two test cases

IntTCRtrnMedBkgMem = $t_1 + t_2$

= *IntLibMedBkgTM:TestCaseRtrnMedia + MemberTM:TestCaseRtrnMedia*

= $(g(I_1) \cup g(I_2), f(E_1) \cup f(E_2), f(R_1) \cup f(R_2))$

g(I_1 *) U g(* I_2 *) = g(= {booking, media, tci}) U g({memTstCntrl, member}) = {booking, media, tci} U {tci, member} = {booking, media, member, tci}*

f(E_1 *) U f(* E_2 *) = f({e127,e128,e129,e151,e154,e156,e157,e158,e159}) U f({e85,e86, e87,e88,e89,e90,e91,e92,e93}) = {e89,e128,e129,e91,e92,e156,e157,e158,e159} U {e85,e156,e159,e88,e89,e90,e91,e92,e93} = {e128,e129,e156,e157,e158,e159, e85,e88,e89,e90,e91,e92,e93}*

f(R_1 *) U f(* R_2 *) = f({(e151,e128), (e128,e129), (e129,e154), (e154,e127), (e156,e157), (e157,e158), (e158,e159), (e151,e156), (e157,e128), (e129,e158), (e159,e154), (e151,e129), (e128,e154), (e128,e158), (e129,e127), (e156,e158), (e156,e128), (e157,e159), (e158,e154), (e151,e157), (e157,e129), (e129,e159), (e159,e127), (e151,e154), (e151,e158), (e128,e127), (e128,e159), (e156,e159), (e156,e129), (e157,e154), (e158,e127), (e151,e127), (e151,e159), (e156,e154), (e157,e127), (e156,e127)}) U f({(e85,e86), (e86,e87), (e87,e88), (e88,e89), (e90,e91), (e91,e92), (e92,e93), (e85,e90), (e91,e86), (e87,e92), (e93,e88), (e85,e87), (e86,e88), (e86,e92), (e87,e89), (e90,e92), (e90,e86), (e91,e93), (e92,e88), (e85,e91), (e91,e87), (e87,e93), (e93,e89), (e85,e88), (e85,e92), (e86,e89), (e86,e93), (e90,e93), (e90,e87), (e91,e88), (e92,e89), (e85,e89), (e85,e93), (e90,e88), (e91,e89), (e90,e89)}) = {(e91,e128), (e128,e129), (e129,e92), (e92,e89), (e156,e157), (e157,e158), (e158,e159), (e91,e156), (e157,e128), (e129,e158), (e159,e92), (e91,e129), (e128,e92), (e128,e158), (e129,e89), (e156,e158), (e156,e128), (e157,e159), (e158,e92), (e91,e157), (e157,e129), (e129,e159), (e159,e89), (e91,e92), (e91,e158), (e128,e89), (e128,e159), (e156,e159), (e156,e129), (e157,e92), (e158,e89), (e91,e89), (e91,e159), (e156,e92), (e157,e89), (e156,e89)} U {(e85,e156), (e156,e159), (e159,e88), (e88,e89), (e90,e91), (e91,e92), (e92,e93), (e85,e90), (e91,e156), (e159,e92), (e93,e88), (e85,e159), (e156,e88), (e156,e92), (e159,e89), (e90,e92), (e90,e156), (e91,e93), (e92,e88), (e85,e91), (e91,e159), (e159,e93), (e93,e89), (e85,e88), (e85,e92), (e156,e89), (e156,e93), (e90,e93), (e90,e159), (e91,e88), (e92,e89), (e85,e89), (e85,e93), (e90,e88), (e91,e89), (e90,e89)} = {(e91,e128), (e128,e129), (e129,e92), (e92,e89), (e156,e157), (e157,e158), (e158,e159), (e91,e156), (e157,e128), (e129,e158), (e159,e92), (e91,e129), (e128,e92), (e128,e158), (e129,e89), (e156,e158), (e156,e128), (e157,e159), (e158,e92), (e91,e157), (e157,e129), (e129,e159), (e159,e89), (e91,e92), (e91,e158), (e128,e89), (e128,e159), (e156,e159), (e156,e129), (e157,e92), (e158,e89), (e91,e89), (e91,e159), (e156,e92), (e157,e89), (e156,e89), (e85,e156), (e159,e88), (e88,e89), (e90,e91), (e92,e93), (e85,e90), (e93,e88), (e85,e159), (e156,e88), (e90,e92), (e90,e156), (e91,e93), (e92,e88), (e85,e91), (e159,e93), (e93,e89), (e85,e88), (e85,e92), (e156,e93), (e90,e93), (e90,e159), (e91,e88), (e85,e89), (e85,e93), (e90,e88), (e90,e89), (e129,e93), (e129,e88), (e158,e88), (e158,e93), (e128,e93), (e128,e88), (e157,e88), (e157,e93), (e85,e157), (e85,e158), (e85,e128), (e85,e129), (e90,e128), (e90,e129), (e90,e157), (e90,e158)}*

After generating the test behavior, the tool generates the test structure as follows:

T = { IntTCRsrvMedBkgMem, IntTCRtrnMedBkgMem }

P = (TCi, {}, {Booking, Media, Member})

IntLibMedBkgMemTM = (P, T)

The intermediate generated integration test model *IntLibMedBkgMemTM* is shown in Figure 72. After that, we examine the carried-on component test models: *LibrarianTM*, *BookingTM* and *MediaTM*, against the intermediate generated test model *IntLibMedBkgMemTM* and the currently integrated component test model *MemberTM* to generate additional test cases. Hence, we move to the second test integration. In the second test integration, the tool integrates the component test model *LibrarianTM* to *IntLibMedBkgMemTM* and *MemberTM*. The identification process does not detect shared test objects between *LibrarianTM* and *IntLibMedBkgMemTM*, but it detects shared test objects between *LibrarianTM* and *MemberTM*. The tool detects that the test control *LibTstCntrl* emulates the CUT *Member* through the following events: (*e15*,*e55*), (*e16*,*e56*), (*e18*,*e57*), (*e37*,*e61*) and (*e38*,*e62*). It also detects that the test control *MemTstCntrl* emulates the CUT *Librarian* through the following events: (*e51*,*e22*), (*e52*,*e23*), (*e53*,*e25*), (*e58*,*e42*) and (*e59*,*e43*). In the selection process, the tool selects four test cases as complete integration test cases: *LibrarianTM:TestCaseAddMember*, *LibrarianTM:TestCaseBrowseMembers*, *MemberTM:TestCaseAddMember* and *MemberTM:TestCaseBrowseMembers*. Next, the tool builds EDTs for the test cases to detect complement integration test cases. The tool selects two pairs of complement integration test cases: *(LibrarianTM:TestCaseAddMember*, *MemberTM:TestCaseAddMember)* and *(LibrarianTM:TestCaseBrowseMembers*, *MemberTM:TestCaseBrowseMembers)*. The tool excludes the complete integration test cases since they are involved in the complement integration test cases. The next step is to generate the third integration test cases by merging the first pair; the tool creates the integration test control *TCi* and creates the shared events set, Definition 6, using the event matching expression, Definition 5.

se = { (e15,e55), (e16,e56), (e18,e57), (e51,e22), (e52,e23), (e53,e25), (e54, e20) }

Then, the tool generates the first integration test case by applying Definition 7:

IntTCAddMem = t1 + t2

= LibrarianTM:TestCaseAddMember + MemberTM:TestCaseAddMember

= (g(I1) U g(I2) , f(E1) U f(E2), f(R1) U f(R2))

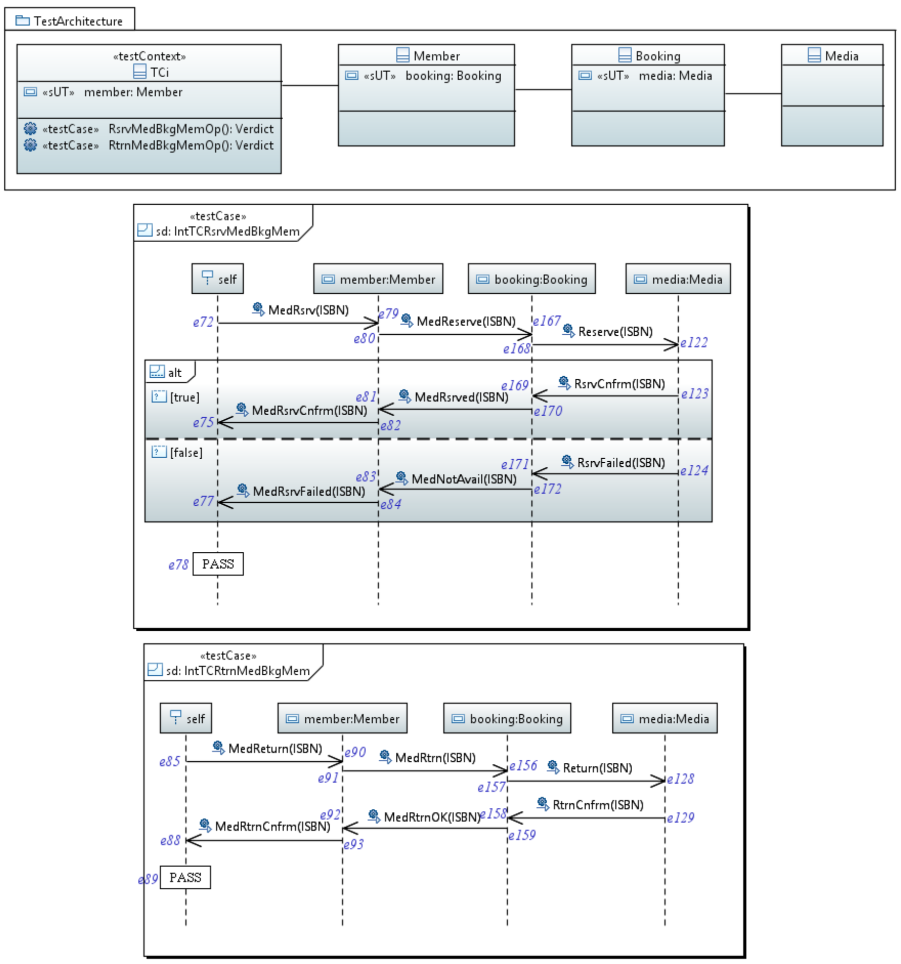


Figure 72. Intermediate generated test model (*IntLibMedBkgMemTM*)

g(I1) U g(I2) = g({libTstCntrl, librarian}) U g({memTstCntrl, member}) = {tci, librarian} U {tci, member} = {librarian, member, tci}

f(E1) U f(E2) = f({e14,e15,e16,e17,e18,e19,e20,e21,e22,e23,e24,e25,e26}) U f({e51,e52,e53,e54,e55,e56,e57}) = {e14,e55,e56,e17,e57,e19,e20,e21,e22,e23, e24,e25,e26} U {e22,e23,e25,e20,e55,e56,e57} = {e14,e17,e19,e20,e21,e22,e23, e24,e25,e26,e55,e56,e57}

f(R1) U f(R2) = f({(e14,e15), (e15,e16), (e16,e17), (e17,e20), (e15,e18), (e18,e19), (e19,e20), (e21,e22), (e22,e23), (e23,e24), (e22,e25), (e25,e26), (e14,e21), (e22,e15), (e16,e23), (e24,e17), (e18,e25), (e26,e19), (e14,e16), (e14,e18), (e15,e17), (e15,e23), (e16,e20), (e15,e19), (e15,e25), (e18,e20), (e21,e23), (e21,e25), (e21,e15), (e22,e24), (e23,e17), (e22,e26), (e25,e19), (e14,e22), (e22,e16), (e22,e18), (e16,e24), (e24,e20), (e18,e26), (e26,e20), (e14,e17), (e14,e23), (e14,e19), (e14,e25), (e15,e20), (e15,e24), (e15,e26), (e21,e24), (e21,e26), (e21,e16), (e21,e18), (e22,e17), (e23,e20), (e22,e19), (e25,e20), (e14,e20), (e14,e24), (e14,e26), (e21,e17), (e21,e19), (e22,e20), (e21,e20)}) U f({(e51,e52), (e51,e53), (e52,e54), (e53,e54), (e55,e56), (e55,e57), (e51,e55), (e56,e52), (e57,e53), (e51,e54), (e55,e52), (e55,e53), (e51,e56), (e51,e57), (e56,e54), (e57,e54), (e55,e54)}) = {(e14,e55), (e55,e56), (e56,e17), (e17,e20), (e55,e57), (e57,e19), (e19,e20), (e21,e22), (e22,e23), (e23,e24), *(e22,e25), (e25,e26), (e14,e21), (e22,e55), (e56,e23), (e24,e17), (e57,e25), (e26,e19), (e14,e56), (e14,e57), (e55,e17), (e55,e23), (e56,e20), (e55,e19), (e55,e25), (e57,e20), (e21,e23), (e21,e25), (e21,e55), (e22,e24), (e23,e17), (e22,e26), (e25,e19), (e14,e22), (e22,e56), (e22,e57), (e56,e24), (e24,e20), (e57,e26), (e26,e20), (e14,e17), (e14,e23), (e14,e19), (e14,e25), (e55,e20), (e55,e24), (e55,e26), (e21,e24), (e21,e26), (e21,e56), (e21,e57), (e22,e17), (e23,e20), (e22,e19), (e25,e20), (e14,e20), (e14,e24), (e14,e26), (e21,e17), (e21,e19), (e22,e20), (e21,e20)} U {(e22,e23), (e22,e25), (e23,e20), (e25,e20), (e55,e56), (e55,e57), (e22,e55), (e56,e23), (e57,e25), (e22,e20), (e55,e23), (e55,e25), (e22,e56), (e22,e57), (e56,e20), (e57,e20), (e55,e20)} = { (e14, e55), (e55, e56), (e56, e17), (e17, e20), (e55, e57), (e57, e19), (e19, e20), (e21, e22), (e22, e23), (e23, e24), (e22, e25), (e25, e26), (e14, e21), (e22, e55), (e56, e23), (e24, e17), (e57, e25), (e26, e19), (e14, e56), (e14, e57), (e55, e17), (e55, e23), (e56, e20), (e55, e19), (e55, e25), (e57, e20), (e21, e23), (e21, e25), (e21, e55), (e22, e24), (e23, e17), (e22, e26), (e25, e19), (e14, e22), (e22, e56), (e22, e57), (e56, e24), (e24, e20), (e57, e26), (e26, e20), (e14, e17), (e14, e23), (e14, e19), (e14, e25), (e55, e20), (e55, e24), (e55, e26), (e21, e24), (e21, e26), (e21, e56), (e21, e57), (e22, e17), (e23, e20), (e22, e19), (e25, e20), (e14, e20), (e14, e24), (e14, e26), (e21, e17), (e21, e19), (e22, e20), (e21, e20) }*

Next, the tool generates the fourth integration test case by merging the second pair. The tool starts by creating the shared events set.

se = { (e37,e61), (e38,e62), (e58,e42), (e59,e43), (e60,e40) }

Then, the tool generates the second integration test case by applying Definition 7:

IntTCBrwMem = $t_1 + t_2$

= *LibrarianTM:TestCaseBrowseMembers + MemberTM:TestCaseBrowseMembers*

= ($g(I_1)$ U $g(I_2)$, $f(E_1)$ U $f(E_2)$, $f(R_1)$ U $f(R_2)$))

$g(I_1)$ U $g(I_2)$ = *g({libTstCntrl, librarian}) U g({memTstCntrl, member}) = { tci, librarian } U { tci, member } = { librarian, member, tci }*

$f(E_1)$ U $f(E_2)$ = *f({e36,e37,e38,e39,e40,e41,e42,e43,e44}) U f({e58,e59,e60,e61, e62}) = {e36,e61,e62,e39,e40,e41,e42,e43,e44} U {e42,e43,e40,e61,e62} = {e36, e39,e40,e41,e42,e43,e44,e61,e62}*

$f(R_1)$ U $f(R_2)$ = *f({ (e36,e37), (e37,e38), (e38,e39), (e39,e40), (e41,e42), (e42,e43), (e43,e44), (e36,e41), (e42,e37), (e38,e43), (e44,e39), (e36,e38), (e37,e39), (e37,e43), (e38,e40), (e41,e43), (e41,e37), (e42,e44), (e43,e39), (e36,e42), (e42,e38), (e38,e44), (e44,e40), (e36,e39), (e36,e43), (e37,e40), (e37,e44), (e41,e44), (e41,e38), (e42,e39), (e43,e40), (e36,e40), (e36,e44), (e41,e39), (e42,e40), (e41,e40) }) U f({ (e58,e59), (e59,e60), (e61,e62), (e58,e61), (e62,e59), (e58,e60), (e61,e59), (e58,e62), (e62,e60), (e61,e60) }) = {(e36,e61), (e61,e62), (e62,e39), (e39,e40), (e41,e42), (e42,e43), (e43,e44), (e36,e41), (e42,e61), (e62,e43), (e44,e39), (e36,e62), (e61,e39), (e61,e43), (e62,e40), (e41,e43), (e41,e61), (e42,e44), (e43,e39), (e36,e42), (e42,e62), (e62,e44), (e44,e40), (e36,e39), (e36,e43), (e61,e40), (e61,e44), (e41,e44), (e41,e62), (e42,e39), (e43,e40), (e36,e40), (e36,e44), (e41,e39), (e42,e40), (e41,e40) } U { (e42,e43), (e43,e40), (e61,e62), (e42,e61), (e62,e43), (e42,e40),* *(e61,e43), (e42,e62), (e62,e40), (e61,e40) } = { (e36, e61), (e61, e62), (e62, e39), (e39, e40), (e41, e42), (e42, e43), (e43, e44), (e36, e41), (e42, e61), (e62, e43), (e44, e39), (e36, e62), (e61, e39), (e61, e43), (e62, e40), (e41, e43), (e41, e61), (e42, e44), (e43, e39), (e36, e42), (e42, e62), (e62, e44), (e44, e40), (e36, e39), (e36, e43), (e61, e40), (e61, e44), (e41, e44), (e41, e62), (e42, e39), (e43, e40), (e36, e40), (e36, e44), (e41, e39), (e42, e40), (e41, e40) }*

After generating the test behavior, the tool updates the test structure as follows:

T = {IntTCRsrvMedBkgMem, IntTCRtrnMedBkgMem, IntTCAddMem, IntTCBrwMem}

P = (TCi, {}, {Booking, Media, Member, Librarian})

IntLibMedBkgMemTM = (P, T)

The intermediate generated integration test model *IntLibMedBkgMemTM is* shown in Figure 73. Next, we move to the next test integration. In the third test integration, the tool integrates the component test model *MediaTM* to *IntLibMedBkgMemTM* and *MemberTM*. Test cases of *MediaTM*, which are integrated with *IntLibMedBkgMemTM*, are not used to integrate with *MemberTM* test cases. The identification process detects shared test objects between *MediaTM* and *IntLibMedBkgMemTM*. The CUT *Media* is defined in both test packages. The test control *medTstCntrl* emulates the CUT *Booking*. However, the tool does not generate/update any test cases since the generated test cases include the test cases of the *MediaTM*. Next, the identification process detects shared test objects between *MediaTM* and *MemberTM*. This integration have been discussed in Section C.1.2 and the generated test case *IntTCBrwMemMed* is shown in Figure 66. The tool updates the test architecture. Consequently, the intermediate generated test model would be shown in Figure 74. Next, we move to the next test integration.

T = { IntTCRsrvMedBkgMem, IntTCRtrnMedBkgMem, IntTCAddMem, IntTCBrwMem, IntTCBrwMemMed }

P = (TCi, {}, {Booking, Media, Member, Librarian})

IntLibMedBkgMemTM = (P, T)

In the fourth test integration, the tool integrates the component test model *BookingTM* to *IntLibMedBkgMemTM* and *MemberTM*. Test cases of *BookingTM* that are integrated with *IntLibMedBkgMemTM* are not used in the integration with *MemberTM*.

The tool detects shared test objects between *BookingTM* to *IntLibMedBkgMemTM*. The CUT *Booking* is specified in both test models and the test control *bookTstControl* emulates the CUT *Member* through the following events: *(e160,e80), (e163,e81), (e165,e83), (e151,e91) and (e154,e92)*, and emulates the CUT *Media* through the following events: *(e161,e122), (e162,e123), (e164,e124), (e152,e128) and (e153,e129)*. However, the tool does not generate/upgrade any test cases since the test cases of *BookingTM* are already included in the generated test cases. Therefore, the generated integration test model is not modified. It remains the same as shown in Figure 74.

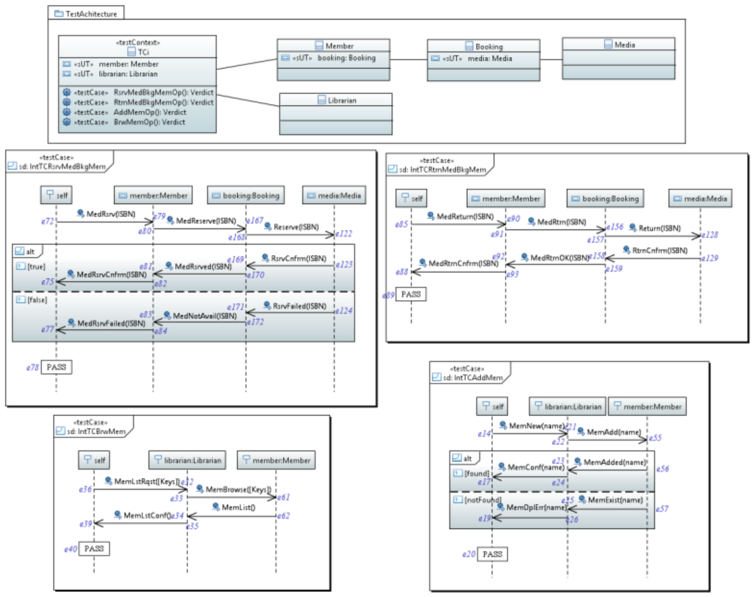


Figure 73. Intermediate generated test model (*IntLibMedBkgMemTM*)

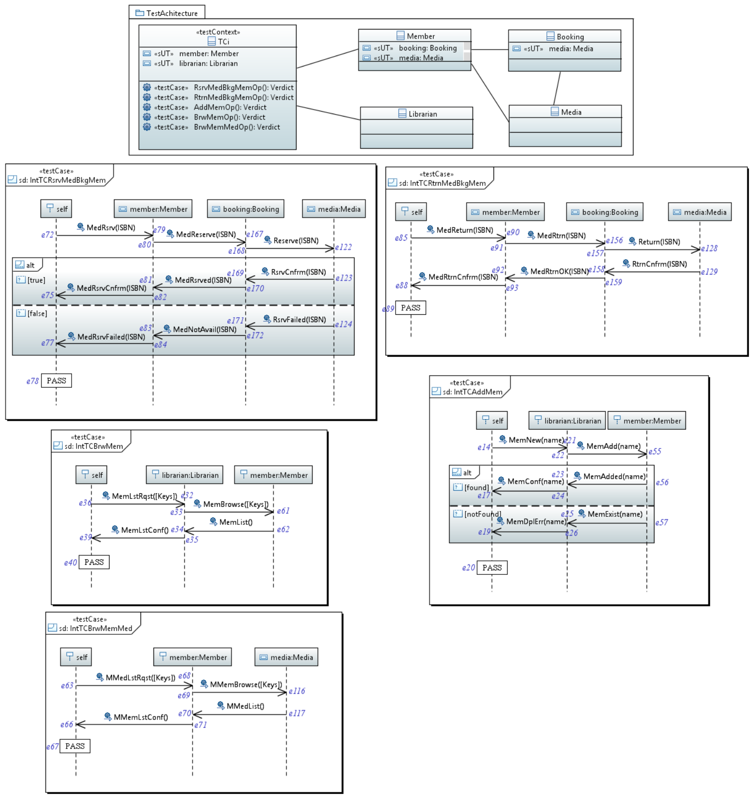


Figure 74. Generated test model (*IntLibMedBkgMemTM*)