TOWARDS CHANGE PROPAGATING TEST MODELS IN AUTONOMIC AND ADAPTIVE SYSTEMS

A Dissertation
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

Mohammed Abd Alwahab Akour

In Partial Fulfillment of the Requirements
for the Degree of
DOCTOR OF PHILOSOPHY

Major Program:
Software Engineering

October 2012

Fargo, North Dakota

North Dakota State University
Graduate School

Title

TOWARDS CHANGE PROPAGATING TEST MODELS IN AUTONOMIC AND ADAPTIVE SYSTEMS

By

Mohammed Abd Alwahab Akour

The Supervisory Committee certifies that this ***disquisition*** complies with North Dakota State University's regulations and meets the accepted standards for the degree of

DOCTOR OF PHILOSOPHY

SUPERVISORY COMMITTEE:

Dr. Kendall Nygard
Chair

Dr. Jun kong

Dr. Tariq King

Dr. Limin Zhang

Approved:

10/30/2012
Date

Dr. Kenneth Magel
Department Chair

ABSTRACT

The major motivation for self-adaptive computing systems is the self-adjustment of the software according to a changing environment. Adaptive computing systems can add, remove, and replace their own components in response to changes in the system itself and in the operating environment of a software system. Although these systems may provide a certain degree of confidence against new environments, their structural and behavioral changes should be validated after adaptation occurs at runtime.

Testing dynamically adaptive systems is extremely challenging because both the structure and behavior of the system may change during its execution. After self adaptation occurs in autonomic software, new components may be integrated to the software system. When new components are incorporated, testing them becomes vital phase for ensuring that they will interact and behave as expected. When self adaptation is about removing existing components, a predefined test set may no longer be applicable due to changes in the program structure. Investigating techniques for dynamically updating regression tests after adaptation is therefore necessary to ensure such approaches can be applied in practice.

We propose a model-driven approach that is based on change propagation for synchronizing a runtime test model for a software system with the model of its component structure after dynamic adaptation. A workflow and meta-model to support the approach was provided, referred to as Test Information Propagation (TIP). To demonstrate TIP, a prototype was developed that simulates a reductive and additive change to an autonomic, service-oriented healthcare application.

To demonstrate the generalization of our TIP approach to be instantiated into the domain of up-to-date runtime testing for self-adaptive software systems, the TIP approach was applied to the self-adaptive JPacman 3.0 system.

To measure the accuracy of the TIP engine, we consider and compare the work of a developer who manually identifyied changes that should be performed to update the test model after self-adaptation occurs in self-adaptive systems in our study. The experiments show how TIP is highly accurate for reductive change propagation across self-adaptive systems. Promising results have been achieved in simulating the additive changes as well.

ACKNOWLEDGMENTS

It is a pleasure to express my gratitude to the many people who were abundantly helpful and offered invaluable assistance and made this dissertation possible.

It is difficult to exaggerate my gratitude to my Ph.D. advisor, Prof. Dr. Kendall Nygard. With his encouragement, his suggestions, and his time. I hope that one day I could become as good an advisor to my students as Dr. Nygard has been to me and his students. Special thanks to my supervisory committee, Assoc. Prof. Dr. Jun Kong and Dr. Limin Zhang for their support, guidance and helpful suggestions. Without their comments and assistance this dissertation would not have been successful.

I would like to express my sincere gratitude to my Ph.D. Co-supervisor, the best friend, and old brother Dr. Tariq King for his enthusiasm, guidance, inspiration, patience, and his great efforts to explain things clearly and simply. Through his pain and being away for treatment after my preliminary exam, he provided encouragement, sound advice, lots of good ideas and advise necessary for me to proceed through the doctoral program and complete my dissertation. For everything you have done for me, Dr. King, I thank you.

The deepest gratitude are due to my beloved parents, Abd Alwahab and Sameera, for their praying, endless love and supporting throughout my whole life; this accomplishment is simply impossible without them. I owe them everything and wish I could show them how much I love and appreciate them. I would like to thank my wife, Saja Wardat, for her encouragement, tolerance, and patientce that allowed me to continue and finish graduate trip. She already has my heart so I will just give her a heartfelt thanks.

I thank my friends, for their faith in me and supporting me in several way to be as ambitious as I wanted, for helping me defeat hard times, and for all the emotional support, entertainment, and caring they provided. Special thanks to my friend Mathew Warner.

Lastly, I wish to thank my entire family for being a constant source of encouragement during my graduate study. My brothers Ahmed and Osama, my sisters Ahlam, Ateka, Hanan, Rabaa, Sumaia, Noor, Slam and Sajeda.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1. INTRODUCTION

The systems, technologies, and enterprises of today have become highly complex and heterogeneous. Traditional approaches to managing this complexity have focused on manual configuration, integration, and maintenance. However, due to increasingly rapid changes in the context, goals, and requirements of software systems, there is a demand to perform such tasks automatically during runtime [91].

Major engineering performers have recognized the need to shift the onus of support tasks such as configuration, maintenance and fault management from people to technology [35, 51, 77]. Microsoft developed the Dynamic Systems Initiative [77]; Hewlett-Packard proposed the Adaptive Infrastructure [51]; and in 2001, IBM introduced the Autonomic Computing (AC) paradigm [62]. A central theme within each initiative is the concept of self-managing software (i.e., software system able to control their own support tasks).

Autonomic and adaptive computing seeks to meet this demand by specifying systems that can self-configure, self-optimize, self-heal, and self-protect [62]. However, the development of such systems has been shown to be significantly more challenging than traditional software systems, that are relatively more static and predictable [5]. These types of systems can add, remove, or replace their own components at runtime, referred to in this dissertation as additive, reductive, and mutative changes.

In comparison with conventional software systems, one of the characteristics of a self-adaptive software system is that not all activities of its life cycle occurs during the development time, much is left to be conducted during runtime (e.g. reconfiguration). During runtime a self-adaptive software system observes and analyses itself and its environment for any changes, and if an adaptation becomes necessary, a set of actions will be generated and then taken to adapt itself in an effective manner.

As dynamic adaptation occurs, the dependency relationships between the component targeted for adaptation and other system components should be updated, to be consistent with the new changes. As a result of a new adaptation, new errors can be introduced after changes have been made to the adaptive system. In order to determine whether a new change doesn't affect other parts of the software system, regression testing becomes necessary. Regression testing is a technique which determines whether modifications to software have introduced new errors into previously tested code [9]. This may involve re-running the entire test suite (retest-all) or selecting a subset of the initial test suite for execution (i.e., selective regression testing) [46]. The developers write and maintain test cases continually in order to reflect changes in the source code to keep an effective regression suite. Techniques for regression test selection include dataflow, random, safe and test minimization [46].

In software testing several axiomatic theories and practices reveal why runtime testing should be an integral part of dynamic adaptive software systems. For example, even when two components are logically equivalent (anti-extensionality), or have the same structural shape (general multiple change), a test set for one is not necessarily adequate for the other [102]. Furthermore, a test set that is adequate for validating a component in isolation, may not be adequate for testing the component's behavior as part of an enclosing component (anti-composition) [102]. This is because errors can arise due to interactions between components.

In practice, anti-extensionality and general multiple change axioms are the reasoning behind testing software product lines (i.e., that share signiflcant functional and structural commonalities [21]), while anti-composition represents the traditional need for integration testing. For autonomic software, system configuration that has never been tested could be viewed as a member in the same software product line requiring validation.

Furthermore, even if the newly introduced or adapted components have been tested separately, the interaction between components could introduce new errors. Hence runtime testing should be incorporated into autonomic software to evade harmful and exorbitant system failures.

In spite of the increased need for runtime testing in autonomic software, there is little research in the area of testing adaptive system dynamically. Most of the AC research is concentrating on how to integrate the autonomic features of self-configuration, self-optimization, self-protection, and self-healing into domains such as networking, grid computing, and database management, etc.. The pioneers of AC state that one of the major challenges associated with building and maintaining autonomic software is validating its correctness [61, 62]. The proposed approach in this dissertation for a self-adaptive system provides good support for runtime testing.

Testing dynamically adaptive systems is extremely challenging because both the structure and behavior of the system may change during its execution. Existing test cases may no longer be applicable due to changes in program structure, thereby requiring dynamic generation of new test cases, updating or removing existing ones automatically. Certainly, maintaining dependencies between test cases and classes under test can help to maintain the consistency between the autonomic software and its runtime test models. Indeed, refactoring of the code should be followed by refactoring of the tests [26]. Refactoring of many of these dependent tests could be automated or at least made easier, if the exact relationships between the unit tests and the corresponding tested classes would be known [87].

In the xUnit testing environment [47, 76] there is no pre-defined structure, nor does there exist explicit links between code and test cases. Some guidelines and naming conventions that describe the testing environment have been proposed to facilitate the identification of tested classes [76, 36].

Strategies for establishing traceability links between production classes and xUnit test cases in object-oriented systems include test case naming convention, explicit fixture declaration, static test call graphs, last call before assert, lexical analysis, and co-evolution logs [90].

Multi-shot transformation approaches such as change propagation could be effective for synchronizing the component model of an adaptive software system with its runtime test model after dynamic adaptation. Change propagation becomes a central aspect for tackling the problem of conveying structural changes in autonomic software to runtime test models. These changes will be propagated to ensure that the runtime test model for the system is made consistent with its new structure after dynamic adaptation.

In our previous work [2], we proposed a model-driven approach that is based on change propagation for synchronizing a runtime test model for a software system with the model of its component structure after dynamic adaptation. Traceability relationships were handled through the naming conventions strategy. The approach is referred to as Test Information Propagation (TIP). To demonstrate TIP, a prototype was developed that simulates a reductive change to an autonomic, service-oriented healthcare application.

To demonstrate the generalization of our TIP approach into the domain of runtime testing for self-adaptive software systems, the experiment was performed on other self-adaptive sysetm (i.e., a different application domain). During our search for a sample self-adaptive system we forced the three following obstacles:

- We need access to the source code, so as to verify the presence of a considerable test suite and next to apply the naming convention strategy to handle the traceability relationship, profile and trace the junit tests execution, and finally trace the internal component interaction to build the component model that handles the component dependency relationship.

- The implementations we made are currently targeted towards self-adaptive systems developed in Java and the dynamic language groovy.

- The approach we used to build self-adaptive software system is based on the Spring framework which provides a core application container that allows us to specify components (called beans) using XML configurations. We used groovy to write these Beans. Using the groovy dynamic language allows spring to act as an adaptive framework since the container can be set to monitor beans for code changes, and to dynamically use any new source code implementations.

To measure the performance of the propagation engine two main concepts of information retrieval were used: recall and precision. We compare the retrieved propagated changes with a manual developer oracle, (i.e., we consider and compare the work of a developer who manually identified changes that should be performed to update the test model after self-adaptation occurs). This duty of identifying changes is part of a short questionnaire we generated and gave to the developer. The experiments show how TIP is maximally accurate for Reductive change propagation across self-adaptive systems.

The main contributions of this dissertation are that it:

- Provides a generalization of the proposed approach in [2] to be instantiated into the domain of up-to-date runtime testing for self-adaptive software systems;

- Extends the TIP prototype in [2] by including the additive change propagation capability;

- Elaborates on the transformative action update to provide more detailed information and to focus the updating process;and

- Measures the accuracy of the TIP engine in propagating both dynamic reductive and additive changes to the corresponding runtime test model after self adaptation takes place.

The domain of this research dissertation is limited to the investigation of techniques for automatically updating runtime test models after self-adaptation occurs, for making the runtime test model for the system consistent with its new structure after dynamic adaptation. Therefore, the automatic synchronization of a runtime test model for a software system with the model of its component structure after dynamic adaptation is the primary focus of the work. Measurements for change propagation is provided as a means to demonstrate the performance of the proposed approach in updating the runtime test model, and the results are evaluated against the recall and precision critieria. Dynamically test case generation to improve the adequacy and effectiveness of runtime testing is outside the scope of this dissertation.

The rest of this dissertation is organized as follows: Chapter Two provides the background information for autonomic systems, software testing and model-driven engineering. The systematic literature review was performed to survey research on self-testing in autonomic systems and model driven change propagation that this research builds upon. Chapter Three provides some motivations for the research in the area of dynamically testing autonomic computing and describes the problems to be investigated. Chapter Four outlines the approach to achieve the goals of the research, describes the detailed system design aspect of the prototype, adaptation scenarios along with the simulation environment and models instantiation. In Chapter Five we describe the experimental, manual developer evaluation oracle and evaluation criteria. After interpreting the experimental results and discussing the threat to validity in Chapter Six, Chapter Seven concludes the dissertation and outlines future works.

CHAPTER 2. LITERATURE REVIEW

This chapter provides a survey and discussion of the literature in autonomic computing, software testing, mode driven engineering, autonomic self-testing, change propagation and model synchronization.

2.1. Background

This section describes background material on autonomic and adaptive computing, software testing, and change propagation as an emerging field of model-driven engineering to understanding the problem under investigation.

2.1.1. Autonomic and Adaptive Computing

Autonomic computing (AC) is IBM's proposed solution to facilitate the problems of managing the growing complexity of computing systems, and the progressing nature of software system. The beginning of the AC was started in October 2001 and depicted a vision of computing systems [62] that manage themselves according to high-level objectives. The paradigm seeks to alleviate the burden of integrating and managing highly complex systems through increased automation and goal specification.

Autonomic systems, inspired by the concept of human autonomic nervous system, facilitate the paradigm shift from conventional human-manage technology era to technology-manage-technology era [49]. Autonomic Computing is a potential solution to the problem of increasing system complexity and costs of maintenance. It is an approach where the ultimate goal is to create computer systems that can manage themselves while hiding their complexity from the end users [52, 62]. While autonomic computing is shifting the burden of the detailed software operation and maintenance from human beings to technology, the human is only responsible to identify system objectives as high level policies, and have the system adapt its own components in response to changes in the operating environment [62].

AC comprises the activities self-configure, -optimize, -heal, and -protect [62, 80]:

- Self-Configuration: the ability to dynamically configure and reconfigure itself under changing the conditions, and in accordance with high-level policies representing business-level objectives; item Self-Healing: the ability to detect failed components and remove or replace them with other components without disrupting the system. In addition, this characteristic may involve the prediction of problems to avoid failures;
- Self-Protection: the ability to identify and detect attacks and cover various aspects of system security at the platform, operating system, and application levels; and
- Self-Optimization: the ability to maximize resource allocation and utilization for satisfying user requests.

Concept computing	**Current computing**	**Autonomic**
Self-configuration	Corporate data centers have vendors and platforms. Installing, configuring, and integrating systems is time consuming and error prone.	Automated configuration of components and systems follows high-level policies. Rest of system adjusts automatically and seamlessly.
Self-optimization	Systems have hundreds of manually set, nonlinear tuning parameters, and their number increases with each release.	Components and systems continually seek opportunities to improve their own performance and efficiency.
Self-healing	Problem determination in large, complex systems can take a team of programmer's weeks.	System automatically detects, diagnoses, and repairs localized software and hardware problems.
Self-protection	Detection of and recovery from attacks and cascading failures is manual.	System automatically defends against malicious attacks or cascading failures. It uses early warning to anticipate and prevent system wide failures.

Table 1. Four aspects of self-management with and without autonomic computing [62]

Table 1 presents the core features that IBM frequently cites, for supporting self-management in autonomic software. The main characteristic of autonomic computing software systems is self-management.

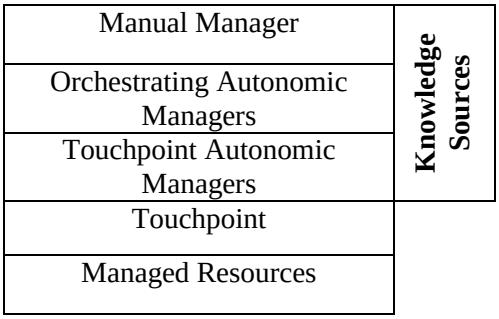

Figure 1. Autonomic computing reference architecture [55]

The aim is to shift the burden of details system operation and maintenance to the system itself and free the system administrators to perform higher level duties.

Earlier, the autonomic systems might deal with these characteristics separately [62]. Lately, these aspects will be essential properties of a general architecture of any computing software system.

The architectural blueprint for AC [55] organized a common layered approach for developing self-managing systems as shown in Figure 1. The horizontal layers (from bottom to top) include: managed resources, touchpoints, touchpoint autonomic managers, orchestrating autonomic managers, and a manual manager. The knowledge in the vertical layer knowledge sources (top-left of Figure 1) is composed of particular types of management data with architected syntax and semantics, such as symptoms, policies, requests for change, and change plans, this knowledge can be shared among autonomic managers (i.e., the top three horizontal layers), so that an autonomic manager can load knowledge from one or more knowledge sources, and the autonomic managers can activate that knowledge, allowing the autonomic manager to perform additional management tasks (such as recognizing particular symptoms or applying certain policies).

The lowest layer (i.e., Managed Resources) contains the system components that make up the IT infrastructure. These managed resources can be any type of resource (hardware or software) and may have embedded self-managing attributes.

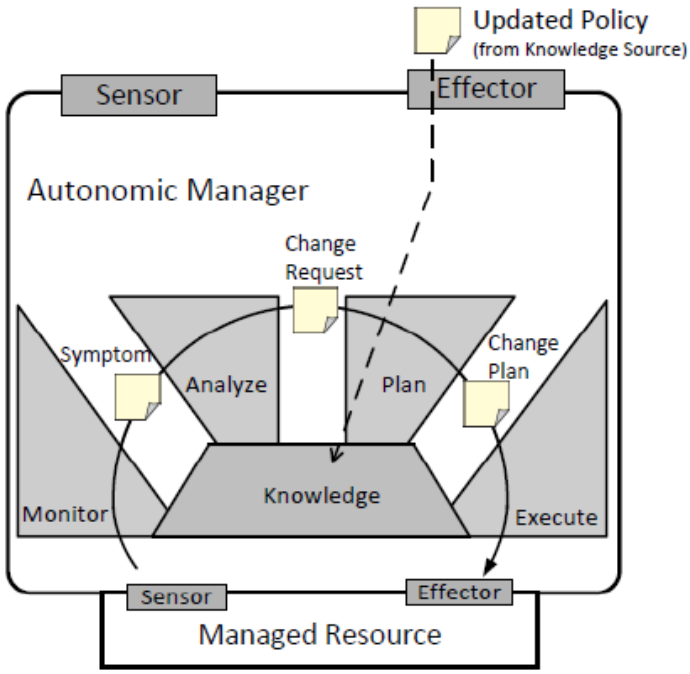

Figure 2. Closed loops of control in autonomic manager [67]

Immediately above the managed resources are manageability interfaces for accessing and controlling the managed resources, which are called touchpoints. Touchpoints implement the sensor and effector behaviors necessary to automate low-level management tasks [55, 62].

Sensors provide mechanisms for observing the state of managed resources, while effectors facilitate the implementation of runtime changes. Autonomic managers are categorized into two classes: (1) Touchpoint AMs, and (2) Orchestrating AMs [55]. Touchpoint AMs work directly with managed resources through their touchpoints. Orchestrating AMs manage pools of resources or optimize the Touchpoint AMs for individual resources. Orchestration may therefore occur within a single discipline for multiple resources (e.g. Self-Configuration only), or across multiple disciplines for a single resource. The uppermost layer is an implementation of the user interface that enables an IT professional to perform some management functions manually. Called the Manual Manager, these functions enable an IT professional to delegate management functions to autonomic managers.

Self-management in autonomic software is realized through a series of intelligent closed control loops [55] within Autonomic Managers (AM). Figure 2 [67] shows the structure of autonomic managers with respect to these closed control loops. AMs are responsible for implementing closed control loops that monitor, analyze, plan, and execute (MAPE) changes to achieve self-management goals.

The MAPE functions of AMs collaborate to manage state changes to the resource as follows:

- Monitor: continuously polls the managed resource for this state information, and correlates it into symptoms for analysis.
- Analyze: determines if the current state is undesirable, and generates a change request to be passed to the plan function.
- Plan: specifies the set of actions needed to remedy the state condition of the managed resource, and formalizes them into a plan for execution.
- Execute: implements change plans on the managed resource through its effectors, for the purpose of acquiring some desired state.
- Knowledge: coordinates access to data shared among the MAPE functions.

Sensors are built on managed resources to provide AMs with mechanisms for introspection, while effectors provide intercession mechanisms. In addition, the self-management policies that guide the behavior of AMs may be dynamically updated through these top sensors and effectors. During self-management AMs may dynamically add, remove, or replace components of the AC system, a process known as Dynamic Software Adaptation (DSA).

The movement towards autonomic computing [62] has led to the development of systems that can add, remove, and replace their own components at runtime.

Dynamic adaptation enables software to respond to changes in its environment, and seeks to improve the way in which systems are configured, managed, and integrated [62]. Salehie and Tahvildari [91] summarized the adaptation processes as follows:

1. The monitoring process: is responsible for collecting and correlating data from sensors and converting them to behavioral patterns and symptoms;

2. The detecting process: is responsible for analyzing the symptoms provided by the monitoring process and the history of the system, in order to detect when a change (response) is required. It also helps to identify where the source of a transition to a new state;

3. The deciding process: determines what needs to be changed, and how to change it to achieve the best outcome; and

4. The acting process: is responsible for applying the actions determined by the deciding process. This includes managing non-primitive actions through predefined workflows, or mapping actions to what is provided by effectors and their underlying dynamic adaptation techniques.

To enhance reliability, self-adaptive software should employ a safe process for dynamic adaptation, and be able to validate or verify its own behavior at runtime [107], [68], [108]. Runtime validation of adaptive software can be achieved by deploying the system with built-in tests, and mechanisms for automatically executing those tests and evaluating the results [68]. Dynamic adaptation is said to be safe if it does not violate dependencies between components, or interrupt critical communications [107].

The safe adaptation process in [107] encompasses three phases: analysis, detection and setup, and realization. The analysis phase occurs during development time, where the developers prepare a data structure that helps to keep detailed information such as component configurations, dependency relationships predicates, and adaptive actions.

The detection and setup phase occurs at runtime. Once the system detects a condition warranting adaptation, the adaptation manager should generate a safe adaptation path. In the realization phase, the adaptation manager and the agents coordinate at runtime to achieve the adaptation along the safe adaptation path established during the previous phase. The major states in the realization phase are as follows:

1. Running state : Every component in the process is running in its full operation.
2. Resetting state: The process is only partially operating and some functionalities related to the adapted component are disabled.
3. Safe state: Hold the system in a safe state while adaptive actions are performed.
4. Resuming state: Resume the system's partial operation once all adaptive actions are complete.
5. Original state: Perform a local-post action to return the system to a fully-operational state (i.e. first state).

If the failure occurs after the manager has sent out a resume message, then the adaptation should run to completion. If failure occurs during an adaptation step, the manager can retry the same step, attempt to return to the source configuration, or remain at the current safe conguration and wait for user intervention.

2.1.2. Software Testing

Software Testing involves executing a program on specified inputs, recording the results, and making an evaluation to determine whether the software behaves as intended [56]. Software testing is one of the V&V (verification and validation) software practices. Verification is the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase [1].

Validation is the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements [1]. Boehm [13] has informally defined verification and validation as follows: Verification: Are we building the product right?, Validation: Are we building the right product?.

Testing may be performed using black box or white box techniques [9]. Black box testing assumes no knowledge of the internal structure of the program. On the other hand, white box testing derives testing requirements from how thoroughly the program structure has been exercised [109]. Hence, for white box techniques, test adequacy is usually specified in terms of coverage of elements of the program, (e.g., branches, paths, statements and internal logic of the code, etc.).

Black box testing is mostly applicable to higher levels of testing such as system testing, while white box testing is applicable to lower levels of testing such as unit and integration testing. In white box testing the tester should have full visibility of the internal workings of the software product, therefore the tester should have programming and testing knowledge, on other hand in black box testing these knowledge's are not required.

Software components may be tested independently at the unit level; or as a set of partially connected building blocks during integration; or all together to validate the behavior of the entire system [17]. During testing, it may be necessary to develop scaffolding code. This includes stubs and drivers required for testing. A test stub is a mock implementation that simulates some behavioral aspect of a component under test (CUT), and a test driver is a program that executes test cases on the CUT [17]. The set of drivers and other tools to support test execution is called a test harness. Although black box as well as white box testing is equally essential, using only one is insufficient. So, a combination of black box as well as white box testing called as Gray box testing has been used in this dissertation proposal.

Test cases can be developed using black box methodologies or white box ones, using both methodologies are recommended. In order to have a rationally strict test, test cases can be developed by using some black box methodologies and then supplementing these test cases by examining the structure of the software, using white box methodologies. Several black box testing techniques are introduced and discussed in [86](graph-based testing, equivalence partitioning, boundary value analysis, comparison testing, orthogonal array testing). Techniques for white box test cases oriented include statement coverage, decision coverage, condition coverage, decision-condition coverage, multiple-condition coverage [82].

After releasing the software system to be used, post-delivery activities could start to keep the system operational and meet user need. Software evolution means that systems typically require perfective, adaptive, or corrective maintenance after delivery [94]. There are four types of maintenance according to Lientz and Swanson [71]:

- Corrective maintenance: deals with the repair of faults or defects found.
- Adaptive maintenance: consists of adapting software to changes in the environment, such as the hardware or the operating system.
- Perfective maintenance: mainly deals with accommodating new or changed user requirements.
- Preventive maintenance: concerns activities aimed at increasing the systems maintainability, such as updating documentation, adding comments, and improving the modular structure of the system.

Regression testing is that test could be run after changes are made to the software to ensure that it behaves as intended and that the modifications have not had an adverse impact on the quality of the softwar [19]. This may involve re-running the entire test suite (retest-all), or a strict subset (selective retest) [46].

A selective retest method commonly used in practice is firewall regression testing [103]. Firewall regression testing uses change impact analysis to identify the set of components affected by the change [103]. The identified components are then retested to ensure the system still behaves as intended.

In manual testing, software testers should play the role of an end user, and use almost all of the features of the software system to ensure the correct and intended behavior of the system. However, there are several tools in the literature that provided tangible assistance in the automation of the testing process [14, 20, 28, 43]. Test automation involves creating test scripts; and setting up a test harness for executing tests, logging the results, and performing a post-test evaluation [79]. If the post-test evaluation passes then the test harness should automatically terminate, otherwise additional test cases should be selected and fed through the harness to improve the testing effort.

In software testing, one of the critical concerns is the quality of the test. Test data adequacy criterion is a rule used to determine whether or not sufficient testing has been performed. This criterion is considered as quality measurement.

If P is a set of programs, and S is a set of specifications, and T is a set of test cases, we can formally define a test data adequacy criterion C as follows [109]:

- Testing criteria as measurements:

  A test adequacy criterion C is a function

  C : P $\times S \times T \rightarrow [0, 1]$

  C (p, s, t) = r means that the adequacy of testing program p against specification s using the test set t is of degree r according to criterion C. The greater the value of r, the more adequate the testing.

- Testing criteria as generators:

  A test data adequacy criterion C is a function

  C : P $\times S \rightarrow 2^T$

  A test set t $\in C\,(p, s)$

  means that t satisfies C with respect to p and s. In other words, t is adequate for (p, s) according to criterion C.

- Testing criteria as stopping rules:

  A test criterion C is a function

  C : P $\times S \times T \rightarrow [true, false]$

  C (p, s, t) = true means that t is adequate for testing program p against specification s according to criterion C. Otherwise, t is inadequate. as a stopping rule, a test data adequacy criterion C is a special case of measurements with the range [0,1].

2.1.3. Model Driven Engineering

The term Model-Driven Engineering (MDE) is typically used to describe software development approaches in which abstract models of software systems are created and systematically transformed to concrete implementations [41].

One of the main processes of the Systems Development Life Cycle (SDLC) is the design process, which starts with the construction of an abstract model of the new system along with the desired features.

The development of abstraction techniques aid to improve the programming practice. It has provided programming language constructs, specification techniques, program structures such as algorithms and data types, strategies for modular decomposition, and more [93]. The object-oritented paradigm becomes very common and frequently used in manufacturing.

Object-oriented software can be described as a set of interacting objects that communicate and collaborate in order to perform specific tasks. Each object can associate with [70]:

- Properties: characterize the object, describing its current state.
- Behavior: is the way an object acts and reacts, possibly changing its state.

The component standpoint provides a higher-level of abstraction than objects [16]. Component provides a modular part of a system, that encapsulates several objects into one unit. A component may be replaced by another if and only if their provided and required interfaces are identical. Therefore developers can use commercial-off-the-shelf (COTS) components, and customize the services provided to the specific requirements of the application. Software reuse comprises the following forms [44]:

- Interface reuse: reusing the signatures available for message passing;
- Code reuse: reusing classes or collections of procedures and functions; and
- Pattern reuse: reusing solutions to well-known problems.

Reuse is a principal subject in object-oriented software engineering, and component-based software engineering. One concept of object-oriented that helps objects to work together is inheritance. Inheritance in OO can be used to reuse code of existing objects, or to establish a subtype from an existing object. In inheritance, subclasses/child classes can inherit properties and methods from pre-existing classes called superclasses/parent classes. Beside the key benefit of inheritance which is to minimise the amount of duplicate code in an application by sharing common code amongst several subclasses, inheritance can also make application code more flexible to change because classes that are inherited from a common superclass can be used interchangeably, If the return type of a method is superclass.

The concepts of composition and delegation are common in object-oriented environment. ClassA is composed with ClassB if ClassA has a ClassB or ClassB* member (i.e., one of its attribute); then we can say ClassA has a ClassB. And, as before, ClassA inherits from ClassB if ClassA is derived from ClassB as a child class; then we can say ClassA is a ClassB. The passing of method calls to a composed object is called delegation.

Design patterns are optimized, reusable solutions to the programming problems that developers and engineers might encounter. These problems have been faced before by other engineers, and solutions have been designed and implemented to treat these problems. If developer encounter these problems, why recreate a solution when the developer can use already proven ones. A design pattern is not just a class or a library that we can simply incorporate into our system; it is much more than that. It is a template that has to be implemented in the right time and situation. Design patterns occur at several levels of abstraction, including [40, 44]:

- Architectural styles: which are patterns at the architectural level;
- Mid-level design patterns (or just design patterns): which are patterns involving classes and their interactions;
- Data structure and algorithms: which are patterns for implementing abstract data types and efficient operations; and
- Programming idioms: which are patterns for using a particular programming language.

One of the general purpose modeling language for modeling object oriented software system is the Unified Modeling Language (UML).

UML diagrams classified into three categories as follows [4]:

- Behavior diagrams: Describe the behavioral features of a whole system, a particular process in the system, or a specific object in the system. This includes activity, state machine, and use case diagrams as well as the interaction diagrams;
- Interaction diagrams: Describe the communication and the collabarotion between the various system components. This includes communication, interaction overview, sequence, and timing diagrams; and
- Structure diagrams: Describe the static composition of the system components. This includes class, composite structure, component, deployment, object, and package diagrams.

MDE is based on very general concepts that can be applied across many different disciplines. The basic set of concept includes Models, Meta-models and Transformations [74]. The goal of model-driven engineering (MDE) is to instate models as first-class citizens throughout the software process [41]. Transformations between models is therefore one of the key goals of MDE.

The major challenges that researchers face when attempting to realize the MDE vision can be grouped into the following categories [41]:

- Modeling language challenges: These challenges arise from concerns associated with providing support for creating and using problem-level abstractions in modeling languages, and for rigorously analyzing models.
- Separation of concerns challenges: These challenges arise from problems associated with modeling systems using multiple, overlapping viewpoints that utilize possibly heterogeneous languages.

- Model manipulation and management challenges: These challenges arise from problems associated with (a) defining, analyzing, and using model transformations, (b) maintaining traceability links among model elements to support model evolution and roundtrip engineering, (c) maintaining consistency among viewpoints, (d) tracking versions, and (e) using models during runtime.

Meta-modeling has been recognized as a standard technique for representing and transforming software artifacts [7]. However, many approaches only allow one-shot transformations to be expressed (i.e., single conversion of a source model into a target model). For one-shot approaches, subsequent changes in the source cannot be mapped to the target without reconstructing the entire target model. Change propagation, an emerging field of MDE, overcomes this limitation by allowing updates to be made to models after initial transformation [99, 105].

When developing an approach based on change propagation, the following factors should be considered [99]:

- Checking or Updating: an approach may simply indicate to a user where in the target changes should be made or, on the other extreme, make updates to the target without notifying the user as to which changes were made;

- Automatic or Manual: it may be possible to automatically extract and convert source model changes into transformations for the target, otherwise target transformations must be written manually; and

- Immediate or Batch: an approach may propagate changes to the target as soon as the source is changed, or propagate multiple changes when applied.

In the case of MDE research on runtime models, the goal is to produce technologies that hide the complexities of runtime phenomena from agents responsible for managing the runtime environment, and for adapting and evolving the software during runtime [41].

2.2. Related Work

This section presents, first, a systematic literature review performed prior to the start of this dissertation, followed by a brief description of research works that have been done in the area of testing adaptive system and model driven change propagation in order to position our work relative to previous research, and to identify those contributions that are complementary to our approach.

2.2.1. Preliminary Investigation

As a preliminary step in our investigation, a systematic literature review [3] was performed to determine the current landscape surrounding the research problem.

To properly focus the review, the following high-level research question was formulated:

> *Are there any approaches in the literature that can automatically synchronize a runtime test model for a software system with the model of its component structure after dynamic adaptation?*

This question was then expanded into the series of questions provided in Table 2. Each top-level question in the series represents a general research inquiry within the problem area, which was then refined with the (more specific) sub-questions that follow. Motivation behind each question in the series is shown in the rightmost column of the table.

The first question in Table 2 aims to find approaches that have been used to maintain synchronization between software models at runtime. Its sub-questions refine this objective to identify works that specifically address the research inquiry in the context of maintaining up-to-date runtime test models for adaptive software. The second question seeks to assess the extent to which existing MDE approaches provide a formidable solution to the research problem. The third question attempts to determine the practicality of implementing such MDE approaches.

Research Questions	Motivation
1. Are there approaches in the literature that focus on maintaining up-to-date models at runtime? 1.1. Are any of these approaches applied in the context of adaptive software? 1.2. 1.2 Are any of these approaches applied in the context of updating test models?	Identify works related to the idea of synchronizing test models at runtime in adaptive software.
2. Is there any evidence in the literature that multi-shot transformation approaches such as change propagation are effective for synchronizing different software models at runtime 2.1. Does any of the evidence demonstrate that such approaches are useful for ensuring completeness and consistency of runtime models in software?	Assess usefulness of approaches in the literature for synchronizing runtime models without having to completely re-construct the target model.
3. Are there any modeling tools, frameworks, or languages to support implementing approaches that synchronize runtime models 3.1. Are there any prototypes or case study applications that were built using these tools?	Assess practicality of developing a prototype of a solution to our specific problem using the approaches from (2.)

Table 2. Research questions and motivation to guide the systematic review [3]

Conducting the systematic review led us to several articles on the use of models at runtime, as well as current research directions in the area of MDE. The works on models at runtime included papers that harness executable models for dynamic adaptation and software testing, as separate issues. There was a noticeable lack of research being performed in the area of testing autonomic and adaptive systems [91, 68]. No works that address the problem of automatically synchronizing runtime test models in adaptive software were found during the literature search. Except for the research on change propagation [99, 105], most of the MDE approaches were focused on one-shot transformations that generate program source code from platform-independent models. Change propagation research appears to be in its early stages, and therefore does not have much direct tool and language support [99]. However, there appears to be a plethora of general MDE tools [39, 32, 100] that could be used to implement practical ideas on change propagation.

In summary, the results of the systematic literature review indicated that the proposed research direction may lead to advances in two relatively open fields of software engineering research: (a) Runtime testing of autonomic and adaptive systems; and (b) Change propagation. The next sections summarize research in the areas of self testable system, testing adaptive software systems, and model driven change propagation.

2.2.2. Self Testable System

There are several researchers have described that how software system could be a self-testable software or include self-testable components [75, 12, 98, 64, 10, 24].

Martins et al. [75] presented an approach to improve component testabilily by integrating testing resources into it, and getting a self-testable component. They intended to increase components testability to improve reliability of the component itself and of the applications using it. A self-testable component comprises a specification from which test cases can be derived in addition to its implementation. A prototyping tool was developed to support some activities of the proposed approach to show its feasibility. This prototyping tool named Concat. The tool is intended for OO components implemented in C++. For evaluating the fault revealing effectiveness of the test selection strategy they used a class from the Microsoft Foundation Class (MFC) library, CObList, which implements a linked list, and one derived class, CSortableObList, obtained through the Internet, which implements an ordered linked list. the results showed that the test strategy has a good potential in detecting methods interaction faults. Furthermore they were able to indicate the need to retest inherited features in the context of a subclass, even if they do not interact with modified or newly introduced features, among other reasons, to avoid that faults introduced during base class maintenance remain unrevealed in the subclass.

Blum et al. [12] presented a general technique which uses self-testing/correcting pairs to verify a variety of numerical functions.

In their work a user can take any program P and its self-testing/correcting pair of programs, if program P passes the self-test then, on any input, the user can call the self-correcting program which in turn makes calls to the original program P, to correctly compute the value. The techniques have been applied on integer multiplication, the mod function, modular multiplication, integr division, polynomial multiplication, modular exponentiation, matrix multiplication, determinant, matrix inversion and matrix rank. If this notion of selftesting/ correcting program pairs worked for complex programs then self-testing for autonomic computing systems would become petty. However, this technique only works for very well defined functions.

Le Traon et al. [98] presented self-testable OO components which embed their specification (documentation, methods signature and invariant properties) and test cases. Their pragmatic approach linked design and test of classes, seen as basic unit test components. Their approach has been implemented in Eiffel, Java, Per1 and C++ languages. As the authors admit, due to the direct support for Design-by-ContractTM in the Eiffel language, they detailed the Eiffel implementation since it makes the introduction of built-in test capabilities straightforward. Test cases are generated manually and embedded into the component. In Eiffel language the assertions can be used as oracle, but manually generated oracles can also used in complement, in case post-conditions and invariants are not sufficient to express functional dependencies between methods. The test quality estimate in their approach can be associated to each self-test for two benefits: (a) To help in the choice of a component, or (b) To guide reaching a test adequacy criteria when generating test cases. In their approach several mutation operators that applicable to different OO languages have been presented. Test case selection can be driven either by quality or by the maximum number of test cases desired. In their approach, test cases generated during class development are embedded into the class, and not the test model, as in our approach.

King et al. [64] presented and described a comparative case study performed on three autonomic applications that were engineered to include an implicit self-test characteristic (i.e., autonomic container (ACT) [95], Communication Virtual Machine (CVM) [25], and Autonomic Job Scheduler (AJS) [88]. Their experiments are divided into three parts as follows:

- Development experiments: Involved assessing the development effort for the three applications, focusing on the autonomic and self-testing features,
- Performance experiments: Comparing a non-self-test variant of AJS with a self-test variant that uses a distributed testing process and comparing a self-test variant of CVM with a thread-enhanced self-test variant, and
- Test set quality experiments: Measuring the effectiveness of test sets in revealing faults, and exercising program code.

Conducting their case study provides evidentiary insight into the benefits and software engineering challenges associated with developing autonomic systems with implicit self-test characteristic.

Beydeda and Gruhn [11] concentrate on overcoming the problem that caused by a limited exchange of information between the component provider and component user . A limited exchange and thereby a lack of information can have several consequences, one of the most important one is the requirement to test a component prior to its integration into a software system. Furthermore, a lack of information could complicate testing task. They proposed a new strategy called self-testing COTS components (STECC) to testing components and making components testable. The essence of this strategy is to augment a component with functionality similar to that of analysis and testing tools. Their strategy allows the component user to test the component with respect to information which is not directly accessible to the component user.

The information is either generated by the component itself on-demand or is encapsulated in it.

In [10] Beydeda extends the previous work by presenting an approach to self testability which encompasses test case generation and test evaluation. The new approach integrates the self-testing COTS components (STECC) method and the metamorphic testing approach. To demonstrate the feasibility of the proposed approach, a component has been implemented which provides basic functionality for number computing together with the functionality required by the STECC framework and for metamorphic testing the states entered and outputs produced. He concludes the main differences between the STECC approach and built-in testing approaches in the literature as follows: (1) Built-in testing approaches are static in that the component user cannot influence the test cases employed in testing. Precisely, the component user cannot identify the adequacy criterion to be used for test case generation. In STECC approach the adequacy criteria can be freely specified; (2) Built-in testing approaches using a predefined test case set generally require more storage than the STECC approach; and (3) Built-in testing approaches using a predefined test case set generally require less computation time than STECC at component user site.

Denaro et al. [24] proposed an approach that automatically synthesizing assertions that evolve over time and adapt to the new context-dependent interactions.

The synthesized assertions capture the semantics of the observed application behaviors, and can be integrated in the networking infrastructure to detect violating interactions at runtime and trigger self-management mechanisms correspondingly. Their approach embeds assertions in the communication infrastructures, to describe the legal interactions between the communicating entities. Such assertions are then checked at runtime to reveal misbehaviors, incompatibilities and unexpected interactions that may be due to hidden faults, changes in some components or malicious code. The synthesis of assertions at runtime can support the self-testing of adaptive systems by providing a way to create new test cases for validating components after an autonomic change occurs.

Deveaux et al. [27] started with simple self-testing of individual classes to optimized integration testing. They proposed a general Design-for-Trustability methodology (DFT) that produces self-testable components for ensuring test quality and increase software trustability. The self-testable concept has been implemented in the Eiffel, java and perl languages.To demonstrate their work some examples in java languages that are extracted from two packages in STclass web distribution site. The implementation of self-test is based on two parts:

- Preprocessor to build an instrumented source:
  - Preprocessor is written as a perl script for diffusion and portability.
  - IContract preprocessor for processing several source files together.
- Test API that supported by a simple library of four classes that provides three services:
  - A message center,
  - Method profiler and a set of useful methods to manage traces,
  - Assertions and test output.

The proposed framework for java implementation was usable in real life practice and implies not any overload on final code efficiency.

2.2.3. Testing Adaptive System

There are several approaches describing how systems can perform self-adaptation at runtime. Such approaches proposed self-adaptation processes, frameworks, or architectures able to investigate the need for the adaption when it is necessary and then perform the adaptation effectively. However, There is a noticeable lack of research being performed in the area of testing autonomic and adaptive systems [91, 68]. This section presents a brief literature review in the area of testing self-adaptive system in order to position our work, and to identify those contributions that are complementary to our approach.

Carlos and Rogrio [23] presented an approach for the dynamic generation of plans for conducting the integration testing of self-adaptive software systems. Their framework depends upon a combination of three techniques,

- Workflows: used as a means to implement the plans,
- AI planning: used to dynamically generate the plans, and
- Model transformation: used for supporting the translation between domain specific models into planning problems.

To demonstrate the feasibility of the proposed approach, a prototype has been built and applied into a component-based web application. The case study that was used in their work is a simple web shop application, which can be employed to sell goods on the Internet. The experiments performed demonstrated that the proposed approach was able to generate workflows for managing integration testing, that the most time consuming activity when generating a workflow is related to the architectural reconfiguration of the system being tested.

Arun Mishra and Arun K. Misra [78] proposed a formal approach that can be applied for validation of the system after component integration in the dynamic adaptive environment. They highlighted the importance of gaining the trace of the inter-component interactions for assuring the validation of component based dynamic adaptive systems. A general profiler with distillation characteristics has been developed to find out that how a component is affected by the other components through its interfaces. The profiler gathers all the traces of execution at the runtime and makes it possible to successfully trace the interactions among components across all threads of the adaptive system. In prior to use Mobility Workbench Model checking tool for validating adaptive system there was a need to transpose the collected interactions from trace file into formal specification. Thereby after implementing the adaptive software to apply their approach on and capturing the inter component interactions, the formal modeling pi-calculus has been used. Validating the implemented system based upon two factors: 1) assessment of external behavior and 2) checking of temporal properties (i.e. safety and liveliness) against a formal model of the system.

Da Costa et al. [22] have extended the adaptive system framework JAAF [29] by introducing a new activity called self-test. Self-testing in JAAF is embedded within the control closed loop of collect, analyze, plan and execute components. The self-test activity has ability to validate the new behavior and checks for its adequacy with the new environment before adapting it. The feasibility of the proposed approach is demonstrated by a case study where a system responsible for generating susceptibility maps. The susceptibility maps application makes use of different web-services that are capable of dynamic adaptation.

Prior to making an adaptive change, a set of test cases is applied on the given behavior and the execution takes place on the result of pass and fail of test cases.

Hu et al. [53] presented a new adaptive software testing approach in the context of an improved Controlled Markov Chain model.

They proposed a new set of basic assumptions on the software testing process, and replaced several unrealistic assumptions that have been used in the previous studies on adaptive testing rely on a simplified CMC model. To evaluate the effectiveness of the Adaptive Testing (AT) strategy, a case study on SPACE program has been used. The experimental showed that under different scenarios, the proposed strategy AT outperforms the traditional ones in both number of detected/removed defects and the cost of the testing process.

King et al. [68] proposed a self-testing framework, which is capable of dynamically validating the behavior of changed components through regression testing in autonomic computing systems. This process of validating is based on two key strategies: (1) Safe adaptation with validation: tests autonomic changes directly on managed resources during the adaptation process, and (2) Replication with validation: tests autonomic changes using copies of managed resources.

The Safe Adaptation with Validation strategy must be used only if the process of duplicating managed resources in autonomic computing system is expensive. To support their research investigations, the authors have developed prototype that implement autonomic self-testing according to the proposed validation strategies [67, 65, 95].

Munoz and Baudry [81] concentrated on testing the adaption policy and proposed a strategy for the selection of environmental variations that can reveal faults in the policy. They developed an approach called artificial shaking table testing (ASTT) for testing adaptation policies and their realization. ASTT consists in laying a DAS into a virtual shaking table, which produces artificial earthquakes (AEQ) that test its adaptation capabilities.

The main contribution of their work is the ability of ASTT to automatically generate AEQs in such a way that they simulate representative environmental changes. The experimental results of performing mutation analysis over an adaptive web server indicate that automatically generating violent and smooth environmental variations are beneficial to uncover faults in adaptation policies and their realization. The experimental results showed that out of 90 faults introduced into an adaptation policy realization, ASTT was capable of detecting 100% of them. ASTT are complementary to the research problem addressed in this dissertation. Validating the correctness of dynamically adaptive software is divided into two distinct research topics:

1. Determining whether or not the software is adapting correctly to environmental changes;
2. Ensuring that the software behaves correctly after an adaptive change takes place.

The work by King et al. [63] extends their previous work [65] on Autonomic Self-Testing (AST) by overcoming two of its limitations.

- To narrow the gap between online testing and other advances in autonomic computing, they address the need for system-wide validation in autonomic software through the description of a runtime integration testing approach. Their approach treats with the autonomic system as an interconnected set, Self-Testable Autonomic Components (STACs), and emphasizes operational integrity during the runtime testing process.
- They addressed AST of self-testing autonomic communication virtual machine application for which it is expensive to maintain test copies of managed resources. Their application motivates the need for Safe Adaptation with Validation (SAV) and system-wide AST, and is used as a platform for investigating their feasibility.

Niebuhr and Rausch [83] integrated runtime testing into component infrastructure DAiSI, to guarantee correctness of component bindings in dynamic adaptive systems.

Runtime testing enables them to detect incompatibilities of provided and required services before they occur. Runtime testing has been integrated into their prototype of DAiSI. Their work is complementary to ours as they guarantee system correctness and to support binding of components during runtime. One of the limitations of their work is not tackling the problem of cyclic dependencies of components. In future work they intend to investigate test case generation, to enable component developers to provide a single specification of their components and assure good test cases while trading-off test case execution overhead. If they accomplish this task, we could use the same technique to improve our approach in generating test case dynamically after adding new component.

None of the aforementioned approaches and frameworks describe how the test model is made consistent with the new structure of the autonomic system after adaptation takes place.

2.2.4. Model Driven Change Propagation

To the best of our knowledge, the approach presented in this dissertation is the first attempt at tackling the research problem under investigation. Following are some of the researches in the literature, which described models synchronization, change propagation, and traceability link establishment between test cases and classes under test in object-oriented systems.

Falleri et al. [89] implemented a simple traceability framework in the model oriented language Kermeta. This framework is based on a model definition, which allows a basic trace meta-model to be defined. They implemented the following features in the traceability framework:

- Generic traceability items;
- Trace serialization (in XMI 2.0, thanks to EMF); and
- Simple transformation from a trace to graphvizs dot language, in order to allow trace vizualisation.

With their proposed framework, it was possible to trace transformations within Kermeta. But it is still possible to improve the Trace metamodel and thus the framework.

Xiong et al. [105] is closely related to our work. They proposed an approach that automatically synchronizes two models related by transformations described in Atlas Transformation Language (ATL) [32]. Their clarification of the semantics of model synchronization under the context is precisely characterizing the behavior of the synchronization process with four important properties, namely stability, information preservation, modification propagation and composability. These properities give the users a clear view of what models will be after synchronization. These properties were much motivated by studies on updating semantics of database views [8] and the well-definedness of bidirectional tree transformation [38, 72]. They were the first who adapted these results to solve the model synchronization problem. An example that synchronizes class diagrams with relational database models has been used to demonstrate their approach. Although the semantics of the approach are similar to TIP, we address the specific problem of updating runtime test models after component-based adaptations.

Jean-remy et al. [89] have proposed a traceability framework for model transformation using Kermeta language. Their framework is based on a model definition which allows a basic trace meta-model to be defined. Several definitions have been provided to describe how their approach deals with different elements in the proposed framework. To demonstrate their approach, a very simple transformation example has been used where a class hierarchy turned into a database.

Ivkovic and Kontogiannis [58] proposed a framework called Model-Driven Software Evolution (MDSE) for model synchronization to achieve traceability of changes of software models that occur during software evolution and maintenance.

Whereby software artifacts at different levels of abstraction such as architecture diagrams, object models, and abstract syntax trees are represented by graph-based MOF compliant models that can be synchronized using model transformations. Their work is closely related to ours as they invistigated the problem of synchronization between software models when one is altered due to evolution or maintenance activities. Their view of software is in terms of models, each at a different level of abstraction (i.e., requirements, architecture, design, implementation). Each such model conforms to and is an instance of a corresponding metamodel. In TIP all models are at the same level of the abstractin (implementation). For model synchronization they employ an intermediary Graph Metamodel(GMS). This metamodel is an instance of MOF but is less abstract and more capable of providing desired semantic detail. They depict each model modification as a combination of graph changes: insert node/edge, modify node/edge, and delete node/edge. Finally, they provide a synchronization algorithm that is based on dependency relations implicitly defined by mapping source and target metamodels as graphs using GMS.

Synchronization approach between a feature model and its specializations is given by Hwan et al. in [54]. In their research work, to handle the consistency among the involved models their synchronization approach is based on traceability links between the interrelated models. These links are introduced during the generation of an initial specialization model by cloning all features in the original model feature. Once traceability links are created, the unidirectional synchronization between the two models is applied (i.e., propagate changes made in the feature model to the corresponding specialization models but not vice versa). Moreover the traceability links in their work represent only one-to-many relationships. Currently model synchronization problem requires a bidirectional approach, source and target models are mutable, that is capable of handling many-to-many relationships between elements of models.

Engels et al. in [33] have addressed the transformation of Class and Collaboration Diagrams into Java source code. They have presented how to deal with both the structural and behavioral mapping problems between UML and Java using a pattern-based transformation algorithm. The pattern used is an instance of a metamodel from which one can identify parts of the source diagram that is to be transformed. The main objective of their work is preserving the semantic information through transformation, they did not discuss how the defined transformations could be used in model synchronization.

The work by Giese and Wagner [45] is related to our work. They presented an approach to incremental model synchronization that is based on the declarative, visual, and bidirectional transformation technique of triple graph grammars. Triple graph grammars consist of three graph grammars which describe how to derive in parallel a source model, target model and correspondence model between source and target. Their work revealed how correct bidirectional model transformations can be derived from the declarative specification formalism [92] and how they provide an effcient and incremental model synchronization. Their work is complementary to ours as they addressed the the efficient execution of the transformation rules and how to achieve an incremental model transformation for synchronization purposes. Large model transformation could make the model synchronization inefficient and worthless, in their approach the incremental processing in the average case even larger models can be tackled. We could benefit from their approach to increase the efficiency of the syncronization process for large adaptive system.

Tratt [99] presented PMT as a new approach to change propagating model transformations. The main stages of a PMT transformation are as follows:

1. Take a source model, and an empty target model and transform the source model,

2. The user may make arbitrary changes to both the source and target models, independent from one another,

3. The user then requests that the changes they have made to the source model are propagated non-destructively to the target model.

The transformation is reinitialized with the updated source and target models, and the tracing information from the previous execution. The execution of the transformation then propagates changes from the source model to the target model. After the transformation has executed, the source and target models, together with the new tracing information created are once again stored. To demonstrate his work, the standard class to relational model transformation example has been used and described in detailed.

Chechik et al. [18] have taken a model-based approach and provided an algorithm for propagating changes between requirements and design models. Their approach propagates changes between requirements-level activity diagrams, and design-level sequence diagrams. They start with a set of models that describe a system at different levels of abstraction and/or from different perspectives. The main target was to provide a technique for propagating changes across these models. The key feature of their work was to explicate relationships between these models, and then utilize these relationships to propagate changes automatically, if possible, and to localize the regions in other models that should be modified by hand. Our approach differs from theirs in that the models in TIP are at the same level of abstraction (implementation-level). We made this decision with the hope of achieving higher levels of automation. This rationale is consistent with the findings of Chechik et al. [18], who reason that propagating changes between models at different levels of abstraction is impossible.

Hassan and Holt [48] addressed the question: "How does a change in one source code entity propagate to other entities?". They proposed several heuristics which could be used to predict change propagation by suggesting entities that should change based on an entity that has changed.

In order to validate their approach, they have collected a large data set which is based on the development a history of five large open source software systems, developed for a total of over 40 years by hundreds of developers spread around the globe. They studied changes to these large code bases using data derived from their source control repositories. Using this large data set, they empirically studied several general heuristics that predict change propagation. Then their newly acquired understanding has been used to build enhanced heuristics and measure their effectiveness in predicting change propagation. Their work is highly complementary to ours as the proposed heuristics may also be applicable to changes in test code. Applying these heuristics in both our component and test implementations may improve the overall approach.

Bart Van and Serge [90] established traceability links between test cases and classes under test in object-oriented systems explicitly. They used and evaluated six different traceability strategies that rely on naming conventions, static call graph, fixture element types, lexical analysis, Co-evolution and Last Call Before Assert. The authors analyzed the accuracy and the applicability of the proposed strategies on three systems JPacman, ArgoUML4, and Mondrian. The results revealed the strategy that is based on naming conventions achieved the highest accurate. For this reason, we choose this strategy to handle traceability relationships for test-related entries within an artifact, and across multiple artifacts.

Qusef et al. [87] presented a traceability recovery approach based on Data Flow Analysis. The approach identifies the tested classes by looking at all the classes that might affect the results of the last assert statement in each method of a unit test. To evaluate the accuracy of the proposed DFA-based recovery method, two system have been used, an open source system, namely Mondrian, and an industrial system, namely AgilePlanner. They compared the accuracy of the proposed approach with the approach based on naming conventions and LCBA presented in [90].

The comparison revealed that detecting the class under test cannot be fully automated and some issues should be better investigated.

CHAPTER 3. RESEARCH PROBLEM

This chapter describes the problems to be investigated and presents a detailed problem statement. The research focus is in the areas of software testing, autonomic computing, and model synchronization. The primary goal of the research presented in this dissertation is to formulate a model-driven approach that can automatically synchronize a runtime test model for a software system with the model of its component structure after dynamic adaptation, so that test model for the system can become consistent with its new structure after dynamic adaptation.

The next section provides the motivation for this research by emphasizing the need for automatically updating runtime test models after self-adaptation occurs. Section 3.2 concisely describes the problems to be investigated.

3.1. Research Motivations

Although researchers have developed many tools and techniques for building adaptive systems [107, 101, 69], there has been little research on assuring their quality and reliability [91]. More specifically, only a few researchers have addressed the need for runtime validation and verification (V&V) in self-adaptive software [68, 108]. However, since self-adaptation modifies the structure and behavior of the system, runtime V&V is necessary to ensure that errors are not introduced as a result of the adaptation process.

As long as technologies of AC continue to advance, it is vital that researchers interchange thoughts on how to validate the self-adaptive system dynamically. This includes: formulating approaches for integrating runtime testing into autonomic software; studying the detailed designs and prototype implementations that realize these approaches; and sharing the software engineering experiences of conducting such research studies.

King et al. [68] proposed the use of a runtime testing framework for validating changes in self-adaptive software. Their approach introduces an implicit self-test characteristic into autonomic and adaptive systems, which validates changes made to the software during dynamic adaptation.

Mishra et al. [78] defined a formal approach that can be applied for validation of the system after component integration in the dynamic adaptive environment. They proposed and developed a technique to capture the runtime components interactions using CLR mechanism (middleware of .NET Framework).

Carlos et al. [23] proposed an approach for the dynamic generation of plans for conducting the integration testing of self-adaptive software systems. However, none of the above approaches describe how the runtime test model for the system is made consistent with its new structure after dynamic adaptation.

To ensure that runtime testing of autonomic software can be applied in practice, it is necessary to investigate techniques for automatically updating runtime test models after self-adaptation occurs. For example, if self-adaptation introduces a new component, new integration test cases should be generated to validate its interactions with existing components. Similarly, if an existing component is removed, some test cases may no longer be applicable, or adequate for testing, due to changes in program structure. Such test cases would therefore have to be updated or pruned from the runtime test model.

3.2. Problem Statement

The problems under investigation are in the areas of software testing, autonomic computing, and model-driven development. More precisely, the core problem is to develop a model-driven approach that can automatically synchronizes a runtime test model for a software system with the model of its component structure after dynamic adaptation.

Currently, runtime testing of autonomic software systems has received little attention in the research community, and there is a general lack of freely available prototypes of the projects based on autonomic computing.

To properly address the problems under investigation, a high level research problem was formulated; this main problem was then expanded into a series of sub-problem as following:

1. Formulating a new approach for propagating structural changes in autonomic software to runtime test models. Although some research has been performed in the area of a self-testing framework in autonomic systems, none of these works have specifically targeted the problem of how the test model is made consistent with the new structure of the autonomic system after dynamic adaptation occurs. Special considerations need to be made when dynamically validating autonomic software systems in order for testing to be consistent with the new structure of system. In addition, runtime testing of an autonomic computing system should free system administrators from the burdensome details of updating and uploading the new test model after dynamic adaptation takes place.

    1.1. Investigating reductive and additive change propagations in autonomic software to runtime test models. As dynamic adaptation occurs additive or reductive changes could happen to adaptive software. Additive changes introduce new component interfaces and implementations into the system at runtime. So, the propagation of additive changes will require conveying detailed information about the new componens into the run time test model. On other side, reductive changes remove existing component interfaces and their implementations from the system at runtime. However, less attention has been given to the problem of automatically removing tests that may no longer be applicable due to changes in program structure.

1.2. Developing a prototype for the proposed approach to support the research in the area of testing autonomic software. Unfortunately, there is a general lack of freely available real-world autonomic systems for evaluating runtime autonomic software system approaches. In order to evaluate the proposed research ideas, a healthcare based prototype of test information propagation approach (TIP) will be developed, in which self-adaptation and self-testing could be practically useful.

1.3. Designing and performing controlled experiments to evaluate the TIP approach performance. The Propagation of dynamic changes in autonomic systems for updating built-in regression tests is necessary to get up-to-date test model. However, the performance of the technique for automatically propagating reductive and additive changes to the runtime test model needs to be measured to ensure that only relevant changes have been propagated. A set of experiments have to be conducted to evaluate approach performance.

CHAPTER 4. THE TIP APPROACH AND HEALTH CARE PROTOTYPE

To address the research problem, a model-driven approach was proposed for updating the runtime tests of a software system after dynamic adaptation. The approach, referred to as Test Information Propagation (TIP) [2], uses change propagation to synchronize elements of the adaptive system's component model, with related elements in its runtime test model.

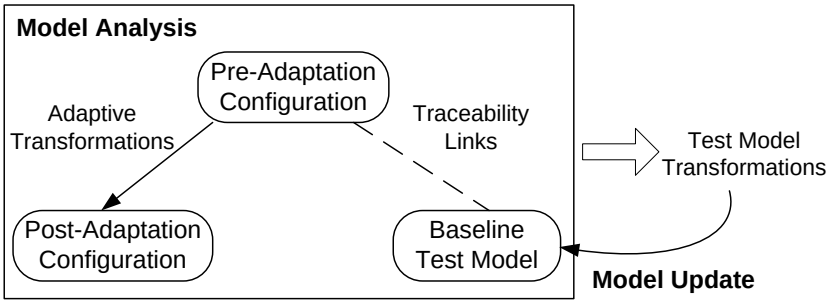

Figure 3. A test information propagation approach for adaptive software

Figure 3 provides an overview of the TIP approach. Under TIP, the transformation that maps the pre- to the post-adaptation component configuration is analyzed together with traceability links to the baseline test model. Analysis generates a set of transformations that are applied to the baseline test model to produce an updated test model.

As the first step in the formulation of TIP, we describe the high-level activities that can be performed to update the runtime test model of an adaptive system after additive and reductive changes. Propagating additive changes will require conveying detailed information about the new componens into the run time test model. However, less attention has been given to the problem of automatically removing tests that may no longer be applicable due to changes in program structure. Therefore, to gain some insights into the latter problem, our technical details and discussions focus on propagating reductive and additive changes.

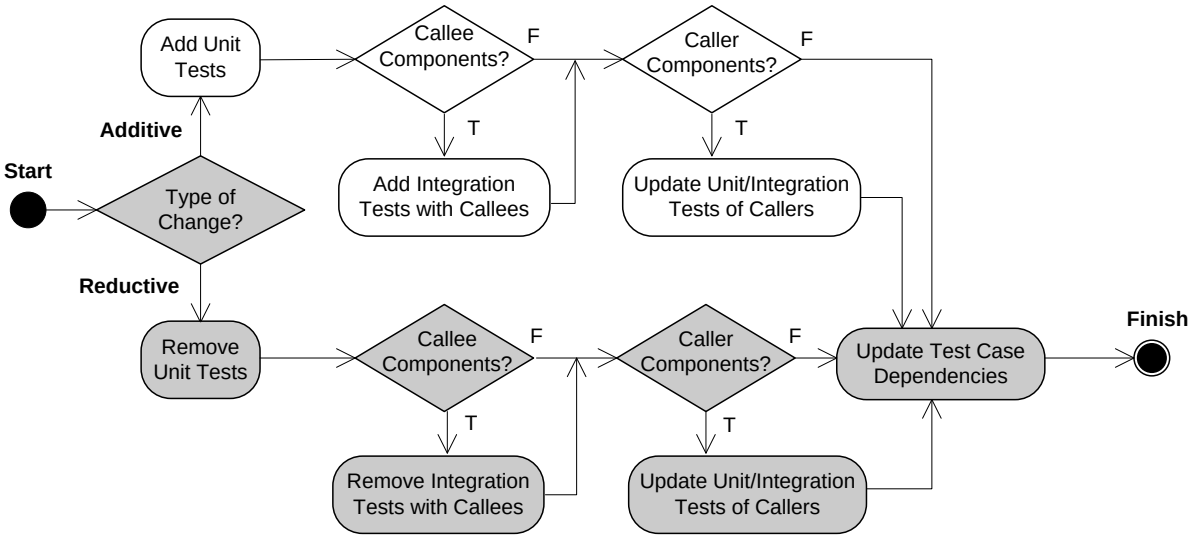

Figure 4. Major decisions and actions for updating runtime test models

Figure 4 provides a workflow of the major test-related decisions (diamond boxes) and actions (rounded rectangles) to be made when propagating additive and reductive changes. In our previous work [2], we have focused on Handling Reductive Changes since the initial version of the prototype is targeted on assessing the feasibility of automated propagation of reductive changes. In this dissertation the prototype is extended to include additive changes as well. Subsection 4.1 describes the workflow for additive changes, while Subsection 4.2 pertains to reductive changes. Steps that are common to both types of changes are described in Subsection 4.3. Note that mutative changes are not included within the scope of this dissertation but will be addressed in future work.

4.1. Handling Additive changes

Additive changes introduce new component interfaces and implementations into the system at runtime. The unshaded nodes in Figure 4 represent unique aspects of the workflow related to additive changes. The workflow for additive changes is described as follows, starting from left to right after the type of change has been identified:

(1) New Black Box Tests and Coverage Criteria: A component interface provides black box test information that be used to support dynamic test case generation for the new component. However, the level of automation is typically limited to: (a) generating test input values for the new component based on the data types of its operation signatures; and (b) defining test selection criteria based on the new component interface and using it as a generator rule.

(2) Is Implementation Accessible?: Details on the internal workings of the newly added component can be used to support dynamic generation of white box tests aimed at exercising the components structure. However, due to the widespread use of components-off-the-shelf (COTS) and the trend towards service-oriented architectures, such implementation details may not be accessible by the calling program. Therefore, the test update engine must be able to determine whether the implementation of the new component is readily available for structural analysis.

(3) White-Box Tests and Coverage Criteria: If the component implementation is accessible, its structure should be harnessed for dynamic test case generation and code coverage analysis. Full access to the source code provides a wealth of test information, and facilitates automating many existing white box testing techniques. For situations where the source is unavailable, researchers have been investigating approaches that automate white box testing techniques at the byte code level.

4.2. Handling Reductive Changes

Reductive changes remove existing component interfaces and their implementations from the system at runtime. The shaded nodes in Figure 4 represent elements of the workflow for these types of changes, which is described as follows:

(1) Remove Unit Tests: Unit-level test cases associated with the component targeted in the reductive change can be removed from the test model without many considerations.

This is because unit tests validate the behavior of a component in isolation, and are therefore independent of other components and tests.

(2) Does Target have Callee Components?: Dependency relationships between the component targeted for removal and other components, directly impact the changes that should be made to the test model. In general, the test model may contain integration tests involving callees and callers of the component targeted in the adaptation. Callees are components that are invoked by the adaptation target, while callers are components that invoke the adaptation target.

(3) Remove Integration Tests with Callees: If the adaptation target has callees, integration tests that validate the behavior of the target with its callees can also be readily removed from the test model. Since the adaptation target will be removed, tests that validate it with its dependents will not affect other parts of the test model. This assumes a software design in which there are no cyclic dependencies, i.e., component A depends on B but not vice-versa, and therefore A can be removed without affecting B or B's callees. Similarly, tests that validate A using B can be removed without affecting the tests of B or its callees.

(4) Does Target have Caller Components?: Removal of a component will have a great impact on the behavior of its caller components, thereby requiring updates to be made to tests that validate the behavior of these components with their own callers. If the adaptation target has many caller components, we anticipate that a significant number of changes would have to be made to the test model.

(5) Update Unit and Integration Tests of Callers: Both unit and integration tests of caller components must be updated after the adaptation target is removed. At the unit level, tests will no longer require calls to stubs of the adaptation target. Such stubs are also not necessary for integration-level configurations of the caller components. For integration tests, function calls to the actual adaptation target, as opposed to its stub, will also have to be removed.

4.3. Considering Test Case Dependency

Test cases typically depend on a number of entities. These range from test data stored in files or databases, to software components and frameworks, to physical hardware devices such as printers. In addition, before a specific test is run, it may be necessary to execute one or more related tests and verify that they have passed. An automated test harness is generally implemented to enforce this type of hierarchical test structure, where one test depends on the successful execution of other tests.

To achieve checking-level propagation we had to identify enough meta-data .The propagation engine to be able to identify general points of change in the test model, we had to maintain highly detailed information on both the adaptable components and their associated tests. This information included a list of the components test cases, along with the filenames, locations, and access information for the: (1) Test scripts that contain the tests, (2) Test drivers that make calls to the tests, and (3) Test stubs and/or data files used by the tests. The test metamodel should composed of the above mentioned objects which describe our domain model. The metamodel acts as a repository of these metamodel objects and provides direct access to them.

Figure 5 provides a meta-model showing the various types of dependencies in a test model for a software system. Such a meta-model can be used to support updating different elements of the runtime tests after dynamic software adaptation. As shown at the top-left of the figure, each test case in the model is composed of multiple dependencies. Dependencies are divided into three categories:

- Hierarchical: other tests that must be executed and pass the test in order for a test to run,
- Internal: entities that are implemented as part of the software, and
- Environmental: entities that are external to the software under test.

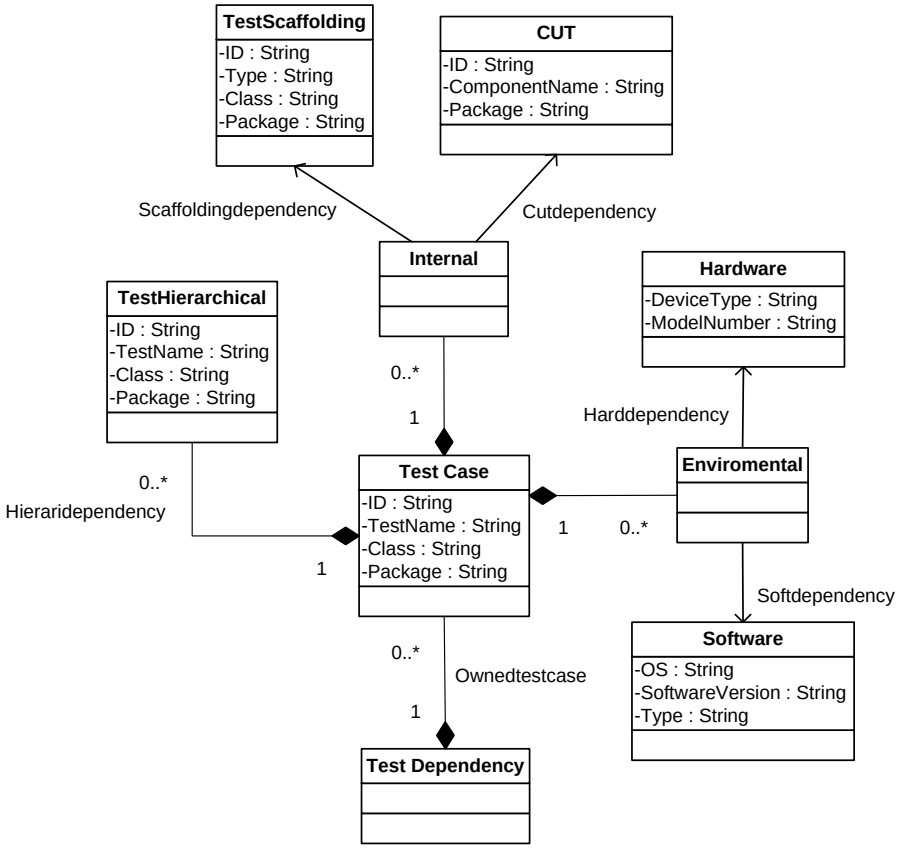

Figure 5. A meta-model to support test information propagation in self-adaptive systems

The hierarchical structure of a test harness is exhibited through the order in which drivers make calls to execute individual test cases. In our metamodel, hierarchical dependencies indicate the required test cases (if any) for each test to be run. Keeping track of this information facilitates locating and updating the test model elements associated with the constraints on execution order.

Internal test case dependencies include the component under test (CUT), test drivers, and test stubs. Storing information on the CUT allows adaptations in the systems component model to be directly traced to elements in the test model. If test cases are added, removed, or modified, the associated drivers and stubs can be updated by following the traceability links to these entities in the meta-model.

Environmental test case dependencies include hardware devices, other software systems, and stored test data. Adaptation may require updates to test information on specific hardware and network devices, or their pre- and post- test states. In addition, software frameworks and libraries to support automated testing may need to migrate to new versions or platforms as the software evolves. Lastly, test data stored in files or databases will also need to be updated to ensure adequate testing of data-dependent paths.

4.4. Modeling Tools, Frameworks, and Languages of TIP

In order to demonstrate and evaluate the proposed approach, our investigation involves the development of a prototype of TIP. This section discusses in details of healthcare based prototype of TIP and its features, tools, frameworks, and languages to support autonomic computing to improve the quality of patient care. Towards building self-adaptive system, the Tools, languages, and frameworks that would be used during the development should be able to provide the following features: (1) ability to replace the old source code file with the new one; (2) ability to build policies to provide self-Configuration, -Protection, -Healing, and -Optimization behaviors; (3) support runtime testing; and (4) support model driven engineering, (i.e., model driven tools that allows changes to be propagated between two models).

During our investigation we found several frameworks and tools to support the development of a prototype of TIP. These tools and frameworks are belonging to four categories: (1) Component-Based Adaptation Framework; (2) Policy-Driven Management Framework; (3) Model-Driven Development Tools; and (4) Automated Testing Tools. Next subsections describe some of the frameworks and tools found in each category.

4.4.1. Component-Based Adaptation Frameworks

Self-adaptive software system should be able to modify its own structure and behavior during runtime in order to cope with changes in its specification, environment or the system itself. The Spring Framework [101] is one of such framework that provides a core application container that allows you to specify components (called beans) using XML configurations. Beans can be written in Java and/or using dynamic programming languages such as Groovy [69] and Ruby [37]. Using these dynamic languages allows spring to act as adaptive framework since the container can be set to monitor beans for code changes, and dynamically use any new source code implementations. Another component-based frameworks that were found included Struts and Enterprise Java Beans [59, 34].

Spring provides five primary characteristics, Spring is a lightweight, dependency injection, aspect-oriented container and framework[15].

- Lightweight: Spring is lightweight in terms of both size and overhead. The bulk of the Spring Framework can be distributed in a single JAR file that weighs in at just over 2.5 MB. And the processing overhead required by Spring is negligible. Whats more, Spring is nonintrusive: objects in a Spring-enabled application often have no dependencies on Spring-specific classes.

- Dependency Injection: Spring promotes loose coupling through a technique known as dependency injection (DI). When DI is applied, objects are passively given their dependencies instead of creating or looking for dependent objects for themselves. You can think of DI as JNDI in reverse. Instead of an object looking up dependencies from a container, the container gives the dependencies to the object at instantiation without waiting to be asked.

- Aspect-oriented: Spring comes with rich support for aspect-oriented programming (AOP) that enables cohesive development by separating application business logic from system services (such as auditing and transaction management). Application objects do what they supposed to do, perform business logic, and nothing more. They are not responsible for (or even aware of) other system concerns, such as logging or transactional support.

- Container: Spring is a container in the sense that it contains and manages the lifecycle and configuration of application objects. In Spring, you can declare how each of your application objects should be created, how they should be configured, and how they should be associated with each other.

- Framework: Spring makes it possible to configure and compose complex applications from simpler components. In Spring, application objects are composed declaratively, typically in an XML file. Spring also provides much infrastructure functionality (transaction management, persistence framework integration, etc.), leaving the development of application logic to you.

4.4.2. Model-Driven Development Tools

Model-Driven Engineering enhances the notion of reusability and automation by the extensive use of models, meta-model and model transformations. The advantage of using a runtime explicit meta-model is that it allows new kinds of resource or meta-data information to be smoothly integrated into the system in a dynamic matter.

In order to design and build a meta-model we have chosen Eclipse Modeling Framework [39], EMF is a modeling framework and code generation facility for building tools and other applications based on a structured data model. From a model specification described in XMI, EMF provides tools and runtime support to produce a set of Java classes for the model, a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor.

The EMF code generation facility is capable of generating everything needed to build a complete editor for an EMF model. It includes a GUI from which generation options can be specified, and generators can be invoked. The generation facility leverages the JDT (Java Development Tooling) component of Eclipse. Three levels of code generation are supported: Model: provides Java interfaces and implementation classes for all the classes in the model, plus a factory and package (meta data) implementation class; Adapters: generates implementation classes (called ItemProviders) that adapt the model classes for editing and display; and Editor: produces a properly structured editor that conforms to the recommended style for Eclipse EMF model editors and serves as a starting point from which to start customizing.

Kermeta [100] is a metamodeling language which allows describing both the structure and the behavior of models. It has been designed to be fully compliant with the OMG metamodeling language EMOF (part of the MOF 2.0 specification) and provides an action language for specifying the behavior of models.

Kermeta is intended to be used as the core language of a model oriented platform. It has been designed to be a common basis to implement Metadata languages, action languages, constraint languages or transformation language. Support for meta-modeling was provided by the Eclipse Modeling Framework (EMF). Models instantiation and transformation was achieved using Kermeta, which facilitates the programmatic manipulation of EMF models (.ecore files).

4.4.3. Policy-Driven Management Frameworks

Policy driven management frameworks are an administrative technique to simplify the definition of autonomic management of a given exertion by launching policies to cope with circumstances that are expected to occur. Policies are set of instructions that can be referred to as a way to maintain order, security, consistency, etc.

Such policies are guidelines that would be specified by administrators and used by AMs to provide self -configuration, -healing, -protection and -optimization behaviors.

Ponder2 [85] is one of a policy-driven self-management framework that involves a self-contained, stand-alone, general-purpose object management system with message passing between objects. It incorporates an awareness of events and policies and implements a policy execution framework. It has a high-level configuration and control language called PonderTalk and user-extensible managed objects are programmed in Java. Ponder2 supports access control by providing authorization, delegation, information filtering, and refrain policies as described below:

- Authorisation policies: Define what activities a member of the subject domain can perform on the set of objects in the target domain. These are essentially access control policies, to protect resources and services from unauthorized access;

- Delegation policies: Delegation is often used in access control systems to cater for the temporary transfer of access rights. However the ability of a user to delegate access rights to another must be tightly controlled by security policies;

- Information filtering policies: Are needed to transform the information input or output parameters in an action. Some databases support similar concepts of views onto selective information within records for example a payroll clerk is only permitted to read personnel records of employees below a particular grade; and

- Refrain policies: Define the actions that subjects must refrain from performing (must not perform) on target objects even though they may actually be permitted to perform the action. Refrain policies act as restraints on the actions that subjects perform and are implemented by subjects.

The last policy it was about determining set of actions that must be performed by authorised manager within the system when certain events occur and provide the ability to respond to changing circumstances. These policies called *Obligation Policies*

King et al. [67] provide an XML-based policy driven framework for performing autonomic self-management. In their work the dynamic test model for autonomic systems was extended by applying the concepts of knowledge sources to testing activities, and explicitly describes the interdependency relationships of the model components. They provide a highly reusable detailed design for autonomic managers, test managers, touchpoints, and self-management policies that facilitate automation. To demonstrate the feasibility of their approach, a case study was developed that applies the features of self-configuration, self-optimization, and self-testing in the context of job scheduling.

4.4.4. Automated Testing Tools

JUnit [43] is an open source unit testing framework for Java programs, it is integrated with several IDEs such as Eclipse. Used for writing, exectuting automated tests and revealing the test results. Unit testing is an important step in order to validate that individual units of your system source code are working correctly. By unit we mean the smallest testable part of a program, function, application, etc..

In Test-Driven Development (TDD) technique for software development, the unit test is continuously performed on source code, the purpose of is to have something working at the current point and make it perfect later. After each test, refactoring is done and then the same or a similar test is performed again. The process is iterated as many times as necessary until each unit is functioning according to the desired specifications.

Junit provides several features that empower the tester to create and run tests easily, the following is some of that features:(1) API for easily creating Java test cases, (2) Comprehensive assertion facilities to verify expected versus actual results, (3) Test runners for running tests, (4) Aggregation facility(test suites), and (5) Reporting.

JUnit 4 provides several annotations, for example:

- @Test: The Test annotation is used the to tell the JUnit framework that the following method can be run as a test case.

- @Before and @After: These annotations are used the to tell the JUnit framework that the annotated methods with @After should be run after the @Test method, while the annotated methods with @Before should be run before the @Test method. They can be used to setup or tear down the test environment.

- @BeforeClass and @AfterClass: These are pretty similar to the above annotations, but they are only run only once. Annotated method with @BeforeClass means this method should be run only once before any of the @Test methods in the class. In opposite the method that annotated with @AfterClass means this method should be run after all the tests in the class have been executed. An application for these kind of methods could be used to loggin into a database, setup the database connections and loggout.

- @Ignore: This annotation would be used to tell the framework to temporarily ignore and not execute the methodes annotated with @Test. JUnit 4 test runners is able to report the number of ignored tests along with the number of tests that ran and the number of tests that failed.

Figure 6 presents an example of a simple JUnit test.

Cobertura [28] is a free Java tool that calculates the percentage of code accessed by tests. It is used to identify which parts of your Java program are lacking test coverage, it is a free plug-in for Eclipse IDEs. Cobertura provides sevral reports based on the coverage criteria has been requested to measure how well the program is exercised by a test suite. One or more coverage criteria are used.

```
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

public class CopyFileTest {

    @Before public void createOutputFile() {

    }
    @Test public void copyFile() {

    }
    @After public void deleteOutputFile() {

    }
    @BeforeClass
    public static void setUpClass() throws Exception {
    }

    @AfterClass
    public static void tearDownClass() throws Exception {
    }

}
```

Figure 6. Simple Junit testcase example

There are a number of coverage criteria, the main ones being [82]: (1)Function coverage: Has each function (or subroutine) in the program been called?, (2)Statement coverage: Has each node in the program been executed?, (3) Decision coverage: Has every edge in the program been executed?

For instance, have the requirements of each branch of each control structure (such as in IF and CASE statements) been met as well as not met?, (4) Condition coverage (or predicate coverage): Has each boolean sub-expression evaluated both to true and false? This does not necessarily imply decision coverage, and (5) Condition/decision coverage: Both decision and condition coverage should be satisfied.

Some of the features supported by Cobertura are: (1) Can be executed from ant or from the command line; Instruments Java byte code after it has been compiled;

(2) Can generate reports in HTML or XML; (3) Shows the percentage of lines and branches covered for each class, each package and for the overall project; (4) Shows the McCabe cyclomatic code complexity of each class, and the average cyclomatic complexity for each package, and for the overall product; (5) Can sort HTML results by class name, percent of lines covered, percent of branches covered, etc. and can sort in ascending or descending order. Several coverage tools for Java can be found in the literature such as Clover, Emma, Jtest, and Serenity.

4.5. Application Description

Using the approach by King et al. [68], we implemented a small autonomic system with runtime testing capabilities for evaluation purposes. To provide a realistic context for the prototype, we developed the application based on a healthcare scenario in which self-adaptation and self-testing could be practically useful. Our scenario conveys the idea of a service-oriented healthcare solution.

Scenario. A person takes ill while abroad and is admitted to a local clinic. A service-oriented software solution provides the admitting doctor with services for electronically: (a) retrieving and updating the patient's medical records stored at hospitals or clinics in his/her hometown; (b) scheduling an appointment with another physician or specialist on the patient's behalf; and (c) requesting that a pharmacist fills a prescription for medical drugs to treat the patient's condition.

The goal of the described application is to improve the overall healthcare process from the perspective of patients, doctors, and other stakeholders, while reducing the burden of system administration. Automatic service integration and configuration through self-adaptation are therefore key characteristics of the application.

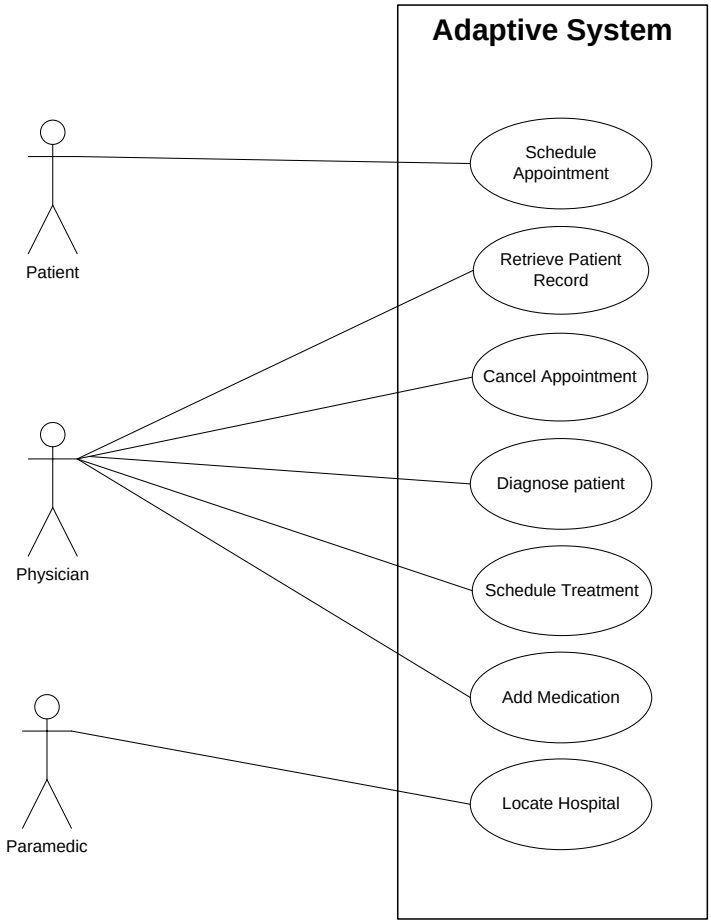

Figure 7. Basic use case diagram of EMR and appointment services

In the presence of medical emergencies such a system is mission-critical, and hence integrated runtime testing is vital to ensure that system operations are reliable after adaptation occurs.

4.6. Adaptation Scenarios

Health care adaptive system is composed of 8 use cases as shown in Figure 7. The use cases represent some core functionality of the system. Based on these core functionalities, the following scenarios might be happened in the healthcare services application:

(a) Automatically diagnosing patients scenario: The scenario is based on a situation where doctor needs to diagnose a patient based on a list of symptoms.

The doctor enters the patients symptoms and the autonomic system compares those symptoms against known diseases. The diagnosis results are returned to the doctor and if applicable, the doctor can request clinical trials related to the patients medical conditions. (b) automatically locating hospital scenario: The scenario is based on a situation where a patient is to be admitted to a hospital as he is seriously injured in a road accident. The attending ambulance service personnel have to determine the location of the hospital where the patient can be treated. The decision to choose a hospital depends on various factors such as the criticality of the patient, distance of hospital from the accident location, requirement for specialty services, availability of doctors and so on. The autonomic system plays a vital role in this decision making mechanism.

(c) Automatically retrieving patients record scenario: The scenario is based on a situation where a patient undergoes heart surgery at Fargo Childrens Hospital (FCH) and is moved to Merit care at the familys request. The doctor at Merit care asks patients father to provide the following information: the name of the attending physician at FCH, an x-ray, a summary of heart related medical data. His parents never got a copy of medical records.

Thus the role played by an autonomic system here is to authorize the hospital to retrieve patients record from other hospitals system.

(d) Automatically re-Schedule appointment scenario: The scenario is based where a patient logs into a hospital website to schedule an appointment with a doctor. The registration form requires the patient to choose from a list of symptoms. Based on the symptoms, the autonomic system schedules an appointment with a specialist doctor whom the system thinks might have experience with treating the given symptoms. Once the appointment is scheduled, the autonomic system sends a confirmation through an email, text message and automatic voice message. However due to some reasons the doctor has to cancel all the appointment of that particular day.

Thus autonomic system will automatically schedule an alternative appointment for the patient when the original appointment gets cancelled.

(e) automatically prioritizing available services scenario: The scenario is based on the situation where a patient is undergoing a critical heart surgery, the case is difficult, and need a second opinion from another remote expert specialist through video conferencing. Assume that the broadband network of the hospital is supporting simultaneous streaming services such as security camera feed, Internet telephone communication etc. A failure occurs, due to some unexpected reasons, resulting in a reduced network bandwidth provided by the broadband network. In this situation an autonomic system helps in dynamically prioritizing available services depending on the criticality factor.

4.7. System Development and Architecture

This section provides an overview of different languages, tools and frameworks that have been used in order to build an autonomic and adaptive system, and then described the architecture of service-oriented healthcare Prototype of TIP.

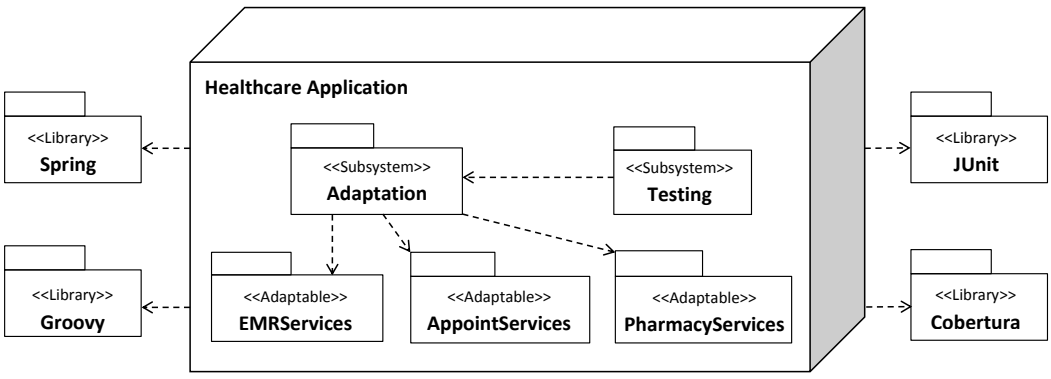

Figure 8. Architecture of service-oriented healthcare prototype

An application was developed in Java [96], using the Eclipse IDE [31] and the tools/libraries to support adaptation, testing, and change propagation.

As shown at the top-left of Figure 8, we used the Spring Framework [101] to provide a component-based application container for the three major application services. These services were: EMRServices, AppointServices, and PharmacyServices. Services were made adaptable using the dynamic language Groovy [69] (bottom-left), which allows components to be specified as beans within the application container. At runtime, the container was set to monitor the Groovy beans for source code changes, and automatically reload them to use the new implementations. The Adaptation subsystem (center-left) included a manager that was responsible for updating the component source (.groovy files) at runtime. The runtime test model for the application was defined in the Testing subsystem, and consisted of 29 test cases for validating the implemented services. Tests were developed using a combination of black box and white box techniques. Since JUnit [43] is built into the Groovy runtime, we created automated tests for each bean by scripting JUnit tests in the Groovy syntax. Cobertura [28] was used to collect line and branch coverage of the application services. This was achieved by instrumenting the Groovy byte code (.class files), executing the tests, and exporting the results to a coverage report in XML format. Support for meta-modeling was provided by the Eclipse Modeling Framework (EMF) [39]. Model instantiation and transformation was achieved using Kermeta [100], which facilitates the programmatic manipulation of EMF models (.ecore files). Kermeta therefore provided us with a programming environment with which we could set up our simulation.

4.8. Detailed Object Design

This section presents the detailed system design aspect of the EMR and Appointment systems along with class and sequence UML diagrams.

4.8.1. EMR Service Subsystem

Figure 9 shows the detailed object design of the EMRService in the TIP prototype.

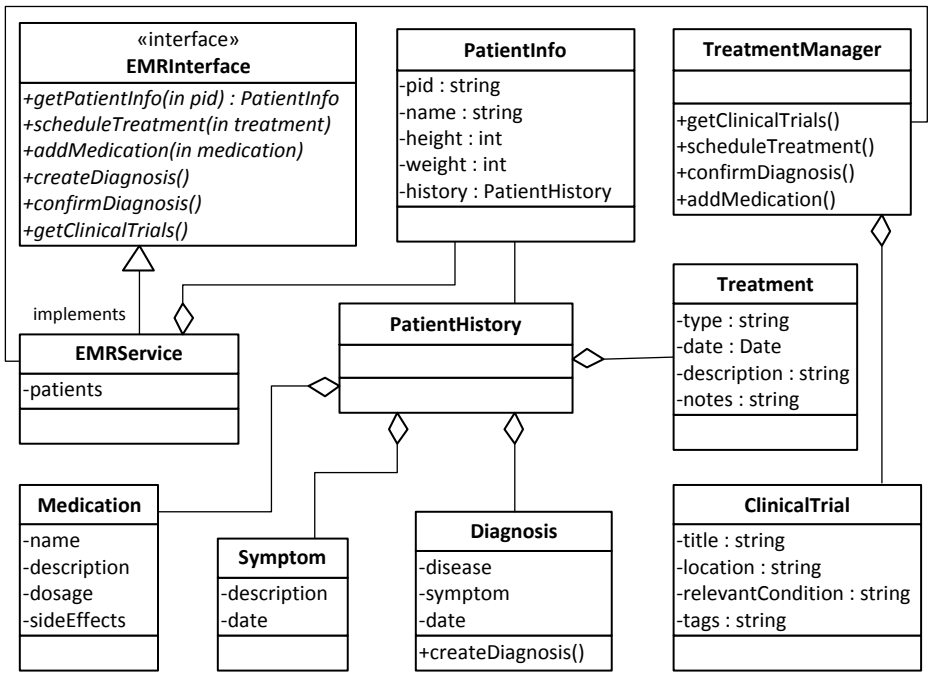

Figure 9. Design of EMRservice (based on data analysis spec. [30])

Our object design is based on a software requirements specification for EMR data analysis [30], which was elicited from a domain expert as part of a software engineering course project.

As shown at the top-left of Figure 9, the interface for the EMRService consists of the following six operations: getPatientInfo - retrieves the patient's medical information; scheduleTreatment - schedules a treatment to address the patient's condition; addMedication - prescribes medication as part of a patient's treatment; createDiagnosis - allows the doctor to enter their medical diagnosis of a patient's condition; confirmDiagnosis - checks whether a patient's symptoms are consistent with the diagnosis; and getClinicalTrials - querying clinical trials that may be relevant to the patient's case. The class labeled EMRService implements the operations in the EMRInterface. Upon receiving a request for service, this class orchestrates a series of calls to the other classes in Figure 5, in order to realize the needs of the client. Recall that classes within the EMR subsystem were made adaptable via the dynamic language Groovy [69].

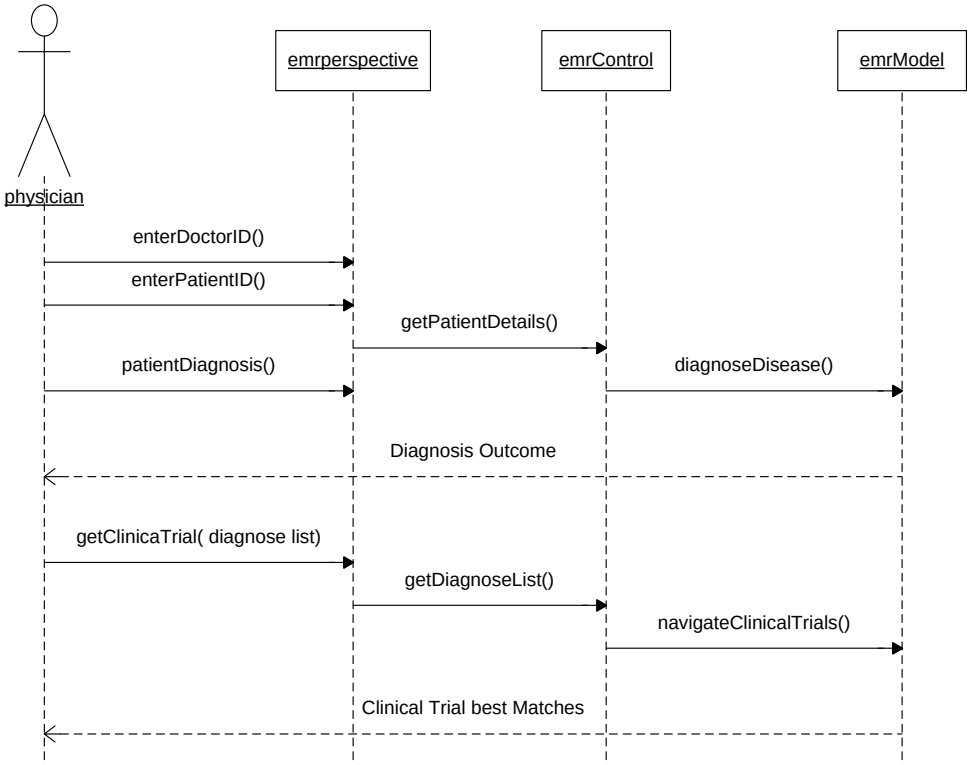

Figure 10. Diagnosing patient sequence diagram

The Sequence Diagram in Figure 10 shows the whole message passing between the doctor and EMR Service objects to retrieve the patient diagnoses based on the list of the symptoms. If that doctor is authorized to search through the clinical trial, the doctor can request the best clinical trials, then the system searches for the clinical trial based on the diagnoses results and returns the doctor with valuable detailed information.

4.8.2. Appointment Service Subsystem

As shown in Figure 11, the detailed object design of the Appointment service in the TIP prototype. The class diagram for the Appointment service consists of the following six classes: user, doctor, patient, hospital, department, calendar. For faster service, instead of entering waiting room of outpatient departments the user could log in the system and make an appointment before he visits a hospital.

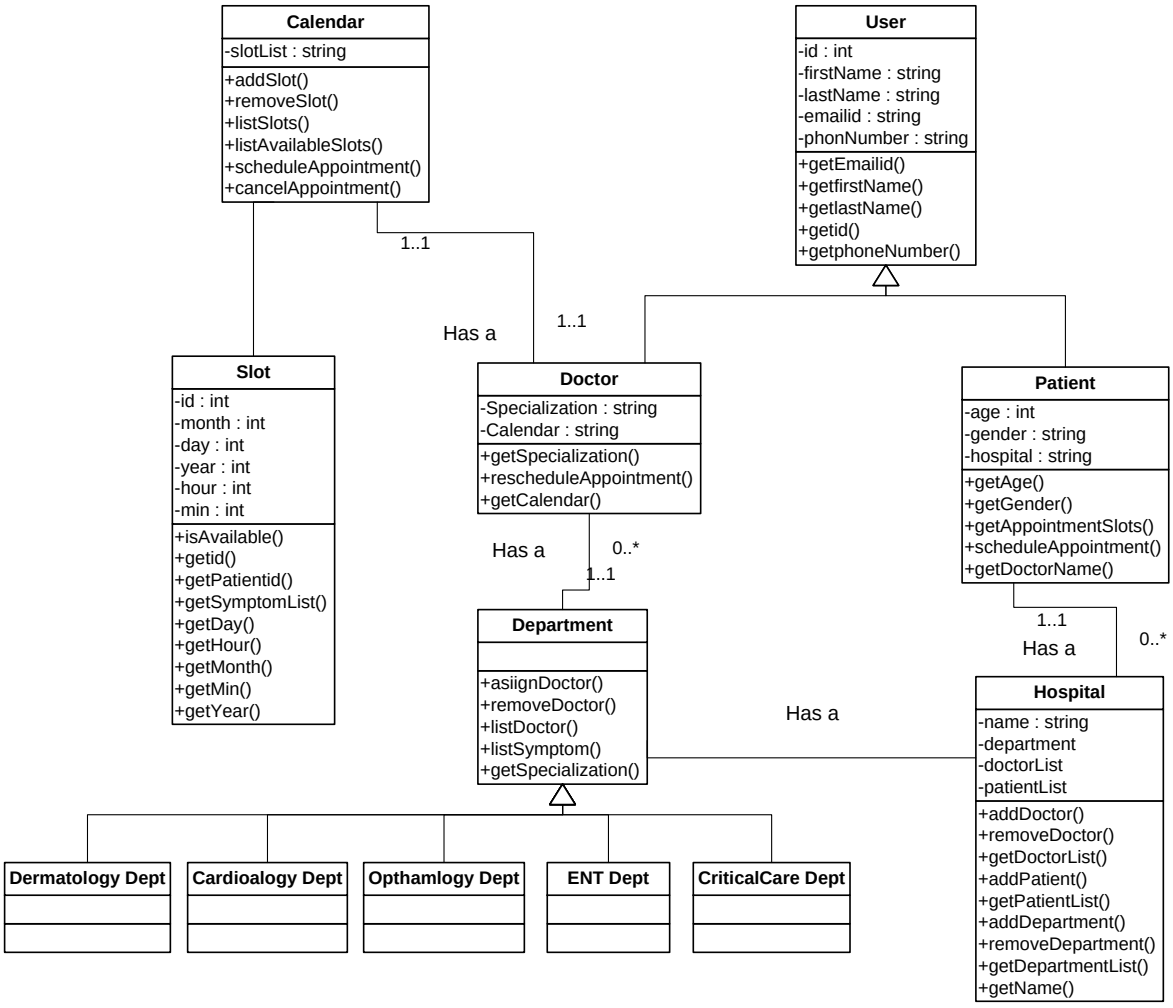

Figure 11. Detailed class diagram of appointment service

The appointment subsystem would have all departments and doctors schedules, once the user get in the appointment system he could schedule an appointment within one of the available slots of the required doctor.

Figure 12 shows the interaction and the sequence of messages exchanged between the Patient and the Appointment system that needed to carry out the functionality of scheduling an appointment scenario. When a patient chooses the desired slot, the system enables the schedule button and let patient schedule his appointment. When the appointment is scheduled successfully, the system sends the patient a confirmation mail.

Figure 13 presents the sequence diagram for finding the best competent doctor.

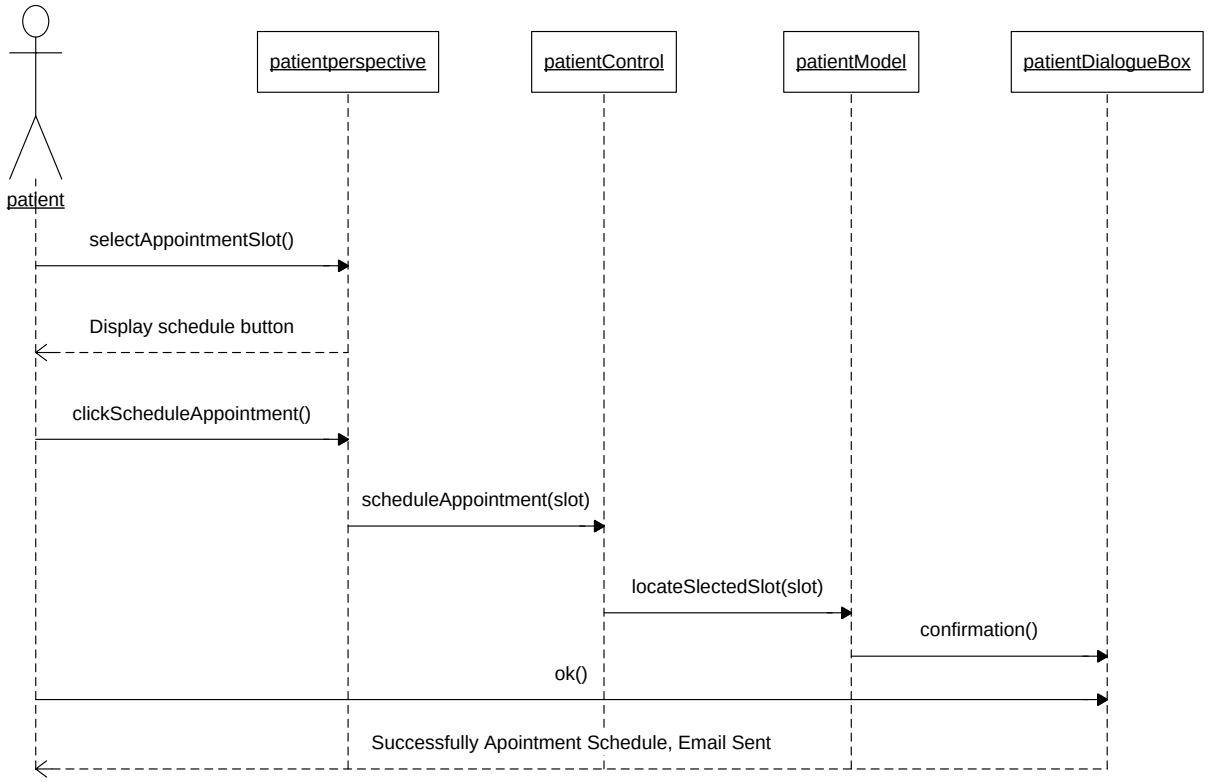

Figure 12. Scheduling appointment sequence diagram

Once the patient selects his symptoms the system returns the associated physicians with their calendars, the calendars highlights the availability date and time for each doctor.

Figure 14 shows the interaction of the doctor with the appointment system for cancelling/rescheduling an appointment. When a doctor selects cancel button to cancel all the appointments for a particular day, the system automatically reschedule appointment for the patient with another competent doctor and sends the confirmation of rescheduled appointment via email.

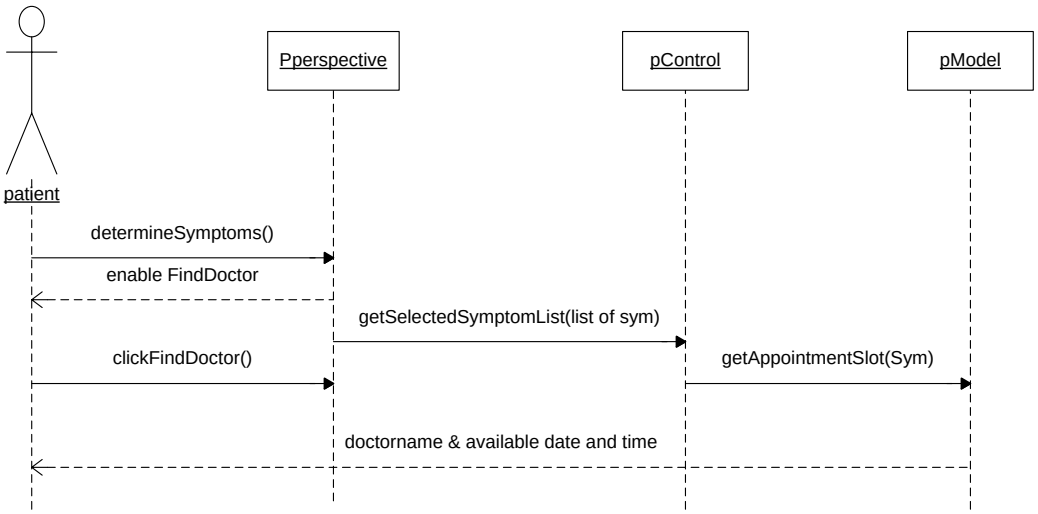

Figure 13. Exploring competent physician sequence diagram

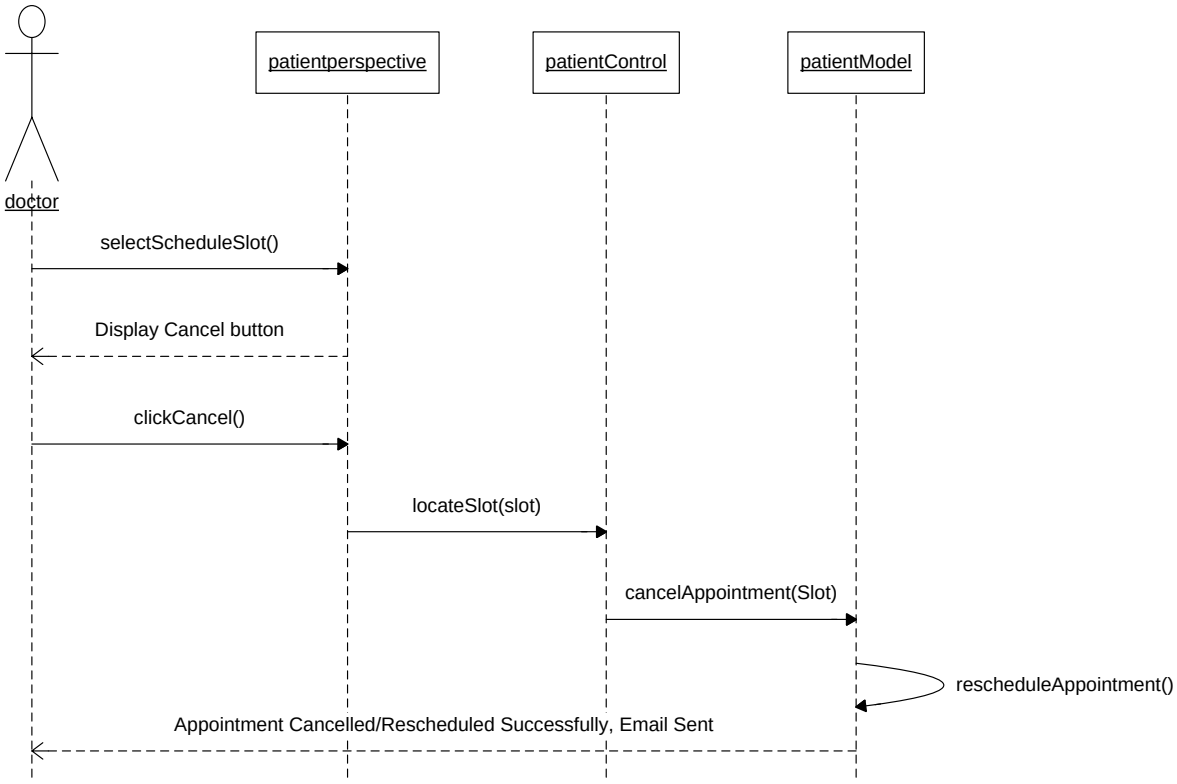

Figure 14. Cancelling/rescheduling appointment sequence diagram

4.9. Dynamic Models Instantiation

We performed a simulation to propagate additive and reductive changes in the EMR service implementation to its associated test implementation. To set up the simulation, we used Kermeta [100] to specify and instantiate two models associated with the EMR service: a component model and a test model. The EMR service component model was initialized with the names of the classes within the subsystem, as well as the dependency relationships among them. Both the class names and dependency relationships were captured automatically using Classycle [42].

Classycle dependencies are classified into three types: (1) usedBy: other software components are using the adaptation target, (2) usesExternal: adaptation target uses external java packages and libraries, and (3) usesInternal: adaptation target uses other software components.

Figure 15 provides an overview of our proposed approach to instantiate and then load the two models associated with the EMR service. A general parser has been built to populate the required information from XML reports that was generated by Classycle analyzer, and automatically create the dynamic instance of EMR component model (i.e. xmi file). Although research is advancing in change propagation and testing adaptive systems during runtime, there is a lack of development in the area of creating a comprehensive dynamic instance of the test model automatically. While the automated generation of the dynamic test model can handle the most important test information (i.e. callee, caller, hierarchical test structures), some other test information needs to be injected manually (i.e. hardware, software, stub and driver dependencies). Hence at this stage our methodology provides a semi-automatic approach for building the test model.

The EMR test model conformed to the structure of the meta-model defined in Figure 5. Traceability relationships were handled through naming conventions, which we elaborate on as part of the lessons learned in Subsection 6.

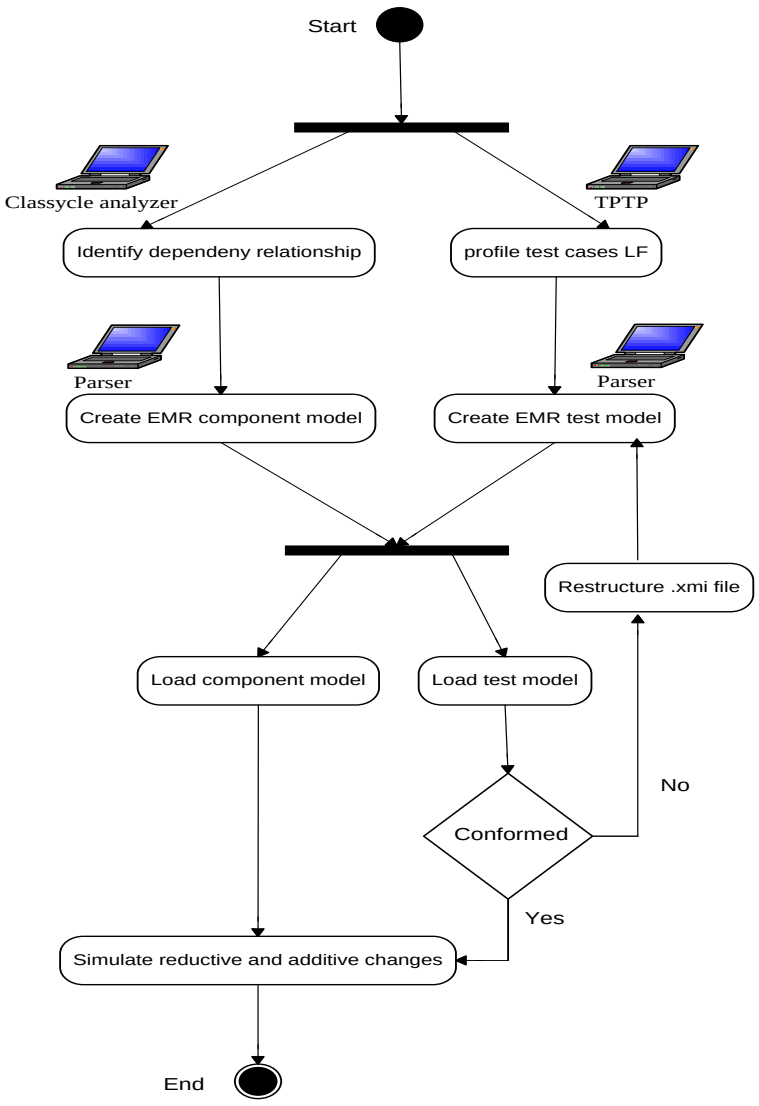

Figure 15. Component and test models instantaition

We build an GUI in java that allows testers to feed the rest of the test model dependencies for each test case. After saving the dependencies, an xmi file will be created that contains the new feeded test information, the dynamic instance test model then might be updated with the new test dependencies to get a full test case dependencies information.

As shown in figure 16 the tester could select the test method that required to update its test dependencies from test method ComboBox, also he could select the class and package which that test belongs to as well.

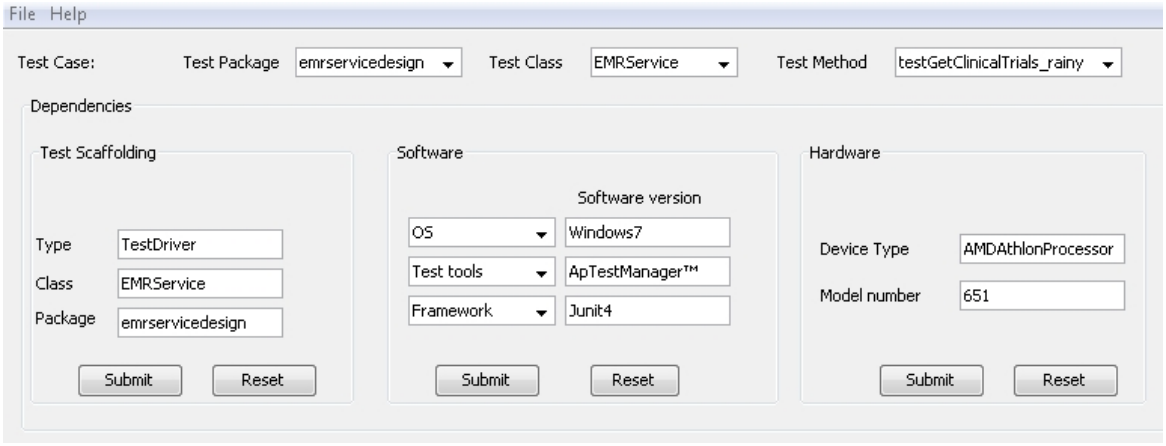

Figure 16. Test case dependencies GUI

After adding all test dependencies and submitting the information, an .xmi file will be created which would be used for updating the existed dynamic test model instance.

The Eclipse Test and Performance Tools Platform (TPTP) [97] Project provides a way to address the entire test and performance life cycle, from early testing to production application monitoring, including test editing and execution, monitoring, tracing and profiling. TPTP offers different report views, the Method Invocation Details view used in our approach to see the detailed information about all test cases that related to EMR service, including test case name, class name, package name, and all methods have been invoked during test case execution.

In order to create EMR dynamic instance test model, we parse each detailed test case execution .xml file, extract only required information, and then merge all restructured execution files into one .xml file to build an test model(.xmi) that conformed to the structure of the meta-model defined in Figure 5.

CHAPTER 5. EXPERIMENTAL SETUP

This section presents the setup of the experiments and the procedures for measuring the performance of the approach for automatically updating runtime test models after self-adaptation occurs. After self-adaptation introduces the removal or addition of components from/to a program structure, some test cases may no longer be applicable, and some existing integration test cases should be updated to validate its interactions with the remaining components in the software model. Alternatively, new test cases should be added to validate the new component behavior and its interaction with other existing components.

We applied our approach to an adaptive java healthcare application described in Chapter 4. In order to evaluate the generality of our approach, we applied our approach to adaptive java jpacman application as well. We then compare the outcome against the result of an evaluation performed by a developer. Our approach to measuring the performance of the change propagation engine will be described later. The change propagation engine in the exemplary state could correctly propagate all the test cases that correspond to the changes in the component structure.

Finally, we apply Size, complexity, and Performance metrics to compare the two adaptive software systems that were used in our study.

5.1. Propagating Reductive Changes Simulation

Using the Eclipse Modeling Framework (EMF) allowed us to load, change and save existing models by using Kermeta [100]. Hence we used Kermeta to simulate a reductive change in the EMR service related to ClinicalTrial, Treatment, PatientHistory, PatientInfo, and finally EMRService features. This was achieved by creating and applying a transformation to the EMR component model that removed the {ClinicalTrial, Treatment, PatientHistory, PatientInfo, EMRService} classes, and its associated dependency relationships. Our change propagation engine then generated a set of transformative actions for synchronizing the test model with the adapted component model.

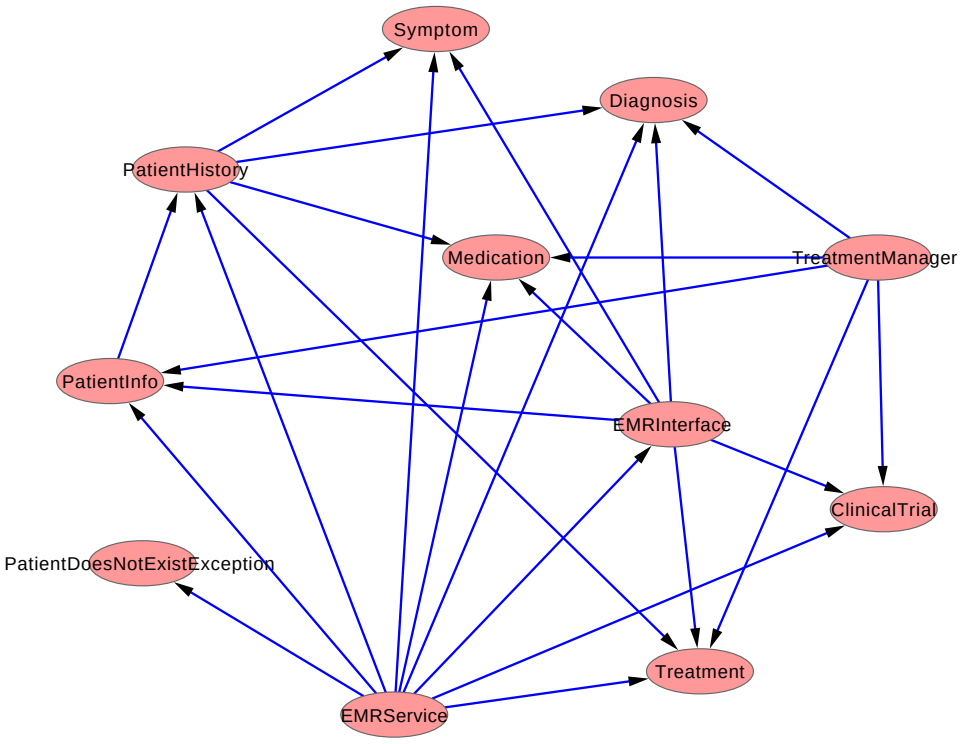

Figure 17. EMR component model dependencies call graph

After loading the dynamic instance of EMR component model, and before loading the dynamic instance of EMR test model, based on the component dependency relationships, our engine highlights in general a set of changes that must be propagated to correspondent test model.

Figure 17 shows EMR component model dependencies call graph. Since Classycle's Analyzer is helpful for finding cyclic dependencies between classes and packages, as a first step of change propagation, our engine only could propagate the integration tests that need to be updated or removed based on the reductive target component. Here the propagation engine skipped all unit tests, and detailed information about the integration test such as test case name. For example after removing PatientHistory component our engine identifies that this component is a caller of Symptom, Medication, Diagnosis, and Treatment, and is a callee for PatientInfo and EMRService components as shown in Figure 17.

As general transformative actions the engine propagates two changes to the test model: (1) removing the integration test with all callers (Symptom, Medication, Diagnosis, and Treatment), (2) updating the integration test with all callees (PatientInfo and EMRService).

Reductive target Component	Caller	Artifact type	Actual Propagation	Necessary Propagation	Callee	Transformative Action	Statement
PatientInfo	1	integration test	4	4	1	Update	4
EMRService		unit test	1	1		Remove	
		3* integration test	1	1	6		
		4* integration test	1	1	2		
		1* integration test	1	1	1		
PatientHistory	1	integration test	3	3	1	Update	3
ClinicalTrial	1	integration test	4	4	1	Update	4
Treatment	1	integration test	3	3	1	Update	3

Table 3. Detailed test model updates for reductive change simulation

Propagating detailed changes to the test model, required loading the EMR dynamic instance test model that described in subsection 4.9. Utilizing naming conventions strategy to handle the traceability relationship within the artifacts that make up the system allows the change propagation engine to automatically lookup of specific test-related entries within an artifact, and across multiple artifacts. To handle the consistency among the involved models (i.e., component and test models) our synchronization approach is based on traceability links between the these interrelated models.

Once the propagation engine identified all Unit-level test cases that associated with the component targeted in the reductive change, remove transformative action will be generated to remove these test cases from the test model without many considerations. This is because unit tests validate the behavior of a component in isolation, and are therefore independent of other components and tests.

If the propagation engine detects any callee of the adaptation target, remove transformative action will be generated to remove the integration tests that validate the behavior of the target with its callees (i.e. components that are invoked by the adaptation target) from the test model. Since the adaptation target will be removed, tests that validate it with its dependents will not affect other parts of the test model. The engine assumes a software design in which there are no cyclic dependencies. As long as the target component would have a great impact on the behavior of its caller components, the engine will generate update transformative action to update the integration tests of caller components after the adaptation target is removed. Table 3 summarizes the actions that were generated by our change propagation simulation. We have simulated five reductive changes.

The component in Reductive target Component column is a component targeted in the reductive change, Caller column points to the caller component for targeted component, Artifact Type represents different type of test cases, Actual Propagation reveals the total number of suggested changes, Necessary Propagation column shows the set of changes that needed to be propagated, Callee column specifies all callee components for each test case, Transformative action shows all actions that performed by our propagation engine to update the test model, the last column Statement represents the total number of the exact statement in the source code where the developers have to pay attention on to update the integration test after a callee component is removed. This column is added to refine the update procedure in column transformative action, and provide more detailed information to the developer after reductive change takes place. Our engine is able to identify the exact statement which needs to be modified in each test case instead of just generating update action. The number of statements that need to be focus on for updating the integration test are varies from the number of integration test itself, after removing a callee component, the engine could highlight 4 statements as a total to be modified for 2 integration test, because it depends on how many calls occurs in each integration test.

```
String type="GENERAL_CHECKUP";
String description="you should use Morphine to treat moderate to severe pain";
String notes="";
Treatment treat=new Treatment(type, date, description, notes);

symptomlist.add(sym);
medicationlist.add(medication);
diagnosislist.add(diag);
treatmentlist.add(treat);
```

Figure 18. Major statements for updating the test model after reductive changes

Figure 18 shows part of the source code of the integration test (TestAddMedication()) that needs to be updated after removing Treatment component. the engine as shown in Table 3 highlighted three statements where the developer should focus on to update the test model, the highlighted statement in Figure 18 is one of three that are located in three test cases.

Our engine propagates set of changes to the test model after simulating reductive change to several EMR component model. As shown in Table 3 after removing PatientInfo component, the engine doesn't catch any Unit-level test cases associated with PatientInfo, removal of PatientInfo will have a great impact on the behavior of its caller components, thereby our engine asked for updating all tests that validate the behavior of PatientInfo with EMRservice component. Finally our engine hasn't detected any callee of PatientInfo component.

5.2. Propagating Additive Changes Simulation

Prior to simulating additive changes to EMR service we had to maintain the detailed dependency relationships information for all EMR components, and then build the component model to be used for additive simulation. We used Kermeta to simulate additive changes in the EMR service related to Medication, Diagnosis, Symptom and finally EMRService features. This was achieved by creating and applying a transformation to the EMR component model that add the { Medication, Diagnosis, Symptom and EMRService} classes respectively, and its associated dependency relationships.

Our change propagation engine then generated a set of transformative actions for synchronizing the test model with the adapted component model.

Additive target Component	Artifact type	Actual Propagation	Necessary Propagation	Callee	Transformative Action	Calls
Diagnosis	Unit test	7	7		Add	
EMRService	unit test	3	5		Add	
	1* Integration test	1	1	1		1
	4* Integration test	1	1	2		2
	1* Integration test	1	1	2		4
Medication	Unit test	4	4		Add	
Symptom	Unit test	4	4		Add	

Table 4. Detailed test model updates for additive change simulation

Table 4 summarizes the actions that were generated by our change propagation simulation. We have simulated four additive changes.

The component in Additive target Component column is a component targeted in the Additive change, Artifact Type represents different type of test cases, Actual propagation reveals the total number of suggested changes, Necessary Propagation column shows the set of changes that needed to be propagated, Callee column specifies all callee components for each test case, Transformative action shows all actions that performed by our propagation engine to update the test model, the last column calls represents the total number of the calls occurs in each test case to execute and pass that test case.

Our engine propagates set of changes to the test model after simulating additive change to several EMR component model. As shown in table 4 after adding Medication component, the engine doesn't catch any Integration-level test associated with Medication where Medication is a caller, the only transformative action were generated by the engine were about adding 4 unit level test associated with Medication.

After Adding the EMRservice the engine identifies 3 Unit-level test out of 5 need to be added, while identifies 6 Integration-level test associated with EMRservice need to be added out of 6 where the EMRservice is a caller.

5.3. Evaluation of Generalization

In our previous paper [2], in order to demonstrate and evaluate the proposed approach we implemented a small autonomic system with runtime testing capabilities for evaluation purposes. To provide a realistic context for the prototype, we developed the application based on a healthcare scenario in which self-adaptation and self-testing could be practically useful. The initial version of the prototype is focused on assessing the feasibility of automatically propagating reductive changes; in this work we extended the prototype to include the additive change as well.

To demonstrate the generalization of our TIP approach into the domain of runtime testing for self-adaptive software systems, the experiment should be performed on other self-adaptive system (i.e., different application domain), our research investigation resulted that there has been general lack of freely available real-world autonomic systems for evaluating self-testing approaches. Most products are commercial (i.e., closed source).

The three prototypes we investigated were:

- Carlos et al. [23] Dynamic Plans for Integration Testing of Self-adaptive Software Systems, for approach evaluation purposes, they have developed a prototype application that has been used to conduct some experiments, and to demonstrate the feasibility of their approach. They have applied their prototype to a case study, the case study used is a simple web shop application, which can be employed to sell goods on the Internet. The software architecture of this application involves five component types.

- Steven et al. [95] A Self-Testing Autonomic Container, which can be defined as a data structure, called as stack which has ability to reconfigure itself at runtime. When the stack reaches its full capacity that is at 80%, it will reconfigure itself by increasing the capacity. And to validate the newly configured stack, they applied the approach of Replication with Validation, that is validation will be done on the copy of stack.

- Ramirez et al. [88] A Self-Testing Autonomic Job Scheduler also applies the same concept of Autonomic Container but in more realistic way, which will hold the a collection of job request and a pool of software agents for handling request. However the prototype was not very realistic and it was just a simulation.

During our search for sample self-adaptive system we forced the following obstacles: (1) We need access to the source code, as to verify the presence of a considerable test suite and next to apply naming convention strategy to handle the traceability relationship, profile and trace the junit tests execution, and finally trace the internal component interaction to build the component model that handles the component dependency relationship; (2) The implementations we made are currently targeted towards self-adaptive systems developed in Java and dynamic language groovy; and (3) The approach we used to build an self-adaptive software system is based on Spring framework which provides a core application container that allows us to specify components (called beans) using XML configurations, we used groovy to write these Beans. We used the Spring Framework to provide a component-based application container for the three major application services. These services were: EMRServices, AppointServices, and PharmacyServices. Services were made adaptable using the dynamic language Groovy, which allows components to be specified as beans within the application container. At runtime, the container was set to monitor the Groovy beans for source code changes, and automatically reload them to use the new implementations.

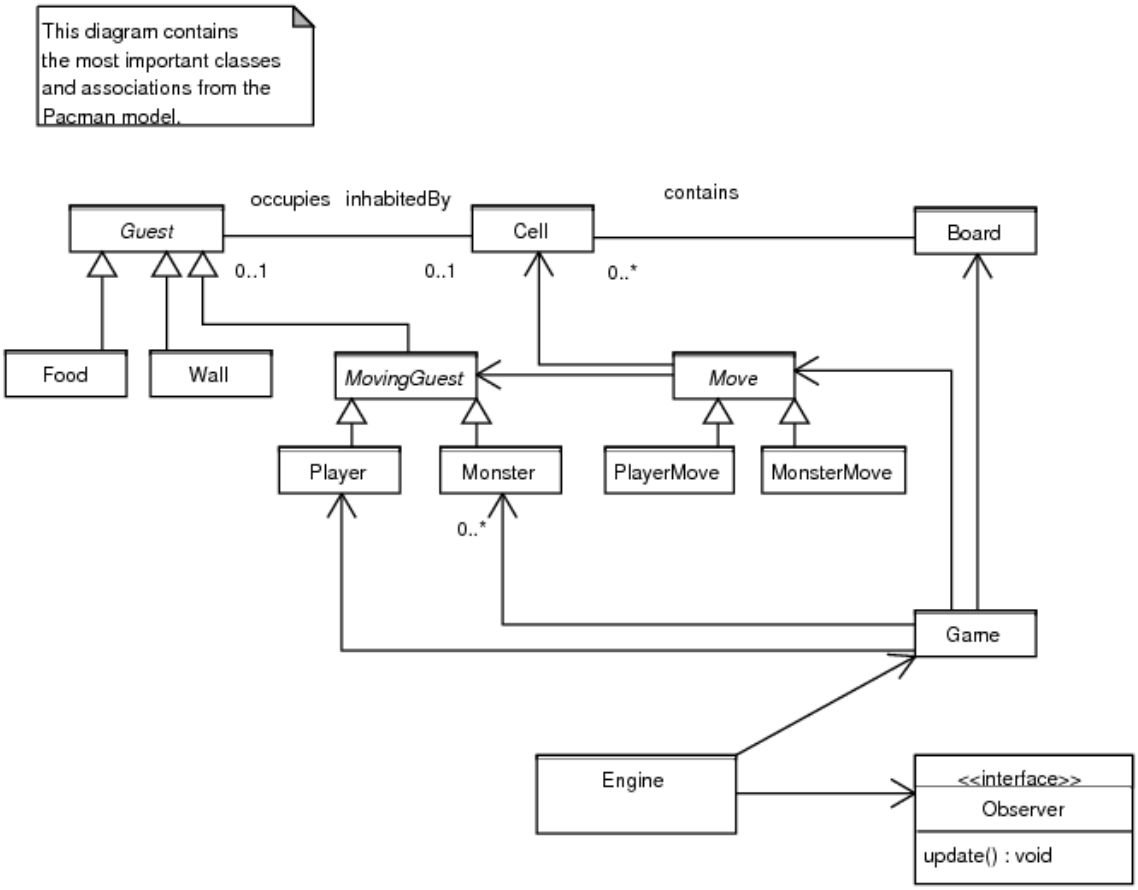

Figure 19. Jpacman partial class diagram

The Adaptation subsystem included a manager that was responsible for updating the component source (.groovy files) at runtime, and keeping a backup of the old one.

5.3.1. Sample Self-Adaptive System

we searched in Sourceforge for systems with a considerable JUnit test suite, to re-implement it and get a self-adaptive version with runtime testing capabilities by using the approach proposed by King et al. [68]. We successfully found JPacman 3.0 java application which used for educational purposes.

The JPacman 3.0 system is a teaching example at the TU Delft used during a course about software testing. Its implementation is an example of best practice Java, JUnit, design-by-contract, etc.

It has been developed using a test-intensive XP-style process, featuring *unit* and *integration* tests achieving a high level of test coverage, Its test suite is based on JUnit 4, but is compatible with older versions of JUnit. The source repository consists of 22 classes and 16 test cases, totaling 2.3 kSLOC, Figure 19 presents the most important classes and association dependencies from the pacman model, not all classes, methods, and attributes have been added.

By using the same tools, languages, and frameworks for building an autonomic and adaptive java healthcare application that described in Chapter 4, we build a self-adaptive jpacman release with runtime testing capabilities.

5.4. Manual Developer Evaluation Oracle

To measure the performance of the proposed TIP approach, we compare the retrieved propagated changes with a manual developer oracle, i.e., we consider and compare the work of a developer who manually identified changes that should be performed to update the test model after self-adaptation occurs as the objective baseline for this experiment. This duty of identifying changes is part of a short questionnaire we generated and gave to the developer as shown in Figure 20.

In the main questions of the questionnaire (question 4 and 5), we asked the developers to identify all associated unit and integration tests for component class in the given list. The list of component classes is randomly selected. We first ask the developer to go through additive part, and then the reductive part for both adaptive systems (Healthcare and jpacman).

5.5. Evaluation Criteria

As shown in Table 4 the following example shows simple scenario occurs after runtime self-adaptation by adding component EMRService.

1. What is your favorite editor/IDE you use to develop software system? Have you used testing tools (testing framework, browsing, coverage, etc.)?

2. How are you familiar with the (unit) and (integration) testing of the system?
   Do you have any experience in unit, integration testing and testing frameworks such as JUnit?

3. How are you involved in the self-adaptive software system?
   How much experience do you have with dynamic language such as the groovy language?

4. Additive part, please provide for each component class in the given list, the associated unit and integration tests.
   Based on the source code for each component class identify,
- First, all Unit tests need to be added to the test suite.
- Second, all Integration tests need to be added to the test suite, where the component class is a caller, and all its callee.
- Third, all Integration tests need to be added to the test suite, where the component class is a callee, and its entire caller.

5. Reductive part, please provide for each component class in the given list, the associated unit and integration tests.
   Based on the test suite identify,
- First, all Unit tests need to be removed from the test suite if the component class is removed.
- Second, all Integration tests need to be removed from the test suite, where the component class is a caller, and all its callee.

Figure 20. Developer questionnaire

After component EMRService is added, our change engine propagates the additive change and generates the following transformative action to be performed: First, six integration tests associated with EMRService component class should be added, our engine detects that for all six tests the EMRSercive component is a caller. Second, three unit tests associated with the EMRService component should be added to the test model without many considerations. After comparing the changes generated and propagated by our engine and the manual developer evaluation, we found that the engine missed two unit test should be added after adding EMRService component.

To measure the performance of propagation engine two main concepts of Information Retrieval have been used: Recall and Precision. Where *RECALL* is the ratio of the number of relevant propagated changes to the total number of relevant changes should perform on test model, or

$$RECALL = \frac{|relevantUT\&IT \cap propagatedUT\&IT|}{relevantUT\&IT}$$

PRECISION is the ratio of the number of relevant propagated changes to the total number of irrelevant and relevant changes propagated, or

$$PRECISION = \frac{|relevantUT\&IT \cap propagatedUT\&IT|}{propagatedUT\&IT}$$

The total set of propagated updates will be called the Propagated set; Propagated = 3 unit and 6 integration tests. The set of updates that required to be propagated will be called the Demanded set; Demanded = was 5 unit and 6 integration tests. These sets don't contain that component has been changed (EMRService). We define the number of elements in propagated as P (P =9), and the number of elements in Demanded as D (D = 11). The number of elements in the intersection of Propagated and Demanded as PD (PD = 9). Based on these definitions, we define: *Recall* = *PD/D*, *Precision* = *PD/P*. In the above scenario, *Recall* = 9/11 = 82% and *Precision* = 9/9 = 100%.

The following example shows simple scenario occurs after runtime self-adaptation by removing component ClinicalTrial, as shown in Table 3 after component ClinicalTrial is removed, our change engine propagates 4 integration test need to be updated, since the engine detects that the targeted component is a callee for other components. After comparing the changes generated and propagated by our engine and the manual developer evaluation for ClinicalTrial, we found that the engine propagated all required changes that needed to be performed to get up-to-date test model.

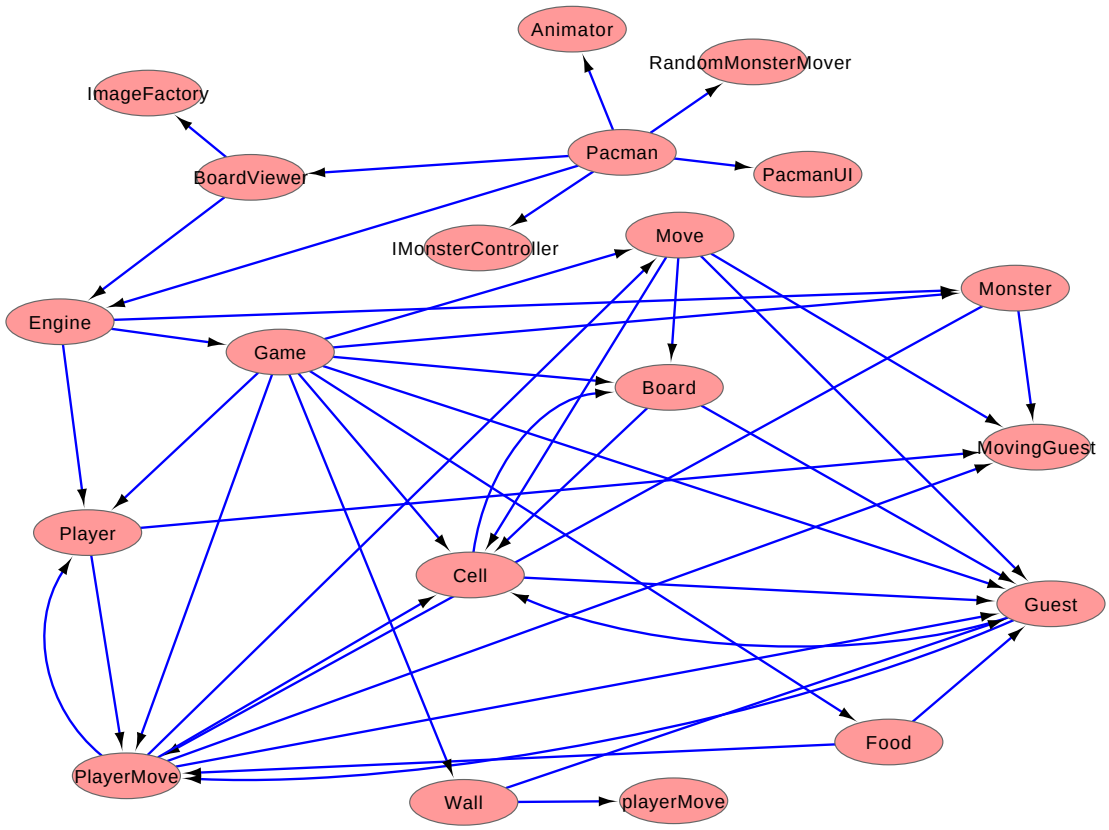

Figure 21. Jpacman component model dependencies call graph

In the above scenario, p=4, D=4, PD=4, *Recall* = 4/4 = 100% and *Precision* = 4/4 = 100%.

5.6. Jpacman Additive and Reductive Changes Simulation

In order to propagating detailed changes to the test model of the Jpacman system, we had to build the dynamic instance model of both component and test model. We have generated the component and test model of adaptive jpacman in the same way we had did for EMR.

After loading the dynamic instance of Jpacman component model, based on the component dependency relationships, our engine highlights in general a set of changes that must be propagated to correspondent test model.

Figure 21 shows Jpacman component model dependencies call graph. For example after removing Engine component our engine identifies that this component is a caller of Player, Game, Monster, and is a callee for Pacman and BoardViewer components as shown in Figure 21. As general transformative actions the engine propagates two changes to test model: (1) removing the integration test with all callers *(Player, Game, Monster)*, (2) updating the integration test with all callees *(Pacman and BoardViewer)*.

Reductive target Component	Caller	Artifact type	Actual Propagation	Necessary Propagation	Callee	Transformative Action	Statement
Board		Unit test	1	1		Remove	
		2*Integration test	1	1	1		2
		Integration test	1	1	2		4
	1	Integration test	1	1	1	Update	1
Cell		Integration test	1	1	1	Remove	1
	1	Integration test	1	1	1	Update	2
	1	Integration test	1	1	1		1
Engine	1	integration test	1	1	1	Update	5
Game		Unit test	3	3		Remove	
Guest		integration test	1	1	1	Remove	1
	1	integration test	1	1	1	Update	2
ImageFactory		Unit test	2	2		Remove	
Move	1	Integration test	3	3	1	Update	3
Observer		Integration test	1	1	1	Remove	5
Pacman		Unit test	1	1		Remove	

Table 5. Jpacman detailed test model updates for reductive change simulation

The dynamic test model is conformed the meta-model that we have described earlier in Section 4.3, after loading the Jpacman dynamic instance test model, our propagation engine generated a set of actions for synchronizing the test model with the adapted component model. Table 5 summarizes the actions that were generated by our change propagation simulation. We have simulated Nine reductive changes.

Our engine propagates set of changes to the test model after simulating reductive change to several Jpacman component model. After removing Cell component, the engine doesn't catch any Unit-level test cases associated with Cell, removal of Cell will have a great impact on the behavior of its caller components, thereby our engine asked for updating all tests that validate the behavior of Cell with its caller component. Finally our engine detects one integration test of Cell component with one callee needs to be removed.

In order to simulat additive changes to Jpacman we had to maintain the detailed dependency relationships information for all EMR components, and then build the component model to be used for additive simulation. We used Kermeta to simulate additive changes in the Jpacman, this was achieved by creating and applying a transformation to the Jpacman component model that add the { Board, BoarViewer, Food, ImageFactory, Monster, Move, MovePlayer, Pacman, Player, Wall} classes respectively, and its associated dependency relationships. Our change propagation engine then generated a set of transformative actions for synchronizing the test model with the adapted component model.

Table 6 summarizes the actions that were generated by our change propagation simulation. We have simulated Ten additive changes. Our engine propagates set of changes to the test model after simulating additive change to several Jpacman component model. As shown in table 6 after Adding Move component, the engine identifies six Unit-level tests associated with Move out of 8 unit tests, and four Integration-level tests need to be added to the test model. The first integration test has two callees and four calls occurs in there, two integration test has the same number of callees and calls which is one, and the last integration test has two callees and two calls occurs in there.

Additive target Component	Artifact type	Actual Propagation	Necessary Propagation	Callee	Transformative Action	Calls
Board	Unit test	4	5		Add	
	Integration test	1	1	1		2
	Integration test	1	1	4		4
	Integration test	1	2	1		1
BoardViewer	unit test	6	7		Add	
	2* Integration test	1	1	1		2
	Integration test	1	1	2		5
	2* Integration test	1	1	1		1
Food	Unit test	3	3		Add	
	3*Integration test	2	3	1		1
ImageFactory	Unit test	6	7		Add	
Monster	Unit test	1	1		Add	
	2*Integration test	1	1	1		1
Move	Unit test	6	8		Add	
	Integration test	1	1	2		4
	2*Integration test	1	1	1		1
	Integration test	1	1	2		2
PlayerMove	Unit test	3	3		Add	
	2*Integration test	1	1	1		2
	Integration test	1	1	2		3
	Integration test	1	1	1		1
Pacman	Unit test	1	1		Add	
	4*Integration test	1	1	1		1
	Integration test	1	1	2		3
	Integration test	1	1	1		2
	Integration test	1	1	2		5
	2* Integration test	1	1	3		3
Player	Unit test	8	8		Add	
	Integration test	1	2	1		1
Wall	Unit test	2	2		Add	
	Integration test	1	1	1		1

Table 6. Jpacman detailed test model updates for additive change simulation

CHAPTER 6. RESULTS AND DISCUSSION

This section presents the results of the survey that has given to the developer, report on the accuracy of the proposed TIP approach. For both self-adaptive systems in our study, the developer completed the questionnaire, and the result used to compare and evaluate the performance of our work.

Q1.An IDE (Eclipse or NetBeans) is used in the two systems. JUnit is a unit testing framework for the Java programming language is commonly used for testing, For code coverage Cobertura was used to focus on lines and branches while EclEmma to concentrate on bytecode instructions and get line metrics.

Q2.The developer is working in testing field for more than 5 years, experience gained with JUnit has been important in the development of test-driven development. As long as his projects based on different languages he used to get to know some of the family members of unit testing frameworks which referred to collectively as xUnit, such as, PHP (PHPUnit), C# (NUnit), Python (PyUnit), Fortran (fUnit), Perl (Test::Class and Test::Unit) and C++ (CPPUnit).

Q3.With this question, we intended to make sure the developer has at least little bit of the experience in self-adaptive system. In general, the developer did not seem to have much problem understanding the code and test suite, since groovy language builds upon the strengths of Java but has additional power features inspired by languages like Python, Ruby and Smalltalk, seamlessly integrates with all existing Java classes and libraries.

Q4.Q5.The developer gave the impression that identifying the unit and integration test that need to be removed or add based on a specific component class was not hectic task. To get the detail about some of reductive and additive developer manual evaluations please refers to Appendix A and B.

	Additive		Reductive	
Precision	**Unit**	**Integration**	**Unit**	**Integration**
EMRService	100%	100%	100%	100%
Jpacman	100%	100%	100%	100%

	Additive		Reductive	
Recall	**Unit**	**Integration**	**Unit**	**Integration**
EMRService	90%	100%	100%	100%
Jpacman	81%	80%	100%	100%

Table 7. Mean precision and mean recall of the TIP approach

We calculated the recall and precision for each component was targeted in the reductive and additive simulation, then we calculate the mean recall and precision for the unit and integration test in both the reductive and additive change simulations. Table 7 shows how TIP change propagation engine provides the highest accurate precision measure on both Jpacman and EMR Service in both Reductive and Additive simulations, 100% on Jpacman for both unit and integration tests, and 100% on EMRService for both unit and integration tests. The result based on recall measure is divided into several areas, for EMR Service Additive simulation the engine provides high recall for integration tests 100% and a good recall for unit tests 90%, while for Jpacman Additive simulation the engine provides a good recall for both unit and integration tests 81% and 80% respectively. Back to the Reductive simulation the engine provides high accuraacy on both Jpacman and EMR Service, 100% on Jpacman for both unit and integration tests, and 100% on EMRService for both unit and integration tests. From Table 7, we observe how TIP engine achieves high accuracy in synchronizing component structure model with its test model. 100% in recall means the engine propagate all relevant changes that needed to be propagated to updat the run time test model, and 100% in precision means the engine propagates only relevant changes (i.e. none of the propagated changes are irrelevant) to updat the run time test model.

Size metric of the two software artifacts produced during implementation were generated using the Eclipse Metrics vl.3.6 [14] plugin. The data was automatically exported from Metrics to XML format as an estimate of development overhead. In total, the Jpacman system consists of 2.3 kSLOC and HealthCare system consist of 1.2 kSLOC. The time spent to trace the execution of all test cases was acquired via the Eclipse Test and Performance Tools Platform (TPTP) [97], TPTP spent 1.11 second to trace all the execution of the test cases for the HealthCare application and 2.95 seconds for JPacman system. In addition to the recall and precision performance measurement of our TIP engine we computed the elapsed time, in seconds, taken to propagate the necessary changes and generate the transformative actions to update the run time test model. After simulating the additive and reductive changes in both systems in our study, we calculated the mean time that required by our engine to propagate the changes, the performance results showed that the TIP engine for the Jpacman system took 5 seconds in propagating the additive changes and 7 seconds in propagating reductive changes, while took 4 seconds in propagating the additive changes and 5 seconds in propagating reductive changes for HealthCare application. A windows-based Intel Core i7 6GHz PC with 8GB RAM was used to collect the development and performance metrics.

Developing the prototype provided us with much insight into the complexity of implementing an automated solution to the research problem. Although our reductive and additive examples of the EMR service and Jpacman were not a very complex scenarios available, it allowed us to identify enough meta-data to achieve checking-level propagation. We discovered that even for the propagation engine to be able to identify general points of change in the test model, we had to maintain highly detailed information on both the adaptable components and their associated tests.

This information included a list of the components test cases, along with the file-names, locations, and access information for the: (1) test scripts that contain the tests, (2) test drivers that make calls to the tests, and (3) test stubs and/or data files used by the tests. Our simulation results and experience also revealed the central role of the meta-model in enabling change propagation.

Using naming convention resolution strategy to handle the traceability relationship represents a valuable support for the developer during the identification of relationships between unit tests and component class under test. One of the lessons learned in our work was the importance of utilizing naming conventions within the artifacts that make up the system. The purpose of the naming conventions was to allow the automatic lookup of specific test-related entries within an artifact, and across multiple artifacts. Conventions included the use of unique identifiers for all components and test cases, and the reuse of component IDs within test IDs for traceability.

6.1. Threats to Validity

In this section we describe the threats to validity that could affect our results. Regarding construct validity, we have contacted three developers to identify manually all tests that need to be added or removed based on target component class for both self-adaptive systems. We got a response from two developers. Inexperience, lack of motivation or human mistake factors could impact the correctness of the manual evaluation. Therefore we counter the experience argument by the selection of developer. The developers that we have chosen are a project leader with more than 5 years experience in the testing field, and the second developer is still junior with no experience in adaptive system. The developer was motivated to do his task so we got a quick response from his side.

Recall and Precision are widely used metrics for assessing relevant retrieval approaches and traceability recovery techniques [6, 73, 50, 106].

Thereby we used these metrics to measure the accuracy of our engine to retrieve the only associated tests and dependencies that need to be updated or generated to get consistent component and test model after self-adaptation occurs.

Although if naming convention strategy seems to be the most accurate strategy for establishing traceability link between unit tests and component under test [90], the relationships between tests and component class under test are not always one-to-one. Therefore our approach used naming convention strategy to handle the traceability relationship but does not rely completely on this strategy. The TIP approach was able to identify all classes that might be targeted in each test, by tracing all interactions among system components and tests cases.

Finally, Regarding the generality of the approach and result (external validity), an important threat is related to the case studies used in the evaluation. To reduce such a threat we use open source system jpacman, and a real service oriented healthcare application. Note that Jpacman has been previously used to evaluate the accuracy of traceability recovery approaches based on naming conventions strategy [90, 87].

CHAPTER 7. CONCLUSIONS AND FUTURE WORK

In this dissertation we investigated in the importance of having *First*, Runtime validation and verification (V&V) in self-adaptive software. Since self-adaptation modifies the structure and behavior of the system, runtime V&V is necessary to ensure that errors are not introduced as a result of the adaptation process; *Second*, Up-to-date test models. If self-adaptation introduces a new component, new integration test cases should be generated to validate its interactions with existing components. Similarly, if an existing component is removed, some test cases may no longer be applicable or adequate for testing, due to changes in program structure. Such test cases would therefore have to be updated or pruned from the runtime test model. As a result, the runtime test model for the system will be made consistent with its new structure after dynamic adaptation.

To ensure that runtime testing of autonomic software can be applied in practice, it was necessary to investigate techniques for automatically updating runtime test models after self-adaptation occurs. Using multi-shot transformation approaches such as change propagation, an emerging field of MDE could be effective for synchronizing different software models at runtime.

As a preliminary step in our investigation, we performed a systematic literature review to determine the current landscape surrounding the research problem. To properly focus the review, we formulated the high-level research question, and then we expanded it into a series of questions. The main motivations behind our research questions were: (1) Identify works related to the idea of synchronizing test models at runtime in adaptive software; (2) Assess the usefulness of approaches in the literature for synchronizing runtime models without having to completely re-construct the target model; and (3) Assess the practicality of developing a prototype of a solution to our specific problem using the approaches from 2.

Conducting the systematic review led us to several articles on the use of models at runtime, as well as current research directions in the area of MDE. The works on models at runtime included papers that harness executable models for dynamic adaptation and software testing, as separate issues. No works that specifically address the problem of automatically synchronizing runtime test models in adaptive software were found during the literature search. The results of the systematic literature review indicated that the proposed research direction may lead to advances in two relatively open fields of software engineering research: (1) runtime testing of autonomic and adaptive systems, and (2) change propagation.

In this dissertation we have developed, described, and evaluated a model-driven approach that is based on change propagation for synchronizing component models and runtime test models in autonomic software after dynamic adaptation. To investigate practical issues surrounding the research problem, a prototype of the approach was developed and used to demonstrate its feasibility. Traceability relationships were handled through the naming conventions strategy. The approach is referred to as Test Information Propagation (TIP). We extended the TIP prototype in [2] by including the additive change propagation capability. The TIP was able to identify and then propagate the new component detailed information such as interfaces, implementations, callers, and callees into the test model after additive changes occur at runtime. The prototype simulates reductive and additive changes to an autonomic, service-oriented healthcare application. We elaborated on the transformative action update to provide more detailed information and to focus the updating process.

Designing and building a self-adaptive Health care application allowed us to identify major software components, and to create appropriate test cases. Testing tools were applied to assess coverage of the test suite.

To demonstrate the generalization of our TIP approach into the domain of runtime testing for self-adaptive software systems, the experiment was performed an another self-adaptive system in a different application domain. To measure the performance of the propagation engine two main concepts of information retrieval were used: Recall and Precision. We compared the retrieved propagated changes with a manual developer oracle, i.e., we consider and compare the work of a developer who manually identified changes that should be performed to update the test model after self-adaptation occurs as the objective baseline for this experiment. This duty of identifying changes is part of a short questionnaire we generated and gave to the developer. The developer has worked in the testing field for more than 5 years, including experience gained with JUnit which is important in the development of test-driven development. To the best of our knowledge, the approach presented in this dissertation is the first attempt addressing the research problem under investigation. The experiments performed demonstrated that our approach is able to propagate reductive and additive changes with high accuracy.

As future work, we intend to fully improve the TIP approach in which the suggested new test cases by our engine are dynamically generated by exploring some of the approaches for automatically generating test cases from source code. Pacheco et al. [84] presented an automatic unit test generator for Java called Randoop. It automatically creates unit tests for java classes, in JUnit format. Randoop generates unit tests using feedback-directed random test generation. Their technique generates sequences of methods and constructor invocations for the classes under test, and uses the sequences to create tests.Randoop tool can do the following: 1) Generate unit tests; 2) Captures behavior of existing code; 3) Produce random test data; and 4) Automatically execute tests. Whitney [104] presented a tool for automatic JUnit test creation as a part of his thesis. His tool provides a GUI that allow tester to input test values and specify expected results. The GUI allows developers to work at the problem domain level of abstraction.

This will reduce cognitive load since developers will no longer have to worry about coding the test harness at the same time they are deriving the test. We also intend to extend the TIP prototype to include the mutation changes, so that our prototype will be able to propagate all types of changes (i.e., add new components, remove exist components, or modify existing ones). More case studies will be used to evaluate the accuracy of TIP approach.

Another beneficiary of TIP is merging the cloud computing infrastructures to improve the healthcare industry. As a part of the future work, the prototype can be served as a concrete real world problem in the healthcare domain. Each healthcare provider can make use of services available in a cloud environment. To ensure feasibility of the application in the cloud environment, the approach used by King et al. [66] can be used to perform adequate testing. In their approach, testing would be performed on copy of the service to avoid interruption, (i.e., Replication with validation), due to high availability and requirements of the service. To achieve the replication with validation and test environment the design is incorporated in virtual environment. In this test environment each service provider host Test Support as a Service [66], which would help in accessing the test model of the different service provider and also helps in updating the test model if needed.

In this dissertation we have evaluated how built-in regression tests for autonomic systems can be updated after dynamic changes have been made to the software systems. Regression testing is an expensive maintenance process directed at validating modified software. Regression test selection techniques attempt to reduce the cost of regression testing by selecting tests from a programs existing test suite [46]. Several regression test selection techniques have been depicted in the testing literature, (e.g., Minimization, Safe, Dataflow-Coverage-Based, Ad Hoc/Random, Retest-All techniques). As future work, we intend to conduct an experiment to examine the relative costs and benefits of some of these regression test selection techniques and our technique for updating regression test.

In the experiment we would concentrate on the relative abilities of the examined techniques and our in reducing regression testing effort and detect faults in modified programs.

One of the important means of assuring the validation of the dynamic adaptive system is by obtaining the trace of the inter-component interactions [78]. Interactions are the most commonly used tools to model behavioral aspects of the system [57]. In this dissertation we used TPTP profiler to trace all components interaction and test cases execution for validating dynamic adaptive system by analyzing execution traces. As future work, we intend to build a general profiler that collects all the traces of execution at the runtime and makes it possible to successfully trace the interactions among components with minimal overheads and gets the output in the form of a trace file.

Software metrics are quantitative measures of some properties of a part of software. Coupling is defined as the degree to which each program module depends on the other modules. Low coupling is desired among the modules of an object-oriented application and it is a sign for a good design. High coupling may lower the understandability and the maintainability of a software system [60]. For testing, four unordered types are needed, These four coupling types were used to define coupling-based testing criteria for integration testing. The four types are defined between pairs of units (A and B) as follows [60]. (1) Call coupling refers to calls between units (unit A calls unit B or unit B calls unit A) and there are no parameters, common variable references, or common references to external media between the two units. In this dissertation we focus on this type of coupling to determine components dependencies, as future work we intend to consider the following remaining three types in capturing components dependencies, (2) Parameter coupling refers to all parameter passing, (3) Shared data coupling refers to procedures that both refer to the same data objects, (4) External device coupling refers to procedures that both access the same external medium.

REFERENCES

[1] *IEEE standard glossary of software engineering terminology*, December 1990.

[2] Mohammed Akour, Akanksha Jaidev, and Tariq M. King, *Towards change propagating test models in autonomic and adaptive systems*, Proceedings of the 2011 18th IEEE International Conference and Workshops on Engineering of Computer-Based Systems (Washington, DC, USA), ECBS '11, IEEE Computer Society, 2011, pp. 89–96.

[3] Mohammed Akour, Gursimran Walia, and Tariq M. King, *A systematic review of runtime test case synchronization in adaptive software*, Tech. report, NDSU Dept. of Computer Science, May 2010.

[4] Scott W. Ambler, *The object primer: Agile model-driven development with uml 2.0*, Cambridge University Press, New York, NY, USA, 2004.

[5] Jesper Andersson, Rogerio de Lemos, Sam Malek, and Danny Weyns, *Reflecting on self-adaptive software systems*, Proceedings of the 2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (Washington, DC, USA), IEEE Computer Society, 2009, pp. 38–47.

[6] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo, *Recovering traceability links between code and documentation*, vol. 28, IEEE Press, October 2002, pp. 970–983.

[7] Colin Atkinson and Thomas Kühne, *Model-driven development: A metamodeling foundation*, vol. 20, IEEE Computer Society Press, September 2003, pp. 36–41.

[8] F. Bancilhon and N. Spyratos, *Update semantics of relational views*, vol. 6, 1981, pp. 557–575.

[9] Boris Beizer, *Software testing techniques*, second ed., Van Nostrand Reinhold, New York, 1990.

[10] Sami Beydeda, *Self-metamorphic-testing components*, Proceedings of the 30th Annual International Computer Software and Applications Conference - Volume 02 (Washington, DC, USA), COMPSAC '06, IEEE Computer Society, 2006, pp. 265–272.

[11] Sami Beydeda and Volker Gruhn, *Merging components and testing tools: The self-testing cots components (stecc) strategy*, In EUROMICRO Conference Component-based Software Engineering Track. IEEE Computer, Society Press, 2003, pp. 107–114.

[12] M. Blum, M. Luby, and R. Rubinfeld, *Self-testing/correcting with applications to numerical problems*, vol. 45, 1993, p. 549?595.

[13] Barry W. Boehm, *Software engineering economics*, Prentice Hall, Englewood Cliffs, NJ, 1981.

[14] Guillaume Boissier, *Metrics 1.3.6*, 2005, http://metrics.sourceforge.net/ (July 2010).

[15] Ryan Breidenbach and Craig Walls, *Spring in action*, Manning Publications Co., Greenwich, CT, USA, 2007.

[16] Bernd Bruegge and Allen H. Dutoit, *Object-oriented software engineering: Using uml, patterns and java, second edition*, Prentice Hall, September 2003.

[17] Ilene Burnstein, *Practical software testing: A process-oriented approach*, Springer-Verlag, New York, NY, USA, 2003.

[18] Marsha Chechik, Winnie Lai, Shiva Nejati, Jordi Cabot, Zinovy Diskin, Steve Easterbrook, Mehrdad Sabetzadeh, and Rick Salay, *Relationship-based change propagation: A case study*, MISE '09 (Washington, DC, USA), IEEE Computer Society, 2009, pp. 7–12.

[19] Pavan Kumar Chittimalli and Mary Jean Harrold, *Regression test selection on system requirements*, Proceedings of the 1st India software engineering conference (New York, NY, USA), ISEC '08, ACM, 2008, pp. 87–96.

[20] Mike Clark, *JUnitPerf 1.9*, 2005, http//www.clarkware.com/software/JUnitPerf.html (July 2010).

[21] Paul Clements and Linda Northrop, *Software product lines: Practices and patterns*, third ed., Addison-Wesley Professional, Boston, MA, USA, 2001.

[22] Andrew Diniz da Costa, Camila Nunes, Viviane Torres da Silva, Baldoino Fonseca, and Carlos J. P. de Lucena, *Jaaf+t: a framework to implement self-adaptive agents that apply self-test*, SAC '10 (New York, NY, USA), ACM, 2010, pp. 928–935.

[23] Carlos Eduardo da Silva and Rogério de Lemos, *Dynamic plans for integration testing of self-adaptive software systems*, Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (New York, NY, USA), SEAMS '11, ACM, 2011, pp. 148–157.

[24] Giovanni Denaro, Leonardo Mariani, Mauro Pezzè, and Davide Tosi, *Adaptive runtime verification for autonomic communication infrastructures*, First International IEEE WoWMoM Workshop on Autonomic Communications and Computing (ACC'05), 2005, pp. 553–557.

[25] Yi Deng, S. M. Sadjadi, Peter J. Clarke, Vagelis Hristidis, Raju Rangaswami, and Yingbo Wang, *CVM-a communication virtual machine*, vol. 81, Elsevier Science Inc., 2008, pp. 1640–1662.

[26] Arie Van Deursen and Leon Moonen, *The video store revisited - thoughts on refactoring and testing*, 2002.

[27] Daniel Deveaux, Patrice Frison, and Jean-Marc Jézéquel, *Increase software trustability with self-testable classes in java*, Proceedings of the 13th Australian Conference on Software Engineering (Washington, DC, USA), ASWEC '01, IEEE Computer Society, 2001, pp. 3–11.

[28] Mark Doliner, Grzegorz Lukasik, and Jeremy Thomerson, *Cobertura 1.9*, 2002, http://cobertura.sourceforge.net/ (July 2010).

[29] Baldoino F. dos S. Neto, Andrew Diniz da Costa, Manoel T. de A. Netto, Viviane Torres da Silva, and Carlos José Pereira de Lucena, *Jaaf: A framework to implement self-adaptive agents*, SEKE '09, Knowledge Systems Institute Graduate School, 2009, pp. 212–217.

[30] James Drallos, Jordan Clare, Joseph Korolewicz, Daniel Laboy, and Betty H.C. Cheng, *SRS: EMR Data Analysis*, Tech. report, Michigan State University, East Lansing, MI, Fall 2009, http://www.cse.msu.edu/ cse435/Projects/F09/EMR-Analysis/web/ (Nov 2010).

[31] Eclipse Foundation, *Eclipse 3.2*, November 2001, http://www.eclipse.org/ (July 2010).

[32] Eclipse Foundation., *ATL: A Model Transformation Technology*, Jan 2004, http://www.eclipse.org/atl/ (July 2010).

[33] Gregor Engels, Roland Hücking, Stefan Sauer, and Annika Wagner, *Uml collaboration diagrams and their transformation to java*, Proceedings of the 2nd international conference on The unified modeling language: beyond the standard (Berlin, Heidelberg), UML'99, Springer-Verlag, 1999, pp. 473–488.

[34] Enterprise Java Beans., http://www.java.sun.com/products/ejb/docs.html (October 2011).

[35] Enterprise Managment Associates, *Practical autonomic computing: Roadmap to self managing technology*, Tech. report, IBM, Boulder, CO, Jan. 2006.

[36] Michael Feathers, *Working effectively with legacy code*, Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[37] David Flanagan and Yukihiro Matsumoto, *The ruby programming language - everything you need to know: covers ruby 1.8 and 1.9*, O'Reilly, 2008.

[38] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt, *Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem*, vol. 29, ACM, May 2007.

[39] Eclipse Foundation, *Eclipse Modeling Framework*, August 2003, http://www.eclipse.org/modeling/emf/ (July 2010).

[40] Christopher Fox, *Introduction to software engineering design: Processes, principles and patterns with uml2*, Addison Wesley, 2006.

[41] Robert France and Bernhard Rumpe, *Model-driven development of complex software: A research roadmap*, FOSE '07 (Washington, DC, USA), IEEE Computer Society, 2007, pp. 37–54.

[42] Franz-Josef Elmer, *Classycle 1.4*, 2011, http://classycle.sourceforge.net/ (October 2011).

[43] Erich Gamma and Kent Beck, *JUnit 3.8.1*, 2005, http://www.junit.org/index.htm (July 2010).

[44] Erich Gamma, John Vlissides, Ralph Johnson, and Richard Helm, *Design patterns: Elements of reusable o-o software*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[45] Holger Giese and Robert Wagner, *From model transformation to incremental bidirectional model synchronization*, vol. 8, Springer Berlin / Heidelberg, February 2009, pp. 21–43.

[46] Todd Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, and Gregg Rothermel, *An empirical study of regression test selection techniques*, no. 2, April 2001, pp. 184–208.

[47] Paul Hamill, *Unit test frameworks*, 2004.

[48] Ahmed E. Hassan and Richard C. Holt, *Predicting change propagation in software systems*, vol. 0, IEEE Computer Society, 2004, pp. 284–293.

[49] Shenin Hassan, Dhiya Al-Jumeily, and Abir Jaafar Hussain, *Autonomic computing paradigm to support system's development*, vol. 0, IEEE Computer Society, 2009, pp. 273–278.

[50] Jane Huffman Hayes, Alex Dekhtyar, and James Osborne, *Improving requirements tracing via information retrieval*, in Proceedings of the International Conference on Requirements Engineering (RE, 2003, pp. 151–161.

[51] Hewlett-Packard Development Company, L.P., *HP Adaptive Infrastructure: Accelerating adoption of next-generation data center technologies and services to optimize business outcomes*, Tech. report, HP, February 2009.

[52] Paul Horn, *Autonomic Computing: IBM's perspective on the State of Information Technology*, October 2001.

[53] Hai Hu, Chang-Hai Jiang, and Kai-Yuan Cai, *Adaptive software testing in the context of an improved controlled markov chain model*, Proceedings of the 2008 32nd Annual IEEE International Computer Software and Applications Conference (Washington, DC, USA), COMPSAC '08, IEEE Computer Society, 2008, pp. 853–858.

[54] Chang Hwan, Peter Kim, and Krzysztof Czarnecki, *Synchronizing cardinality-based feature models and their specializations*, Proceedings of the First European conference on Model Driven Architecture: foundations and Applications (Berlin, Heidelberg), ECMDA-FA'05, Springer-Verlag, 2005, pp. 331–348.

[55] IBM Autonomic Computing Architecture Team, *An architectural blueprint for autonomic computing*, Tech. report, IBM, Hawthorne, NY, June 2006.

[56] IEEE Computer Society, *Std 610.12-1990(r2002): Glossary of software engineering terms*, Tech. report, 2002.

[57] Paola Inverardi, Henry Muccini, and Patrizio Pelliccione, *Automated check of architectural models consistency using spin*, vol. 0, IEEE Computer Society, 2001, p. 346.

[58] Igor Ivkovic and Kostas Kontogiannis, *Tracing evolution changes of software artifacts through model synchronization*, Proceedings of the 20th IEEE International Conference on Software Maintenance (Washington, DC, USA), ICSM '04, IEEE Computer Society, 2004, pp. 252–261.

[59] Java Struts., http://www.struts.apache.org/ (October 2011).

[60] Zhenyi Jin and A. Jefferson Offutt, *Coupling-based criteria for integration testing*, vol. 8, 1998, pp. 133–154.

[61] Jeffrey O. Kephart, *Research challenges of autonomic computing*, ICSE '05: Proceedings of the 27th international conference on Software engineering, 2005, pp. 15–22.

[62] J.O. Kephart and D.M. Chess, *The vision of autonomic computing*, vol. 36, January 2003, pp. 41–52.

[63] Tariq King, Andrew Allen, Rodolfo Cruz, and Peter Clarke, *Safe runtime validation of behavioral adaptations in autonomic software*, Autonomic and Trusted Computing (Jose Calero, Laurence Yang, Felix Marmol, Luis Garcia Villalba, Andy Li, and Yan Wang, eds.), Lecture Notes in Computer Science, vol. 6906, Springer Berlin / Heidelberg, 2011, 10.1007/978-3-642-23496-53, pp. 31–46.

[64] Tariq M. King, Andrew A. Allen, Yali Wu, Peter J. Clarke, and Alain E. Ramirez, *A comparative case study on the engineering of self-testable autonomic software*, vol. 0, IEEE Computer Society, 2011, pp. 59–68.

[65] Tariq M. King, Djuradj Babich, Jonatan Alava, Ronald Stevens, and Peter J. Clarke, *Towards self-testing in autonomic computing systems*, ISADS '07 (Washington, DC, USA), IEEE Computer Society, 2007, pp. 51–58.

[66] Tariq M. King and Annaji Sharma Ganti, *Migrating autonomic self-testing to the cloud*, Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops (Washington, DC, USA), ICSTW '10, IEEE Computer Society, 2010, pp. 438–443.

[67] Tariq M. King, Alain Ramirez, Peter J. Clarke, and Barbara Quinones-Morales, *A reusable object-oriented design to support self-testable autonomic software*, Proceedings of the 2008 ACM symposium on Applied computing (New York, NY, USA), SAC '08, ACM, 2008, pp. 1664–1669.

[68] Tariq M. King, Alain E. Ramirez, Rodolfo Cruz, and Peter J. Clarke, *An integrated self-testing framework for autonomic computing systems*, vol. 2, 2007, pp. 37–49.

[69] Dierk Koenig, Andrew Glover, Paul King, Guillaume Laforge, and Jon Skeet, *Groovy in action*, Manning Publications Co., Greenwich, CT, USA, 2007.

[70] Timothy Lethbridge and Robert Laganiere, *Object-oriented software engineering: Practical software development using uml and java*, 1 ed., McGraw-Hill, Inc., New York, NY, USA, 2002.

[71] Bennett P. Lientz and E. Burton Swanson, *Software maintenance management*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1980.

[72] Dongxi Liu, Zhenjiang Hu, and Masato Takeichi, *Bidirectional interpretation of xquery*, Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation (New York, NY, USA), PEPM '07, ACM, 2007, pp. 21–30.

[73] Andrian Marcus and Jonathan I. Maletic, *Recovering documentation-to-source-code traceability links using latent semantic indexing*, Proceedings of the 25th International Conference on Software Engineering (Washington, DC, USA), ICSE '03, IEEE Computer Society, 2003, pp. 125–135.

[74] Jean marie Favre, *Towards a basic theory to model model driven engineering*, In Workshop on Software Model Engineering, WISME 2004, joint event with UML2004, 2004.

[75] Eliane Martins, Cristina Maria Toyota, and Rosileny Lie Yanagawa, *Constructing self-testable software components*, Proceedings of the 2001 International Conference on Dependable Systems and Networks (formerly: FTCS) (Washington, DC, USA), DSN '01, IEEE Computer Society, 2001, pp. 151–160.

[76] Gerard Meszaros, *Xunit test patterns: Refactoring test code*, Addison-Wesley, 2007.

[77] Microsoft Corporation, *Dynamic Systems Initiative Overview*, Tech. report, Microsoft, March 2007.

[78] Arun Mishra and Arun K. Misra, *Formal aspects of specification and validation of dynamic adaptive system by analyzing execution traces*, vol. 0, IEEE Computer Society, 2011, pp. 49–58.

[79] Daniel J. Mosley and Bruce Posey, *Just enough software test automation*, Prentice Hall PTR, Upper Saddle River, NJ, USA, 2002.

[80] H. A. Muller, L. O'Brien, M. Klein, and B. Wood, *Autonomic Computing*, Tech. report, Carnegie Mellon Univeristy and SEI, April 2006.

[81] Freddy Munoz and Benoit Baudry, *Artificial table testing dynamically adaptive systems*, HAL - CCSD, 2009.

[82] Glenford J. Myers, Tom Badgett, Todd M. Thomas, Corey Sandler, and Inc ebrary, *The art of software testing*, 2nd ed ed., John Wiley & Sons, Hoboken, N.J, 2004.

[83] Dirk Niebuhr and Andreas Rausch, *Guaranteeing correctness of component bindings in dynamic adaptive systems based on runtime testing*, Proceedings of the 4th international workshop on Services integration in pervasive environments (New York, NY, USA), SIPE 09, ACM, 2009, pp. 7–12.

[84] Carlos Pacheco and Michael D. Ernst, *Randoop: feedback-directed random testing for Java*, OOPSLA 2007 Companion, Montreal, Canada, ACM, October 2007.

[85] Ponder2, http://www.ponder2.net/cgi-bin/moin.cgi/Ponder2Overview (July 2011).

[86] Roger S. Pressman, *Software engineering: A practitioner's approach*, 5th ed., McGraw-Hill Higher Education, 2001.

[87] Abdallah Qusef, Rocco Oliveto, and Andrea De Lucia, *Recovering traceability links between unit tests and classes under test: An improved method*, Proceedings of the 2010 IEEE International Conference on Software Maintenance (Washington, DC, USA), ICSM '10, IEEE Computer Society, 2010, pp. 1–10.

[88] Alain Ramirez, Barbara Morales, and Tariq M. King, *A self-testing autonomic job scheduler*, ACM-SE 46 (New York, NY, USA), ACM Press, 2008, pp. 304–309.

[89] Jean rmy Falleri, Marianne Huchard, and Clmentine Nebut, *C.: Towards a traceability framework for model transformations in kermeta*, In: ECMDA-TW Workshop, 2006.

[90] Bart Van Rompaey and Serge Demeyer, *Establishing traceability links between unit test cases and units under test.*, CSMR (Andreas Winter, Rudolf Ferenc, and Jens Knodel, eds.), IEEE, 2009, pp. 209–218.

[91] Mazeiar Salehie and Ladan Tahvildari, *Self-adaptive software: Landscape and research challenges*, vol. 4, ACM, 2009, pp. 1–42.

[92] Andy Schrr, *Specification of graph translators with triple graph grammars*, in Proc. of the 20th Int. Workshop on Graph-Theoretic Concepts in Computer Science (WG '94), Herrsching (D, Springer, 1995.

[93] M. Shaw, *Abstraction techniques in modern programming languages*, vol. 1, IEEE Computer Society, 1984, pp. 10–26.

[94] Ian Sommerville, *Software engineering: Seventh edition*, Addison-Wesley, Essex, England, 2004.

[95] Ronald Stevens, Brittany Parsons, and Tariq M. King, *A self-testing autonomic container*, ACM-SE 45 (New York, NY, USA), ACM Press, 2007, pp. 1–6.

[96] Sun Microsystems, Inc., *Core Java J2SE*, February 2005, http://java.sun.com/j2se/ (July 2009).

[97] The Eclipse Foundation, *Test and Performance Tools Platform*, Nov. 2001, http://www.eclipse.org/tptp/ (July 2011).

[98] Yves Le Traon, Daniel Deveaux, and Jean-Marc J233;z233;quel, *Self-testable components: From pragmatic tests to design-for-testability methodology*, vol. 0, IEEE Computer Society, 1999, p. 96.

[99] Laurence Tratt, *A change propagating model transformation language*, vol. 7, March 2008, pp. 107–126.

[100] Triskell Team, *Kermeta - Breathe life into your metamodels*, October 2005, http://www.kermeta.org/ (July 2010).

[101] Craig Walls and Ryan Breidenbach, *Spring in action*, Manning Publications Co., Greenwich, CT, USA, 2005.

[102] E J Weyuker, *Axiomatizing software test data adequacy*, vol. 12, IEEE Press, 1986, pp. 1128–1138.

[103] Lee White and Brian Robinson, *Industrial real-time regression testing and analysis using firewalls*, ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance (Washington, DC, USA), IEEE Computer Society, 2004, pp. 18–27.

[104] William Whitney, *Automatic JUnit Creation Tool: An Exploration in High Level Process Driven Automatic Test Case Creation*, June, http://sourceforge.net/projects/amaticjunittool/ (June 2010).

[105] Yingfei Xiong, Dongxi Liu, Zhenjiang Hu, Haiyan Zhao, Masato Takeichi, and Hong Mei, *Towards automatic model synchronization from model transformations*, ASE '07 (New York, NY, USA), ACM, 2007, pp. 164–173.

[106] Suresh Yadla, Jane Huffman Hayes, and Alex Dekhtyar, *Tracing requirements to defect reports: an application of information retrieval techniques*, 2005, pp. 116–124.

[107] Ji Zhang, Betty H. C. Cheng, Zhenxiao Yang, and Philip K. McKinley, *Enabling safe dynamic component-based software adaptation.*, WADS, 2004, pp. 194–211.

[108] Ji Zhang, Heather J. Goldsby, and Betty H.C. Cheng, *Modular verification of dynamically adaptive systems*, AOSD '09 (New York, NY, USA), ACM, 2009, pp. 161–172.

[109] H. Zhu, P. A. V. Hall, and J. H. R. May, *Software unit testing coverage and adequacy*, vol. 29, December 1997, pp. 366–427.

APPENDIX A. MANUAL EVALUATION ADDITIVE ORACLE

- Class Name: Diagnosis

Unit test needs to be added	**Integration test needs to be added(class here is a caller)**		**Integration test needs to be added (class here is a callee)**	
	Test	**Callee/method**	**Test**	**Caller/method**
Test createDiagnosis(java.lang.String, java.lang.String, java.util.Date)			Test confirmDiagnosis(emrservicedesign.Diagnosis)	emrservicedesign.EMRService/getSymptom()
Test getDate()			Test confirmDiagnosis(emrservicedesign.Diagnosis)	emrservicedesign.EMRService/getDisease()
Test getDisease()				
Test getSymptom()				
Test setDate(java.util.Date)				
Test setDisease(java.lang.String)				
Test setSymptom(java.lang.String)				

- Class Name: Medication

Unit test needs to be added	**Integration test needs to be added(class here is a caller)**		**Integration test needs to be added (class here is a callee)**	
	Test	**Callee/method**	**Test**	**Caller/method**
test getDosage()				
test getSideEffects()				
test setDosage(java.lang.String)				
test setSideEffects(ja				

va.lang.String)				
test getDosage()				

- Class Name: EMRService

Unit test needs to be added	Integration test needs to be added(class here is a caller)		Integration test needs to be added (class here is a callee)	
	Test	**Callee/method**	**Test**	**Caller/method**
test setClinicaltrials(java.util.ArrayList)	Test addMedication(java.lang.String, emrservicedesign.Medication)	emrservicedesign.PatientInfo/getHistory()		
test setPatients(java.util.ArrayList)	Test addMedication(java.lang.String, emrservicedesign.Medication)	emrservicedesign.PatientHistory/getMedicationlist()		
test setResult(java.util.ArrayList)	Test confirmDiagnosis(emrservicedesign.Diagnosis)	emrservicedesign.Diagnosis/ getSymptom()		
	Test confirmDiagnosis(emrservicedesign.Diagnosis)	emrservicedesign.ClinicalTrial/ getSymptomDescription()		
	Test confirmDiagnosis(emrservicedesign.Diagnosis)	emrservicedesign.Diagnosis) / getDisease()		
	Test confirmDiagnosis(emrservicedesign.Diagnosis)	emrservicedesign.ClinicalTrial/ getBestDiagnosis()		
	Test createDiagnosis(java.lang.String, emrservicedesign.Diagnosis)	emrservicedesign.PatientInfo / getHistory()		
	Test createDiagnosis(j	emrservicedesign.PatientHistory /		

	ava.lang.String, emrservicedesign .Diagnosis)	getDiagnosislist( )		
	Test getClinicalTrials(emrservicedesign .Symptom)	emrservicedesign .ClinicalTrial/ getSymptomDes cription()		
	Test getClinicalTrials(emrservicedesign .Symptom)	emrservicedesign .Symptom/ getDescription()		
	Test getPatientInfo(ja va.lang.String)	emrservic edesign.PatientIn fo/ getPid()		
	Test scheduleTreatme nt(java.lang.Strin g, emrservicedesign .Treatment)	emrservicedesign .PatientInfo/ getHistory()		
	Test scheduleTreatme nt(java.lang.Strin g, emrservicedesign .Treatment)	emrservicedesign .PatientHistory/ getTreatmentlist( )		

- Class Name: Symptom

Unit test needs to be added	**Integration test needs to be added(class here is a caller)**		**Integration test needs to be added (class here is a callee)**	
	Test	**Callee/method**	**Test**	**Caller/method**
test getDate()			Test getClinicalTrials(emrservicedesign .Symptom)	emrservicedesign .EMRService/ getDescription()
test getDescription()				
test setDate(java.util. Date)				
test setDescription(ja va.lang.String)				

- Class Name: Board

Unit test needs to be added	Integration test needs to be added(class here is a caller)		Integration test needs to be added (class here is a callee)	
	Test	**Callee/method**	**Test**	**Caller/method**
test getCell(int, int)	test Board(int, int)	jpacman.model.Cell/ Cell(int, int, jpacman.model.Board)	test addGuestFromCode(char, int, int)	jpacman.model.Game/ getCell(int, int)
test getHeight()	Test Board(int, int)	jpacman.model.Cell/ -clinit-()	test boardHeight()	jpacman.model.Game/ getHeight()
test getWidth()	Test getGuest(int, int)	jpacman.model.Cell/ getInhabitant()	test boardWidth()	jpacman.model.Game/ getWidth()
test withinBorders(int, int)	Test guestCode(int, int)	jpacman.model.Wall/ guestType()	Test cellAtOffset(int, int)	jpacman.model.Cell/ getCell(int, int)
	Test guestCode(int, int)	jpacman.model.Food/ guestType()	Test cellAtOffset(int, int)	jpacman.model.Cell/ withinBorders(int, int)
	Test guestCode(int, int)	jpacman.model.Monster/ guestType()	Test getGuestCode(int, int)	jpacman.model.Game/ guestCode(int, int)
	Test guestCode(int, int)	jpacman.model.Player/ guestType()	test loadWorld(java.lang.String[])	jpacman.model.Game/ Board(int, int)
			test loadWorld(java.lang.String[])	jpacman.model.Game/ -clinit-()

- Class Name: BoardViewer

Unit test needs to be added	Integration test needs to be added(class here is a caller)		Integration test needs to be added (class here is a callee)	
	Test	**Callee/method**	**Test**	**Caller/method**
test cellHeight()	test BoardViewer(jpacman.model.Engine)	jpacman.controller.ImageFactory/ ImageFactory()	test display()	jpacman.controller.PacmanUI/ windowWidth()
Test cellWidth()	test	jpacman.controll	Test	jpacman.c

	BoardViewer(jpacman.model.Engine)	er.ImageFactory/ -clinit-()	display()	ontroller.PacmanUI/ windowHeight()
test createGraphics2D(int, int)	test drawCell(int, int, java.awt.Graphics2D)	jpacman.model.Engine/ getGuestCode(int, int)	Test PacmanUI(jpacman.model.Engine, jpacman.controller.Pacman)	jpacman.controller.PacmanUI/ BoardViewer(jpacman.model.Engine)
Test drawCells(java.awt.Graphics2D)	test drawCell(int, int, java.awt.Graphics2D)	jpacman.controller.ImageFactory/ monster(int)	Test PacmanUI(jpacman.model.Engine, jpacman.controller.Pacman)	jpacman.controller.PacmanUI/ -clinit-()
test paint(java.awt.Graphics)	test drawCell(int, int, java.awt.Graphics2D)	jpacman.model.Engine/ getPlayerLastDx()		
Test windowWidth()	test drawCell(int, int, java.awt.Graphics2D)	jpacman.controller.ImageFactory/ player(int, int, int)		
	test drawCell(int, int, java.awt.Graphics2D)	jpacman.model.Engine/ getPlayerLastDy()		
	Test nextAnimation()	jpacman.controller.ImageFactory/ monsterAnimationCount()		
	Test nextAnimation()	jpacman.controller.ImageFactory/ playerAnimationCount()		
	Test worldHeight()	jpacman.model.Engine/ boardHeight()		
	Test worldWidth()	jpacman.model.Engine/ boardWidth()		

- Class Name: Food

Unit test needs to be added	Integration test needs to be added(class here is a caller)		Integration test needs to be added (class here is a callee)	
	Test	**Callee/method**	**Test**	**Caller/method**
Test Food()	Test Food(int)	jpacman.model. Guest/ Guest()	test createFood()	jpacman.model. Game/ Food()
test getPoints()	Test meetPlayer(jpacman.model.PlayerMove)	jpacman.model.PlayerMove/ setFoodEaten(int)	test createFood()	jpacman. model.Game/ getPoints()
test guestType()			test createFood()	jpacman. model.Game/ -clinit-()
			Test guestCode(int, int)	jpacman.model.Board/ guestType()

- Class Name: ImageFactory

Unit test needs to be added	Integration test needs to be added(class here is a caller)		Integration test needs to be added (class here is a callee)	
	Test	**Callee/method**	**Test**	**Caller/method**
test getImage(java.lang.String)			Test BoardViewer(jpacman.model.Engine)	jpacman.controller.BoardViewer/ ImageFactory()
test ImageFactory()			Test BoardViewer(jpacman.model.Engine)	jpacman.controller.BoardViewer/ -clinit-()
Test monsterAnimationCount() int			Test drawCell(int, int, java.awt.Graphics2D)	jpacman.controller.BoardViewer/ monster(int)
Test monster(int)			Test drawCell(int, int, java.awt.Graphics2D)	jpacman.controller.BoardViewer/ player(int, int, int)
Test playerAnimationCount()			Test nextAnimation()	jpacman.controller.BoardViewer/ monsterAnimationCount()

Test player(int, int, int)			Test nextAnimation()	jpacman.controller.BoardViewer/ playerAnimationCount()

- Class Name: Monster

Unit test needs to be added	**Integration test needs to be added(class here is a caller)**		**Integration test needs to be added (class here is a callee)**	
	Test	**Callee/method**	**Test**	**Caller/method**
Test guestType()	Test meetPlayer(jpacman.model.PlayerMove)	jpacman.model.Move/die()	test createMonster()	jpacman.model.Game/ Monster()
	Test Monster()	jpacman.model.MovingGuest/ MovingGuest()	test createMonster()	jpacman.model.Game/ -clinit-()
			test guestCode(int, int)	jpacman.model.Board/ guestType()

- Class Name: Move

Unit test needs to be added	**Integration test needs to be added(class here is a caller)**		**Integration test needs to be added (class here is a callee)**	
	Test	**Callee/method**	**Test**	**Caller/method**
Test die()	test apply()	jpacman.model.Cell/ getInhabitant()	test apply()	jpacman.model.PlayerMove/ apply()
test getMovingGuest()	test apply()	jpacman.model.Guest/ occupy(jpacman.model.Cell)	Test invariant()	jpacman.model.PlayerMove/ moveInvariant()
test movePossible()	test apply()	jpacman.model.Guest/ deoccupy()	Test invariant()	jpacman.model.PlayerMove/ getMovingGuest()
Test Move(jpacman.model.MovingGuest,	test apply()	jpacman.model.Guest/ getLocation()	Test meetPlayer(jpacman.model.PlayerMove)	jpacman.model.Monster/ die()

jpacman.model.Cell)				
test playerDies()	Test moveDone()	jpacman.model.Guest/ getLocation()	Test movePlayer(int, int)	jpacman.model.Game/ -clinit-()
test withinBorder()	Test moveInvariant()	jpacman.model.Guest/ getLocation()	test PlayerMove(jpacman.model.Player, jpacman.model.Cell)	jpacman.model.PlayerMove/ precomputeEffects()
	Test precomputeEffects()	jpacman.model.Cell/ getInhabitant()	test PlayerMove(jpacman.model.Player, jpacman.model.Cell)	jpacman.model.PlayerMove/ Move(jpacman.model.MovingGuest, jpacman.model.Cell)
	Test precomputeEffects()	jpacman.model.PlayerMove/ tryMoveToGuest (jpacman.model.Guest)		

- Class Name: MovePlayer

Unit test needs to be added	**Integration test needs to be added(class here is a caller)**		**Integration test needs to be added (class here is a callee)**	
	Test	**Callee/method**	**Test**	**Caller/method**
test getFoodEaten()	Test apply()	jpacman.model.Move/ apply()	test meetPlayer(jpacman.model.PlayerMove)	jpacman.model.Food/ setFoodEaten(int)
Test getPlayer()	Test apply()	jpacman.model.Player/ eat(int)	Test movePlayer(int, int)	jpacman.model.Game/ PlayerMove(jpacman.model.Player, jpacman.model.Cell)
test setFoodEaten(int)	Test apply()	jpacman.model.Player/ getPointsEaten()	test movePlayer(int, int)	jpacman.model.Game/ -clinit-()

	Test invariant()	jpacman.model.Move/ moveInvariant()	test precomputeEffects()	jpacman.model.Move/ tryMoveToGuest(jpacman.model.Guest)
	Test invariant()	jpacman.model.Move/ getMovingGuest()		
	Test PlayerMove(jpacman.model.Player, jpacman.model.Cell)	jpacman.model.Move/ precomputeEffects()		
	Test PlayerMove(jpacman.model.Player, jpacman.model.Cell)	jpacman.model.Move/ Move(jpacman.model.MovingGuest, jpacman.model.Cell)		
	Test tryMoveToGuest(jpacman.model.Guest)	jpacman.model.Wall/ meetPlayer(jpacman.model.PlayerMove)		

- Class Name: Pacman

Unit test needs to be added	**Integration test needs to be added(class here is a caller)**		**Integration test needs to be added (class here is a callee)**	
	Test	**Callee/method**	**Test**	**Caller/method**
test exit()	Test down()	jpacman.model.Engine/ movePlayer(int, int)	test keyPressed(java.awt.event.KeyEvent)	jpacman.controller.PacmanUI/ up()
	test Pacman(jpacman.model.Engine)	jpacman.controller.AbstractMonsterController/ -clinit-()	test keyPressed(java.awt.event.KeyEvent)	jpacman.controller.PacmanUI/ down()
	test Pacman(jpacman	jpacman.controller.RandomMonst	test keyPressed(java.	jpacman.controller.Pacman

	.model.Engine)	erMover/ RandomMonster Mover(jpacman. model.Engine)	awt.event.KeyEv ent)	UI/ right()
	test Pacman(jpacman .model.Engine)	jpacman.controll er.RandomMonst erMover/ -clinit- ()	test keyPressed(java. awt.event.KeyEv ent)	jpacman.c ontroller.Pacman UI/ left()
	Test Pacman()	jpacman.model.E ngine/ Engine()		
	Test Pacman()	jpacman.model.E ngine/ -clinit-()		
	Test Pacman(jpacman .model.Engine, jpacman.controll er.IMonsterContr oller)	jpacman.controll er.PacmanUI/ PacmanUI(jpacm an.model.Engine, jpacman.controll er.Pacman)		
	Test Pacman(jpacman .model.Engine, jpacman.controll er.IMonsterContr oller)	jpacman.controll er.PacmanUI/ display()		
	Test Pacman(jpacman .model.Engine, jpacman.controll er.IMonsterContr oller)	jpacman.controll er.Animator/ Animator(jpacma n.controller.Boar dViewer)		
	Test Pacman(jpacman .model.Engine, jpacman.controll er.IMonsterContr oller)	jpacman.controll er.PacmanUI/ - clinit-()		
	Test Pacman(jpacman .model.Engine, jpacman.controll er.IMonsterContr oller)	jpacman.controll er.PacmanUI/ getBoardViewer( )		
	Test quit()	jpacman.controll		

		er.Animator/ stop()		
	Test quit()	jpacman.model.Engine/ quit()		
	Test quit()	jpacman.controller.AbstractMonsterController/ stop()		
	Test right()	jpacman.model.Engine/ movePlayer(int, int)		
	Test start()	jpacman.model.Engine/ start()		
	Test start()	jpacman.controller.AbstractMonsterController/start()		
	Test start()	jpacman.controller.Animator/start()		
	Test up()	jpacman.model.Engine/ movePlayer(int, int)		
	Test left()	jpacman.model.Engine/ movePlayer(int, int)		

- Class Name: player

Unit test needs to be added	**Integration test needs to be added(class here is a caller)**		**Integration test needs to be added (class here is a callee)**	
	Test	**Callee/method**	**Test**	**Caller/method**
Test die()	Test Player()	jpacman.model.MovingGuest /MovingGuest()	Test apply()	jpacman.model.PlayerMove/ eat(int)
Test eat(int)			Test apply()	jpacman.model.PlayerMove /getPointsEaten()
Test getLastDx()			Test	jpacman.

			getFoodEaten()	model.Engine/ getPointsEaten()
Test getLastDy()			Test getPlayerLastDx( )	jpacman. model.Game/ getLastDx()
Test getPointsEaten()			Test getPlayerLastDy( )	jpacman.model. Game/ getLastDy()
Test guestType()			Test guestCode(int, int)	jpacman. model.Board/ guestType()
Test living()			Test movePlayer(int, int)	jpacman.model. Game/ setLastDirection(int, int)
Test setLastDirection(int, int)			Test playerDied()	jpacman.model. Game/ living()
			Test playerWon()	jpacman.model. Game/ getPointsEaten()

- Class Name: Wall

Unit test needs to be added	**Integration test needs to be added(class here is a caller)**		**Integration test needs to be added (class here is a callee)**	
	Test	**Callee/method**	**Test**	**Caller/method**
Test guestType()	Test Wall()	jpacman.model. Guest/Guest()	test addGuestFromC ode(char, int, int)	jpacman.model. Game/ Wall()
Test meetPlayer (jpacman.model. PlayerMove)			test addGuestFromC ode(char, int, int)	jpacman.model. Game/ -clinit-()
			test guestCode(int, int)	jpacman.model.B oard/ guestType()
			Test tryMoveToGuest (jpacman.model. Guest)	jpacman.model.P layerMove/ meetPlayer(jpac man.model.Playe rMove)

APPENDIX B. MANUAL EVALUATION REDUCTIVE ORACLE

- Class Name: ClinicalTrial

Unit test needs to be removed	Integration test needs to be updated(class here is a callee)		Integration test needs to be removed(class here is a caller)	
	Test	**Caller**	**Test**	**Callee**
	testConfirmDiagnosis_rainy()	emrservicedesign.EMRServic/ClinicalTrial(java.lang.String, java.lang.String, java.lang.String, java.lang.String, java.lang.String)		
	testConfirmDiagnosis_sunny()	emrservicedesign.EMRServic/ClinicalTrial(java.lang.String, java.lang.String, java.lang.String, java.lang.String, java.lang.String)		
	testGetClinicalTrials_rainy()	emrservicedesign.EMRServic/ClinicalTrial(java.lang.String, java.lang.String, java.lang.String, java.lang.String, java.lang.String)		
	testGetClinical	emrservicedesi		

	Trials_sunny()	gn.EMRServic/ ClinicalTrial(java.lang.String, java.lang.String , java.lang.String , java.lang.String , java.lang.String)		

- Class Name: Diagnosis

Unit test needs to be removed	**Integration test needs to be updated(class here is a callee)**		**Integration test needs to be removed(class here is a caller)**	
	Test	**Caller**	**Test**	**Callee**
testGetDate()	testAddMedication()	emrservicedesign.EMRService / Diagnosis(java.lang.String, java.lang.String , java.util.Date)		
testGetDisease( )	testConfirmDiagnosis_rainy()	emrservicedesign.EMRService / Diagnosis(java.lang.String, java.lang.String , java.util.Date)		
testGetSymptom()	testConfirmDiagnosis_sunny()	emrservicedesign.EMRService / Diagnosis(java.lang.String, java.lang.String , java.util.Date)		
testSetDate()	testCreateDiagnosis()	emrservicedesign.EMRService / Diagnosis(java.lang.String, java.lang.String , java.util.Date)		

testSetDisease()	testScheduleTreatment()	emrservicedesign.EMRService/Diagnosis(java.lang.String, java.lang.String, java.util.Date)		
testSetSymptom()				

- Class Name: EMRService

Unit test needs to be removed	**Integration test needs to be updated(class here is a callee)**		**Integration test needs to be removed(class here is a caller)**	
	Test	**Caller**	**Test**	**Callee**
testGetPatientInfo_rainy()			testAddMedication()	emrservicedesign.Diagnosis/Diagnosis(java.lang.String, java.lang.String, java.util.Date)
			testAddMedication()	emrservicedesign.Symptom/Symptom(java.lang.String, java.util.Date)
			testAddMedication()	emrservicedesign.Medication/Medication(java.lang.String, java.lang.String)
			testAddMedication()	emrservicedesign.Treatment/Treatment(java.lang.String, java.util.Date, java.lang.String, java.lang.String)
			testAddMedication()	emrservicedesign.PatientHistory/PatientHistory(j

				ava.util.ArrayList, java.util.ArrayList, java.util.ArrayList, java.util.ArrayList)
			testAddMedication()	emrservicedesign.PatientInfo/PatientInfo(java.lang.String, java.lang.String, int, int, emrservicedesign.PatientHistory)
			testConfirmDiagnosis_rainy()	emrservicedesign.Diagnosis/Diagnosis(java.lang.String, java.lang.String, java.util.Date)
			testConfirmDiagnosis_rainy()	emrservicedesign.ClinicalTrial/ClinicalTrial(java.lang.String, java.lang.String, java.lang.String, java.lang.String, java.lang.String)
			testConfirmDiagnosis_sunny(	emrservicedesign.Diagnosis/Diagnosis(java.lang.String, java.lang.String, java.util.Date)
			testConfirmDiagnosis_sunny(	emrservicedesign.ClinicalTrial/ClinicalTrial(ja

				va.lang.String, java.lang.String , java.lang.String , java.lang.String)
			testCreateDiagnosis()	emrservicedesign.Diagnosis/ Diagnosis(java.lang.String, java.lang.String , java.util.Date)
			testCreateDiagnosis()	emrservicedesign.Symptom/ Symptom(java.lang.String, java.util.Date)
			testCreateDiagnosis()	emrservicedesign.Medication/ Medication(java.lang.String, java.lang.String)
			testCreateDiagnosis()	emrservicedesign.Treatment/ Treatment(java.lang.String, java.util.Date, java.lang.String , java.lang.String)
			testCreateDiagnosis()	emrservicedesign.PatientHistory/ PatientHistory(java.util.ArrayList, java.util.ArrayList, java.util.ArrayList,

				java.util.ArrayList)
			testCreateDiagnosis()	emrservicedesign.PatientInfo/ PatientInfo(java.lang.String, java.lang.String, int, int, emrservicedesign.PatientHistory)
			testGetClinicalTrials_rainy()	emrservicedesign.Symptom/ Symptom(java.lang.String, java.util.Date)
			testGetClinicalTrials_rainy()	emrservicedesign.ClinicalTrial/ ClinicalTrial(java.lang.String, java.lang.String, java.lang.String, java.lang.String, java.lang.String
			testGetClinicalTrials_sunny()	emrservicedesign.Symptom/ Symptom(java.lang.String, java.util.Date)
			testGetClinicalTrials_sunny()	emrservicedesign.ClinicalTrial/ ClinicalTrial(java.lang.String, java.lang.String, java.lang.String, java.lang.String,

				java.lang.String)
			testGetPatientInfo_sunny()	emrservicedesign.PatientInfo/ PatientInfo(java.lang.String, java.lang.String, int, int)
			testScheduleTreatment()	emrservicedesign.Diagnosis/ Diagnosis(java.lang.String, java.lang.String, java.util.Date)
			testScheduleTreatment()	emrservicedesign.Symptom/ Symptom(java.lang.String, java.util.Date)
			testScheduleTreatment()	emrservicedesign.Medication/ Medication(java.lang.String, java.lang.String)
			testScheduleTreatment()	emrservicedesign.Treatment/ Treatment(java.lang.String, java.util.Date, java.lang.String , java.lang.String)
			testScheduleTreatment()	emrservicedesign.PatientHistory/ PatientHistory(java.util.ArrayList, java.util.ArrayList, java.util.ArrayList, java.util.ArrayList)

			testScheduleTreatment()	emrservicedesign.PatientInfo/ PatientInfo(java.lang.String, java.lang.String, int, int, emrservicedesign.PatientHistory)

- Class Name: Medication

Unit test needs to be removed	**Integration test needs to be updated(class here is a callee)**		**Integration test needs to be removed(class here is a caller)**	
	Test	**Caller**	**Test**	**Callee**
testGetDosage()	testAddMedication()			
testGetSideEffects()	testCreateDiagnosis()			
testSetDosage()	testScheduleTreatment()			
testSetSideEffects()				

- Class Name: PatientHistory

Unit test needs to be removed	**Integration test needs to be updated(class here is a callee)**		**Integration test needs to be removed(class here is a caller)**	
	Test	**Caller**	**Test**	**Callee**
	testAddMedication()	emrservicedesign.EMRService/ PatientHistory(java.util.ArrayList, java.util.ArrayList, java.util.ArrayList)		
	testCreateDiagnosis()	emrservicedesign.EMRService		

		/ PatientHistory(java.util.ArrayList, java.util.ArrayList, java.util.ArrayList, java.util.ArrayList)		
	testScheduleTreatment()	emrservicedesign.EMRService / PatientHistory(java.util.ArrayList, java.util.ArrayList, java.util.ArrayList, java.util.ArrayList)		

- Class Name: Patientinfo

Unit test needs to be removed	**Integration test needs to be updated(class here is a callee)**		**Integration test needs to be removed(class here is a caller)**	
	Test	**Caller**	**Test**	**Callee**
	testAddMedication()	emrservicedesign.EMRService / PatientInfo(java.lang.String, java.lang.String, int, int, emrservicedesign.PatientHistory)		
	testCreateDiagnosis()	emrservicedesign.EMRService / PatientInfo(java.lang.String, java.lang.String, int, int,		

		emrservicedesign.PatientHistory		
	testGetPatientInfo_sunny()	emrservicedesign.EMRService/PatientInfo(java.lang.String, java.lang.String, int, int)		
	testScheduleTreatment()	emrservicedesign.EMRService/PatientInfo(java.lang.String, java.lang.String, int, int, emrservicedesign.PatientHistory)		

- Class Name: Symptom

Unit test needs to be removed	**Integration test needs to be updated(class here is a callee)**		**Integration test needs to be removed(class here is a caller)**	
	Test	**Caller**	**Test**	**Callee**
testGetDate()	testAddMedication()	emrservicedesign.EMRService/Symptom(java.lang.String, java.util.Date)		
testGetDescription()	testCreateDiagnosis()	emrservicedesign.EMRService/Symptom(java.lang.String, java.util.Date)		
testSetDate()	testGetClinicalTrials_rainy()	emrservicedesign.EMRService/Symptom(java.lang.String, java.util.Date)		
testSetDescripti	testGetClinical	emrservicedesi		

on()	Trials_sunny()	gn.EMRService / Symptom(java.lang.String, java.util.Date)		
	testScheduleTreatment()	emrservicedesign.EMRService / Symptom(java.lang.String, java.util.Date)		

- Class Name: Treatment

Unit test needs to be removed	**Integration test needs to be updated(class here is a callee)**		**Integration test needs to be removed(class here is a caller)**	
	Test	**Caller**	**Test**	**Callee**
	testAddMedication()	emrservicedesign.EMRService / Treatment(java.lang.String, java.util.Date, java.lang.String, java.lang.String)		
	testCreateDiagnosis()	emrservicedesign.EMRService / Treatment(java.lang.String, java.util.Date, java.lang.String, java.lang.String)		
	testScheduleTreatment()	emrservicedesign.EMRService / Treatment(java.lang.String, java.util.Date, java.lang.String,		

		java.lang.String)		

- Class Name: Board

Unit test needs to be removed	Integration test needs to be updated(class here is a callee)		Integration test needs to be removed(class here is a caller)	
	Test	**Caller**	**Test**	**Callee**
testGettingWidthHeight()	testCellAtOffset()	jpacman.model.Cell/getCell(int, int)	testFailingBoardCreation()	TestUtils.jpacman/-clinit-()
			testFailingBoardCreation()	TestUtils.jpacman/assertionsEnabled()
			testGettingCellsFromBoard()	jpacman.model.Cell/getY()
			testGettingCellsFromBoard()	jpacman.model.Cell/getX()
			testOccupy()	jpacman.model.Guest/-clinit-()
			testOccupy()	jpacman.model.Guest/occupy(jpacman.model.Cell)
			testOccupy()	jpacman.model.Food/Food()
			testOccupy()	jpacman.model.Food/-clinit-()

- Class Name: Cell

Unit test needs to be removed	Integration test needs to be updated(class here is a callee)		Integration test needs to be removed(class here is a caller)	
	Test	**Caller**	**Test**	**Callee**
	testGettingCellsFromBoard()	jpacman.model.Board/ getY()	testCellAtOffset()	jpacman.model.Board/ getCell(int, int)
	testGettingCellsFromBoard()	jpacman.model.Board/ getX()		
	testOccupyDeoccupy()	jpacman.model.Guest/ getInhabitant()		

- Class Name: Engine

Unit test needs to be removed	Integration test needs to be updated(class here is a callee)		Integration test needs to be removed(class here is a caller)	
	Test	**Caller**	**Test**	**Callee**
	testUpdates()	jpacman.model.Observer/ inPlayingState()		
	testUpdates()	jpacman.model.Observer/ start()		
	testUpdates()	jpacman.model.Observer/ movePlayer(int, int)		
	testUpdates()	jpacman.model.Observer/ quit()		
	testUpdates()	jpacman.model.Observer/ inHaltedState()		

- Class Name: Game

Unit test needs to be removed	Integration test needs to be updated(class here is a callee)		Integration test needs to be removed(class here is a caller)	
	Test	**Caller**	**Test**	**Callee**
testDxDyImpossibleMove()				
testDxDyPossibleMove()				
testGetMonsters()				

- Class Name: Guest

Unit test needs to be removed	Integration test needs to be updated(class here is a callee)		Integration test needs to be removed(class here is a caller)	
	Test	**Caller**	**Test**	**Callee**
	testOccupy()	jpacman.model.Board/-clinit-()	testOccupyDeoccupy()	jpacman.model.Cell/getInhabitant()
	testOccupy()	jpacman.model.Board/occupy(jpacman.model.Cell)		

- Class Name: ImageFactory

Unit test needs to be removed	Integration test needs to be updated(class here is a callee)		Integration test needs to be removed(class here is a caller)	
	Test	**Caller**	**Test**	**Callee**
`testMonster()`				
`testPlayer()`				

- Class Name: Move

Unit test needs to be removed	Integration test needs to be updated(class here is a callee)		Integration test needs to be removed(class here is a caller)	
	Test	**Caller**	**Test**	**Callee**
	testSimpleGetters()	jpacman.model.PlayerMove/movePossible()		
	testSimpleGetters()	jpacman.model.PlayerMove/-clinit-()		
	testSimpleGetters()	jpacman.model.PlayerMove/playerDies()		

- Class Name: Observer

Unit test needs to be removed	Integration test needs to be updated(class here is a callee)		Integration test needs to be removed(class here is a caller)	
	Test	**Caller**	**Test**	**Callee**
			testUpdates()	jpacman.model.Engine /inPlayingState()
			testUpdates()	jpacman.model.Engine/ movePlayer(int, int)
			testUpdates()	jpacman.model.ObserverTest$MyObserver/ access$1(jpacman.model.ObserverTest$MyObserver)
			testUpdates()	jpacman.model.Engine/ quit()
			testUpdates()	jpacman.model.Engine/ inHaltedState()
			testUpdates()	jpacman.model.Engine/ start()

- Class Name: pacman

Unit test needs to be removed	Integration test needs to be updated(class here is a callee)		Integration test needs to be removed(class here is a caller)	
	Test	**Caller**	**Test**	**Callee**
testTopLevelAlphaOmega()				