Towards Change Propagating Test Models in Autonomic and Adaptive Systems

Mohammed Akour, Akanksha Jaidev, and Tariq M. King
Department of Computer Science
North Dakota State University
Fargo, ND 58108, USA
E-mail: {mohammed.akour, akanksha.jaidev, tariq.king}@ndsu.edu

***Abstract*—Autonomic and adaptive computing systems can add, remove, and replace their own components in response to a changing environment. Self-adaptation facilitates the performance of automated maintenance and configuration tasks, but makes it possible for faults to be introduced into the software at runtime. To address this issue, researchers have developed approaches for integrating runtime testing into autonomic and adaptive software systems.**

An important aspect of runtime testing approaches for autonomic software is the provision of a framework for regression testing, which determines whether modifications have introduced faults into previously tested components. However, after adaptation occurs in autonomic software, 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.

In this paper we describe a model-driven approach that maps structural adaptations in autonomic software, to updates for its runtime test model. We provide a workflow and meta-model to support the approach, referred to as Test Information Propagation (TIP). To demonstrate TIP, we have developed a prototype that simulates a reductive change to an autonomic, service-oriented healthcare application. Conducting the simulation has provided us with much insight into this highly challenging research problem.

***Keywords*-testing; autonomic software; self-adaptation; change propagation; model-driven engineering**

I. 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 [1]. Autonomic and adaptive computing seeks to meet this demand by specifying systems that can *self-configure*, *self-optimize*, *self-heal*, and *self-protect* [2]. These types of systems can add, remove, or replace their own components at runtime, referred to as *additive*, *reductive*, and *mutative* changes.

Although researchers have developed many tools and techniques for building adaptive systems [3], [4], [5], there has been little research on assuring their quality and reliability [1]. More specifically, only a few researchers have addressed the need for runtime validation and verification (V&V) in self-adaptive software [6], [7]. 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.

King et al. [6] 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. This is achieved by linking the core system to an automated testing harness, consisting of a set of baseline test cases, test drivers, test stubs, and other tools to support testing. However, their approach does not 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.

In this paper we describe an approach for synchronizing component models and runtime test models in autonomic software. Our approach is based on *change propagation* [8], an emerging field of model-driven engineering (MDE) that supports making multiple updates to models at runtime. The major contributions of this work are: (1) formulation of a new approach for propagating structural changes in autonomic software to runtime test models, and (2) description of the findings and experiences from conducting a prototype simulation that demonstrates the approach.

The rest of this paper is organized as follows: the next section provides background material for understanding the research problem. Section III summarizes a systematic literature review which was performed to survey the research area. Section IV presents our model-driven approach. Section V describes the prototype. Section VI provides related work, and in Section VII we conclude the paper.

II. BACKGROUND

This section provides background material on autonomic and adaptive computing, software testing, and the model-driven engineering field of change propagation.

A. Autonomic and Adaptive Computing

A software system can be modeled as a set of interconnected components that communicate across one or more processes [3]. The movement towards autonomic computing [2] 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 [2].

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 [3], [6], [7]. Dynamic adaptation is said to be safe if it does not violate dependencies between components, or interrupt critical communications [3]. 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 [6].

B. Software Testing

Testing involves executing a program on specified inputs, recording the results, and making an evaluation to determine whether the software behaves as intended [9]. Testing may be performed using *black box* or *white box* techniques [10]. 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 [11]. Hence, for white box techniques, test adequacy is usually specified in terms of coverage of elements of the program, e.g., all statements.

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* [12]. 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 [12]. The set of drivers and other tools to support test execution is called a *test harness*.

C. Change Propagation

The goal of model-driven engineering (MDE) is to instate models as first-class citizens throughout the software process [13]. Transformations between models is therefore one of the key goals of MDE. Meta-modeling has been recognized as a standard technique for representing and transforming software artifacts [14]. 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 [8], [15].

When developing an approach based on change propagation, the following factors should be considered [8]: (1) *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; (2) *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 (3) *Immediate or Batch* – an approach may propagate changes to the target as soon as the source is changed, or propagate multiple changes when applied.

III. PRELIMINARY INVESTIGATION

As a preliminary step in our investigation, we performed a systematic literature review [16] to determine the current landscape surrounding the research problem.

A. Research Questions

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 I. 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 I 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.

B. Review Results

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

Research Questions	Motivation
1. Are there approaches in 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 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 I
RESEARCH QUESTIONS AND MOTIVATION USED TO GUIDE THE SYSTEMATIC LITERATURE REVIEW [16]

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 [1], [6]. 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 [8], [15], 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 [8]. However, there appears to be a plethora of general MDE tools [17], [18], [19] 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: (1) runtime testing of autonomic and adaptive systems, and (2) change propagation.

IV. The TIP Approach

To address the research problem, we propose a model-driven approach for updating the runtime tests of a software system after dynamic adaptation. Our approach, referred to as *Test Information Propagation* (TIP), uses change propagation to synchronize elements of the adaptive system's

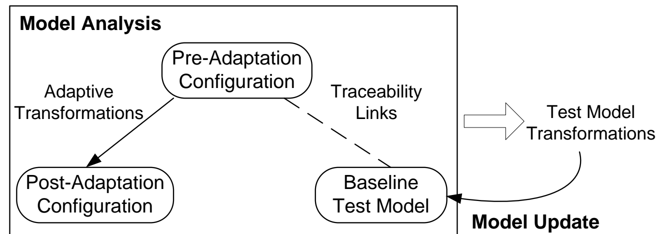

Figure 1. A Test Information Propagation Approach for Adaptive Software

component model, with related elements in its runtime test model. Figure 1 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 *dynamic test case generation*, an extremely challenging research problem that continues to be studied in the literature. 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 changes.

Figure 2 provides a workflow of the major test-related decisions (diamond boxes) and actions (rounded rectangles) to be made when propagating additive and reductive changes. Subsection IV-A describes the workflow for additive changes, while Subsection IV-B pertains to reductive changes. Steps that are common to both types of changes are described in Subsection IV-C. Note that mutative changes are not included within the scope of this paper but will be addressed in future work.

A. Handling Additive Changes

Additive changes introduce new component interfaces and implementations into the system at runtime. The unshaded nodes in Figure 2 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

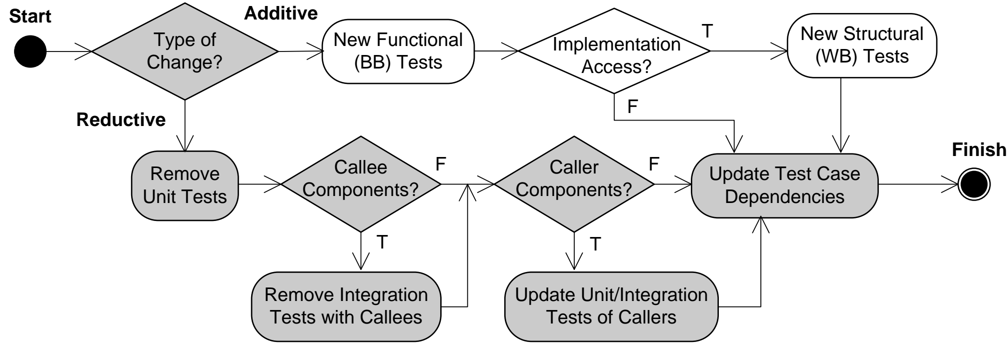

Figure 2. Major decisions and actions for updating runtime test models after additive and reductive changes

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 component's 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.

B. Handling Reductive Changes

Reductive changes remove existing component interfaces and their implementations from the system at runtime. The shaded nodes in Figure 2 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.

C. Considering Test Case Dependencies

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.

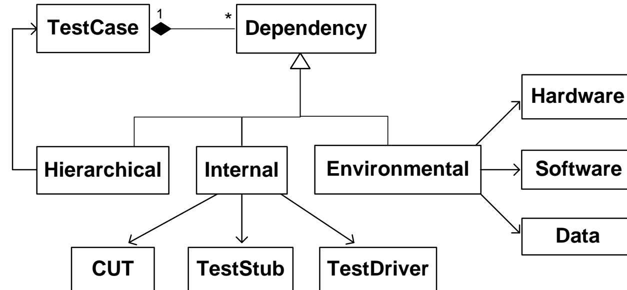

Figure 3. A meta-model to support test information propagation in self-adaptive systems

Figure 3 provides a meta-model showing the various types of dependencies in a test model for a software system. Such a metamodel 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: (1) *Hierarchical* – other tests that must be executed and pass the test in order for a test to run, (2) *Internal* – entities that are implemented as part of the software; and (3) *Environmental* – entities that are external to the software under test.

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 system's 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.

V. Prototype

In order to demonstrate and evaluate the proposed approach, our investigation involves the development of a prototype of TIP. The initial version of the prototype is focused on assessing the feasibility of automatically propagating reductive changes, i.e., the shaded portion of Figure 2.

A. Application Description

Using the approach by King et al. [6], 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. 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.

B. System Development and Architecture

An application with the features described in the above scenario was developed in Java [20], using the Eclipse IDE [21] and the tools/libraries to support adaptation, testing, and change propagation. As shown at the top-left of Figure 4, we used the Spring Framework [4] to provide a component-based application container for the three major application services. These services were: `EMRServices`, `AppointServices`, and `PharmacyServices`, corresponding to features (a), (b), and (c) from the scenario.

Services were made adaptable using the dynamic language `Groovy` [5] (bottom-left), which allows components to be specified as *beans* within the application container. At

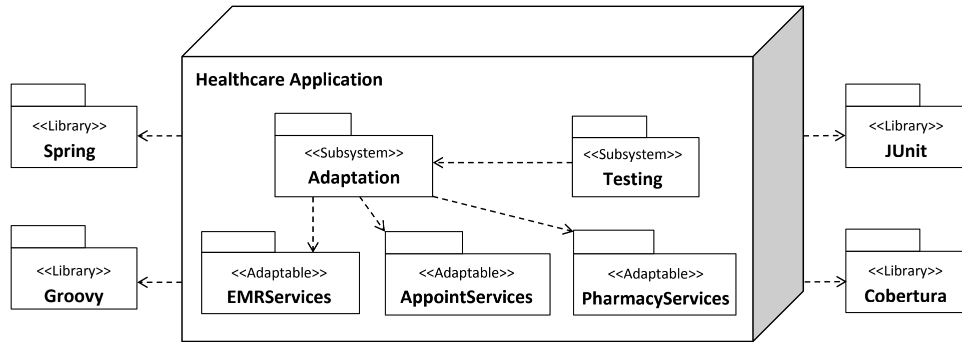

Figure 4. Architecture of Service-Oriented Healthcare Prototype

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 [22] is built into the Groovy runtime, we created automated tests for each bean by scripting JUnit tests in the Groovy syntax. Cobertura [23] 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) [17]. Model instantiation and transformation was achieved using Kermeta [19], which facilitates the programatic manipulation of EMF models (.ecore files). Kermeta therefore provided us with a programming environment with which we could set up our simulation, which is described in Subsection V-D.

C. Detailed Object Design

Figure 5 shows the detailed object design of the `EMRService` in the TIP prototype. Our object design is based on a software requirements specification for EMR data analysis [24], which was elicited from a domain expert as part of a software engineering course project.

As shown at the top-left of Figure 5, 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.

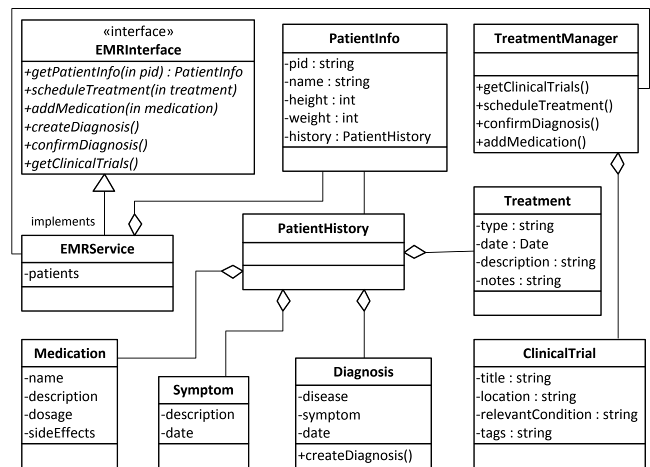

Figure 5. Design of EMRService (based on data analysis spec. [24])

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 [5].

D. Simulation Environment

We performed a simulation to propagate a reductive change in the EMR service implementation to its associated test implementation. To set up the simulation, we used Kermeta [19] to specify and instantiate two models associated with the EMR service: a *component model* and a *test model*.

The EMR component model was initialized with the names of the classes within the subsystem, and the dependency relationships among them. Both the class names

and dependency relationships were captured automatically using JDepend [25]. The EMR test model conformed to the structure of the meta-model defined in Figure 3 but, due to lack of freely available test management tools, had to be initialized manually with the EMR test information. Traceability relationships were handled through naming conventions, which we elaborate on as part of the lessons learned in Subsection V-F.

We used Kermeta [19] to simulate a reductive change in the EMR service related to the clinical trials feature. This was achieved by creating and applying a transformation to the EMR component model that removed the `ClinicalTrial` class, 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.

E. Results

Table II summarizes the actions that were generated by our change propagation simulation. The first action identified removal of the unit test associated with the `ClinicalTrial` class. Since this class did not invoke any other components, it was not necessary to update any callee-related aspects of the test model. The second action indicated that the stub component `TrialsDBSimulator`, which mimics the behavior of clinical trials database, could be removed. This stub was no longer needed because the only test driver that depended on it, `UT007-ClinicalTrial`, was pruned from the model by the first action. Lastly, the third action identified that updates should be made to the drivers `UT003-TreatmentManager` and `IT001-EMRService`, which both reference the `ClinicalTrial` class in their test code.

Action	Artifact Type	Source File / Location
1. Remove	Unit Test	UT007-ClinicalTrial.java
2. Remove	Test Stub	TrialsDBSimulator.java
3. Update	Unit Test	UT003-TreatmentManager.java
	Integration Test	IT001-EMRService.java

Table II
TEST MODEL UPDATES FOR REDUCTIVE CHANGE SIMULATION

F. Discussion

Developing the prototype provided us with much insight into the complexity of implementing an automated solution to the research problem. Although our reductive example of the EMR service was not a very complex scenario 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 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.

Our simulation results and experience also revealed the central role of the meta-model in enabling change propagation. While we were able to successfully automate the first two actions in Table II, this was not possible for the third action. This is because our meta-model did not provide us with sufficient details on the structure of the test sets to be able to remove the appropriate references. Furthermore, even if the structural information had been maintained, there was no way for us to guarantee that the test would still be meaningful if the class references were taken out.

A major lesson learned 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. In addition, test IDs were used in drivers and stubs so that these dependencies could be easily located.

VI. RELATED WORK

As mentioned in the summary of our preliminary investigation, no works that address the problem of synchronizing runtime test models in self-adaptive software were found in the literature. To the best of our knowledge, the approach presented in this paper is the first attempt at tackling the research problem under investigation.

The work by Xiong et al. [15] is closely related to our work. They propose an approach that automatically synchronizes two models related by transformations described in Atlas Transformation Language (ATL) [18]. 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.

Chechik et al. [26] 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. 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. [26], who reason that propagating changes between models at different levels of abstraction is impossible.

Hassan and Holt [27] 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. 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.

VII. Conclusion

This paper presented a model-driven approach that synchronizes runtime tests 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. The prototype simulated a reductive change to the electronic medical record service of an adaptive healthcare application. Conducting the simulation provided many insights into the complexity of the research problem. Future work calls for performing controlled experiments to evaluate the approach, and extending the prototype to include additive and mutative changes.

Acknowledgments

The authors would like to thank Dr. Gursimran Walia, and members of NDSU Spring 2010 course on Empirical Software Engineering for their contributions to this work.

References

[1] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Trans. Auton. Adapt. Syst.*, vol. 4, no. 2, pp. 1–42, 2009.

[2] J. Kephart and D. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–52, January 2003.

[3] J. Zhang, B. H. C. Cheng, Z. Yang, and P. K. McKinley, "Enabling safe dynamic component-based software adaptation." in *WADS*, 2004, pp. 194–211.

[4] C. Walls and R. Breidenbach, *Spring in Action*. Greenwich, CT, USA: Manning Publications Co., 2005.

[5] D. Koenig, A. Glover, P. King, G. Laforge, and J. Skeet, *Groovy in Action*. Greenwich, CT, USA: Manning Publications Co., 2007.

[6] T. M. King, A. E. Ramirez, R. Cruz, and P. J. Clarke, "An integrated self-testing framework for autonomic computing systems," *JCP*, vol. 2, no. 9, pp. 37–49, 2007.

[7] J. Zhang, H. J. Goldsby, and B. H. Cheng, "Modular verification of dynamically adaptive systems," in *AOSD '09*. New York, NY, USA: ACM, 2009, pp. 161–172.

[8] L. Tratt, "A change propagating model transformation language," *Journal of Object Technology*, vol. 7, no. 3, pp. 107–126, March 2008.

[9] IEEE Computer Society, "Std 610.12-1990(r2002): Glossary of software engineering terms," Tech. Rep., 2002.

[10] B. Beizer, *Software Testing Techniques*, 2nd ed. New York: Van Nostrand Reinhold, 1990.

[11] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit testing coverage and adequacy," *ACM Computing Surveys*, vol. 29, no. 4, pp. 366–427, December 1997.

[12] I. Burnstein, *Practical Software Testing: A Process-Oriented Approach*. New York, NY, USA: Springer-Verlag, 2003.

[13] R. France and B. Rumpe, "Model-driven development of complex software: A research roadmap," in *FOSE '07*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 37–54.

[14] C. Atkinson and T. Kühne, "Model-driven development: A metamodeling foundation," *IEEE Softw.*, vol. 20, no. 5, pp. 36–41, 2003.

[15] Y. Xiong, D. Liu, Z. Hu, H. Zhao, M. Takeichi, and H. Mei, "Towards automatic model synchronization from model transformations," in *ASE '07*. New York, NY, USA: ACM, 2007, pp. 164–173.

[16] M. Akour, G. Walia, and T. M. King, "A systematic review of runtime test case synchronization in adaptive software," NDSU Dept. of Computer Science, Tech. Rep., May 2010. [Online]. Available: http://cs.ndsu.edu/research/reports.htm

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

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

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

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

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

[22] E. Gamma and K. Beck, "JUnit 3.8.1," 2005, http://www.junit.org/index.htm (July 2009).

[23] M. Doliner, G. Lukasik, and J. Thomerson, "Cobertura 1.9," 2002, http://cobertura.sourceforge.net/ (July 2009).

[24] J. Drallos, J. Clare, J. Korolewicz, D. Laboy, and B. H. Cheng, "SRS: EMR Data Analysis," Michigan State University, Tech. Rep., 2009, http://www.cse.msu.edu/~cse435/Projects/F09/EMR-Analysis/web/ (Nov 2010).

[25] Clarkware Consulting Inc., "JDepend 2.9," 2009, http://www.clarkware.com/software/JDepend.html (July 2009).

[26] M. Chechik, W. Lai, S. Nejati, J. Cabot, Z. Diskin, S. Easterbrook, M. Sabetzadeh, and R. Salay, "Relationship-based change propagation: A case study," in *MISE '09*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 7–12.

[27] A. E. Hassan and R. C. Holt, "Predicting change propagation in software systems," *Software Maintenance, IEEE International Conference on*, vol. 0, pp. 284–293, 2004.