2016 IEEE 17th International Symposium on High Assurance Systems Engineering

Using Models to Validate Unanticipated, Fine-Grained Adaptations at Runtime

Mohammed Al-Refai
Computer Science Department
Colorado State University, USA
Email: al-refai@cs.colostate.edu

Walter Cazzola
Department of Computer Science
Università degli Studi di Milano
Milan, Italy
Email: cazzola@di.unimi.it

Sudipto Ghosh and Robert France
Computer Science Department
Colorado State University, USA
Email: {ghosh,france}@cs.colostate.edu

***Abstract*—An increasing number of modern software systems need to be adapted at runtime while they are still executing. It becomes crucial to validate each adaptation before it is deployed to the running system. Models are used to ease software maintenance and can, therefore, be used to manage dynamic software adaptations. For example, models are used to manage coarse-grained anticipated adaptations for self-adaptive systems. However, the need for both fine-grained and unanticipated adaptations is becoming increasingly common, and their validation is also becoming more crucial.**

This paper proposes an approach to validate unanticipated, fine-grained adaptations performed on models before the adaptations are deployed into the running system. The proposed approach exploits model execution where model representations of the test suites of a software system are executed. The proposed approach is demonstrated and evaluated within the Fine Grained Adaptation (FiGA) framework.

***Index Terms*—model-based validation, model-based dynamic adaptation, executable UML models, system validation, software analysis and visualization, unanticipated adaptation.**

I. Introduction

The ability to perform runtime adaptations is becoming a requirement for many critical software systems. These systems need to be adapted without being stopped. For example, intelligent transportation systems must provide continuous services to human and software clients for safety reasons. Online gaming systems with thousands of users interacting with each other at any given time cannot be stopped for economic or business reasons.

Runtime adaptation is needed based on changes in a software system context and requirements. These changes can be foreseen at design time, or may be unforeseen and disclosed only at runtime. Anticipated adaptations of the foreseen changes are prepared during development and included in the system design. Unfortunately, this approach cannot support unforeseen changes. When such changes are needed at runtime, unanticipated adaptations must be planned and deployed into the running system. In both cases, the dynamic adaptation process is complex, and models can be used to manage this complexity by representing aspects of a running system at a higher abstraction level [1].

Models@RunTime (M@RT) research [2] uses component-based models to support dynamic adaptation in autonomous systems [3], [4]. These approaches support coarse-grained and anticipated adaptations that are controlled and automated by the MAPE feedback loop in autonomous systems [5]. Adaptations in these approaches are restricted to adding/removing components and links between components. Unanticipated adaptations require human intervention to implement them. They can involve code changes, and can be fine-grained at the level of classes, methods, and statements.

Models that provide a fine-grained view of the running system and its implementation (e.g., activity models) can be used by developers to plan fine-grained adaptations to support unforeseen changes. For example, the *fine grained adaptation* (FiGA) framework supports unanticipated and fine-grained adaptation of a running system through model adaptation [7], [8]. These models can be used to ease system maintenance by minimizing the interventions performed directly on the code. For example, a developer can use a class model to understand static relations between objects and make changes in object attributes and references, and an activity model to make changes to the system behavior [7].

Using design time models to describe anticipated adaptations enables developers to reason about and validate these models during development time [6]. Since unanticipated adaptation is not planned *a priori*, it needs to be validated at runtime before its deployment to the running system.

In this work, we propose a new model-based validation approach that uses test models to validate unanticipated and fine-grained adaptations at the model level. The proposed approach consists of three steps: (1) test models are generated from the original test suites of the running software system that will be adapted. (2) Developers adapt these test models (and create new test models) to specify new test configurations and assertions if the adaptation to the models of the software system introduces new behavior or changes existing behavior. (3) The models representing the test cases are executed to validate the adapted models of the software system before the adaptations are deployed to the running software system. The proposed validation approach is integrated with the (FiGA) framework [7], [8].

Sect. II provides a motivating example and introduces the proposed validation approach. Sect. III describes the FiGA framework with the integration to support validating models at runtime. An explanation of how model execution of JUnit

1530-2059/16 $31.00 © 2016 IEEE
DOI 10.1109/HASE.2016.40

test suites is supported in the FiGA framework is provided in Sect. IV. A demonstration and evaluation of our approach are presented in Sect. V. Related work is presented in Sect. VI, and Sect. VII concludes with plans for further work.

II. Motivating Example and Proposed Solution

It is evident that any human activity is potentially error-prone. This naturally includes the process of planning the adaptation of a running system using a model abstraction. For systems that cannot be stopped, the adaptation must be deployed immediately after the planning process. This results in the need to validate the adaptation before its deployment to avoid potential failures. To demonstrate the critical need for validation, we show in Sect. II-A a small scenario where the adaptation of a model introduces some logic faults and how these propagate to the code. Section II-B gives a general overview of how we could approach this problem.

A. Motivating Example

*Airline reservation system*s (ARS) are a typical example of highly-available systems used by thousands of users everyday that for economic reasons, cannot be stopped to permit any kind of maintenance. As the name suggests, ARSs implement features related to flight reservations. A flight is characterized by the airline, the departure and destination airports, and its time schedule. Each customer can look for a flight and reserve a seat on it. In our design, a `SystemManager` class provides access to all the ARS functionality; including the access to airport, airline, and flight instances.

Runtime adaptation to ARS. The first version of our ARS implementation has a limitation in its *flight reservation* functionality: a search for a trip between two airports provides only direct flights, if any. Clearly this means that when the two airports are not connected by a direct flight, the reservation functionality fails even though the two airports may be connected through another airport.

The existing functionality has to be adapted to deal with such a limitation. The adaptation affects some methods including the `Airline::bookSeat()` method that implements the flight reservation functionality. Given the departure and destination airports, and some other information about the travel schedule, the `Airline::bookSeat()` method searches for a flight whose departure *and* destination airports coincide with those given. Such a behavior is described by the framed portion of the activity diagram in Fig. 1. The previously described limitation is overcome by looking for flights that have the given airports as departure *or* destination airport and then trying to combine them in a journey from the departure airport to the destination airport with a layover at some intermediate airport. Priority is given to direct flights over flights that involve a layover.

The adapted activity model (Fig. 1) has a logic fault. The *else* transition flow of the decision node ($\texttt{test}_1$) that checks the departure airport is linked to the decision node ($\texttt{test}_2$) that checks the destination airport. As a consequence, the destination airport is not checked for all flights and some

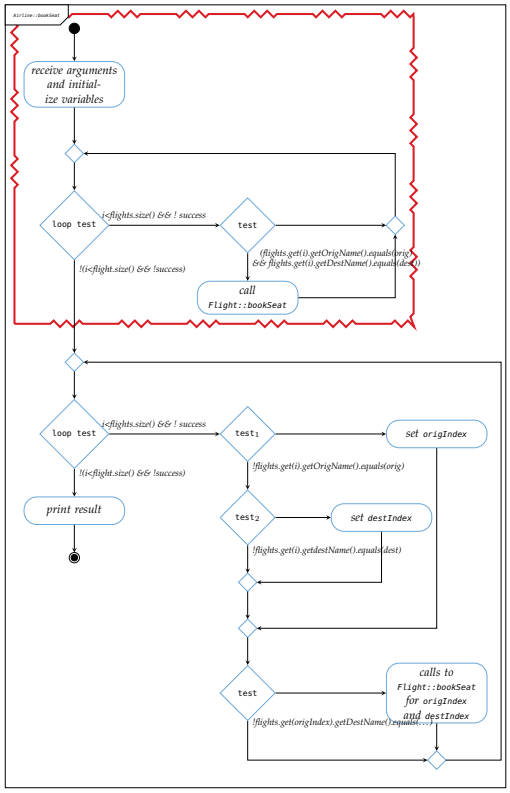

Fig. 1. Adapted activity model for the `Airline::bookSeat()` method.

```
........
for(int i=0; (i<flights.size() && !success) ;i++){
  if(flights.get(i).getOrigName().equals(orig)){
    origIndex = i;
  } else if(flights.get(i).getDestName().equals(dest)){ destIndex = i;}
  if(flights.get(origIndex).getDestName().equals(
              flights.get(destIndex).getOrigName())){
    flights.get(origIndex).bookSeat(row, col, s);
    flights.get(destIndex).bookSeat(row, col, s);
```

Listing 1: Portion of adapted code with logic fault.

connections are still missing from the final result. This logic fault propagates from the adapted model to the generated code that will be used to update the running software. Listing 1 shows the generated code that contains the logic fault. The fault is at line 5, where the IF statement at this line is nested inside the *else* branch of another IF statement.

From the example, it should be clear how faults introduced in the models propagate to the running system. Therefore, validating the adapted behaviors and fixing faults at the model level is crucial for generating well designed and correct code before we can deploy the changes to the running system.

B. Proposed Solution

Unit and system test cases validate the software system at the code-level. While code-level test suites are available for testing the implementation of a software system, model-level test suites are not available for the model representation of

such a system in the current FiGA framework [7], [8]. Reverse engineering can provide a model of the code-level test suites that can be used to validate the models of the system.

The proposed approach assumes that the running software system to be adapted using models comes with its code-level test suites. In the proposed approach, test models are extracted from the code-level test suites, and these test models are used to validate the adapted models of the system. The model representation of the test suites should be adapted when the system adaptation affects the existing functionality because the available test suites can no longer correctly validate the system. The adaptation of the test suites at the model-level includes modifying existing test cases and creating new ones. Model validation takes place by executing the model test suites.

We propose using UML class and activity models to represent, adapt, and execute test suites. Each test case is represented as an activity model. Generally, the major elements of a code-level test case are as follows: (1) object declaration and initialization statements, (2) statements to call methods under test, and (3) assertion statements. These elements are mapped to action nodes in the activity model representing their test case. For example, object declaration and initialization statements are grouped together and represented as an action node. In the code represented as action nodes, calls to methods are mapped to calls to the corresponding activity models representing these methods.

Validation of the adapted models representing the program is done by executing the models representing the test suites. Therefore, activity models of test cases and models of the program must be executable: activity model elements are associated with executable code statements. For example, each outgoing transition flow of a decision node contains a conditional expression, and each action node contains a set of code statements that represent the action node behavior. Action, merge, and decision nodes are executed according to UML semantics. Executable UML activity models are supported by the Rational Software Architect tool[1].

In the motivating example, the adapted activity model in Fig. 1 of the `Airline::bookSeat()` is validated as follows. Executable activity models are extracted from test cases for the `Airline::bookSeat()` method to test the activity model of this method, which is also executable. Developers adapt the activity models of the test cases when needed. The adapted models are executed to validate the adapted activity model and detect faults. Complete details for validating the adapted ARS using the proposed approach are provided in Sect. V-A.

III. Validation Integration into the FiGA Framework

The *Fine-Grained Adaptation* (FiGA) framework [7], [8] allows a developer to adapt a program running on a standard JVM without stopping it by modifying UML models and propagating model changes to the source code. The program change process is kept separate from the running program instance until the changes are ready to be compiled and loaded into the Java virtual machine, so as not to compromise the service provided by the program. The FiGA framework uses the JavAdaptor [9] tool, which can update a running Java program without stopping it. JavAdaptor works at a low level, requiring as an input the compiled version of the class to update and a connection to the Java virtual machine in which the program is executing. Therefore, the program update is driven by changes to the Java source code. As shown in Fig. 2 the FiGA framework supports the adaptation of a running program through a repetitive loop composed of five steps.

Step (1): Model Generation from Program Code. ReverseЯ [10] is used to generate the UML diagrams from the application source code. The source code is annotated, and these annotations drive the model extraction from the running code of the application, which is performed by ReverseЯ. The annotations are introduced only once by developers during development time.

Step (2): Modification of Program Model. Developers change the models to deal with the needed adaptation. Each model change can be expressed as a sequence of elementary model changes (γ_i) that can be easily and automatically mapped to a code change (δ_i). The model changes are determined by model differencing [12], [13], and mapped into calls to the change operators with the proper parameters.

Step (3): Adaptation Process for Program. The sequence of elementary model changes (Γ) and their mappings to program code changes (Δ) are related using the σ mapping function. Therefore, the following holds:

$$M_1 = M_0 \boxplus \Gamma \equiv S_0 \boxplus \sigma(\Gamma) = S_0 \boxplus \Delta = S_1$$

This relationship is used by the FiGA framework to adapt the source code according to the model changes. Complete details of the adaptation semantics are given in Cazzola *et al.* [7], [8].

Step (4): Propagating Changes to the Program. Model changes are not propagated to the program after every elementary operation. To ensure a consistent adaptation, its deployment to the running program is triggered by the developer only when all required model changes are performed.

Step (5): Updating the Running Application. Finally, the modified Java classes are passed to the JavAdaptor tool, which deploys the program changes to the running application without stopping it [9]. At this point, future adaptations occur using the adaptation cycle starting again from step 1 but with the new source code instead.

The validation approach presented in Sect. II can be smoothly integrated in the adaptation loop for the FiGA framework. The same process is used (1) to extract the models from the code for the test cases, (2) to adapt the models for the test cases, and (3) to maintain models for the test cases synchronized with their code. Only one new step needs to be added: *model validation*. To summarize, the validation process consists of five further steps. The steps labeled with a prime are not new steps; they are an application of the corresponding FiGA steps to a different code portion, i.e., the test cases.

[1] http://www-03.ibm.com/software/products/en/ratisoftarchsimutool

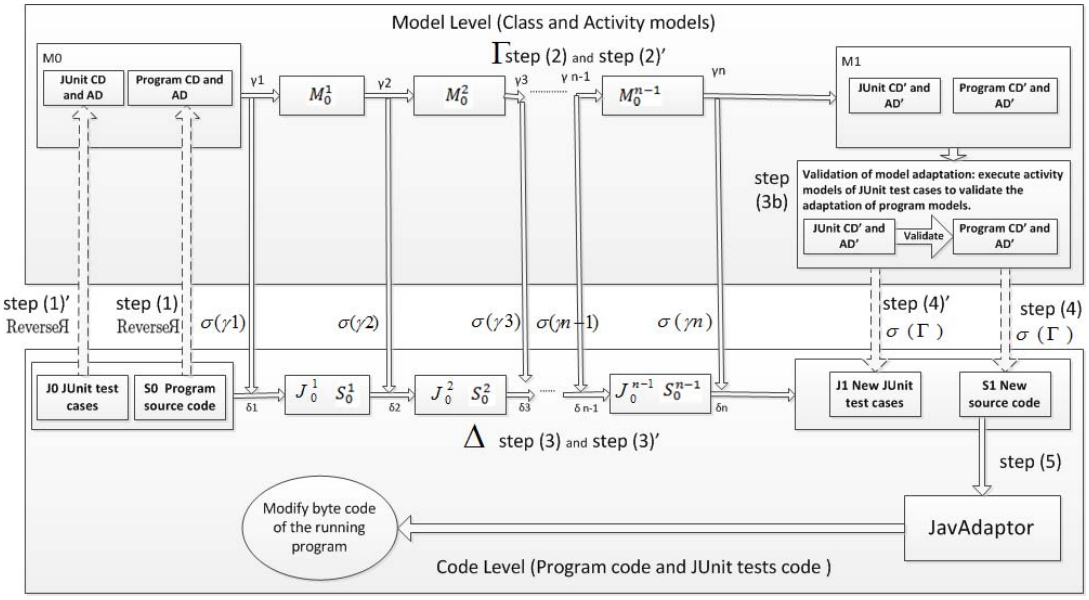

Fig. 2. Overview of the Extended FiGA Approach.

Step (1)': Model Extraction from Test Suites. ReverseЯ extracts UML class and activity models from the application test suites. The test suite code is annotated, and these annotations drive the model extraction performed by ReverseЯ. The annotations are introduced in the test suites only once by developers during development time. Each individual test case is mapped into an activity model. In the experiment we considered JUnit test suites.

Step (2)': Modification of Test Suites Model. The developer adapts the models for the test suites. As for the program, any model change is decomposed into a sequence of elementary changes that are mapped to code changes.

Step (3)': Adaptation Process for Test Suites. The code for the test suites is adapted as described in Step (3).

Step (3b): Validation of Adapted Program Model. The validation process consists of the execution of the models for the test suites against the adapted models for the program. If the execution of some test model fails revealing some faults, the adaptation process is stopped and the developer has to modify the models of the program to fix the faults (i.e., goes back to step 2 and 2'). The validation process is completed when all the models for the test cases pass, after which the adaptation can be deployed.

Step (4)': Propagating Changes to the JUnit Test Suites. Similar to the program code, the code for test suites is also kept synchronized with their models.

IV. Enabling Model Execution

The *Rational software architect* (RSA) simulation toolkit 9.0[2] is used to execute the activity models. It supports model execution via the UML action language. Action code can be associated with action nodes and transition flows of activity models. Each action node has a code snippet associated with it where code statements can be entered, and each transition flow has a code snippet associated with it where a boolean expression can be entered [14]. We added the Java and JUnit libraries to the tool and enabled model execution using Java as the action language instead of using the UML action language. Other libraries can be added and used at the model level in a similar way to the addition of the JUnit library.

In the original FiGA framework [7], [8], the annotated code was mapped to comments associated with action nodes, which permitted changing the code at the model level but not its execution. In order to exploit the RSA model simulation tool, we extended the FiGA framework to support (1) mapping Java statements to code snippets of action nodes and transition flows (Sect. IV-A), and (2) mapping calls between methods to calls between the activity models representing these methods (Sect. IV-B). These new mappings are supported in the extended FiGA framework presented in this paper.

A. Mapping Java Statements to Code Snippets

In the extended FiGA framework, ReverseЯ maps annotated Java statements to code snippets of action nodes and transition flows of decision nodes.

Action nodes. If Java statements are annotated with the `@CallAction` annotation, then ReverseЯ creates an action node and adds the Java statements to the code snippet of the action node. When the model execution flow reaches an action node, the code snippet associated with the node is executed. For example, the activity model in Fig. 3 is extracted from the annotated test case in Listing 2. The `@CallAction` annotation

[2]http://www-03.ibm.com/software/products/en/ratisoftarchsimutool

```
@Test
public void test_bookSeat() {
 @CallAction(id="Declare and initialize objects"){
    Airport Den = new Airport("DEN");
    Airport Dtw = new Airport("DTW");
    Airline Delta = new Airline("DELTA");
    ArrayList<Airport> airports =
          new ArrayList<Airport>(Arrays.asList(Den, Dtw));
    ArrayList<Airline> airlines =
          new ArrayList<Airline>(Arrays.asList(Delta));
    SystemManager reserve = new SystemManager(airports, airlines);
 }

 @CallAction(id="Method calls"){
   @CallBehavior(Behavior="SystemManager::createFlight"){
    reserve.createFlight("DELTA", "DEN", "DTW", 2013, 5, 6, "123");}
   @CallBehavior(Behavior="SystemManager::createSection"){
    reserve.createSection("DELTA","123", 1, 1, SeatClass.business);}
   @CallBehavior(Behavior="SystemManager::bookSeat"){
    reserve.bookSeat("DELTA", "DEN", "DTW", SeatClass.business);}
 }

 @CallAction(id="Assertions"){
    assertEquals(reserve.findAvailableFlights("DEN", "DTW"), false);
 }
}
```

Listing 2: Test case before adaptation: reserving a direct flight.

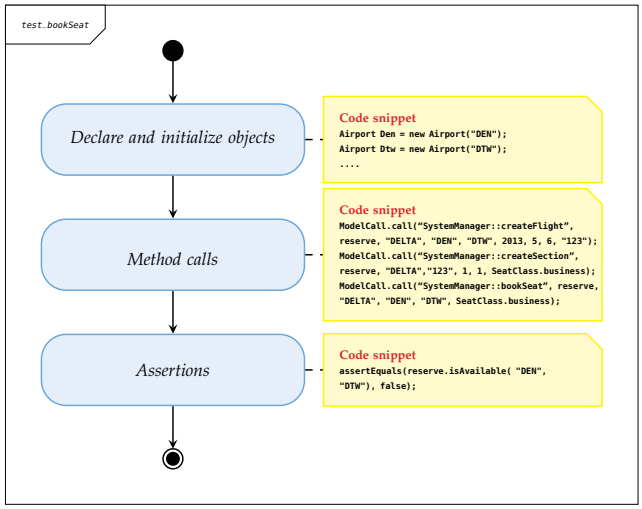

Fig. 3. An activity model extracted from the `test_bookSeat()` test case.

at line 3 decorating the statements commands ReverseЯ to create an action node labeled with `"Declare and initialize objects"` (cf. Fig. 3). The annotated statements (from line 4 to 11) are copied to the code snippet associated with the created action node.

Decision nodes. A Java conditional statement that is annotated with the `@Decision` annotation is mapped to a decision node with outgoing transition flows. The boolean expression of the conditional statement and the negation of the expression are represented as code snippets of the outgoing transition flows. When the model execution flow reaches a decision node, its outgoing transition flows are evaluated and the transition that evaluates to `True` is executed. If developers think that visualizing branches of a conditional statement in the model is not important, then they can annotate the statement with the `@CallAction`. In this case, the conditional statement and its body are copied to a code snippet of an action node, and executed as already explained.

B. Mapping Method Calls to Activity Model Calls

In the extended FiGA framework, formal input arguments of a Java method are represented as local model attributes in the activity model extracted from that method. Additionally, ReverseЯ maps method call statements that are annotated with `@CallBehavior` to calls to activity models representing the called methods. For example, the annotated method call in line 16 of Listing 2 is mapped to a call to the activity model of the `SystemManager::createFlight()` method. In Listing 2, the value of the `Behavior` attribute is the `id` of the called activity model. The corresponding call statement in the activity model of Fig. 3 is as follows:

```
ModelCall.call("SystemManager::createFlight", reserve, "DELTA",
                "DEN", "DTW", 2013, 5, 6, "123")
```

where `ModelCall` is a static class that we implemented. The `call()` method takes as inputs the `id` of the called activity model, the caller object, and the actual arguments of the call statement. The `call()` method contains statements to call the necessary methods from the RSA model simulation library to execute the called activity model.

The RSA model simulation tool does not support passing arguments between activity model calls (as explained in [14] on page 29). Therefore, we implemented this functionality in the `ModelCall` class (within the extended FiGA framework). When an activity model is called for execution by the `ModelCall.call()` method, the called activity model receives the actual arguments from the `ModelCall` class and saves them in its local attributes.

Activity models are only generated from annotated methods. During development time, developers have the choice not to annotate some of the program methods (e.g., constructors, getter, setter, `toString`, and helper methods) if they think that generating activity models for these methods is not important and would just increase the complexity of the software system's full model. In this case, no activity models are extracted for such methods. Signatures of such methods are stored in the class model, and developers can modify their implementation in the class model (e.g., clicking on an operation shows its code view where the implementation can be modified). These methods can be called from activity models, i.e., a statement to call such a method is contained in a code snippet of an action node.

V. Evaluation and Discussion

This section consists of two parts: a demonstration of the validation approach, and a mutation testing experimental study.

A. Demonstration of the validation approach

We demonstrated the validation approach when integrated with the FiGA framework using the ARS example introduced in Sect. II-A and the NanoXML/Java[3] parser. The goal of the

[3]http://nanoxml.sourceforge.net/orig/

TABLE I
ORIGINAL ARS AND NANOXML 2.0, AND THEIR BASELINE TEST CASES

Software System	Number of classes	Number of methods	LOC	Number of baseline tests	Number of activity models of methods	Number of activity models of test cases
ARS	9	58	721	61	43	61
NanoXML	19	147	5827	68	59	68

TABLE II
ADAPTED MODELS OF ARS AND NANOXML 2.0, AND THEIR TEST CASES

Software System	Change type	Classes	Activity models of methods	Activity models of test cases
ARS	*Added*	0	5	8
	Deleted	0	0	0
	Modified	4	14	20
NanoXML	*Added*	2	16	20
	Deleted	0	0	0
	Modified	6	13	0

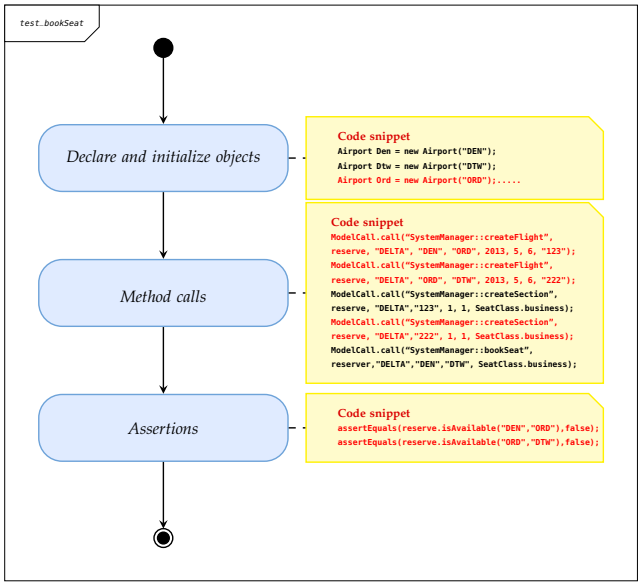

Fig. 4. Adapted activity model to test connecting flights.

demonstration is to show that the proposed validation approach can be used to adapt test cases at the model level, and can validate adapted models of a program and detect failures.

The ARS study: The ARS comes with a JUnit test suite that contains unit and system test cases. The test suite achieves 100% branch coverage. The ARS is dynamically adapted to support (1) reservation on flights with stops as described in Sect. II-A, and (2) the automatic re-booking for passengers whose flights have been canceled.

The class and activity models were generated via ReverseЯ from the ARS and its JUnit test suite. Table I summarizes the data about the original ARS and its generated models. For example, 43 of ARS methods were extracted as activity models (the rest are helper methods represented only in the class model). An activity model was extracted for each test case. For example, Fig. 3 shows the extracted activity model from the annotated test case shown in Listing 2.

The activity models of the ARS and its test cases were modified according to the considered adaptations. Table II summarizes data about the adapted models. For example, the activity model of Fig. 3 tests the functionality of reserving a seat on a direct flight. This activity model was adapted to test the same functionality in case of connecting flights. Fig. 4 shows the adapted activity model, where some initializations were added to the code snippet of the action node labeled with `"Declare and initialize objects"`. New calls to the activity models responsible for creating flights and seat sections were added and assertion statements were also modified. The added/modified statements are shown in red in Fig. 4.

The activity models (both adapted and unchanged) for test cases were executed to validate the adapted ARS models. Faults unintentionally introduced during the adaptation process (16 faults) were disclosed by the test cases. For example, the activity model in Fig. 4 was executed to validate the reservation functionality for connecting flights. The statement

```
ModelCall.call("SystemManager::bookSeat",
reserver,"DELTA","DEN","DTW", SeatClass.business)
```

passes the execution flow to the activity model for the `SystemManager::bookSeat()` method which, in turn, executes the faulty activity model (shown in Fig. 1) of the `Airline::bookSeat()` method. The execution of this activity model violates the assertions of the test case revealing the logic fault (presented in Sect. II-A) in the activity model.

The NanoXML study: The NanoXML is an XML parser for Java. Multiple versions of NanoXML are available[4], each adding new functionality to the previous version. For example, NanoXML 2.1 supports parsing XML namespaces that are not supported in version 2.0. We chose NanoXML in order to demonstrate the proposed approach on software that was implemented by a third party with a size larger than the ARS size, and to show that the adaptation and its validation are feasible at the model level when a NanoXML version is adapted to another version at the model level.

The NanoXML package provides XML files to be used as inputs to test the NanoXML software. Each file contains XML data with different properties than XML data in other files. We created 68 code level test cases for NanoXML 2.0 such that each test case contains (1) statements to read and parse an input XML file, and (2) assertion statements to check the properties of the results.

We used the extended FiGA framework to apply model level adaptation from NanoXML version 2.0 to version 2.1 to support parsing XML files with namespaces. The class and activity models were extracted from NanoXML 2.0 and its test cases. Table I summarizes the data about NanoXML 2.0

[4]http://sir.unl.edu/portal/bios/nanoxml.php

and its models. Among the 147 methods, 59 methods were extracted as activity models. The rest are helper methods that are represented and can be adapted in the class model as explained in Sect. IV-B.

Based on code differences between NanoXML version 2.0 and 2.1, the class and activity models of version 2.0 were adapted to get to version 2.1 at the model level. Table II summarizes the data about the adaptation. New activity models (20 models) were added for creating test cases that tests parsing XML files with namespaces. After the adaptation process, the 88 activity models representing the test cases (20+68) were executed to validate the adapted models representing version 2.1. Faults unintentionally introduced during the adaptation process (18 faults) were disclosed by the test cases.

The ARS and NanoXML studies showed that modifying existing test cases and creating new ones were feasible at the model level, and executing these test cases disclosed faults unintentionally introduced in the adapted models. The studies also showed that the process for model-based adaptation within the FiGA framework works correctly.

B. Mutation Testing Experiment

The goal of the mutation testing experiment is to compare the effectiveness of testing at the model level using the proposed approach with the effectiveness of testing at the code level. The effectiveness is measured by the number of faults that can be found at the model and at the code levels.

We applied *mutation testing* [15] in this experiment. Mutation testing is a fault-based testing technique that measures the effectiveness of test cases. Mutation testing uses mutation operators to apply syntactic changes in a program to introduce simple faults and create a set of faulty versions of the program, called *mutants*. Test cases of the original program are used to execute these mutants, and if these test cases expose faults in a mutant, then the test cases are said to *have killed the mutant*. The ARS is the subject of this experiment that consists of the following four steps:

First, `MuJava` [16] was used to apply first-order method-level mutation operators to the original ARS program. Different mutants were created for the ARS. The set of all mutated versions is called **P**. The applied mutation operators are: (1) replacement/insertion/deletion of arithmetic/conditional/logical operators, (2) relational operator replacement, (3) assignment operator replacement, (4) statement deletion, and (5) predicates with a boolean constant replacement.

In the second step, the mutation in each p_i of **P** was repeated on a copy of the class model and activity models representing the ARS as follows. For each mutated statement at the code level, we found its corresponding element at the model level and applied the same mutation operator to it. Note that code statements are associated with action nodes and transition flows in activity models. Each mutated copy of the ARS class and activity models based on p_i is called m_i, and the set of all m_i is called **M**.

The baseline JUnit test suite for the ARS is called **PT**, and the set of activity models of the ARS test cases (extracted by ReverseЯ from the ARS baseline JUnit test suite) is called **MT**. No changes were made to the test cases in **MT** and **PT**, so they were not able to kill all mutants generated by the `MuJava`. In the third step, the test cases of **MT** were used to evaluate all mutants in **M**, and the killed mutants were reported. Similarly, the test cases of **PT** were used to evaluate all mutants in **P**, and the killed mutants were reported. The results are shown in Table III.

TABLE III
EXPERIMENTAL RESULTS

Mutations Introduced/Killed	Number
Mutants in **M**	654
Mutants in **P**	654
Mutants killed by **MT**	541
Mutants killed by **PT**	541

The set of faults corresponding to the killed mutants of **M** is called **FM**, and the set of faults corresponding to the killed mutants of **P** is called **FP**. In the fourth step, faults in **FM** were traced to those in **FP** in order to find if the testing at the code level and model level disclosed the same faults, and to verify that each fault in **FM** corresponds to a fault in **FP**. Each fault in **FM** was traced to a fault in **FP** through the FiGA mapping operators (details of these operators are in [7] and [8]) that map a model change to a corresponding code change. If a model fault in **FM** is mapped to a corresponding code fault that is listed in **FP**, then the two faults can be considered similar even though they are represented at different abstraction levels.

Discussion of results. The results of the experiment showed that at the model level, **MT** killed exactly the same set of mutants that were killed by **P** at the code level (541 mutants). We reviewed the remaining 113 mutants that were not killed, and found that they included 81 equivalent mutants that cannot be killed at the model level or code level. Therefore, the mutation score was 541/(654-81)=94.4%. The remaining 32 mutants can be killed by **MT** and **PT** if they are improved by adding more test cases to check additional internal states that are not checked by **MT** and **PT**.

Moreover, by applying the fourth step we found that each fault in **FM** has a similar fault in **FP** but they are represented at different abstraction levels (i.e., a sequence of statements at the code level are grouped as an action node at the model level). Therefore, the sets **FM** and **FP** are isomorph. According to that, the test cases at the model level showed no loss of effectiveness compared to the test cases at the code level. The reason is that the same set of faults existed at two different abstraction levels and the test cases at both levels disclosed these related faults.

The FiGA mapping operators were used to trace all faults in **FM** to faults in **FP** and verify that with model level testing we can disclose the same set of faults that can be disclosed with code level testing. For example, the fault explained in Sect. II-A can be traced from the model to the code as follows. The test case of Fig. 4 failed when executed on the ARS models, and the fault causing the failure was detected in the

activity model of Fig. 1. The fault was the addition of a new decision node ($\texttt{test}_2$) and connecting the *else* transition flow of the decision node ($\texttt{test}_1$) to it. This fault was mapped to a corresponding code using the FiGA mapping operators that specify the corresponding code change and determine its location in the program (in which method and after which statement). The code corresponding to the fault disclosed at the model level is the one at line 5 in Listing 1. If the code level testing disclosed a fault in such a piece of code, then the two faults can be considered the same.

VI. Related Work

Most of the existing model-based approaches for runtime adaptation support coarse-grained reconfiguration of software structure [3], [4], [17]. These approaches support anticipated adaptations. Morin *et al.* [4] use a runtime model that conforms to a user defined meta-model. Architectural invariants are anticipated and defined during design time on this meta-model, and used to check the validity of every constructed configuration during runtime. Garlan *et al.* [3] use a runtime model that maintains explicit architectural constraints at design time, and at runtime, these constraints are periodically checked to make sure that they are not violated by model adaptations. Song *et al.* [17] apply structural model differencing to check the syntactic validity of architectural changes to a system.

Amoui *et al.* [18] proposed an approach for fine-grained adaptations at runtime. This approach utilizes TGraphs to represent runtime models for self-adaptive systems, and it supports fine-grained adaptation at the method and field level. However, this approach does not support or address unanticipated adaptations and their validation at runtime. Other approaches use models during development time to validate all possible configurations of an adaptive system, such as Zhang *et al.* [6]. However, enumerating adaptation rules and system configurations is not possible for unforeseen changes, and thus, unanticipated adaptations are not tested in these approaches. Piechnick *et al.* [19] proposed an approach for applying unanticipated adaptations such that new component types and adaptation plans can be introduced at runtime. However, this work does not propose a technique for validation.

VII. Conclusions and Future Work

We presented a new approach to validate model-based fine-grained and unanticipated adaptations of running Java software systems. The approach validates the functional correctness of UML class and activity models representing fine-grained behaviors of a program. Test cases are represented as activity models, and they are executed to validate runtime adaptations at the model level. The approach was demonstrated and evaluated within the FiGA framework. The results indicated that the approach worked at the model level and showed no loss of effectiveness compared to code level testing, and detected the same failures that were detected at the code level.

We plan to evaluate the validation approach on different software systems and different fault types to investigate the effectiveness and efficiency of the approach. We will extend the validation approach to incorporate a model-based regression test selection technique. We also plan to support automatic annotation of Java code. Currently, annotations are introduced by developers during development time.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant No. CNS 1305381.

We list Robert France as an author because he was involved with this work at its inception. Unfortunately, he passed away in February, 2015.

References

[1] R. B. France and B. Rumpe, "Model-Driven Development of Complex Software: A Research Roadmap," in *Proc. of FoSE'07*, Minneapolis, USA: IEEE, May 2007, pp. 37–54.

[2] G. S. Blair, N. Bencomo, and R. B. France, "Models@run.time," *IEEE Computer*, vol. 42, no. 10, pp. 22–27, Oct. 2009.

[3] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-Based Self Adaptation with Reusable Infrastructure," *IEEE Computer*, vol. 37, no. 10, pp. 46–54, Oct. 2004.

[4] B. Morin, O. Barais, J.-M. Jézéquel, F. Fleurey, and A. Solberg, "Models@Run.time to Support Dynamic Adaptation," *IEEE Computer*, vol. 42, no. 10, pp. 44–51, Oct. 2009.

[5] J. O. Kephart and D. M. Chess, "The Vision of Autonomic Computing," *IEEE Computer*, vol. 36, no. 1, pp. 41–50, Jan. 2003.

[6] J. Zhang and B. H. C. Cheng, "Model-Based Development of Dynamically Adaptive Software," in *Proc. of ICSE'06*. Shanghai, China: ACM, May 2006, pp. 371–380.

[7] W. Cazzola, N. A. Rossini, P. Bennett, S. Pradeep Mandalaparty, and R. B. France, "Fine-Grained Semi-Automated Runtime Evolution," in *MODELS@Run-Time*, LNCS 8378, Springer, Aug. 2014, pp. 237–258.

[8] W. Cazzola, N. A. Rossini, M. Al-Refai, and R. B. France, "Fine-Grained Software Evolution using UML Activity and Class Models," in *Proc. of MODELS'13*, LNCS 8107, Miami, FL, USA: Springer, Oct. 2013, pp. 271–286.

[9] M. Pukall, C. Kästner, W. Cazzola, S. Götz, A. Grebhahn, R. Schöter, and G. Saake, "JavAdaptor—Flexible Runtime Updates of Java Applications," *Software—Practice and Experience*, vol. 43, no. 2, pp. 153–185, Feb. 2013.

[10] W. Cazzola, S. Pini, A. Ghoneim, and G. Saake, "Co-Evolving Application Code and Design Models by Exploiting Meta-Data," in *Proc. of SAC'07*. Seoul, South Korea: ACM Press, Mar. 2007, pp. 1275–1279.

[11] G. C. Murphy, D. Notkin, and K. J. Sullivan, "Software Reflexion Models: Bridging the Gap between Design and Implementation," *IEEE Transactions on Software Engineering*, vol. 27, no. 4, pp. 364–380, Apr. 2001.

[12] Z. Xing and E. Stroulia, "Differencing Logical UML Models," *Automated Software Engineering*, vol. 14, no. 2, pp. 215–259, Jun. 2007.

[13] S. Maoz, J. O. Ringert, and B. Rumpe, "ADDiff: Semantic Differencing for Activity Diagrams," in *Proc. of ESEC/FSE'11*, Szeged, Hungary: ACM, Sep. 2011, pp. 179–189.

[14] E. Anders, *Model Simulation in Rational System Architect: Activity Simulation*, IBM, 2010.

[15] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," *IEEE Trans. Softw. Eng.*, vol. 37, no. 5, pp. 649–678, Sep. 2011.

[16] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "MuJava: An Automated Class Mutation System," *Software Testing, Verification & Reliability*, vol. 15, no. 2, pp. 97–133, Jun. 2005.

[17] H. Song, G. Huang, F. Chauvel, Y. Xiong, Z. Hu, Y. Sun, and H. Mei, "Supporting Runtime Software Architecture: A Bidirectional-Transformation-Based Approach," *J. of Systems and Software*, vol. 84, no. 5, pp. 711–723, May 2011.

[18] M. Amoui, M. Derakhshanmanesh, J. Ebert, and L. Tahvildari, "Achieving Dynamic Adaptation Via Management and Interpretation of Runtime Models," *J. of Syst. & Softw.*, vol. 85, no. 12, pp. 2720–2737, Dec. 2012.

[19] C. Piechnick, S. Richly, S. Götz, C. Wilke, and Aßmann, "Using Role-Based Composition to Support Unanticipated, Dynamic Adaptation," in *Proc. of ADAPTIVE'12*, Nice, France: IARIA, Jul. 2012.