Assertion-Based Test Oracles for Home Automation Systems[*]

Ajitha Rajan, Lydie du Bousquet, Yves Ledru, German Vega, Jean-Luc Richier
Laboratoire d'Informatique de Grenoble (LIG), Grenoble, France
{ajitha.rajan, lydie.du-bousquet, yves.ledru, german.vega, jean-luc.richier}@imag.fr

ABSTRACT

The Home Automation System (HAS) is a service-oriented application that facilitates the automation of a private home to improve the comfort and security of its residents. HAS is implemented using a service-oriented architecture. Many of the services in the HAS dynamically change their configuration during run-time. This occurs due to change in availability and bindings between services. Dynamic reconfigurations of services in the HAS presents several testing challenges, one being the specification of test oracles. In this paper, we give an approach for specifying test oracles for services in the HAS. We formally specify test oracles in the JML specification language. To verify service behavior in the presence of dynamic reconfigurations, we use mechanisms in the service architecture that notify dynamic changes along with run-time evaluation of JML specifications. We illustrate our approach using an example service in the H-Omega HAS developed on the OSGi^{TM}and iPOJO service platform. To evaluate our approach, we developed a testing framework that allows for generation of tests with dynamic service reconfigurations. In addition, we seeded faults into the example service, and evaluated the effectiveness of the test oracles in revealing the faults using the generated tests.

1. INTRODUCTION

Modern day homes are being revolutionized with the advent of devices and technologies that can network and communicate with each other. A Home Automation System (HAS) facilitates the automation of a private home to improve the comfort and security of its residents. It integrates different home appliances via a network to provide services for entertainment, safety, and comfort. For instance, integrating a TV, a DVD player, surround speakers, lights, curtains and an air-conditioner allows to provide an integrated service, that we call *Theater integrated service*, where a user can watch movies in a theater-like atmosphere. HAS is an application in the domain of Service-Oriented Computing (SOC) and is implemented using service-oriented architecture. The HAS, like any other SOC application, utilizes services as the basic units to support development of the distributed application.

Most of the research in SOC has focused on the architecture and framework for developing SOC applications. Research in verification of SOC applications is still in its infancy. The main challenge in verifying SOC applications like the HAS lies in the dynamic reconfigurations that often occur in these applications. In the HAS, new services may appear or existing services may disappear as the application is running. Not only is there a dynamic change in availability of services but the bindings between them also change during run-time. As a result, the architecture and configuration of the HAS and its services evolve dynamically. In the rest of this paper, we refer to this phenomenon in the HAS as its dynamic nature/behavior/reconfigurations. To exemplify, consider the *theater integrated service* mentioned earlier. The service may be required to connect to a mobile video player (like an iPad) if it is available in the room and play videos from it. Thus, if a mobile video player appears in (or disappears from) the room when the theater integrated service is running, the service is required to dynamically bind to (or unbind from) the player service at run-time. Such dynamic changes in service configuration may affect the correctness and quality levels of these applications.

Verification of HAS and other applications with dynamic reconfigurations can be viewed as two testing problems, (1) the need for test oracles that observe and check behavior during dynamic reconfigurations, and (2) the need to generate tests that involve dynamic service reconfigurations. In this paper, we primarily focus on addressing the first testing concern–specifying test oracles for HAS. Nevertheless, to evaluate our approach for test oracles, we also addressed the second testing concern with regard to test generation, albeit in a preliminary manner.

Traditional test oracles that examine outputs at the end of the test execution are not adequate for the HAS since the configurations and context of the service can change dramatically as the service is running. We need test oracles that are run-time monitors, continuously monitoring the behavior of the service, particularly during dynamic reconfigurations. Test oracles that monitor the run-time behavior of a system for consistency with requirements have been proposed in the past [20, 16, 11]. These approaches, however, cannot be directly applied to the HAS since they are not tailored towards monitoring dynamic reconfigurations in the service

[*]Partially supported by the iPOTest Project of the Université Joseph Fourier, and by the ISLE cluster of the Région Rhône-Alpes .

composition and bindings.

To address this issue, we propose test oracles in the form of formal specifications that act as run-time monitors of the services in the HAS. The test oracles monitor whether the services deliver the functions expected from them in the presence of dynamic reconfigurations. Our approach relies on utilizing mechanisms in the service architecture to notify run-time monitors of dynamic reconfigurations. We use the Java Modeling Language (JML) [17] specification language to formally specify test oracles. We illustrate our approach on an example service in the HAS. The HAS we use in this paper is simulated using the H-Omega [6, 9] framework implemented on top of OSGi [2].

To evaluate our approach for test oracles, we generated tests with dynamic service reconfigurations for the HAS. We adapted our existing combinatorial testing tool, TOBIAS [18], to support test generation with dynamic reconfigurations that can be executed on a service-oriented platform. We evaluated the fault revealing capability of our test oracles by seeding faults into the example service and running the TOBIAS test suite against them. Automatically generating tests with dynamic reconfigurations for SOC applications like the HAS has not been explored extensively in the past. We believe our effort at test generation is a useful, although preliminary, step in this direction.

2. BACKGROUND

2.1 Framework for Home Automation Systems

The Adele team at the Laboratoire d'Informatique de Grenoble (LIG) developed a platform, called H-Omega [6, 9], for building home automation systems. The H-Omega gateway eases the creation and deployment of new services by transparently managing service bindings, heterogeneity, and dynamism. The gateway is implemented on top of OSGi^{TM} [2] and iPOJO [10]. The OSGi framework is a system for Java that implements a dynamic component model that can be remotely managed. The service-oriented component model, iPOJO (injected POJO), aims to simplify service-oriented programming on OSGi frameworks by transparently managing service dynamics.

The iPOJO framework allows developers to distinctly separate functional code (i.e., the POJO - acronym for *Plain Old Java Object*) from the non-functional code (for dependency management, service provision, configuration, etc.). All non functional concerns are externalized and managed by the container through handlers (see Fig. 1). The component is the central concept in iPOJO. The description of the component — information on service dependencies, provided services, and callbacks — is recorded in the component's metadata. Using the component metadata, the iPOJO runtime manages the component, i.e., manage its life cycle, inject required services, publish provided services, discover needed services.

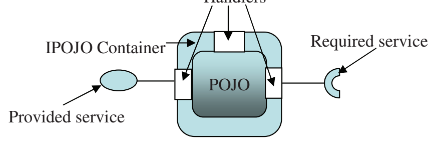


Figure 1: Component in iPOJO

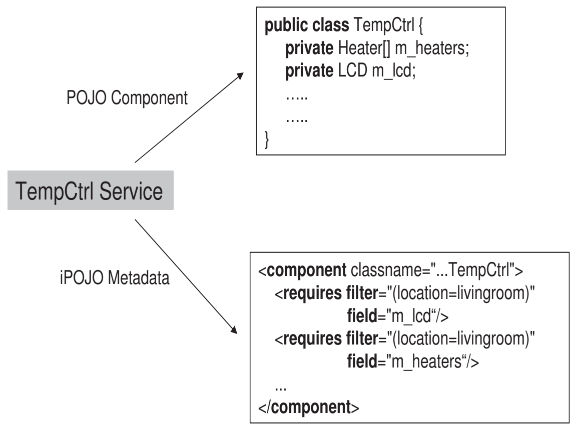
2.2 Example Service in HAS

To help illustrate the principles in iPOJO, consider the following example of an integrated service, termed *Temperature Control*, in the HAS. Note that the HAS and temperature control service are simulations using the H-Omega framework. Real home automation systems and their services are not easily available. We chose to use the temperature control service example simply to illustrate dynamic reconfigurations in services and the need for their continuous monitoring. Other services like the theater integrated service, mentioned in Section 1, may also be used in its place. Regardless of the example service used, the monitoring challenges encountered are similar. The temperature control service controls the temperature of the room, so that the target temperature desired by the user is reached. The service requires heaters, and a display device (termed LCD in the service) that displays the number of heaters active and running. At least one heater and LCD are mandatory requirements for the temperature control service, implying the service will be invalid if either of these devices are unavailable. The service also ensures that the heaters are used economically. The number of heaters that ought to be running for economical usage is controlled based on the difference in temperature between the desired target temperature and current room temperature. The service uses the following conditions for economical usage of heaters:

Temperature Difference	< 10	Turn on 1 heater
Temperature Difference	$10\ to\ 20$	Turn on $<=$ 3 heaters
Temperature Difference	> 20	Turn on All heaters

The service continuously monitors the temperature difference, and controls the available heaters in the room (turning heaters on/off) based on this difference. The *dynamic aspect* in the service is introduced by two factors:

1. Heaters may appear/disappear from the room. (Assuming the heaters are portable heaters)
2. Depending on the temperature difference, the number of active heaters in the room keeps changing. The LCD should display the number of active heaters and update the display as the number of active heaters changes.



```
public class TempCtrl {
    private Heater[] m_heaters;
    private LCD m_lcd;
    .....
    .....
}
```

```
<component classname="...TempCtrl">
   <requires filter="(location=livingroom)"
                 field="m_lcd"/>
   <requires filter="(location=livingroom)"
                 field="m_heaters"/>
   ...
</component>
```

Figure 2: Temperature Control Service - POJO component and metadata

The POJO component of the temperature control service contains the Java class defining functionality of the service. The devices (or services) required by temperature control are simply used as fields in the component class. Figure 2 shows a portion of the POJO component for the temperature control service with field declarations for the required devices. For instance, *m_heaters* in the POJO component in Figure 2 is an array of heaters whose length will vary depending on the number of available heaters at run-time. To enable iPOJO to manage this component, we describe the component in the metadata file. In the metadata, we ask iPOJO to create the temperature control component and an instance, indicate the service provided by temperature control, and indicate fields in the component that represent required services. Figure 2 shows a portion of the iPOJO metadata for the temperature control service with the required services. As seen in the figure, fields for heaters, and LCD in the temperature control component are indicated as required services (using the *requires* tag) to be injected by iPOJO at run-time. If required services are unavailable, the temperature control component instance becomes invalid. When services fulfilling the requirement appear, the instance becomes valid.

2.3 JML: Java Modeling Language

JML is an annotation language used to specify Java programs by expressing formal properties and requirements on the classes and their methods [17]. Our proposed approach for test oracles in the HAS uses JML specifications. We chose to use JML as the formal specification language for the following reasons, (1) Wide range of tools already exist for JML, supporting, runtime assertion checking, static checking, program verification, generation of annotations, specification browsing [17], and (2) The HAS application was implemented in the Java programming language. JML is a natural choice as a formal specification language for Java.

JML specifications appear within special Java comments, /*@ and @*/, or starting with //@. The specifications of each method precede the method declaration. We ask the reader to refer to [17] for a discussion on syntax and usage of JML specifications. An example JML postcondition in the temperature control service is presented below.

```
//@ ensures
    ((isrunning && (m_heaters.length >= 3) &&
        (tempdiff >= 10) && (tempdiff < 20))
      ==> (num_running == 3));
```

The postcondition states that if the service is running and the number of available heaters is greater than or equal to 3 and the temperature difference between current and desired room temperature lies between 10 and 20 degrees, then the number of heaters active in the room should be 3.

Our approach uses JML specifications as run-time monitors. The JML Runtime Assertion Checker (RAC) [7] provides this capability. It translates JML specifications into runtime checking bytecode, and verifies that specifications are satisfied during program execution.

3. TEST ORACLES USING JML SPECIFICATIONS

Test oracles that monitor the run-time behavior of a system for consistency with requirements have been proposed in the past [20, 16, 11]. Additionally, run-time monitoring of JML specifications for use as test oracles in unit testing of programs has been proposed previously [8]. Nevertheless, these existing approaches for run-time monitoring have never been used for applications like the HAS where the architecture is dynamically evolving, i.e., where bindings among components in the application change dynamically, and components available for composition also change. To adopt existing run-time monitoring techniques for the HAS, we need to enhance them with the capability of monitoring service behavior during dynamic service reconfigurations. In this section, we present our approach for doing this using JML specifications along with capabilities in the service architecture.

3.1 Validating Service dynamism using JML

As with any run-time monitoring technique, one of the difficult and important aspects lies in ensuring the assertions are placed and checked at the right points in the execution of the program. This aspect is more challenging in the HAS due to the presence of dynamic reconfigurations in the application and execution context. Broadly, in our approach, we tackle this issue by first identifying potential sources of dynamic behavior in the service, i.e. fields in the service that may change dynamically. We then place probes in the service architecture, so that dynamic changes at the identified sources are communicated to a *listener method* in the service component. The listener method is associated with a set of JML assertions that check the correctness of the service during dynamic reconfigurations. If a JML assertion is violated, a run-time exception is raised to notify the user.

To better understand our approach, we briefly describe how dynamic nature in services is managed by the H-Omega framework. We then discuss the mechanisms we use in the architecture to alert a listener method. The main source of dynamic behavior in the HAS lies in the dynamism in service availability which in turn affects other services depending on or requiring them. When a service (or component instance in the vocabulary of iPOJO) requires another service, the iPOJO framework chooses a suitable service satisfying the requirements and directly injects the required service object inside a field in the component, or invokes a method when the required service appears (or disappears). The dependency handler in iPOJO manages service dependencies/requirements. As stated in [1], the handler manages two types of service injection in the component to handle dependencies:

1. Field injection: a field in the component contains the service object. As soon as the field is used, a consistent service object is injected. This injection type fully hides the dynamism.
2. Method invocation: when a required service appears, or disappears a method in the component is invoked. For each dependency, the component can declare bind and unbind methods that get invoked when the service appears or disappears, respectively.

In the second injection mechanism, method invocation, the dynamics can be managed directly by the developer. Each dependency can declare two callback methods: A bind method, called when a service appears, and an unbind method, called when a service disappears. The two injection mechanisms, field injection and method invocation, can also be used together. In this combined injection mechanism, the field receives the value before the bind method invocation. So, if the field is used in the bind method, the returned value will be up to date. Table 1 presents a portion of the iPOJO metadata for the Temperature Control service

Field Injection Mechanism

```
<component classname="...TempCtrl">
  <requires filter="(location=livingroom)" field="m_heaters">
  </requires>
  ...
</component>
```

Method Invocation Mechanism

```
<component classname="...TempCtrl">
  <requires>
      <callback type="bind" method="bindHeater"/>
      <callback type="unbind" method="unbindHeater"/>
  </requires>
  ...
</component>
```

Combined Injection Mechanism

```
<component classname="...TempCtrl">
  <requires filter="(location=livingroom)" field="m_heaters">
      <callback type="bind" method="bindHeater"/>
      <callback type="unbind" method="unbindHeater"/>
  </requires>
  ...
</component>
```

Table 1: Injection Mechanisms in iPOJO Metadata for TempCtrl Component

component (introduced earlier in Section 2.2) that shows the dependency of the service on heaters. The three mechanisms for injecting the heater service–field injection, method invocation, and combined injection–for this dependency are illustrated in Table 1. In the field injection mechanism, we simply mention the field name *m_heaters* in the *requires* tag for the component, and iPOJO takes care of injecting and updating the field when heaters appear or disappear from the living room. In the method invocation mechanism, we define the callback types *bind* and *unbind* along with their associated methods in the component in the *requires* tag. When a heater appears, the *bindHeater* method defined in the temperature control component is called. In a similar fashion the *unbindHeater* method in the component gets called when a heater disappears[1]. The bind and unbind callback methods are responsible for updating the *m_heaters* field. Finally, in the combined injection mechanism, we use both the field name and callback types in the *requires* tag. When a heater appears or disappears, iPOJO takes care of injecting and updating the *m_heaters* field. After the field update, the callback methods in the component get called.

Our approach uses the combined service injection mechanism, since it allows the component to be notified of the dynamic event, while taking care of the burden of updating the field with a consistent service object automatically. The callback methods in the component that get invoked when the dynamic change in the field occurs are referred to as the *listener methods*. The listener methods usually define actions or updates to be executed in the service after the reconfiguration. We attach JML assertions to these listener methods. Thus, every time the service is dynamically reconfigured, the listener methods are invoked and consequently, JML assertions associated with the listener method are evaluated and checked for violations. Our approach thus validates service behavior during dynamic reconfigurations while the service is running.

Our approach for monitoring services is targeted at helping the creators of integrated services like the temperature control or theater integrated services with testing and monitoring their behavior. Our approach manually inserts JML assertions and tags listener methods in the service implementation. We believe it is reasonable to assume that the creators and testers of the service have access to its implementation when testing the service. Note that our approach views other devices and services used by the service of interest as a black box. We do not intrude into the implementation of these services. The completeness and correctness of the JML assertions would have to be manually ensured by the testers of the service.

3.2 Test oracles for Temperature Control Service

In this section, we illustrate our approach for test oracles using the temperature control service introduced earlier in Section 2.2. The test oracles monitor the dynamic nature of the service in the H-Omega architecture. The dynamic aspect in the service is introduced by the appearance or disappearance of heaters, or due to temperature change resulting in heaters being dynamically switched on/off.

We begin by briefly describing the methods implementing the functionality of the temperature control component. The *execute()* method in the component is responsible for the core functionality of the service. When the temperature control service is activated, the *execute()* method in the service component is called. The method computes and monitors the temperature difference every 2 seconds[2]. Based on the temperature difference and the availability of heaters, the *execute()* method switches on/off the heaters, sets the target temperature, and displays the number of active heaters on the display device (LCD). The *bindHeater()* and *unbindHeater()* methods serve as the listener methods in the service component that are notified of dynamic changes in the heaters. The listener methods are responsible for making the necessary updates to the LCD display when dynamic reconfigurations in heaters occur.

We now proceed to describe the test oracles for this service. For ease in understanding, we split the JML specifications for the service into two: (1) JML specifications that monitor service behavior during normal (no dynamic reconfiguration) service operation, (2) JML specifications that monitor service behavior during dynamic reconfigurations. The JML specifications in (1) are associated with the *execute()* method. The JML specifications in (2) for monitoring dynamic behavior are associated with the listener methods, *bindHeater()* and *unbindHeater()*. To better understand the JML specifications for the service, we give a brief description of the variables used in the specifications.

- *num_running* reflects the number of heaters that ought to be active and running based on the temperature difference and number of available heaters.
- *isrunning* is a boolean variable that reflects whether the temperature control service is running.
- *m_heaters* is an array of available heaters in the room.

[1]The callback methods, *bindHeater* and *unbindHeater*, use the heater service object appearing or disappearing as a parameter in the method definition. iPOJO infers the service object type for the requirement using this parameter.

[2]We found that monitoring every 2 seconds was adequate to detect change in room temperature. Other values that also ensure frequent monitoring can be chosen. Choice of this value is only a service implementation concern, it does not affect the specification of test oracles.

m_heaters.length gives the number of available heaters.

- *m_lcd* represents the LCD device available in the room.
- *tempdiff* represents the temperature difference between the desired and current room temperature.

iPOJO takes care of injecting the fields, *m_heaters* and *m_lcd*, at run-time with available heaters and LCD, respectively. Recall that our approach for handling dynamic reconfigurations in services uses the combined service injection mechanism mentioned in Section 3.1. As a result, when heaters used in the service get dynamically reconfigured, iPOJO automatically updates the *m_heaters* field with the change while also notifying the change to the listener methods. All three methods in the service component, *execute()*, *bindHeater()*, and *unbindHeater()* update the variables *num_running*, the *m_lcd* display, and *tempdiff*. The *isrunning* variable is updated by the *execute()* method.

The first set of JML specifications, invariants and post conditions, to check service behavior during normal operation, are shown in Table 2. The post conditions (using the *ensures* clause) in Table 2 are specified on the *execute()* method and should hold after the *execute()* method call. The invariants (using the *invariant* clause), on the other hand, are checked *before* and *after* every method execution in the service component. The post conditions $N1, N2, N3, N4, N5$ ensure that the number of heaters active in the room correspond to the number of available heaters and the temperature difference conditions described earlier in Section 2.2. The post conditions for the heater ($H1$, $H2$, $H3$) ensure that only *num_running* heaters as per the economic usage conditions are *on*. They also ensure that all of those heaters are set at the desired target temperature. The invariants ($L1, L2$) for the LCD ensure that when the temperature control service is running, the LCD is *on* and displays the number of active heaters. The invariants $L1, L2$ should hold at the beginning and end of the *execute(), bindHeater(), unbindHeater()* method executions. Note that since $L1, L2$ are specified as invariants, they aid in monitoring service behavior during dynamic reconfigurations in addition to normal service operation. It is also worth noting that properties involving the heaters are specified as post conditions, rather than invariants, since dynamic changes in heater availability would cause such invariants to be violated *before* calls to *bindHeater()* and *unbindHeater()* that take care of the necessary service updates during dynamic reconfigurations.

The second set of JML specifications is to monitor the dynamic nature of the service. In the service component, the listener methods *bindHeater()* or *unbindHeater()* respectively get called when heaters satisfying the temperature control service requirements appear or disappear. To monitor dynamic reconfigurations, our approach associates JML specifications to these listener methods. We present the bind/unbind methods along with their JML specifications in Table 3. When a heater appears, the *bindHeater()* method in Table 3 is called by the service. Within the method, if the conditions for economic usage are not violated, then this newly available heater is switched on and set to the target temperature. The LCD display is updated to reflect the change in the number of running heaters. The JML post conditions associated with the *bindHeater()* method check whether the heater is turned on according to the conditions for economic usage. Note that post conditions from the *execute()* method $N1, N2, N3, N4, N5, H1, H2, H3$ are repeated here since they represent the economic usage conditions for the heaters. The JML invariants, $L1$ and $L2$, for the LCD, mentioned earlier in Table 2, also get evaluated to ensure the LCD display is updated correctly.

When a heater disappears, the *unbindHeater()* method in Table 3 is called by the service. To compensate for the unbound heater, the method switches on another heater, if available, in compliance with the economic usage conditions. The number of active heaters in the room and the LCD display are updated. The JML post conditions check whether the heater being unbound is switched off and if the economic usage conditions are obeyed. The JML invariants, $L1$ and $L2$, check whether the LCD display is updated with the correct number of active heaters.

4. TEST GENERATION AND EVALUATION

We evaluated our proposed approach for test oracles in the HAS by testing several dynamic reconfigurations in services. To enable us to evaluate and test our approach thoroughly, we developed a testing framework that generates tests with dynamic service reconfigurations for the HAS from a test pattern. We tailored our existing combinatorial testing tool, TOBIAS [18], to help achieve this. We monitored the JML specifications as the test cases were run to check for violations in service behavior. Additionally, we created several mutated services by seeding faults into the service so that service behavior is altered during dynamic changes. Each mutated service has a single seeded fault. We ran the test suite generated by TOBIAS against the set of mutated services, and checked whether the test oracles in the service could reveal the mutations. We say that the test oracles revealed the service mutation for the given test suite if at least one of the test cases in the test suite violated at least one of the JML specifications in the mutated service.

The tests generated by TOBIAS are sequences of method calls with different combinations of input parameter values for the methods. The input to TOBIAS is a test pattern (also called test schema) that defines the set of test cases to be generated. A test pattern is a bounded regular expression involving the Java methods in the service. TOBIAS unfolds the test pattern into a set of sequences, and then computes all combinations of the input parameters for all the methods in the pattern. The resulting test suite is converted into a JUnit ([14, 4]) file for testing services on the OSGi platform. Note that TOBIAS was previously used as a combinatorial test generation tool for traditional JAVA applications rather than service-oriented applications such as the HAS. We adapted TOBIAS to generate test suites that are executable on the OSGi service-oriented platform. Additionally, we created test patterns that exercised different dynamic reconfigurations and behavior changes in the service by placing calls to methods that made required services appear/disappear or by changing the configuration of the environment during service run-time.

4.1 Temperature Control Service: Test Oracle Evaluation

Due to space limitations, we only briefly illustrate test generation and oracle evaluation using the temperature control service in this Section. The test pattern we used to automatically generate test cases using TOBIAS for the temperature control service is described in Table 4. In the test

```
// Properties for number of active heaters in the room
// (labeled N1, N2, N3, N4, N5)
N1: //@ ensures (isrunning ==> (num_running <= m_heaters.length));
N2: //@ ensures ((isrunning && (m_heaters.length > 0) && (tempdiff < 10))
                    ==> (num_running == 1));
N3: //@ ensures ((isrunning && (m_heaters.length >= 3) && (tempdiff >= 10) && (tempdiff < 20))
                    ==> (num_running == 3));
N4: //@ ensures ((isrunning && (m_heaters.length < 3) && (tempdiff >= 10) && (tempdiff < 20))
                    ==> (num_running == m_heaters.length));
N5: //@ ensures ((isrunning && (tempdiff >= 20)) ==> (num_running == m_heaters.length));

// Heater Properties (labeled H1, H2, H3)
H1: //@ ensures isrunning ==> (\forall int i; 0<=i && i<num_running; m_heaters[i].isOn());
H2: //@ ensures isrunning ==> (\forall int i; num_running<=i && i<m_heaters.length;
                    !(m_heaters[i].isOn()));
H3: //@ ensures isrunning ==> (\forall int i; 0<=i && i<num_running;
                    m_heaters[i].getTargetedTemperature() == targetTemp);

// LCD properties (labeled L1, L2)
L1: //@ invariant (isrunning ==> m_lcd.isOn());
L2: //@ invariant (isrunning ==> m_lcd.getDisplay().equals( "Number of heaters active is " +
                    Integer.toString(num_running)));
```

Table 2: JML assertions to monitor temperature control service behavior

```
// BIND method and specifications
/*@ ensures ((isrunning && (((tempdiff < 10) && (\old(num_running) < 1))
            ||((tempdiff >= 10) && (tempdiff < 20) && (\old(num_running) < 3))
            ||(tempdiff > 20))) <==> (h.isOn() && (num_running == \old(num_running) + 1)));
@*/
//@ Repeat post conditions N1, N2, N3, N4, N5 given earlier
//@ Repeat post conditions H1, H2, H3 given earlier

private synchronized void bindHeater(Heater h) {
    if (isrunning) {
        tempdiff = tempDiff();
        if (((tempdiff < 10) && (num_running < 1))
           ||((tempdiff >= 10) && (tempdiff < 20) && (num_running < 3)) || (tempdiff > 20)){
                System.out.println("Binding Heater: " + h.getFriendlyName());
                h.turnOn();
                h.setTargetedTemperature(targetTemp);
                num_running++ ;
                m_lcd.display("Number of heaters active is " + Integer.toString(num_running));
        }
    }
  // if isrunning is false it means the execute method is not running,
  // so no updates necessary
}

// UNBIND method and specifications
//@ ensures (isrunning ==> (h.isOn() == false));
//@ Repeat post conditions N1, N2, N3, N4, N5 given earlier
//@ Repeat post conditions H1, H2, H3 given earlier

private void unbindHeater(Heater h){
    if (isrunning && h.isOn()) {
        System.out.println("Unbinding Heater: " + h.getFriendlyName());
        h.turnOff();
        num_running--;
        // Turn on another heater, if available, according to
        // temp diff and economic usage conditions
        if ((num_running < m_heaters.length) && !m_heaters[num_running].isOn() &&
           (((tempdiff < 10) && (num_running < 1))
           ||((tempdiff >= 10) && (tempdiff < 20) && (num_running < 3)) || (tempdiff > 20))){
                m_heaters[num_running].turnOn();
                m_heaters[num_running].setTargetedTemperature(targetTemp);
                num_running++ ;
        }
        m_lcd.display("Number of heaters active is " + Integer.toString(num_running));
    }
}
```

Table 3: Listener methods and JML assertions to monitor dynamic reconfigurations in temperature control service

Initial Configuration
Introduce 3 to 5 heaters Set environment temperature to 5, 20, or 80 Set desired room target temperature to 20, 40 or 100 Activate Temperature Control Service Wait for a fixed time
Dynamic Changes
Add/Remove heater Change environment temperature Wait for a fixed time Deactivate Temperature Control Service

Table 4: Informal description of the TOBIAS test pattern for Temperature Control service

pattern in Table 4, the wait times were configured to allow the service to run for a sufficiently long time so that changes in service behavior could be observed. The test pattern illustrated was unfolded into 135 test cases by TOBIAS with different combinations of input parameters. Note that it is possible to create many other test patterns, different from the one in Table 4, for the temperature control service; with different sequences of method calls, different input parameters for heater configurations and temperature settings, different numbers and combinations of dynamic changes. We chose the test pattern in Table 4 to simply illustrate our test generation and evaluation approach. We do not place any claims on the thoroughness and effectiveness of the generated test suite.

We evaluated the effectiveness of our oracles by seeding faults into the service and checking if the oracles were capable of revealing the faults. We created 25 mutated services, by *manually* seeding faults to alter behavior of the service during dynamic reconfigurations. Each mutated service had a single seeded fault. We ran the test suite generated for the test pattern in Table 4 over each of the 25 mutated services and checked if any of the JML specifications in the mutated service were violated. We found that for 23 of the 25 mutated services, JML specifications were violated revealing the mutations in the service. Thus, for the given test suite of 135 test cases, our approach for test oracles was effective in revealing 23 of the 25 seeded faults. On closer examination of the two undetected faulty scenarios, we found that the test suite did not exercise the scenarios involving the two seeded faults. To overcome this weakness in the test suite, we manually created test cases that exercised the two faulty scenarios. The newly created test cases violated the JML specifications for the bindHeater() listener method. We could thus reveal all 25 mutations with our test oracles and the test suite augmented with the newly created test cases. The evaluation clearly showed that the JML specifications associated with the bind/unbind listener methods were effective in revealing erroneous behaviors during reconfigurations.In our future work, we plan to explore test generation for SOC applications like the HAS in more depth.

4.2 Threats to Validity

We face two threats to the validity of our evaluation. The first one is with regard to the properties that can be expressed with JML. In our example, the reconfiguration properties specified with JML were either static properties expressed as invariants, or properties only valid in the initial or final states expressed as pre or postconditions. We also specified dynamic properties involving current and previous states in the example (using the `\old` clause). Nevertheless, we did not explore properties where the current behavior is dependent on behavior that occurred past the previous state, i.e. the system has some memory of the behavior history and reacts differently to an event based on the history. For example in the temperature control service, when a heater appears, the service may be required to react differently based on whether that heater was already seen before or not. The JML specification language does not support operators to express such temporal properties. This issue, however, can be overcome by using the approach proposed by Bellegarde et al. [5] that translates such temporal properties into an equivalent set of JML annotations. The second external threat is that the example service, test suite, and number of mutations used in our evaluation are relatively small when compared to an evaluation over an industrial system. Nevertheless, this is only a preliminary evaluation that helped show that our approach for test oracles holds promise. We plan to pursue a more extensive evaluation on real world examples in the future.

5. RELATED WORK

The run-time monitoring challenges encountered in systems composed of web services are closely related to the monitoring challenges in the HAS since both applications encounter dynamic reconfigurations in services. Run-time monitoring of properties in web services has been explored in the past.

Spanoudakis et al. [21] proposed a framework for run-time monitoring of requirements, expressed in event calculus, for web-service compositions. Baresi et al. [3] also proposed an approach for run-time monitoring web service compositions. They monitor whether the external service selected by the composition process conforms to the behavior expected from it. Ghezzi et al. [12] proposed an approach, Dynamo, to specify constraints and monitor collaborations with external services. Dynamo monitors whether the external services that it collaborates with deliver what is expected of them.

The related work in the web services domain primarily focuses on monitoring web service compositions. Web service compositions are managed by a composition process specified in languages like BPEL [13] or WSCDL [15] (depending on the collaboration model chosen). A composition process, as defined by Mahbub et al. [19], is one that coordinates external web services that get deployed in a service-oriented system. The composition process provides the required system functionality by calling operations in the external web services, receiving and processing the results that these services return, and accepting and/or responding to requests from them. All the proposed monitoring approaches in the web services domain rely heavily on the composition process, and monitor whether the external web service selected by the composition process adheres to the behavior expected from it. Our proposed approach for monitoring the HAS differs from these existing approaches in two fundamental ways. One, our approach aims at monitoring the behavior of the integrated service that uses other external services. Unlike existing approaches, our monitoring approach is not concerned with the selection mechanism and behavior of external services. For instance in the temperature control service, we monitored behavior of the service in the presence of changes to the external heater services. We do not monitor whether the heater and LCD services that it uses are selected according to requirements and function as expected. Two, the H-Omega gateway that we use to deploy and provide services, transparently manages interaction between services. There is no explicit composition process. As a result, our approach for run-time monitoring is independent of a composition process, and instead employs listener methods interacting with the service architecture to help verify dynamic service behavior.

6. CONCLUSION

In this paper, we proposed an approach to address one of the challenges in testing home automation systems—specifying test oracles that monitor service behavior in the presence of dynamic service reconfigurations. We formally specify test oracles for the HAS using the JML specification language. We use JML specifications as run-time monitors of the service behavior. The main challenge in run-time monitoring is in identifying "visible" states for evaluating the specifications during program execution. We provide the visible states for specification evaluation using *listener methods* that are associated to dynamic events in the service architecture. We combine this capability with JML run-time assertion evaluation to monitor service behavior during dynamic reconfigurations. We illustrated our approach with an example service in the HAS.

We conducted an initial evaluation of our proposed approach for test oracles by testing several dynamic service reconfigurations in the example service. We adapted our existing combinatorial testing tool, TOBIAS, over JAVA applications to generate tests with dynamic service reconfigurations for the HAS. We ran the test suite from TOBIAS against 25 mutated versions of the example service to evaluate the fault finding capability of the test oracles. We found that the test oracles could reveal all 25 mutations. From our preliminary evaluation, we believe our proposed approach provides a useful and effective means for defining test oracles that monitor service behavior in the presence of dynamic reconfigurations for the HAS. We plan to conduct a more extensive evaluation of our test oracle approach on real world example systems in our future work.

In this paper, we have only explored the applicability of our approach to the HAS implemented using the H-Omega service architecture. Nevertheless, it is straightforward to see that our approach can be used in a like manner for other service-oriented applications implemented using the H-Omega service architecture. To apply our approach to other service architectures, we would need to utilize the appropriate mechanisms in the underlying architecture for notification of dynamic changes. We plan to extend our approach to include other architectures in our future work.

7. REFERENCES

[1] Apache felix iPOJO website. http://felix.apache.org/site/apache-felix-ipojo.html.

[2] OSGi Alliance. *OSGi Service Platform: Release 3, March 2003*. IOS Press, 2003.

[3] L. Baresi, C. Ghezzi, and S. Guinea. Smart monitors for composed services. In *ICSOCŠ04*, pages 193–202, 2004.

[4] K. Beck and E. Gamma. Test infected: Programmers love writing tests. Java Report 3(7), July 1998.

[5] F. Bellegarde, J. Groslambert, M. Huisman, J. Julliand, and O. Kouchnarenko. Verification of liveness properties with JML. Technical report, INRIA.

[6] J. Bourcier, A. Chazalet, M. Desertot, C. Escoffier, and C. Marin. A dynamic-soa home control gateway. In *IEEE International Conference on Services Computing (SCC 2006)*, 2006.

[7] Y. Cheon and G. T. Leavens. A Runtime Assertion Checker for the Java Modeling Language (JML). In Hamid R. Arabnia and Youngsong Mun, editors, *International Conference on Software Engineering Research and Practice (SERP '02)*, pages 322–328, Las Vegas, Nevada, June 2002. CSREA Press.

[8] Y. Cheon and G.T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In *16th European Conference on Object-Oriented Programming (ECOOP'02)*, number 2374 in LNCS, pages 231–255. Springer, June 2002.

[9] C. Escoffier, J. Bourcier, P. Lalanda, and Jianqi Yu. Towards a home application server. In *5th IEEE Consumer Communications and Networking Conference*, pages 321–325, January 2008.

[10] C. Escoffier, R.S. Hall, and P. Lalanda. iPOJO: an extensible service-oriented component framework. In *IEEE International Conference on Services Computing (SCC 2007)*, pages 474–481, July 2007.

[11] S. Fickas and M.S. Feather. Requirements monitoring in dynamic environments. In *Proc. of the Second IEEE International Symposium on Requirements Engineering*, pages 140–147, March 1995.

[12] C. Ghezzi and S. Guinea. *Run-time monitoring in service-oriented architectures*. Test and Analysis of Web Services, 2007.

[13] IBM, BEA Systems, Microsoft, SAP AG, and Siebel Systems. Business Process Execution Language for Web Services 1.1, 2005.

[14] JUnit. http://www.junit.org.

[15] N. Kavantzas, D. Burdett, and G. Ritzinger. Web Services Choreography Description Language version 1.0, 2004.

[16] M. Kim, S. Kannan, I.Lee, O. Sokolosky, and M. Viswanathan. JAVA-MAC: a runtime assurance tool for java programs. *Electronic Notes in Theoretical Computer Science*, 55, 2001.

[17] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, P. Muller, J. R. Kiniry, and P. Chalin. *JML Reference Manual*. Iowa State University, Jan 2006.

[18] Y. Ledru, L. du Bousquet, O. Maury, and P. Bontron. Filtering TOBIAS combinatorial test suites. In *7th Int. Conf. FASE, Held as Part of ETAPS*, volume 2984 of *LNCS*, pages 281–294, Barcelona, Spain, 2004.

[19] K. Mahbub and G. Spanoudakis. *Monitoring WS-Agreements: An Event Calculus-Based Approach*. Test and Analysis of Web Services, 2007.

[20] D.K. Peters and D.L. Parnas. Requirements-based monitors for real-time systems. *IEEE Trans. Softw. Eng.*, 28(2):146–158, 2002.

[21] G. Spanoudakis and K. Mahbub. Requirements monitoring for service-based systems:towards a framework based on event calculus. In *Proceedings of the 19th International Conference on Automated Software Engineering*, 2004.