Abstract—We built Smart Home applications for the Cognitively Impaired population. We have chosen to work with an existing framework, OSGi, which allows us to develop specific applications more quickly. We use a combination of traditional testing and formal verification to ensure these applications will cause no harm to the cognitively impaired users of our systems. In the process of checking OSGi applications, we soon found ourselves in a dilemma. On the one hand, OSGi applications are based on the OSGi framework; hence, it seems natural to build a formal model paralleling the OSGi framework and reuse the modeling framework to check OSGi applications. On the other hand, this approach is complicated by the very nature of model checking. A major obstacle to formal modeling is state space explosion, where the interleaving of processes leads to an exponential increase of system states that is beyond the capacity of a typical computer. This restriction requires a formal model being reduced to its bare essentials before the verification step. However, since OSGi applications may use different features of OSGi, it is nearly impossible to come up with a modeling framework that has just enough details for all OSGi applications. We will need some way to vary features of a modeling framework to check various applications.

The Object-Oriented nature of Java has offered some help for customizing the modeling framework. For example, one can override slots and hooks to specialize the modeling framework. However, there are some crosscutting concerns that may not naturally fit in this paradigm. For example, when we are interested only in the stale references problem [3][11], we don’t need permission check in the framework and may remove all related fields and statements to save the state space. On the other hand, when we want to assure that no malicious application spoils the OSGi framework, we have to add back the missing fields and statements to enable permission check. Changing a feature is often a tedious task that involves modifications in multiple files, and it will be even worse when we have to come up with a specific combination of different features.

In our experience, such crosscutting concerns have prevented us from effective reuse of formal models and incurred much overhead in model maintenance. In view of these difficulties, we have found that aspect-oriented programming (AOP), using AspectJ [9][10] for Java, is a potential solution. Using AspectJ, we have been able to prune large portions of the OSGi code-base that are not relevant and vary features in a modular way. We report our findings in this paper.

The rest of the paper is organized as follows. We briefly explain the model checker Java Pathfinder in section II, and the application domain, OSGi applications, in section III; in section IV we use real examples of OSGi applications, to show that we can vary features of a formal model with AspectJ in a modular way; in section V, we show a necessary step, the construction of a specialized JPF, to enable verifying AspectJ programs in...
general. We summarize the contributions and the future work in section VI.

II. JAVA PATHFINDER

Java PathFinder is an explicit model checker that directly checks Java bytecode. It has a specialized virtual machine, which takes on the role of a model checking engine like state cache, query, comparison, and restoration. Through a well-defined component architecture, JPF allows one to customize the search procedure. For example, it has a Model Java Interface (MJI) to intercept Java method calls, which may be used to resolve native methods (by executing native methods outside the JVM, but store interesting results in the JVM) and reduce state space (by excluding state information in the transition from the JVM). The extension schemes in JPF also allow expert users to experiment with novel search heuristics and state representations, which can be conveniently reused through the well-defined component interface.

There are several restrictions on JPF. First, it is not able to check platform-specific native methods like I/O methods, since the execution information is not available to the JVM. Second, due to state space explosion, it typically deals with Java programs with no more than 10k lines-of-code [7]. Therefore, before using JPF to check a real system, one would have to (a) resolve Java native methods with pure Java implementations or the MJI, and (b) apply various reduction techniques to construct a model with only the bare essentials for the actual system. In addition, one may also need to specify the interactions with the environments (e.g., user inputs) in a closed form. This is general to model checking since model checkers can only handle finite systems for a complete verification. In JPF this process can often be combined with the step to resolve native methods.

III. THE OSGI FRAMEWORK

For the purpose of explanation, we briefly introduce the application domain, i.e., the OSGi framework and OSGi applications [3]. The OSGi specification articulates a generic component structure for Java modularization. The unit for modularization is a bundle, which is a jar file typically composed of a manifest file, Java classes, native library and other resources. An OSGi framework has a base framework to fulfill the core functionality of the OSGi specification, e.g., manage the lifecycle of bundles. In particular, when starting a bundle, the base framework instantiates a bundle activator designated in the manifest file and invokes a callback function BundleActivator.start (i.e., a bundle activator is a Java class that implements org.osgi.framework.BundleActivator). When stopping a bundle, the framework invokes another callback function BundleActivator.stop. These callback functions are the main places to extend functionalities in a module. Each bundle runs in a separate JVM environment to avoid interferences between bundles; bundles communicate with each other through events delivered by the framework and services managed by the framework. A service is the unit to dynamically import/export functionality by a bundle. The OSGi framework maintains a mapping to associate service names to service references: a bundle (or more exactly, a bundle activator) can register a service by adding a mapping in the table, and use a service by first looking up a service in the mapping table.

A common pitfall in OSGi applications is the stale references problem. It happens when a service used by a consumer bundle has actually been unregistered by the producer bundle. This problem is acknowledged in the OSGi specification, and an auxiliary class, ServiceTracker, has been created to track valid services. We will uncover violations belonging to this category in our case studies. An interested reader can refer to [3] [11] for more information about the stale reference problem and some of its treatment at the application level.

There exist several reference implementations for the OSGi specification. Knopflerfish is a leading implementation and has been certified to be OSGi compliant [5]. It is composed of a base framework to fulfill the OSGi specification, mandatory bundles for basic functionality like input/output and optional bundles for extended functionality like logging. We base our case studies on the Knopflerfish framework. An interested reader can refer to [3] [8] for a full explanation of OSGi and the Knopflerfish framework.

IV. ADVICE VERIFICATION VIA ASPECTJ

We have built a formal model paralleling the OSGi specification, to ease modeling OSGi applications [10]. In this section, we point out some crosscutting concerns in the formal model, and show that we are able to modularize such concerns with AspectJ and conveniently specialize the base model with various features.

A. Add Modeling Code to Check Correctness

For the purpose of testing and model checking, we often add additional code to record system status and check the status in appropriate places. Such code is not part of the base model (not even part of the implementation code) and should be added only when we model check relevant properties.

Consider the example in Figure 1. It is the formal model one would reasonably derive with to check the stale references problem: the statements in italic font are the extra code solely added for verification purpose. It mandates that all services extend a BasicService (line 5), which has a boolean field valid to indicate whether a service is currently available. Upon the invocation of a service, the service will check whether it is currently available to other bundles (line 7). From the point of the framework, the field valid is set to true (line 14-19, line 24-29) when a service is registered successfully (line 13, line 23), and set to false (line 37-40) when a service is removed from the registration table in the framework (line 36). The code, solely for the purpose of model checking (all lines in italic font), spans multiple places of several Java files (e.g., ServiceRegistrationImpl.java, BundleContextImpl.java and all service implementation files), which is cumbersome to vary and interact with other model checking features.
The crosscutting concerns in Figure 1 can be conveniently modularized in a single aspect. As shown in Figure 2, the rule that all services shall additionally extend BasicService is specified by the declare parents phrase (line 1). The rule that the field BasicService.valid shall be set to true when a service is registered is realized by the pointcut that matches the completion of a successful registration (line 18-21) and the advice (line 19-20). The rule that the service’s validity shall be checked upon the invocation to its functions is realized by a pointcut that matches the access of a service’s functions (line 2-4), and an advice that does the assertion (line 5-9). Whether to perform the verification of the stale references problem can be easily specified as inclusion or exclusion of the aspect while compilation.

B. Vary Security Feature

In addition to specifying correctness properties, we can also leverage AOP to conveniently choose a specific combination of features to check. As mentioned, we shall remove all non-mandatory fields and statements from the base model, and only add them back when they implement a specific feature relevant to the particular property to be verified. This methodology is critical to crack the state explosion problem and shall be enforced when possible.

However, we often find that adding back a feature involves adding fields and statements in multiple places of various files, and they are often different from their counterparts in the application code due to the result of abstraction. Enforcing such unfamiliar programming logic across multiple files is a daunting task when we have to check combinations of different features.

Consider the formal model in Figure 3. The statements in italic font enforce permission check for OSGi operations. A permission handler is created in Framework (line 5) and referred in Listeners and Services. The framework checks the permission when a bundle adds itself as a SynchronousBundleListener, to avoid a silly bundle blocking the whole framework (line 14-17). Similarly, the framework also checks the permission for registering a service to avoid a malicious bundle to preempt other valid services (line 31). These statements (in italic font) are not in the base model and are only needed when checking properties related to privileged operations. They scatter in more than ten places of three files (only a fragment of them are shown in Figure3), which is
tedious and error-prone for frequent addition and removal of features in model checking.

1. public class Framework {
2.   PermissionOps perm;
3.   public Framework(Object m) throws Exception {
4.     ...// Do other initialization
5.   perm=new SecurePermissionOps();
6. }
7. }
8. public class Listeners implements {
9.   PermissionOps secure;
10.  Listeners(PermissionOps perm) {
11.    secure = perm;
12. }
13.  void addBundleListener(Bundle bundle, BundleListener listener) {
14.    if (listener instanceof SynchronousBundleListener) {
15.      secure.checkListenerAdminPerm(bundle);
16.      ...// do the actual work to add listeners
17.    } else ... // do the actual work to add listeners
18. }
19. }
20.class Services {
21.  private PermissionOps secure;
22.  Services(PermissionOps perm) {
23.    secure = perm;
24. }
25.  ServiceRegistration register(BundleImpl bundle, String[] classes, Object service, Dictionary properties) {
26.    ...
27.    for (int i = 0; i < classes.length; i++) {
28.      String cls = classes[i];
29.      if (cls == null)
30.        throw new IllegalArgumentException("...");
31.      secure.checkRegisterServicePerm(cls);
32.      ... // do actual registering...
33.    }
34. }
35. }

Figure 3. A formal model to check permission violation in OSGi

1. static PermissionOps Framework.perm=new SecurePermissionOps();
2. pointcut BL (Bundle bundle, BundleListener bl):
3.   execution(* addBundleListener(Bundle, BundleListener))
4.   && args(bundle) && args(bl))
5.   & & args(bundle) & & args(bl)
6.   before(Bundle bundle, BundleListener bl) : BL (bundle, bl)
7.   if (bl instanceof SynchronousBundleListener) {
8.     Framework.perm.checkListenerAdminPerm(bundle);
9. }
10. }
11. }
12. }
13. }
14. public ServiceRegistrationImpl {
15.  Object service;
16.  public void unregister() {
17. ...
18.  if (service instanceof BasicService) {
19.      BasicService bs = (BasicService)service;
20.      bs.service_lock.lock();
21.      if (service.valid)
22.      service.checkWord(...);
23.      bs.service_lock.unlock();
24.      }
25. }

Figure 4. The security model advised with AspectJ

We can modularize the inclusion of security in a single aspect. As shown in Figure 4, we use an intertype declaration to add back the permission handler (line 1). The invocation of permission check when adding a SynchronousBundleListener is matched by a pointcut (line 2-4), and advised before it is actually added (line 5-9). The invocation of permission check when registering a service is matched by a pointcut (line 10-11), and advised before it is actually registered (line 12-16). Compared with the needed modifications across multiple places of three files, adding permission check with AspectJ is modularized in a single aspect and much easier to change.

C. Change the Granularity of Atomicity

Despite the existence of partial order reduction, program slicing and other techniques to save the state space, we often need to manually enforce atomicity in various places of a model. On the one hand, manually enforcing atomicity may significantly reduce the search space, so that errors may be uncovered much faster and with less memory - in many cases it is also the only way to complete the verification run before running out of resource (e.g., memory and time). On the other hand, we may want to change the atomic code block when we are exploring potential solutions to a concurrency error.

1. public class ConsumerActivator implements
2.   BundleActivator{
3.   public void start(BundleContext context) throws
4.   Exception{
5.   ...
6.   DictionaryService service=...//look up service in
7.   framework
8.   if (service!=null && service instanceof BasicService) {
9.      BasicService bs = (BasicService)service;
10.     bs.service_lock.lock();
11.     if (service.valid)
12.     service.checkWord(...);
13.     bs.service_lock.unlock();
14.     }
15. }
16. public ServiceRegistrationImpl {
17.  Object service;
18.  public void unregister() {
19. ...
20.  if (service instanceof BasicService) {
21.      BasicService bs = (BasicService)service;
22.      bs.service_lock.lock();
23.      bundle.framework.listeners.serviceChanged(new
24.      ServiceEvent(ServiceEvent.UNREGISTERING, reference));
25.      bs.service_lock.unlock();
26.      }

Figure 5. Varying atomicity to explore counter examples

For these reasons, we would like to have a mechanism to conveniently vary the granularity of atomicity. However, the conventional approach to do this is fairly involved. For example, Figure 5 shows a potential solution to avoid the stale references by enforcing atomicity when invoking a service (line 5-12) and unregistering a service (line 18-23). We have to entangle the code (that changes atomicity) with other statements in the two files, and similar code will have to spread in each service declaration file if we want to enforce such atomicity for them.
The daunting task of enforcing atomicity across multiple files can be modularized by aspects. For example, we can intercept the execution of a non-atomic block with an around clause, and enforce atomic execution of the block with a shared, exclusive lock. As shown in Figure 6, we use an around pointcut to intercept the intended atomic block (line 1-4), and enforce atomicity for the action sequence (checking the service validity and using the service) in line 5-12. Similarly, the around pointcut in line 13-15 intercepts the intended atomic block, and the advice in line 16-23 enforces the desired atomicity. By this way, we can enforce arbitrary atomicity without scattering code here and there, which will ease feature management at the model level and improve the efficiency in exploring candidate solutions. In our experience, we do find a verified solution for the stale references problem in this approach.

V. MODEL CHECKING ASPECTJ PROGRAMS

In section IV, we have shown that formal models have crosscutting concerns that may be elegantly modularized via AspectJ assistance. However, such superiority of aspects is not practically useful if the Java bytecode woven by AspectJ can’t be checked by JPF. In particular, we are concerned whether AspectJ will introduce extra native code to a pure Java model. This question is also significant for applications developed with AspectJ but now to be checked by JPF.

A simple AspectJ program like Figure 7 shows our fears are well founded. This program merely gets the signature of a joint point and involves no I/O from the surface. After we compile the program with the AspectJ compiler and check it with JPF, JPF reports the error as shown in Figure 8. Since all statements irrelevant to AspectJ are written in pure Java, the symptom shows that AspectJ introduces native code to a pure Java program. In this example, it occurs when the woven bytecode calls the native code in the AspectJ runtime library. A closer look at Figure 8 reveals that the NullPointerException is thrown when System.getProperty(String, String) is called (line 3). This is natural since it is a system-level function, which gets the environmental information outside the scope of the Java virtual machine.

We can use the MJI scheme in JPF to resolve native methods in the AspectJ runtime library, or customize the runtime library with a pure Java implementation. It is beyond the scope of this paper to fully describe the details to resolve native code in the AspectJ runtime library. As an example, we briefly explain the procedure to resolve native methods invoked by the program in Figure 7.

Defining a peer function in MJI is similar to calling a native method in Java Native Interface (JNI). A MJI method for System.getProperty(String, String) is shown in Figure 9. The function signature has name mangling so that it allows proper association in case of function overloading. The particular implementation in Figure 9 allows reading system properties outside the closed JVM.
the result in a special area. JPF fetches the result from this special area as if it is read from the environment.

1. public static int getProperty__Ljava_lang_String_2Ljava_lang_String_2__Ljava_lang_String_2 (MJIEnv env, int clsObjRef, int keyRef, int defRef) {
2.   int r = MJIEnv.NULL;
3.   if (keyRef != MJIEnv.NULL) {
4.     String k = env.getStringObject(keyRef);
5.     String defaultString = env.getStringObject(defRef);
6.     if (k==null)
7.       return MJIEnv.NULL;
8.     String v = System.getProperty(k);
9.     if (v != null)
10.      r = env.newString(v);
11.    else if (defaultString!=null) {
12.      r = env.newString(defaultString);
13.  }
14.  return r;
15.}

Figure 9. Peer method for System.getProperty(String, String)

As a result to date, we have created a MJI abstraction library and customized the AspectJ runtime library, to help resolve native methods introduced by AspectJ. We also created a test suite that includes all sample programs (excluding those that have native code in the Java program) from [9][10]. In our testing, all native code introduced by the AspectJ keywords has been successfully resolved, and we are currently investigating the whole runtime library (~180K) of AspectJ.

VI. SUMMARY

When we use model checking to verify real-world Smart Home applications, we have found a strong need to vary features of a formal model and customize the verification procedure. This is required as part of the efforts to conquer the state space explosion problem, by removing irrelevant details. However, varying features for a formal model tends to result in modifications scattering in multiple Java files, and it is even worse when we vary the combination of different features to study feature interactions. With examples, we have shown that these crosscutting concerns can be modularized with aspects, and can thus be easily enabled and disabled. We also point out that AspectJ may introduce native code to a pure Java program, and show an example that we can leverage the MJI scheme to solve it. We are currently building a MJI abstraction library, which will benefit not only our paradigm to advise model construction with AspectJ, but also AspectJ applications in general that have been developed without the awareness of model checking. In general, if we deem a model checker as an explorer for counter examples, the overall methodology enables easy exploration of solution space with very different features and goals.

We are also investigating the performance overhead introduced by AspectJ and looking for ways to minimize such impact. This occurs because additional state variables are introduced, e.g., aspect initialization. In our current experience, the state space increase of a complex model is relatively small and doesn’t appear to be a show stopper. This problem can be further mitigated with search heuristics, e.g., one can enforce atomicity for aspect initialization. This brings up an interesting distinction between model checking AspectJ programs and using AOP techniques to ease model checking Java programs.

REFERENCES