Formal Modeling: A Framework-based Approach

Zebin Chen
University of Oregon
Computer and Information Science Department
Eugene, OR 97405
1-541-346-0489
zbchen@cs.uoregon.edu

ABSTRACT
The overhead in constructing a workable formal model places a major obstacle in the path of using model checkers. With its success in software development, we hypothesize that a framework-based approach can also be effective in formal model development. We set out to test our hypothesis by using a formal model framework to model check a real-world application in the domain of digital home systems. In this proposal, we explain the motivation, proposed approach, expected contributions and evaluation metric of the proposed research, backed by a case study we have done.

Categories and Subject Descriptors
D.3.3 [Software Engineering]: Reusable Software – *reuse models, domain engineering, reusable libraries.*

General Terms
Verification.

Keywords
Framework, JPF, Model Checking, OSGi, Knopflerfish

1. INTRODUCTION
Our research group has actively participated in the research and design of digital home applications for cognitively impaired populations [1][2]. Typically, applications in this domain are required to satisfy certain properties with respect to privacy, availability, etc., and they are inherently multi-threaded, reactive systems that are prone to concurrency errors. We have a strong need for methodologies to assure system correctness.

We are also advocates of formal methods. We have relied on model checking to uncover software bugs in real systems [3]. However, constructing a checkable formal model for a real system does not come out on top in ROI discussions: the effort needed to build a model from scratch outweighs the benefits it may return. We found the construction of formal models an expensive procedure that requires expertise unavailable for real programmers. Furthermore, most projects are under time constraints. Managers cannot wait for lengthy model building efforts before commencing implementation and deployment.

The difficulty in constructing formal models motivates us to seek a way to leverage model-building experience from experts. There are a plethora of tools and methodologies that have grown up around the program building effort, including both design and implementation tasks. If one views formal modeling as a programming activity, why can't we place the same effort in formal modeling? In my dissertation, I look at one aspect of this problem, the framework-based approach.

Software frameworks are generic, semi-complete solutions that may be specialized for varieties [4]. In Object-Oriented languages, they typically consist of a set of classes in a particular domain, specifying interactions among entities and providing common facilities in the domain. With an existing framework, a developer leverages the interaction patterns and the common facilities provided in the framework, and customizes the framework for specific needs by filling hooks and slots in the framework. The framework-based approach has been well adopted in software development. For example, frameworks like Microsoft Foundation Class (MFC) and Abstract Window Toolkit (AWT) are widely used today. Software frameworks provide benefits along at least two dimensions: (a) they provide pure code reuse, allowing a programmer to code only pieces that are unique to his or her specific application, and (b) they capture good, tried-and-true design decisions that a programmer can build on. In this thesis, I hypothesize that similar benefits can be achieved in the construction of formal models.

This approach rests on the effectiveness of the modeling framework classes to (a) capture important details of the system, and (b) abstract away details that cause model checking to fail. This thesis takes on this dual task. In particular, the thesis is that domain-specific modeling classes can be defined such that they can be substituted into an implementation, turn-up interesting error scenarios, and allow the exhaustive search of model checking to remain feasible. My thesis rests on the postulate that the new breed of Java-based model checkers are a strong foundation from which to start [8]. These new model checkers can use an implementation somewhat directly as the model itself. I will discuss how the framework idea can be used to fill in the missing pieces.

The remainder of the proposal is organized as follows: in section 2 we explain OSGi applications, the application context that we are interested in; in section 3 we explain JPF, a model checker that we use in our project; in section 4, we sketch a modeling framework that we will build for the thesis, and use a case study to show that it can be effectively reused to construct models and find violations; in section 5 we describe the contributions and evaluation metrics; in section 6, we review the related work.

2. THE APPLICATION CONTEXT
The proliferation of digital home applications brings new challenges for software engineering. Without a common

standard, it is difficult to accommodate applications for different devices and different platforms. Subsequently, many commonalities, like device management, admin management, etc., have to be created anew for each project. The Open Service Gateway Initiative (OSGi) aims at providing a standard platform for the network delivery of managed services to local networks and devices [5].

Essentially, the OSGi specification defines a novel dynamic module system for Java [6]. Figure 1 shows the layered structure of OSGi. The framework is built on top of the JVM and allows customization through extensions called *Bundles*. There are four layers in the framework. The layer 0 is the execution environment, which can be any Java 2 configuration and profile. The layer 1 is the modules layer, which adds modularization to the Java classpath by adding private classes and static linking between *Bundles*. The layer 2 is the life cycle layer, which defines dynamic behaviors for a *Bundle* like install, start, stop, update and uninstall. The layer 3 is the service registry layer, which provides a dynamic cooperation model for *Bundles*. The novelty is that a *Service* is fetched by looking up by its name rather than a direct reference to its implementation. *Services* and *Bundles* can communicate via a predefined event scheme. Figure 1 shows the Desktop Bundle using the Console Service provided by the Console Bundle, which is a real example from *Knopflerfish* [7] and will be used in our case study.

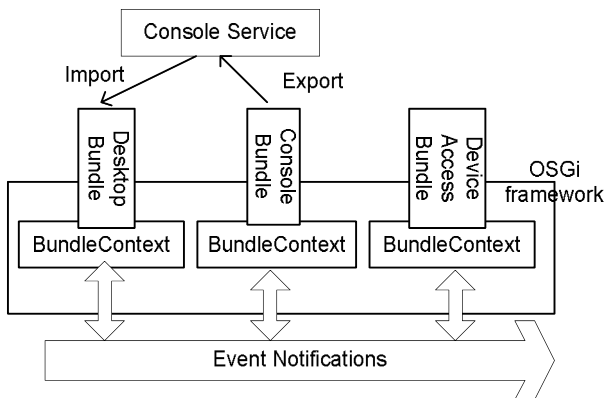

Figure 1. An architectural overview of the OSGi framework

There exist several reference implementations for the OSGi specification. The particular one we are looking at is *Knopflerfish*, which has been certified to be OSGi compliant by the OSGi alliance [7]. An interested reader can refer to [5][6][7] for a full explanation of OSGi and Knopflerfish.

3. JAVA PATHFINDER

Since the Knopflerfish framework is written in Java, we would rather use a model checker that directly takes Java as input. Java PathFinder (JPF) [8] [9] is such a model checker. It is an explicit model checker that directly checks Java bytecode and is extensible with different data and search heuristics. To accomplish these, JPF has a specialized virtual machine (VM), which takes on the role of a model checking engine like state comparison, query, cache and restoration. Furthermore, through a well-defined component architecture, JPF allows one to customize the search procedure. For example, it has a configurable *Choice Generator* to choose and order choices at a branching point, a *Search Listener* to monitor certain execution environment instructions to allow extensions, and a *Model Java Interface (MJI)* to intercept Java method calls. The extension schemes in JPF allow expert users to experiment with novel search heuristics, state representations, etc., which may have significant performance improvements in a specific domain.

There are two restrictions on JPF. First, it is not able to check platform-specific native methods, since the execution information is not available to the JVM. Second, due to state space explosion, it typically deals with Java programs with up to 10k lines-of-code [9]. Therefore, before using JPF to check a real system, one would have to resolve Java native methods with MJI or name substitution, and apply various reduction techniques to construct an abstract model for the actual system so that the verification run may complete within a reasonable time and with a reasonably available main memory. Additionally, one may also need to specify the interactions with the environments (e.g. user inputs) and the emulators for hardware devices (e.g. a set top box), if they are included in the modeling system.

4. A CASE STUDY

In this section, we show a case study of model checking OSGi applications in a framework-based approach. The full life-cycle consists of several stages, as follows.

1). Domain analysis and critical properties identification.

To mitigate state space explosion problem, one needs to abstract away unnecessary details. In general, the abstraction is done under the guidance of the desired properties. Since it is unaffordable to build a formal model for every single desired property, we have to choose a set of correctness properties critical to the proper operation of the system. Domain analysis also helps estimate the complexity of the system, so that the model developer may build the model at the proper level of abstraction.

For example, one novelty of the OSGi framework is that it accommodates “future” devices by separating interfaces (*Services*) from implementations (*Bundles*). However, a service provider may register and unregister services on-the-fly; therefore, a service may become stale if its provider unregisters it. The behavior of a stale service is undefined in the OSGi specification; in the Knopflerfish implementation, it may work properly, throw various exceptions, or behavior unpredictably. Therefore, it is critical to assure the validity of a service.

2). Construct a formal model that corresponds to the OSGi framework. This includes the following steps:

- Specify a set of peer classes that resolve native methods in the JDK library, by either MJI or software simulation. This mainly involves classes in *java.net* and *java.io*. For example, to check proper synchronization of socket communication, we simulate blocking read/write with *Object.wait()* and *Object.notifyAll()* in producer/consumer style.

- Specify a set of hardware emulators. This is needed as the JVM has no execution information about the hardware. For example,

we create an emulator to simulate a set top box. In the simplest case, we ignore the details of communication and use a finite state machine to represent device states.

- Construct a closed environment to run model checking. Environmental inputs, including human interactions and sensor events, shall be explicitly enumerated and non-deterministically injected. For example, starting and stopping bundles are done by humans in OSGi applications, while we specify these actions as non-deterministic choices in the formal models. In many cases, constructing a closed environment is the last step to complete a modeling skeleton with main() method.

- Reduce the system specification to an abstract level. This is a critical step to assure the coverage of model checking. We have tried various methodologies, e.g. automatic reduction like Bandera or manual abstraction with various techniques [10]. There are also different specification patterns that tend to reduce the state space [14]. Knowledge in the application domain also helps make wise decisions towards significant reduction. In general, manual abstraction with expertise in both domains allows more aggressive reduction at large. For example, we know that the OSGi framework has a plug-and-play component structure, thus we can focus on interesting Bundles without plugging in unrelated Bundles. Moreover, to check service validity, bundle serialization is related to bundle activation and service registration thus won't be removed by automatic reduction. However, serialization won't invalidate a service. Therefore, we abstract away all details of bundle serialization by directly instantiating bundle instances.

3). Apply the modeling framework to check framework applications.

As an example, we set out to check the service validity of OSGi applications. There exists service dependency between bundles in the Knopflerfish's implementation, e.g. the Desktop Bundles uses display service provided by the Console Bundle. Implementation of the two bundles needs to go through some modifications: Since we focus on service validity, we remove unrelated code lines in the two bundles. This procedure can be guided by compilation error messages, since the abstraction level is decided by the modeling framework.

Thereafter, we compile the two bundle models and run JPF. JPF reports an error trace like the follow (in a very abstract form):

Install Console Bundle
Check Service Reference Validity (in starting Desktop Bundle)
Get Service (in starting Desktop Bundle)
Uninstall Console Bundle
Use Service (in starting Desktop Bundle)
Error!

For our case study, the modeling code is similar to the implementation, thus we have no difficulty in pin-pointing the same error in the actual system. We decided the cause of the violation is due to the undesired interleaving of the Console Bundle thread and the Desktop Bundle thread, i.e. when a thread has checked the validity of a service, the other thread invalidates the service right away, therefore the first thread ends up with a stale service. A remedy is to force the execution of the start method of Desktop Bundle Activator as atomic.

With such modification, the reported violation disappears. However, JPF catches a *NullPointerException*, with the error trace shown in the below (in a very abstract form):

Install Console Bundle
Cache bundle context (in starting Desktop Bundle)
Start a thread for desktop UI (in starting Desktop Bundle)
Uninstall Desktop Bundle
Get Service reference (in the thread created by Desktop Bundle)
Error!

The violation is due to the delay of the desktop UI thread created in the Desktop Bundle, i.e. it starts execution as late as the Desktop Bundle has stopped. Subsequently, the created thread throws a *NullPointerException* when it uses the invalid Bundle Context. The problem can be eliminated by showing the desktop UI right away and only creating the new thread after the UI has been initialized and is waiting for the inputs. JPF doesn't find any violation in the revised model.

5. EVALUATION

In the above, we have shown an example to catch bugs and pin-point patches based on our modeling framework. With the existence of the modeling framework, the modifications required for the applications (the two Bundles) are modest. However, the catch of violations with such modest modeling effort is somewhat surprising: Knopflerfish is a leading reference implementation of the OSGi specification, and its source code has been publicly available for years. From another angle, it demonstrates the effectiveness of our modeling framework.

As a current result, we have built a modeling framework for OSGi applications as a test bed. It includes peer classes corresponding to the application code roughly in a 1-to-1 relation. The current modeling framework consists of tens of modeling classes for the core infrastructure, and hundreds of classes for the standard bundles accompanied with the core infrastructure. We also model a subset of JDK classes that are used in the modeling framework.

We hypothesize that a modeling framework must be generic to be effectively reused. Considering our case study, the stale service problem is common in OSGi-based applications; therefore, the skeleton to check the Desktop bundle and the Console bundle can be directly reused to check any service dependency. Also, the modeling framework may also be reused to check other aspects of the reminder applications, e.g., it may be reused to check whether two reminders will compete for display. Finally, the peer classes built for JDK and the hardware emulators are generic for applications in the same domain. Many of these components are inherently multi-threaded, have complex semantics and require domain-specific knowledge to specify in the proper level of abstraction.

The following questions will be used to measure the success of the research.

- Is it feasible to specify a modeling framework at a specific abstraction level and be reused later? Will the coverage be so low as to omit violation cases, or will the model be so abstract as to generate too many spurious false alarms? A measurement to

this question is to observe how many errors are reported by our modeling applications, and how much percentage of false alarms out of the total errors is reported. The problem will be less severe if framework applications have similar complexity.

- How much modeling effort can we save, once a modeling framework is built? How much overhead does it bring to understand a modeling framework and link it to the application code? A measurement to this question is to observe the proportion of effort dedicated to the modeling framework compared with to the applications, which can be quantified by code line and human time in an ROI analysis. We are also interested in methodologies and tools to ease modeling the extensions and link the error trace to the application code.

- How often do we need to build a new skeleton to drive a run? What is the better way to reuse a modeling framework, a vertical framework or horizontal framework? Like the first two cases of reuse in the above, a vertical framework allows reuse at large, but it is limited to a certain properties. A horizontal framework (like JDK peer classes in our example) has wider scope, but it takes more efforts to use them. We hypothesize that multiple light-weight vertical skeletons may be the best way out.

To benefit the software community, we will report violations we have uncovered to Knopflerfish and propose our revisions to them. As an example, this may also help the OSGi alliance to standardize the management of services, as service management is so critical and so common in OSGi applications.

6. RELATED WORK

The overhead in model construction has been recognized by the formal modeling community. Researchers have thus proposed various ways to ease formal specifications.

Dwyer et al. pioneered the work by proposing property elicitation with a small number of property patterns [10]. Timeline editor provides a visual, intuitive interface for property specification. Both these works gained popularity as they effectively catch commonality in a specific domain. However, model construction is much more complicated than property specification.

Bandera [12] and MODEX [11] aim at automatic model construction, by translating implementation code into modeling code via a transformation table. While they are useful auxiliary tools for model construction, in general, automatic transformation is a rigid research topic. In most cases, human decision is much needed to apply abstraction aggressive enough to accommodate a complex system, and not all transformations can be expressed in a transformation table.

While new model checkers and new data representation and search heuristics are emerging all the time, there is little work systematically addressing the model construction as a programming activity. In [13], a generic model checking framework is built to catch regularities of publish-subscribe system and ease model construction. The case study is interesting but very small, and the ROI analysis is very limited at the best. In [14], specification patterns are studied to construct efficient Spin models. However, the specification patterns proposed are reused in a small scale and at a conceptual level; we are more interested in reuse at large, code reuse as well as design reuse. The importance of a modeling library is recognized in [8], with the motivation to reduce the overhead in library calls. This is part of the efforts we are pursuing, but it has raised far less attention as in the application domain. To our best knowledge, there is no report about domain-specific modeling framework built for JPF. We thus set out to bring model checking for real programmers via a framework-based approach.

7. REFERENCES

[1] Chen, Z., Fickas, S. The Plain Old Television in a Smart Apartment. In *Proceeding of International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*, San Jose, CA, Dec. 2005.

[2] Fickas, S., Pataky, C., Chen, Z. DuckCall: Tracking the First Hundred Yards Problem. To be presented in *The Eighth International ACM SIGACCESS Conference on Computers & Accessibility*, Portland, OR, in Oct. 2006.

[3] Feather, M., Fickas, S., Razermera, A., Model-Checking for Validation of a Fault Protection System. In *Proceedings of IEEE International Symposium on High Assurance Systems Engineering*. Boca Raton, 2001, 32-41.

[4] Fayad, M., Schmidt, D. and Johnson, R., *Building Application Frameworks: Object-Oriented Foundations of Framework Design*. John Willey & Sons. 1999.

[5] The OSGi Alliance. *OSGi Service Platform (Release 3).*IOS Press, Mar. 2003. http://www.osgi.org/documents/collateral/TechnicalWhitePaper2005osgi-sp-overview.pdf

[6] The OSGi Alliance. *About the OSGi Service Platform (Technical Whitepaper).* Nov. 2005.

[7] Knopflerfish, http://www.knopflerfish.org/

[8] Visser, W., Havelund, K., Brat, G. Park, S. and Lerda, F. Model Checking Programs. In *Automated Software Engineering*, 10, 2 (Apr. 2003), 203-232.

[9] Mehlitz, P., Visser, W. and Penix, J. The JPF Runtime Verification System, manual accompanied in JPF distribution. http://sourceforge.net/projects/javapathfinder

[10] Dwyer, M., Avrunin, G., Corbett, J. Patterns in Property Specifications for Finite-state Verification. In *Proceedings of the 21st International Conference on Software Engineering*,. 1999, 411-420.

[11] Holzmann, G. *The Spin Model Checker: Primer and Refrence Manual*. Addison-Wesley. 2003.

[12] Corbett, J., Dwyer, M., Hatcliff, J., Laubach, S., Pasareanu, C., Robby, Zheng, H. Bandera: Extracting Finite-state Models from Java Source Code. In *Proceedings of the 22nd International Conference on Software Engineering*. 2000, 439-448.

[13] Garlan, D., Khersonsky, S., Kim, J.S., Model Checking Publish-Subscribe Systems. *In Proceedings of the 10th SPIN Workshop*. Portland, OR, 2003, 166-180.

[14] Ruys, T. C. *Towards Effective Model Checking*. Ph.D. Thesis, University of Twente, Deventer, The Netherlands, 2001.