A Formal Specification in JML of Java Security Package

Poonam Agarwal*, Carlos E. Rubio-Medrano, Yoonsik Cheon and Patricia. J Teller
The University of Texas at El Paso,
500 W. University Avenue, El Paso, TX 79968 U.S.A.
{pdagarwal, cerubio, ycheon, pteller}@utep.edu

Abstract-**The Java security package allows a programmer to add security features to Java applications. Although the package provides a complex application programming interface (API), its informal description, e.g., Javadoc comments, is often ambiguous or imprecise. Nonetheless, the security of an application can be compromised if the package is used without a concrete understanding of the precise behavior of the API classes and interfaces, which can be attained via formal specification. In this paper, we present our experiences in formally specifying the Java security package in JML, a formal behavior interface specification language for Java. We illustrate portions of our JML specifications and discuss the lessons that we learned, from this specification effort, about specification patterns and the effectiveness of JML. Our specifications are not only a precise document for the API but also provide a foundation for formally reasoning and verifying the security aspects of applications. We believe that our specification techniques and patterns can be used to specify other Java packages and frameworks.**

I. INTRODUCTION

Java makes it possible to write secure applications by providing a security model based on a sandbox, an execution environment in which a program runs and the program's execution is confined within certain bounds [7]. The classes in the java.security package as well as those in the security extension allow security features to be added to an application [9]. It is necessary to precisely understand the behaviors of these classes and interfaces because a small misuse of an API can significantly compromise the security of an application. However, an informal document, such as Javadoc comments, often is inadequate for precisely specifying the behavior of the API because of the inherent nature of a natural language, i.e., its ambiguity and impreciseness. Even though the source code can provide precise information, in it essential features are tangled with implementation decisions and details.

A formal specification may complement the informal document, since it can document essential features in a concise and precise manner. A formal behavioral interface specification language, such as JML [5], is designed to precisely document both the syntactic interface and the behavior of program modules. A formal behavioral interface specification provides not only a precise document describing the API, but also a formal foundation for rigorously proving properties about the implementation [1] [8].

In this paper, we present a specification case study of the Java security package using JML. This work is currently being done under the Milaap Project, the research goal of which is to unify and integrate several different verification methods. During this study, we specified a significant number of core classes of the security package. In this paper we present some of the interesting classes and discuss the lessons that we learned from this specification effort. The case study allowed us to identify several specification patterns that facilitate writing JML specifications. It also permitted us to critically evaluate the effectiveness of JML as an API documentation language, which led us to a JML wish list. We expect our specifications to be a valuable document to the users of the Java security package, and we believe that our specification techniques and patterns are applicable to the specification of other Java packages and frameworks.

Our work is not the first to formally specify Java packages. The JML distributions are shipped with specifications of several JDK classes, such as collection classes (refer to the JML website at www.jmlspecs.org). Our main contribution is not only the final specifications, but also identification of reusable specification patterns and a critical evaluation of JML. Poll et al. specified in JML all the classes of the Java Card API [8], and their specification made many of the implicit assumptions underlying the implementation explicit; this agrees with our findings. Their specifications are written in a lightweight style, while most of our specifications are written in a heavyweight style. Catano and Huisman found a similar result in that even lightweight use of formal specifications contributed greatly to the improvement of the quality of applications [2].

The remainder of the paper is organized as follows. The Java security package and JML are described in Section II. Sections III and IV focus on the two main aspects of the security package: protection mechanism and cryptographic architecture. A discussion of the results and conclusions of our specification case study conclude the paper.

II. BACKGROUND

A. The Java Security Package

The Java security package (java.security) provides a framework for the Java security architecture, which can be broadly classified into two different aspects of security: protection mechanism and cryptographic architecture (see Fig. 1).

* The work of authors was supported in part by the NSF, CNS-0509299.

K. Elleithy (ed.), Advances and Innovations in Systems, Computing Sciences and Software Engineering, 363–368.
© 2007 *Springer*.

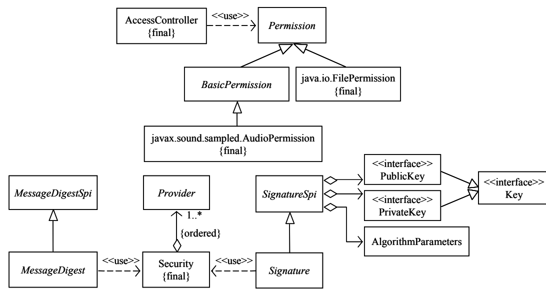

Fig. 1. Security-related classes

The protection mechanism deals with access control and prevention of unauthorized modifications. It is built upon concepts such as code sources (locations from which Java classes are obtained), permissions (requests to perform particular operations), policies (all the specific permissions that should be granted to specific code sources), and protection domains (particular code sources and permissions granted to those code sources) [7]. The class AccessController determines based on the security policy in effect, whether an access to a resource should be granted or denied. Any section of code that performs a security-sensitive operation should consult the access controller to verify if the operation is allowed. An operation or access is represented by permission, and different types of permissions form a class hierarchy rooted at the abstract class Permission.

The Java Cryptography Architecture (JCA) deals with authentication and supports algorithm and implementation independence [7]. Some of the JCA core classes are Security (handling all installed providers and security properties), Provider (interface to the concrete implementation of the cryptographic services), engine classes such as MessageDigest and Signature, and key and parameter classes such as Key and AlgorithmParameters. The code below shows a typical use of the MessageDigest engine class that provides an interface to a message digest (hash) algorithm, such as MD5 and SHA. A message digest is a secure one way hash function that takes arbitrary-sized data and returns a fixed-length hash value.

```
MessageDigest md = MessageDigest.getInstance("MD5","SUN");
md.update("Hi, JCA!".getBytes());
byte[] hash = md.digest();
```

Every engine class implements factory methods, named *getInstance*, each of which returns an instance of the specified algorithm from the optionally specified provider. To find an appropriate algorithm implementation, the factory methods ask the Security class, which, in turn, consults the available providers to check whether they can supply the desired algorithm. The example code gets an instance of Sun's implementation of the MD5 algorithm. The *update* method adds a sequence of bytes to the internal buffer, and the *digest* method computes the message digest of the bytes stored in the internal buffer.

B. JML

The Java Modeling Language (JML) [5] is a formal interface specification language for Java and describes both the syntactic interface and the behavior of Java program modules. The syntactic interface of a Java class or interface, commonly called an application programming interface (API), consists of the signatures of its methods and the names and types of its fields. The behavior of a program module is specified by writing, among other things, pre and postconditions of the methods exported by the module. The assertions in pre and postconditions are written using a subset of Java expressions and are annotated in the source code. The pre and postconditions are viewed as a contract between the client and the implementer of the module. The client must guarantee that, before calling a method *m* exported by the module, *m*'s precondition holds, and the implementer must guarantee that, after such a call, *m*'s postcondition holds. A method specification can consist of, among other things, a precondition (specified in the *requires* clause), a frame axiom (specified in the *assignable* clause), and a normal or exceptional postcondition (specified in the *ensures* or *signals* clause).

III. ACCESS CONTROL

The access control mechanism is built upon concepts such as code sources, permissions, policies, and protection domains [7]. A permission—an encapsulation of a request to perform a particular operation—is a key concept underlying the access control mechanism and provides an interesting aspect on formalization, as there exist several types of permissions organized into a class hierarchy, with some sharing certain common properties. In this section, we specify representative methods of the permission classes.

A. Permission Classes

A permission, a specific action that code is allowed to perform, consists of three components: *type, name,* and *actions.* The *type,* which specifies the type of the permission, is represented by a particular Java class that implements the permission. The *name* is based on permission type, e.g., the name of a file permission is a file or directory name, and a few permissions require no name. The *actions* also vary based on the permission type, and many permissions have no actions associated with them. The *actions* specifies what may be done to the target of the permission, e.g., a file permission may specify that a particular file can be read, written, deleted, executed, or some combination of these. In this paper we specify two representative classes and their superclasses, one for name-based permissions and the other for the name and action-based permissions. We focus only on one method in permission classes, *implies*, that determines whether one permission implies another permission. The *implies* method is one of the most important methods of permission classes because it is the primary method used by the access controller to make access decisions. We illustrate the specification technique that we used to factor out common or general properties into the superclass and leave the specifics to subclasses.

The abstract class Permission is the ancestor of all permission classes (see Fig. 2). In JML, specifications are typically annotated in

source code as special comments, i.e., //@ and a pair of /*@ and @*/, and the specification of a method precedes its declaration. All permissions have a name—the interpretation of which depends on the subclass—and several abstract methods (e.g., *implies*) that define the semantics of the particular subclass. The JML modifier *spec_public* states that the field is treated as public for specification purpose; e.g., it can be used in public specifications. The specification of the *implies* method illustrates a specification pattern that we use to leave the specification of subclass-specific properties to subclasses. The keyword *normal_behavior* specifies the behavior of the method when it terminates normally, i.e., without throwing an exception. The *pure* modifier states that the method has no side-effect; only pure methods can appear in JML assertions. The postcondition states that the method should return true if and only if (a) the given permission, p, is of the same type as the receiver, and (b) jmlImpliesPerm(p) holds. Property (a) is common to all permission classes and, thus, is specified in this class. However, determining whether the receiver implies the argument permission is subclass-specific because different subclasses may have different names or actions, and their interpretations may be different. Thus, it cannot be completely specified in the class Permission. Our approach is to delegate the specification responsibility to subclasses by introducing a model method, i.e., *jmlImpliesPerm*, as shown in property (b). A model method is a specification-only method and can be used only in assertions. Each (concrete) subclass is responsible for overriding this model method to specify the subclass-specific precise meaning of the *implies* method (see BasicPermission and FilePermission below).

```
public abstract class Permission implements Guard, java.io.Serializable {
  private /*@ spec_public @*/ String name;

  /*@ public normal_behavior
    @ assignable this.name;
    @ ensures this.name == name; @*/
  public Permission(String name);

  /*@ public normal_behavior
    @ ensures \result <==> getClass() == p.getClass() && jmlImpliePerm(p);
    @*/
  public abstract /*@ pure @*/ boolean implies(Permission p);

  //@ public model pure boolean jmlImpliesPerm(non_null Permission p);
  // ...
}
```

Fig. 2. Partial specification of class Permission

The class BasicPermission is an abstract subclass and implements permissions that have a name (or target) string but do not support actions. It implements hierarchical property names, i.e., names with dots. An asterisk may appear by itself, or if immediately preceded by a ".", it may appear at the end of the name, to signify a wildcard match (see the *implies* method below). The BasicPermission class does not introduce any new state component. However, it constrains the inherited state by requiring the name to be non-null, as stated in the invariant clause below.

```
//@ public invariant name != null;

/*@ public normal_behavior
  @ assignable this.name;
  @ ensures this.name == name;  @*/
public BasicPermission(/*@ non_null @*/ String name);
```

The BasicPermission class provides a simple wildcarding capability, e.g., "*" implies permission for any target and "x.*" implies permission for any target that begins with "x.". This is implemented by the *implies* method, the behavior of which is specified below indirectly by overriding the inherited model method *jmlImpliesPerm*. The *ensures_redundantly* clause specifies facts that can be inferred from other assertions, such as those inherited from the superclass, and the *old* clause introduces a short name for an expression.

```
/*@ also public normal_behavior
  @ ensures_redundantly \result ==> (p instanceof BasicPermission)
  @     && jmlImpliesPerm(p); @*/
public /*@ pure @*/ boolean implies(/*@ non_null @*/ Permission p);

/*@ also public normal_behavior
  @ old String n1 = name;
  @ old String n2 = p.name;
  @ ensures \result <==> n1.equals(n2) || "*".equals(n1) ||
  @     (n1.endsWith(".*") && n1.length() < n2.length() &&
  @      n1.regionMatches(0, n2, 0, n1.length()-1));
  @ public pure model boolean jmlImpliesPerm
  @     (non_null Permission  p) { /*...*/ } @*/
```

An example of a concrete class that implements a simple named permission is the class AudioPermission defined in the package javax.sound.sampled. It represents access rights to the audio system resources, and the permission string (or name) can be play, record, etc. The class only defines a constructor; all the permission methods are inherited from the superclasses.

The class FilePermission is a final, concrete subclass of Permission and consists of a pathname and a set of actions. An action represents access that can be granted to a pathname, and a possible value is "read", "write", "execute", or "delete". The pathname is modeled by the *name* field inherited from Permission, and the actions are modeled by the model field *actions* of type JMLValueSet (see below).

```
//@ public model non_null JMLValueSet actions;
//@ public invariant actions.isSubset(FILE_ACTIONS);
```

The JML model class JMLValueSet, from the package org.jmlspecs.model, defines immutable sets that use equals for a membership test. The *invariant* clause states that actions can contain only valid action names; FILE ACTIONS, not shown here, is a specification-only constant denoting the set of all file actions. In the implementation, the actions are represented as a bit mask (see below). The JML *represents* clause specifies an abstraction function that maps program values such as a bit mask to abstract values such as a JMLValueSet; the model method *toSet* that does this mapping is not shown in this paper. That is, the value of the model field *actions* is given by the program field *mask*. The *in* clause states that *mask* belongs to the data group of *actions*, meaning that a method that can change the value of *actions* also can change the value of *mask* [6]. Note that the *represents* clause is private and, thus, is for the implementer, not for the client of this class.

```
private transient int mask; //@ in actions;
//@ private represents actions <- toSet(mask);
```

The constructor establishes the invariant about actions by requiring the action string, *acts*, to be a sequence of comma-separated file actions (see below). This is indirectly asserted

by stating that the value toSet(acts) should be a subset of FILE_ACTIONS. The overloaded model method *toSet* converts a sequence of comma-separated strings to a JMLValueSet. As shown below, a model method also may have the method body. The constructor is allowed to mutate the model field *actions* and, thus, is allowed to change the program field *mask* to initialize it.

```
/*@ public normal_behavior
  @ requires toSet(acts).isSubset( FILE_ACTIONS);
  @ assignable name, actions;
  @ ensures name == path && actions.equals(toSet(acts)); @*/
public FilePermission(/*@ non_null @*/ String path,
                      /*@ non_null @*/ String acts);
```

```
/*@ public model pure JMLValueSet toSet(non_null String acts) {
  @   JMLValueSet r = new JMLValueSet();
  @   StringTokenizer tok = new StringTokenizer(acts, ",");
  @   while (tok.hasMoreTokens())
  @      r = r.insert(new JMLString(tok.nextToken().trim().toLowerCase()));
  @   return r;
  @ } @*/
```

The specification of the overriding *implies* method is shown below. As in the class BasicPermission, its precise behavior is specified by overriding the model method *jmlImpliesPerm*.

```
/*@ also public normal_behavior
  @ ensures_redundantly \result <==>
  @    p.getClass() == FilePermission.class  && jmlImpliesPerm(p); @*/
public /*@ pure @*/ boolean implies(/*@ non_null @*/ Permission p);
```

```
/*@ also public normal_behavior
  @ ensures \result <==> actions.isSuperset(((FilePermision) p).actions)
  @    && jmlImpliesPath(p.name);
  @ public model pure boolean jmlImpliesPerm(non_null Permission p);
  @*/
```

The *jmlImpliesPerm* method defines the *implies* relation of file permissions, based on both actions and file path. Its postcondition asserts that the return value be true if and only if the receiver's action set includes all the actions of the argument permission, p, and jmlImpliePath(p.name) is true. A new model method *jmlImpliesPath*, not shown in this paper, formulates the *implies* relation on file path names, e.g., both "/tmp/*"and "/-" imply "/tmp/t.txt".

IV. CRYPTOGRAPHY

The Java Cryptographic Architecture (JCA) includes APIs for message digests, digital signatures, and key and certificate management. JCA is based on the provider architecture to support multiple and interoperable cryptography implementations. In this section we describe key JCA classes such as Security, MessageDigest, and Signature.

A. Security Class

This final class manages the installed providers and the security properties, i.e., it centralizes all security properties and common security methods. For this, it provides a set of static methods, including methods for adding new providers or properties, retrieving existing providers or properties, and removing existing providers. All of these are typical methods for managing a collection of data. The class also defines a method named *getAlgorithms* that returns a set of strings containing the names of all available algorithms or types for the specified cryptographic service, e.g., message digest and signature. In this section, we specify the *getAlgorithms* method, but first let us define the abstract model of the class Security.

```
/*@ public static model non_null JMLObjectSequence jmlPrs;
  @ public static invariant !jmlPrs.has(null)
  @     && (\forall Object o; jmlPrs.has(o); o instanceof Provider)
  @     && jmlPrs.size() == jmlPrs.toSet().size();  @*/
private static /*@ non_null @*/ Provider[] providers; //@ in jmlPrs;
/*@ private static represents jmlPrs <-
  @     JMLObjectSequence.convertFrom(providers);  @*/
```

This specification shows an interesting pattern. For our purpose, it is sufficient and even advantageous to model the class as a sequence of providers, but the implementation uses an array of providers, a private field named *providers*. We may use this array as our abstract model, but we opted for sequences by introducing a model field *jmlPrs*. The resulting specification is more abstract and maintainable. For example, a change of representation has only one affect, i.e., it affects the *represents* clause; the rest of the specification, such as pre and postconditions, remains the same. In addition, sequences are a lot easier to manipulate in pre and postconditions. As stated in the invariant, *jmlPrs* contains only instances of Providers, no null, and no duplicates.

The specification of *getAlgorithms* is given below. It looks a bit complicated due to nested quantifiers, but it precisely documents that the returned value is a set of strings, the elements of which are collected from the keys of the properties (maps) of all providers. The elements are built from the keys that contain no blanks and the prefixes of which match case-insensitively the given service name (n) by discarding the matching prefix and the next character ("."). For example, if the key is "MessageDigest.MD5" and the service name is "MessageDigest", then "MD5" is added to the result.

```
/*@ public normal_behavior
  @ requires n != null && n.length() > 0 && !n.endsWith(".");
  @ ensures (\forall Object o; \result.contains(o); o instanceof String) &&
  @     (\forall String s; \result.contains(s) <==>
  @         (\exists Provider p; jmlPrs.has(p);
  @             (\exists String k; algKeys(p, n).contains(k);
  @                 s.equals(k.toUpperCase().substring(n.length()+1))))); @*/
public static /*@ pure non_null @*/ Set getAlgorithms(String n);
```

Given a provider and a service name, the model method *algKeys* returns the set of keys of the provider that contain no blanks and the prefixes of which match case insensitively the given service name.

B. Engine Classes

The engine classes provide different cryptographic services, such as message digests and digital signatures. All engine classes are abstract and define several factory methods named *getInstance* to create objects that implement specific cryptographic algorithms. All factory methods take an algorithm name, and some also take an optional provider name or object. If the provider is not specified, an algorithm available from the default provider is returned. In addition to factory methods, each engine class defines a set of service-specific API methods.

The MessageDigest engine class provides an interface to a secure one-way hash function that takes arbitrary-sized data and returns a fixed-length hash value, called a *message digest*. We specify all the methods (i.e., *getInstance*, *update*, and *digest*) used in the example in Section II.A. For these methods, it is sufficient to model a message digest object as a sequence of bytes to be digested, as shown below; i.e., we ignore other implementation fields.

```
//@ public model non_null JMLValueSequence data; initially data.isEmpty();
```

The JML model class JMLValueSequence, imported from the org.jmlspecs.models package, defines an immutable sequence of values. Initially, the model field data is empty, i.e., there is nothing to digest.

The class MessageDigest defines several factory methods to create new instances. Specified below is the one that takes an algorithm name (e.g., SHA and MD5) and a provider name.

```
/*@ public normal_behavior
  @   requires Security.getProvider(p) != null && (* alg available from p *);
  @   ensures \fresh(\result) && \result.data.isEmpty() &&
  @        \result.getAlgorithm().equals(alg) &&
  @        \result.getProvider().getName().equals(p);
  @ also public exceptional_behavior
  @   requires Security.getProvider(p) != null && (* no such alg from p *);
  @   signals (NoSuchAlgorithmException e1);
  @ also public exceptional_behavior
  @   requires Security.getProvider(p) != null;
  @   signals (NoSuchProviderException e2); @*/
public static /*@ pure @*/ MessageDigest getInstance(
   /*@ non_null @*/ String alg, /*@ non_null @*/ String p)
   throws NoSuchAlgorithmException, NoSuchProviderException;
```

This specification uses informal descriptions in its preconditions. In JML, one can escape from formality by enclosing plain English or other applicable languages in a pair of (* and *). If the named algorithm is available from the named provider, this method should return a new object of type MessageDigest; otherwise, it should throw an exception. The *\fresh* expression in the *ensures* clause asserts that the object is newly created, i.e., it does not exist in the pre-state but does exist in the post-state. The specification also shows that one can mix formal and informal descriptions in JML assertions.

One uses *update* methods to append data to a message digest object and then calls *digest* methods to actually compute a message digest of the accumulated data. There are several forms of *update* and *digest* methods. The ones used in Section II.A are specified below.

```
/*@ public normal_behavior
  @ assignable data;
  @ ensures data.equals(\old(data.concat(toSeq(d)))); @*/
public void update(/*@ non_null @*/ byte[] d);

/*@ public static model pure JMLValueSequence toSeq(non_null byte[] b) {
  @   JMLValueSequence r = new JMLValueSequence();
  @   for (int i = 0; i < b.length; i++)
  @       r = r.insertBack(new JMLByte(b[i]));
  @   return r;
  @ } @*/

/*@ public normal_behavior
  @ assignable data;
  @ ensures data.isEmpty() && (* \result is hash of \old(data) *); @*/
public /*@ non_null @*/ byte[] digest();
```

The *update* method is specified in terms of the model method *toSeq*, which converts an array of bytes to a sequence of bytes. The postcondition states that the argument bytes are appended at the end; the expression *\old(e)* denotes the value of *e* evaluated in the pre-state. The digest method has a side-effect in that it also trashes the accumulated data.

The Signature engine class provides an interface to compute and verify digital signatures. It uses a private key to sign or produce a new digital signature and a public key to verify a signature; both keys are implemented as interfaces, i.e., PrivateKey and PublicKey. The API of this class is similar to that of MessageDigest, however, an interesting aspect of this class from a specification perspective is that the same object may be used for both signing data and verifying signed data. This means that methods should be called in a particular order. For example, to sign data, one must first call one of the *initSign* methods that initialize the object by supplying a private key; then call the *update* methods, possibly several times, to append data; and finally call one of the *sign* methods. A similar sequence of method calls (i.e., *initVerify*, *update*, and *verify*) are required to verify signed data. But how can this protocol property, i.e., ordering dependency among method calls, be specified? Since JML does not allow a protocol property to be expressed explicitly, we encode it as a finite state machine in pre and postconditions. For example, the following gives partial specifications of the *initSign* and *sign* methods.

```
protected /*@ spec_public @*/ int state;
//@ initially state == UNINITIALIZED;

//@ assignable state;
//@ ensures state == SIGN;
public final void initSign(PrivateKey k) throws InvalidKeyException;

//@ requires state == SIGN;
public final byte[] sign() throws SignatureException;
```

The field *state* represents the current protocol state and the allowed transitions are specified by manipulating it in pre and postconditions of the methods involved. For example, the *initSign* method sets the state's value to the constant SIGN, and the *sign* method requires that it be called in a state where the *state*'s value is SIGN.

V. DISCUSSION

An interface specification of a module may be written at different abstraction levels for different users, e.g., a highly abstract specification for the module's client and a more implementation-oriented specification for the module's implementer. JML facilitates writing specifications at different abstraction levels and, through specification visibility and model elements, mixing them in a single file. In our case study, we attempted to produce client-oriented specifications by defining abstract models for classes. Occasionally, however, we made connections to the representations through private abstraction functions (e.g., the Security class in Section IV.A).

Writing an interface specification is an iterative and incremental process, often starting from the definition of an abstract model to the specification of each method. We found that a concise mathematical notation such as Z, which makes

it easy to see commonalities and variations of different classes, especially when a complex class hierarchy is involved, is an excellent tool to sketch out abstract models. We also used UML class diagrams to identify abstract models: First, we reverse engineered a class diagram from source code to highlight various relationships among classes, and then extended or decorated it with model elements such as model fields and classes. Since JML facilitates incremental development of specifications, often we started with an informal/lightweight specification, which evolved into a formal/heavyweight one. Although JML also allows multiple specifications of a module to be in separate files, we didn't explore this refinement feature in our incremental development.

Another feature of JML that we used heavily in our case study is informal descriptions. This feature allowed us to tune the level of formality and mix formal and informal text in our specifications. We prefer a simple English description when a formalization does not add much or is practically impossible to completely formalize (e.g., the *digest* method in Section IV.B).

Additionally, we identified several features missing in JML. For example, we could not separate and specify cleanly the ordering of dependencies among methods [4]. We had to mix the so-called protocol properties with functional properties in pre and postconditions by encoding them as finite state machines (e.g., the Signature class in Section IV.B). Writing such specifications is laborious and error-prone, and, since the protocol properties have to be inferred, the specifications become hard to understand. Another missing feature may be a more succinct and expressive notation to manipulate mathematical structures such as sets and sequences, e.g., something like Haskell's list comprehension notation. Although JML provides so-called model classes for such mathematical structures, we found that the use of these classes often becomes cumbersome and verbose, e.g., we had to convert values back and forth between Java types and JML types used by model classes.

Finally, we mention several JML specification techniques or patterns that we used recurrently in this case study. The most frequently used pattern is "abstract with model fields", which produces an abstract, client-oriented specification that is easy to maintain and allows runtime assertion checking. The approach is to introduce model fields to define the abstract model of a class and to write pre and postconditions in terms of the model fields (see Section IV.A). If the representations of the model fields are known, private abstraction functions may be specified to make pre and postconditions checkable at runtime [3]. Another pattern illustrated in this paper is "delegate to model methods", which is used to specify method overrides (see Section III.A). The idea is for a superclass to delegate to a subclass the responsibility of specifying an overridden method. The approach is to introduce a model method to specify the behavior of an overridden method and to let a subclass to override the model method to provide a subclass-specific behavior specification. A related pattern, "know your class" also allows the specification of class-specific behavior by using the *getClass* method in assertions.

VI. CONCLUSION

We documented formally in JML a significant portion of the java.security package and its extensions. The starting point of our specifications was informal API descriptions obtained from the Javadoc comments of the source code. We quickly found that in many places the descriptions were incomplete or ambiguous. Thus, we resorted to the source code and experimented with it to resolve ambiguity and to determine the precise behavior of the API. Given our specification, we expect other Java programmers to be able to avoid this "code experimentation". We also expect that our specifications will facilitate the use of the Java security API and increase the reliability of the code on which it is based.

We are in the process of verifying both our specifications and the source code using various JML tools. All our specifications will be soon available from our project website at opuntia.cs.utep.edu/milaap.

REFERENCES

[1] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212–232, June 2005.

[2] N. Catano and M. Huisman. Formal specification of Gemplus's electronic purse case study. In L. H. Eriksson and P. A. Lindsay, editors, *FME 2002*, volume LNCS 2391, pages 272–289. Springer-Verlag, 2002.

[3] Y. Cheon, G. T. Leavens, M. Sitaraman, and S. Edwards. Model variables: Cleanly supporting abstraction in design by contract. *Software—Practice and Experience*, 35(6):583–599, May 2005.

[4] Y. Cheon and A. Perumendla. Specifying and checking method call sequences of Java programs. *Software Quality Journal*, 2006. To appear.

[5] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999.

[6] K. R. M. Leino, A. Poetzsch-Heffter, and Y. Zhou. Using data groups to specify and check side effects. In *ACM PLDI 2002,* volume 37(5) of ACM SIGPLAN Notices, pages 246–257, June 2002.

[7] S. Oaks. *Java Security*. O'Reilly, second edition, 2001.

[8] E. Poll, J. van den Berg, and B. Jacobs. Formal specification of the Java Card API in JML: the APDU class. *Computer Networks*, 36(4):407–421, 2001.

[9] Sun Microsystems, Inc. *Java 2 platform API specification*. Available online from http://java.sun.com (Date retrieved: April 2, 2006).