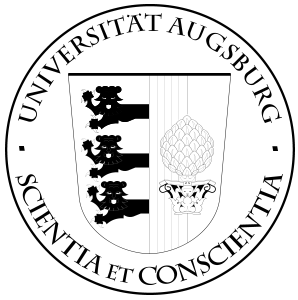
Universität Augsburg



Reasoning about Pointer Structures in Java

Kurt Stenzel, Holger Grandy, Wolfgang Reif

Report 30 2006

Institut für Informatik
D-86135 Augsburg

Reasoning about Pointer Structures in Java

Kurt Stenzel, Holger Grandy, Wolfgang Reif

Lehrstuhl für Softwaretechnik und Programmiersprachen
Institut für Informatik, Universität Augsburg
86135 Augsburg Germany

E-Mail: {stenzel, grandy, reif}@informatik.uni-augsburg.de

Abstract. Java programs often use pointer structures for normal computations. A verification system for Java should have good proof support for reasoning about those structures. However, the literature for pointer verification almost always uses specifications and definitions that are tailored to the problem under consideration. We propose a generic specification for Java pointer structures that allows to express many important properties, and is easy to use in proofs. The specification is part of the Java calculus [22] in the KIV [15] prover.

1 Introduction

The formal verification of pointer algorithms has received quite a lot of treatment in the literature. Often, a special algorithm using a special data structure was verified as a case study. Examples are AVL trees, e.g. [21] [9], or BDD algorithms, e.g. [18]. In the last years, the Schorr-Waite algorithm for garbage collection [20] was an often used example, e.g. [5] [1] [17] [10]. These works use verification techniques that are tailored to some degree to the example algorithm: When reasoning about a pointer structure that represents a binary tree it is helpful to define a correspondence to abstract binary trees, that are specified algebraically (or inductively). However, every pointer structure requires a new specification, and it is not possible to reuse theorems and proofs. Furthermore, normally a heap is used for the pointer structures that abstracts from the difficulties of real programming languages.

Pointer structures in Java can contain **null** pointers, objects can be mixed with arrays in one structure, and subclassing (inheritance) must be taken into account. This is done only in dedicated formal systems for Java, e.g. [24] [11] [25] [23] [3]. Additionally, normal Java programs (in contrast to Java Card [7] programs) often have pointer structures – nested objects – without using specialized algorithms or data structures ([12] contains a nice example with cyclic Java vectors).

We propose a generic specification for Java pointer structures that allows the reuse of theorems and proofs, that uses the same basic specification for a number of different properties of pointer structures, and that is easily extendible for specialized data structures. Section 2 introduces an example pointer structure,

and section 3 defines a formal model of the Java heap. Section 4 describes the basis for the genericity, the generation of axioms from the Java classes, section 5 contains the central specifications, and section 6 concludes.

2 An Example: Documents

When reasoning about the security of cryptographic communication protocols on an abstract level, often typed messages are used [6] [19]. A message can be an *encrypted message*, a *nonce*, a *key*, a *secret*, a list of messages, etc. A Java implementation that is as close as possible to an abstract description of a protocol could use the classes listed in figure 1 (all methods are omitted). The idea is that the application logic (e.g. an e-commerce application) works with the *Document* class while the real data transmission uses some encoding of these documents as a string or a sequence of bytes.

```
abstract class Document { }
class EncDoc extends Document { private byte[] encrypted; }
class DataDoc extends Document { private byte[] value; }
class KeyDoc extends Document { private Key key; }
class NonceDoc extends Document { private Nonce nonce; }
class SecretDoc extends Document { private byte[] secret; }
class Doclist extends Document { private Document[] docs; }
class Nonce { private byte[] nonce; }

abstract class Key { byte[] keyval; }
class SessionKey extends Key { }
class PrivateKey extends Key { }
class PublicKey extends Key { }
```

Fig. 1. Java classes implementing abstract documents (messages). Note that `Doclist` contains a field `docs` of type `Document[]`.

The class `Document` itself is abstract. An `EncDoc` represents an encrypted document, the private field `encrypted` contains the encrypted data (cryptography in Java works on byte arrays). A `DataDoc` contains arbitrary, non-security related data represented with a byte array (a byte array is used instead of a string to make encryption easy). A `KeyDoc` contains a `Key` which can be either a `SessionKey`, a `PrivateKey`, or a `PublicKey`. The itself is represented as one byte array (again assuming a suitable encoding for keys consisting of several parts like RSA keys). A `NonceDoc` contains a `Nonce`, a `SecretDoc` (used to represent passwords or PINs) its secret as a byte array. A list of documents is implemented in `Doclist` with an array of documents. This means that documents can form rather complex pointer structures. Figure 2 shows an example. Elliptical nodes are classes, rectangles are arrays. This pointer structure can be constructed with the following Java code (assuming the appropriate constructors are defined):

```
new Doclist(new Document[]{
  new DataDoc(new byte[]{1,2,3}),
  new KeyDoc(new SessionKey(new byte[]{4,5,6})),
  new Doclist(new Document[]{ null,
                              new EncDoc(new byte[]{7,8,9}),
                              new DataDoc(new byte[]{1,2,3})})}
```

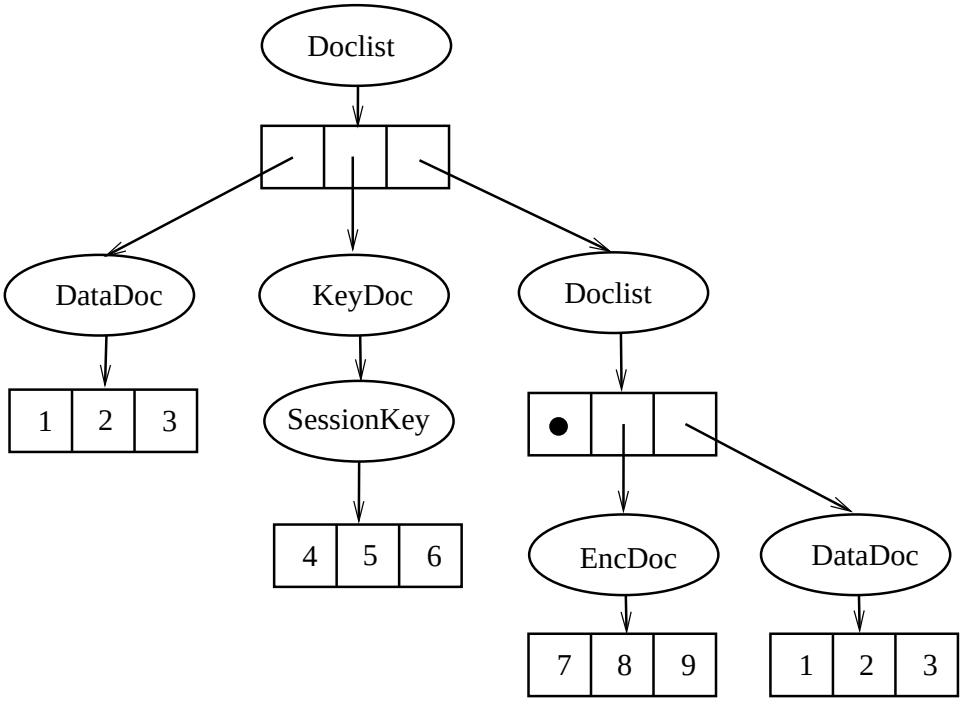


Fig. 2. A document pointer structure. Ellipses show objects, rectangles are arrays. Byte arrays contain numbers, document arrays arrows. The black circle represents **null**.

Figure 3 shows another possible document pointer structure. Note that this structure has null pointers, is cyclic, and contains sharing. A Java programmer must be aware what assumptions can be made about the structure, and what properties must be guaranteed. If the structure can contain null pointers, this must be checked (otherwise a NullPointerException may occur). If the structure can be cyclic, termination of recursive methods becomes an issue. If the structure contains sharing, destructive updates may have undesired effects. From a formal point of view the question is: How can these properties be expressed? But there is more to consider. The pointer structure resides in the heap of the Java virtual machine. Assuming the heap can be corrupted, the (invalid) pointer structure shown in figure 4 could occur.

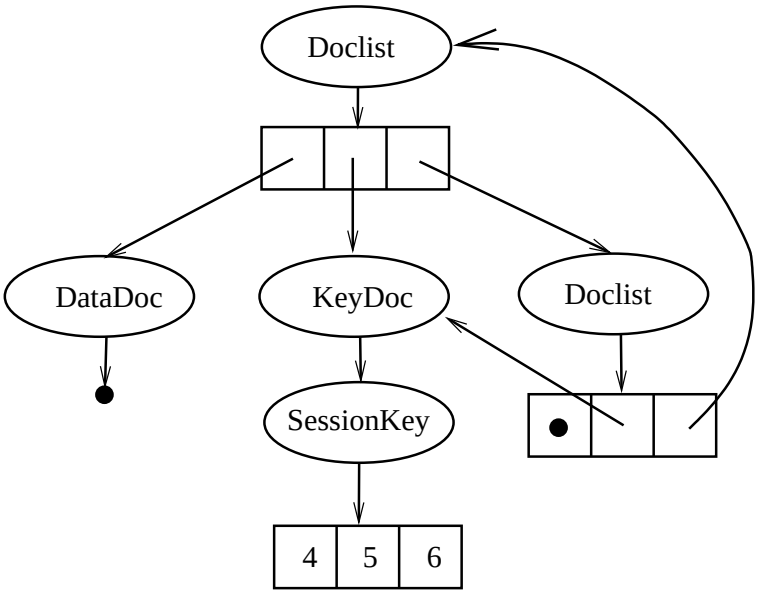


Fig. 3. Another valid document pointer structure. The `value` field of the left DataDoc object is null; the KeyDoc object is shared, and the whole structure is cyclic.

A Java programmer does not have to worry about invalid pointer structures because they cannot occur (assuming the JVM works correctly). However, when doing formal Java program verification, an invalid pointer structure can occur. This is due to the formal representation of the heap, and will be explained in the next section.

3 A formal model of the Java heap

A formal Java semantics must include an explicit model of the Java heap to model pointer structures, creation of new objects, updates etc. (see e.g. [4] [8] [2] [25] [11] [23]), even though several calculi – especially for Java Card – work without one (e.g. [3]). We present an algebraic specification of a heap that is an instantiation of an abstract *store* that uses *keys* to store *values*. For a store st and key k, a value val can be stored or updated, $st[k, val]$. Given a key k, the stored value can be retrieved, $st[k]$. A key (together with its value) can be deleted from the store, $st - k$, and it can be tested if a key is contained in the store, $k \in st$. Stores are finitely generated by an empty store, and the update operation $st[k, val]$. To model the Java heap, the (abstract) keys and values are instantiated. Values are instantiated with Java values, a union type that comprises all Java values: The primitive values boolean, byte, short, int, etc., references (as pointers to objects and arrays), Java types (needed for bookkeeping), and other information (e.g. the current mode of execution, see [14] 14.1). Keys are instan-

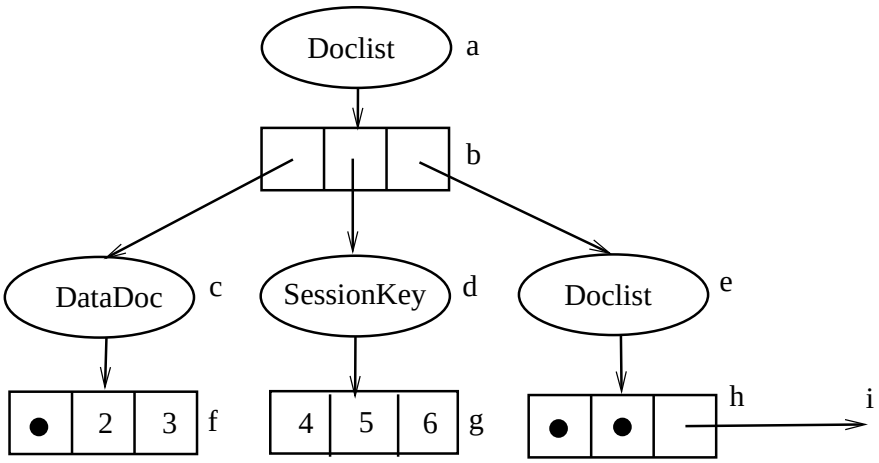


Fig. 4. An invalid (corrupted) document pointer structure: A byte array contains a **null** pointer, the Document array points not to a document, but to a SessionKey, and another entry points to nothing.

tiated a little more complicated. A key is a pair $r \times sk$, consisting of a *reference* and a *storekey*, which in turn is either a *field specification* or an *index*. A field specification is a triple consisting of the class where the Java field is declared, its type, and its name. An index is an integer. This is a very simple model, because the values are simple. Java Objects are not stored under one key, rather the set of all keys with the same reference represents an object; similarly for arrays. Figure 5 shows an example.

reference	**storekey**	**Java value**
a	docs	refval(b)
a	type	Doclist
b	0	null
b	1	refval(a)
b	2	refval(d)
b	length	intval(3)
b	type	Document[]

Fig. 5. Model of the Java heap. Every object has an additional *type* field, and every array has a *type* and a *length* field. This part of the store can be created with `Doclist x = new Doclist(new Document[3]); x.docs[1] = x; x.docs[2] = y;` (`y` is a reference with value d).

It is now possible to specify auxiliary operations on the store, for example to add an object to the store: addobj(a, Doclist, docs $\times$ refval(b), st) creates the

Doclist object in figure 5 (*docs* is the field, refval(b) its value), and addarray(b, Document[], **null** + refval(a) + refval(d), st) creates the Document array of length 3 with values **null**, refval(a), and refval(b) (+ is list concatenation). These operations are used by the proof rules of the calculus. Other useful operations deal with references. E.g. $a \in st$ is true if the reference a occurs as part of a key in the store, $a \in st \leftrightarrow \exists\ sk.a \times sk \in st$. Another useful predicate is newref(a, st) that is true if a is a new reference for the store st, newref(a, st) $\leftrightarrow$ a $\neq$ **null** $\wedge$ $\neg$ a $\in$ st. Note that a reference is considered new if it does not occur as part of a key in the store; nothing is said about references as values.

A type correct Java program will only create 'valid' pointer structures in the store (this notion will be specified in section 5). However, the store model as presented above also allows 'invalid' entries or pointer structures. For example, an integer field could contain a boolean value, or a field could point to an object of the wrong type, or to a reference that does not occur as a key in the store at all. When proving a property about a method on documents, we do not want to deal with those 'illegal' pointer structures, but we want to assume a 'valid' pointer structure. The next section explains what must be done to achieve a generic specification of a 'valid' pointer structure.

4 Generating Axioms from Java Classes

The idea to achieve a completely generic specification of a 'valid' pointer structure is to generate axioms from the Java classes that are tailored for the specification. Of course, it is possible to include the Java classes as a parameter in the predicates and functions. However, this will lead to serious performance problems for programs that are not just toy examples. First, the classes will appear several times in a proof goal, thereby considerably increasing the size of the goal. Second, more or less complex tests and computations will be performed again and again for the classes (e.g. to determine the subclass relation).

A better solution is to generate the necessary axioms, and omit the classes from the proof goal. The following types of axioms must be generated:

1. Which classes exist (i.e. are declared) in the current context:
   class$\exists$ (name) $\leftrightarrow$ name = Doclist $\vee$ name = DataDoc $\vee$ ...
   (name is a variable for a class or interface name).
2. Which interfaces exist (i.e. are declared) in the current context:
   interface$\exists$ (name) $\leftrightarrow$ false
3. For every class or interface the class hierarchy:
   Doclist $\leq$ name
   $\leftrightarrow$ name = Doclist $\vee$ name = Document $\vee$ name = Object
   $name_1 \leq name_2$ is true if $name_1$ is a subclass of $name_2$ (or a subinterface, or a class that implements the interface $name_1$).
4. For every class the list of instance fields (including fields inherited from super classes):
   instfields(Doclist) = [docs]; instfields(DataDoc) = [value]; etc.

(Here, *docs*, *value*, and *keyval* are field specifications as described above, i.e. they contain the name of the class where the field is declared, its type, and the name of the field.)

5. For every class the list of static fields:
   statfields(Doclist) = []; ([] is the empty list).

It turns out that these simple axioms are enough. Those axioms can be used to define some generic auxiliary predicates:

- A Java type is subtype of (or equal to) another type, $ty_1 \leq ty_2$. Primitive types must be equal; if both types are class types the class or interface names must be $\leq$; for arrays the element types must be $\leq$ (and every array type is $\leq$ java.lang.Object).
- *is_object* is true for a reference if it points to an object in the store that has all instance fields as keys, and all fields have a value that 'fits' its type. (A `boolean` field must have a boolean value, a field with a class type must have a reference or **null** as its value, etc.) The definition is shallow in the sense that a field with a reference type is only required to contain a reference as its value; this reference is not further checked.
- *is_array* is similar to *is_object*, but does similar tests for the array indices and their values.
- is_valid_ref(r, st) $\leftrightarrow$ is_object(r, st) $\vee$ is_array(r, st)

We can now specify a 'valid' and an 'illegal' pointer structure.

5 Junkrefs and Validrefs

A reference r is a *junkref* in the store *st*, junkref(r, st), if there exists a path in the pointer structure beginning with r to a reference that contains something invalid. A path is simply a list of references *refs*. So we have the simple definition

$$\text{junkref}(\text{r}, \text{st}) \leftrightarrow \exists\ \text{refs.refs} \neq [] \wedge \text{refs.first} = \text{r} \wedge \text{junkrefs}(\text{refs}, \text{st}[\text{r} \times \text{type}], \text{st})$$

The list of references *refs* must begin with r (refs.first = r). Then the specification of *junkrefs* contains all checks. *junkrefs* has an additional argument, the expected type of the first reference in *refs*. To be more precise: The first reference should be a subtype the given type. $st[r \times type]$ looks up the type of r in the store. *junkrefs* is specified with five axioms:

1. The empty list of references is not *junkrefs*:
   junkrefs-empty: $\neg$ junkrefs([], ty, st)
2. The **null** reference is not *junkrefs*:
   junkrefs-null: $\neg$ junkrefs(null + refs, ty, st)
   (**null** is a special reference, + (which is overloaded) adds a reference to a list of references.)

3. If the first reference r is not **null**, and either does not point to a valid object or array in the store, or this object/array has a non-existing type, or its type is not a subtype of the given type, then *junkrefs* is true:
   junkrefs-is-junk:
   $$\begin{array}{l} \quad \mathrm{r} \neq \mathrm{null} \\ \wedge\ (\quad \neg\ \mathrm{is\_valid\_ref(r,\ st)} \\ \quad\ \vee \neg\ \mathrm{type}\exists(\mathrm{st}[\mathrm{r} \times \mathrm{type}]) \\ \quad\ \vee \neg\ \mathrm{st}[\mathrm{r} \times \mathrm{type}] \leq \mathrm{ty} \\ \quad\ ) \\ \rightarrow \mathrm{junkref(r + refs,\ ty,\ st)} \end{array}$$
4. If the first reference r points to a valid object then *junkrefs* is true iff this object has a field with a reference that is equal to the remaining list of references, and those references are *junkrefs*:
   junkrefs-class:
   $$\begin{array}{l} \quad \mathrm{is\_obj(r,\ st)} \wedge \mathrm{st}[\mathrm{r} \times \mathrm{type}] \leq \mathrm{ty} \wedge \mathrm{type}\exists(\mathrm{st}[\mathrm{r} \times \mathrm{type}]) \\ \rightarrow (\quad \mathrm{junkrefs(r + refs,\ ty,\ st)} \\ \quad\ \leftrightarrow \quad \mathrm{refs} \neq [] \\ \quad\ \wedge (\exists\ \mathrm{fs}. \quad \mathrm{fs} \in \mathrm{instfields}(\mathrm{st}[\mathrm{r} \times \mathrm{type}].\mathrm{class}) \\ \qquad\qquad \wedge\ \mathrm{is\_referencevalue}(\mathrm{st}[\mathrm{r} \times \mathrm{fs}]) \\ \qquad\qquad \wedge\ \mathrm{refs.first} = \mathrm{st}[\mathrm{r} \times \mathrm{fs}] \\ \qquad\qquad \wedge\ \mathrm{junkrefs(refs,\ fs,\ st)})) \end{array}$$
5. Similarly for arrays: If the first reference points to a valid array then *junkrefs* is true iff the array contains an index with a reference that is equal to the remaining list of references, and those references are *junkrefs*:
   junkrefs-array:
   $$\begin{array}{l} \quad \mathrm{is\_array(r,\ st)} \wedge \mathrm{st}[\mathrm{r} \times \mathrm{type}] \leq \mathrm{ty} \wedge \mathrm{type}\exists(\mathrm{st}[\mathrm{r} \times \mathrm{type}]) \\ \rightarrow (\quad \mathrm{junkrefs(r + refs,\ ty,\ st)} \\ \quad\ \leftrightarrow \quad \mathrm{refs} \neq [] \\ \quad\ \wedge (\exists\ \mathrm{i}. \quad 0 \leq \mathrm{i} \\ \qquad\qquad \wedge\ \mathrm{i} < \mathrm{st}[\mathrm{r} \times \mathrm{length}] \\ \qquad\qquad \wedge\ \mathrm{is\_referencevalue}(\mathrm{st}[\mathrm{r} \times \mathrm{i}]) \\ \qquad\qquad \wedge\ \mathrm{refs.first} = \mathrm{st}[\mathrm{r} \times \mathrm{i}] \\ \qquad\qquad \wedge\ \mathrm{junkrefs(refs,\ st}[\mathrm{r} \times \mathrm{type}],\ \mathrm{st}))) \end{array}$$

This finishes the axiomatization. Consider figure 4. The letters a ...i denote references, i.e. the top-level Doclist object has reference a. Then junkref(a, st) is true because junkrefs([a, b, e, h, i], Doclist, st) is true: From the Doclist object a the Document array b can be reached, from b a pointer to e exists, from there to h, and finally to i. junkrefs([i], ty, st) is true because the reference i does not point into the store, hence is_valid_ref(i, st) is false. junkrefs([a, b, c, f], Doclist, st) and junkrefs([a, b, d], Doclist, st) are also true, junkrefs([a, b, e, h], Doclist, st) is false since nothing 'invalid' occurs.

Experience shows that this specification is very useful, and easy to handle (in spite of the existential quantifiers). When proving a property about a method we can now assume that all inputs (that are references) and the invoking reference are not junkrefs. When the method accesses the pointer structure, no unexpected or undefined results can occur.

Intuitively, if junkrefs(refs, ty, st) is true there is a first reference r in *refs* that is responsible for this, i.e. that causes junk. The preceding references form (in a sense) a valid path through the pointer structure. Using the same specification technique it is possible to specify that a list of references are all a valid path in a pointer structure. This is done by the predicate validrefs(refs, ty, st). The specification consists of five axioms that are inverse to the axioms of *junkrefs*. Therefore we only list the first three of them:

1. The empty list is validrefs: validrefs([], ty, st)
2. If the reference is **null** the list of remaining references must be empty:
   validrefs(**null** + refs, ty, st) $\leftrightarrow$ refs = []
   This is slightly different from the second junkrefs axiom. It implies that **null** can occur only as the last element of a valid path (obviously the path cannot be extended because **null** does not point anywhere).
3. If the first reference does not point to a valid object or array validrefs is false. This axiom is the exact inverse to the third junkrefs axiom.

$$\begin{array}{l} \quad \text{r} \neq \textbf{null} \\ \wedge \ (\quad \neg \ \text{is\_valid\_ref(r, st)} \\ \quad\ \vee \neg \ \text{st[r} \times \text{type]} \leq \text{ty} \\ \quad\ \vee \neg \ \text{type}\exists(\text{st[r} \times \text{type]})) \\ \rightarrow \neg \ \text{validrefs(r + refs, ty, st)} \end{array}$$

The remaining two axioms are inverse to their junkrefs counterpart. There is a correspondence between junkrefs and validrefs (references that are junkrefs begin with references that are validrefs), but it is not useful for proving.

With validrefs some interesting properties of a pointer structure can be expressed. The first is also predefined in JML, the Java Modeling Language [16] [13], as a semantical concept (\reach). Here, we give a precise formal specification for it.

1. A path exists from one reference a to another b if there exists a valid list of references starting with a and ending with b:
   path-exists-def :

$$\begin{array}{l} \quad \text{path}\exists\text{(a, b, st)} \\ \leftrightarrow \exists \ \text{refs.} \quad \text{refs} \neq [] \\ \qquad\qquad \wedge \ \text{refs.first} = \text{a} \\ \qquad\qquad \wedge \ \text{refs.last} = \text{b} \\ \qquad\qquad \wedge \ \text{validrefs(refs, java.lang.Object, st)} \end{array}$$

   For example, the property that an update to the field f of an object o (written as $st[o \times f, val]$, the store is modified at key $o \times f$ to val) does not modify a pointer structure beginning with a, can be expressed as:

$$\begin{array}{l} \forall \ \text{b, sk.} \quad \text{path}\exists\text{(a, b, st)} \wedge \text{b} \neq \textbf{null} \\ \qquad \rightarrow \text{st[b} \times \text{sk]} = \text{st[o} \times \text{f, val][b} \times \text{sk]} \end{array}$$

2. The pointer structure starting with a contains a cycle, cyclic(a, st), iff there exists a list of references beginning with a that are validrefs and contain duplicates.
   cyclic-def :

cyclic(a, st)
$\leftrightarrow \exists$ refs. dups(refs)
$\wedge$ refs.first = a
$\wedge$ validrefs(refs, java.lang.Object, st)

dups(refs) is true if refs contains duplicates. This is a simple operation on lists. Of course, cyclic can be defined using path$\exists$, but the specification above is simpler, and more useful in proofs.

3. The pointer structure starting with a has sharing iff there exist two different valid lists of references $refs_1$ and $refs_2$ starting with r and ending with the same reference $b \neq$ **null**.
   has_sharing-def :
   has_sharing(a, st)
   $\leftrightarrow \exists$ $refs_1$, $refs_2$.
   $refs_1 \neq []$
   $\wedge$ $refs_2 \neq []$
   $\wedge$ $refs_1 \neq refs_2$
   $\wedge$ $refs_1$.first = a
   $\wedge$ $refs_2$.first = a
   $\wedge$ $refs_1$.last = $refs_2$.last
   $\wedge$ $refs_2$.last $\neq$ **null**
   $\wedge$ validrefs($refs_1$, st[a $\times$ type], st)
   $\wedge$ validrefs($refs_2$, st[a $\times$ type], st)
   Whether a pointer structure has sharing or not is important if destructive updates to its objects are used.
   A similar specification can be used to express whether two references contain common parts (i.e. share objects or arrays), or whether they are disjoint. This can be used to prove that updates to one structure do not modify the other.
4. Induction: It is simple to specify the longest, valid, acyclical path in a pointer structure using validrefs. Then induction on this length can be used to prove the termination of recursive algorithms working on a pointer structure.

These definitions are very useful in many applications, and many generic properties can be proven for them. Furthermore, they are a base for application specific definitions.

For example, the document structure in section 2 uses arrays of Documents to represent lists of documents. Therefore, the class `Doclist` has a field `docs` of type `Document[]`. An algorithm working on documents may assume that this field is never **null**, and that its value is indeed an array of type `Document`. (This is not guaranteed. The field can hold an array of type `KeyDoc`, `KeyDoc[]`. In this case, trying to put a `DataDoc` object into the array would raise an `ArrayStoreException`). We can express these two assumptions as:

> Given a reference a to a Document structure. Then the structure fulfills the two assumptions iff for every reference b that is reachable from a (i.e. a path exists from a to b) and has type `Doclist` holds: The `docs` field is not **null** and contains an array of type `Document`:

$$
\begin{array}{ll}
& \text{assumptions\_fulfilled(a, st)} \\
\leftrightarrow \forall\ \text{b.} & \quad \text{path}\exists\ (\text{a, b, st}) \wedge \text{b} \neq \textbf{null} \wedge \text{st}[\text{b} \times \text{type}] = \text{Doclist} \\
\rightarrow & \quad \text{st}[\text{r} \times \text{docs}] \neq \textbf{null} \\
& \quad \wedge\ \text{st}[\text{st}[\text{r} \times \text{docs}] \times \text{type}] = \text{Document}[]
\end{array}
$$

Experience shows that this type of assumptions are needed in many applications. The distinction between generic and application specific properties make their correct specification much easier.

6 Conclusion

We have presented a specification that is very useful to reason about arbitrary pointer structures in Java. As an example, a complex Document structure was presented that can be used to implement cryptographic communication protocols. This structure contains a mixture of different objects, and arrays of those objects.

The specification is generic in the sense that it does not depend on concrete Java classes. This allows the specification to be re-used in many different applications, and many properties can be proved that hold for all applications (the specification is a very useful library specification). By generating some simple axioms for the concrete Java classes the specification is tailored to a given application.

Useful generic properties are: The pointer structure is wellformed (valid, in the sense that it can be generated by a type correct Java program), it is (not) cyclic, it (does not) contain sharing, two pointer structures are disjoint, one reference is reachable from another, an induction principle, etc. Based on these generic properties it is very easy to define application specific properties (e.g. a given field of a given object is never **null**). Experience shows that this noticeably increases the productivity when reasoning about Java programs.

References

1. Jean-Raymond Abrial. Event based sequential program development: Application to constructing a pointer program. In *FME 2003: International Symposium of Formal Methods Europe, Proceedings*. Springer LNCS 2805, 2003.
2. J. Alves-Foss and F. Lam. Dynamic Denotational Semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*. Springer LNCS 1523, 1999.
3. B. Beckert. A dynamic logic for the formal verification of java card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards*. Springer LNCS 2041, 2000.
4. E. Börger and W. Schulte. A Programmer Friendly Modular Definition of the Semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*. Springer LNCS 1523, 1999.
5. Richard Bornat. Proving pointer programs in hoare logic. In *MPC 2000: Mathematics of Program Construction, Proceedings*. Springer LNCS 1837, 2000.
6. Michael Burrows, Martín Abadi, and Roger M. Needham. A Logic of Authentication. Technical report, SRC Research Report 39, 1989.

7. Zhiqun Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. Addison-Wesley, 2000.
8. S. Drossopoulou and S. Eisenbach. Describing the Semantics of Java and Proving Type Soundness. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*. Springer LNCS 1523, 1999.
9. R. Hettler, D. Nazareth, F. Regensburger, and O. Slotosch. Avl trees revisited: A case study in spectrum. In *KORSO: Methods, Languages, and Tools for the Construction of Correct Software*. Springer LNCS 1009, 1995.
10. Thierry Hubert and Claude Marché. A case study of c source code verification: the schorr-waite algorithm. In *SEFM 2005: Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods*. IEEE Computer Society, Washington, DC, USA, 2005.
11. M. Huisman. *Reasoning about JAVA programs in higher order logic with PVS and Isabelle*. PhD thesis, University of Nijmegen, IPA dissertation series, 2001-03, 2001.
12. M. Huisman. Verification of java's abstractcollection class: a case study. In *MPC'02: Mathematics of Program Construction*. Springer LNCS 2386, 2002.
13. JML home page. `http://www.jmlspecs.org/`.
14. Bill Joy, Guy Steele, James Gosling, and Gilad Bracha. *The Java (tm) Language Specification, Second Edition*. Addison-Wesley, 2000.
15. KIV homepage. `http://www.informatik.uni-augsburg.de/swt/kiv`.
16. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of jml: A behavioral interface specification language for java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, March 2006.
17. Farhad Mehta and Tobias Nipkow. Proving pointer programs in higher-order logic. In *CADE-19: 19th International Conference on Automated Deduction, Proceedings*. Springer LNCS 2741, 2003.
18. Veronika Ortner and Norbert Schirmer. Verification of bdd normalization. In *TPHOLs 2005: 18th International Conference on Theorem Proving in Higher Order Logics*. Springer LNCS 3603, 2005.
19. L. C. Paulson. The Inductive Approach to Verifying Cryptographic Protocols. *J. Computer Security*, 6:85–128, 1998.
20. H. Schorr and W. M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *Communications of the ACM*, 10(8), August 1967.
21. Robert F. Stärk. Formal verification of logic programs: foundations and implementation. In *LFCS'97: 4th International Symposium on Logical Foundations of Computer Science*. Springer LNCS 1234, 1997.
22. Kurt Stenzel. A formally verified calculus for full Java Card. In C. Rattray, S. Maharaj, and C. Shankland, editors, *Algebraic Methodology and Software Technology (AMAST) 2004, Proceedings*, Stirling Scotland, July 2004. Springer LNCS 3116.
23. Kurt Stenzel. *Verification of Java Card Programs*. PhD thesis, Universität Augsburg, Fakultät für Angewandte Informatik, URL: `http://www.opus-bayern.de/uni-augsburg/volltexte/2005/122/`, or `http://www.informatik.uni-augsburg.de/forschung/dissertations/`, 2005.
24. J. van den Berg and B. Jacobs. The loop compiler for java and jml. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Software (TACAS)*. Springer LNCS 2031, 2001.
25. D. von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München, 2001.