Generation of JML Specification for Java Card applications [*]

September 19, 2003

Contents

[*]acknowledgments to Marieke Huisman for the excellent ideas who contributed a lot to this work, to Lilian Burdy, Gilles Barthe, Jean-Luois Lanet for the fine advises

1 Introduction

A Java Card applet is an application that is intended to be loaded on a smart card. High quality of applets and their development become a crucial issue as they manage sensitive personal information. This means that smart card applications should respect certain requirements in order to guarantee that this data cannot be corrupted, or accessed by unauthorized persons. In order to verify if a concrete Java Card application respects these requirements, a technique is needed - dynamic or static analysis, model checking or formal method approach.

Program verification is particularly relevant to guarantee such kind of high level properties, existing methods as for example dynamic tests(whenever tests are technically possible) being costly or imprecise. We consider that using formal techniques is a solution that allows to increase the quality, but also to reduce validation costs. The subject of the present work deals with the validation of Java Card applet properties using formal methods - generation of proof obligations for a program and then proving them. There are several phases in the formal validation that we are doing - specification of the source code with a specification language, generation of proof obligations, proof of the proof obligations.

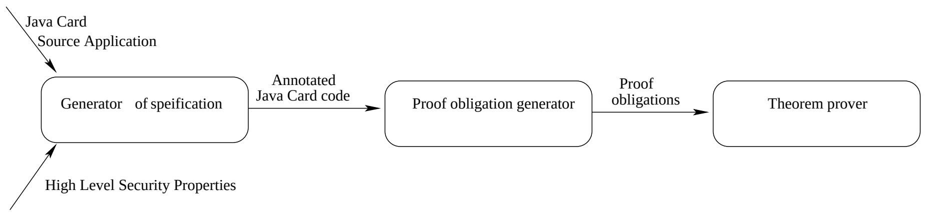
Our study is focused on the specification process. Program annotation being a hard and long task, is one of the unpleasant parts of program verification. The present work proposes a technique for automatically generating annotation for a program. Given certain high level properties for a Java Card applet we translate them into a specification language. Given an application a specification for it is generated starting from this *core translation.* To generate the specification we propagate the *core* translation of the security properties (see section 4) in question over the source code; for the preconditions of a method it is a forward calculation(up from the beginning of the body statement down to its end), the postconditions are calculated in the other direction (from the end of the method body statement up to the beginning of the statement). The language JML - abreviation for Java Modeling Language [7], is the specification language we use and the Java Applet Correctness Kit (JACK), developed at Gemplus[2] to discharge proof obligations.

Section 2 presents the specification language JML, the simple language over which we will reason, section 3 is an overview of Java Card applications, section 4 talks about the properties that we study, section 5 talks about the weakest precondition calculus(*wp*) and an abstract version of it(*wp'*) (which we define because the specification that we generate uses only specification variables and which is proven correct), section 6 focuses on the technique we are using for generating specification and establishes what is the relation with the precondition that we generate and the *wp'*, and we conclude in section 7.

1.1 Architecture

The basic result of the present work is an implementation that generates an annotation in terms of preconditions and postconditions for an input Java program. The tool works on the level of the call graph of a program.
As you can see on the figure below, (the first rectangle on the left represents our

tool) the input is a program - a set of Java files, and the output is a set of Java files annotated with JML annotations. The result then can be submitted to a proof obligation generator and the resulting formulas can be fed into a theorem prover.
So far we deal with annotations that contain only *static ghost variables* - special specification variables (see section 2) which are added to the source code as comments. They do not modify the semantics of the program as they are not seen by the Java compiler.


Java Card
Source Application
Generator of speification
Annotated
Java Card code
Proof obligation generator
Proof
obligations
Theorem prover
High Level Security Properties

The fact that the variables that are permitted in the specification are *static* means that the properties are class properties rather than instance one. For the kind of properties that we study *static* variables are sufficient. Of course it is one of the basic directions for our future work - treating Java instances. We have tested the tool for the transaction properties (see section 4) over two applets - Epurse(Gemplus) and Demoney.

1.2 Related Work

There are several aspects in which one can find similar works: the first one concerns static analysis of Java Card, annotation assistants, automatic Java code generation.

Automatic Generation of Provably Correct Code

Finite State Machines To Java Programs Hubbers,etc. at the University of Nijmegen have worked on the generation a Java Card prototype source code with formal specification in JML starting from a finite state machine description. They generate in a form of a class invariant and historical constraints specification with an application skeleton. The specification that we are able to generate must not obligatory be modeled in a finite transition system. The specification that they generate concern the life cycle of a Java Card application (see [5])

Model Checking Thomas Jensen has theoretical works on verification of high level security properties by using model checking. In [9] is described an analysis made over all possible traces of stacks of execution of a program and temporal logic is used to reason about program correctness. The control flow is abstracting away everything but method calls. This means that data flow analysis is not made and properties concerning information flow are not checked. This technique relies upon the correctness of the graph modelling the program execution.

JML Annotation Assistants

Houdini tool is an automatic generator of code made at the research laboratory of Compaq (see [4]). The specification that is generated specifies local properties - null dereference of objects, arrays, array access out of the array bounds differently from our task which is to specify global properties (as you'll see in section 4). Houdini tries to guess the annotation by generating a finite set of candidate annotations from the source text using heuristics - for example if a derefence of an object is detected then add as precondition that this field is not null while in our approach the user specifies as first phase certain methods of the classes that are the input and then this *initial* specification is propagated. Houdini runs the esc\java as internal routine (see [8]) iteratively over the annotation until a fix point is reached, that is considered to be the searched annotation.

Daikon is a tool made at the MIT laboratory by Jeremy W. Nimmer and Michael D. Ernst. Daikon detects automatically program invariants by using dynamic and static analysis (see [13]). Daikon discovers likely invariants from program executions by instrumenting the target program to trace the variables of interest, running the instrumented program over a test suite, and inferring invariants over the instrumented values. Daikon produces invariants which are approximated based on the result of the dynamic testing and by applying statistical methods, so they are not always sound(for example it can discharge a formula which is an invariant for the concrete test case but not valid for all possible program executions).

2 Java Modeling Language

JML [7] is a specification language that is designed to specification of Java programs by formally expressing preconditions, postconditions and invariants for a program. Some keywords and logical constructions of JML are new to Java, but the core expression language is close to Java. JML has been defined so that specifications are easy to read and write by Java programmers. Tools have been developed around JML annotation as for example: the LOOP tool for generating proof obligations from a JML specification over a Java program, esc\java see [8], Jack [2] - proof obligation generator as the LOOP tool provided also with a friendly user interface, the jml runtime checker that uses JML notation in order to add runtime assertions in the generated code.

2.1 JML Specification

Figure 1 presents an example of a Java code fragment specified with JML. The basic key words for specifying a Java class are :

1. *Invariants* - these are predicates that must hold before and after an execution of a method.
2. *Preconditions* - preconditions for methods are declared with the keyword `requires R(x)`. The predicate `R(x)` must hold for those programs states in which a call to this method with this *requires* clause is made.

3. *Normal Postconditions* - normal postconditions for methods are declared with the keyword `ensures Q(x)`. The predicate `Q(x)` must hold after the normal trmination of the method.
4. *Exceptional Postconditions* - exceptional postconditions for methods are declared with the keyword `exsures(E )P(x)`. The predicate `P` must hold if the execution of the method terminates by throwing an exception `E`.
5. *The Modifies clause* - JML also specifies which variables visible for the outside can be modified by the method. The `modifies` clause is then used.
6. *Model Variables* - One also can declare variables that are significant only to the specification. They are introduced with JML keyword `ghost` or `model`. The good thing for these variables are that they are ¡¡visible¿¿ only for tools that understand JML so the Java compiler ignores them. When generating our specification we also add model variables to the application which does not change the program behavior as the previous fact holds.

```
class C
    int i ;
    int s ;
    //@invariant s != 0 ;


    //@requires i != 1;
   //@modifies i;
   //@ensures i == 3 ;

   public void m() {

       s = 1/(i-1);
       i = 3;
    }
}
```

Figure 1: class specified with JML

The JML annotation are written as comments in the Java source code, so that Java compilers do not notice them, but JML tools can use them. The language contains more complex constructions that allow to model more complex behaviors. It has been used for example for specifying the Java Card API and part of the standard edition Java API, also Java Card applications have been specified with this language. Our work is focused here on these three JML clauses. The specification that we generate involves the mentioned above specification clauses of JML.

2.2 Formal Specification for modular programs

Our work is in strong relation with modular verification of programs. `Java` is an object oriented language- there are two types of types - simple types and

classes. Every class has a set of methods and fields. Specifying a Java program means to specify the classes of this program. Specifying a class means to specify its *invariant* - a predicate which should hold in every programs state after and before every execution of any method of this class, and to specify its methods - the predicates that must hold before calling a method and the predicates that must hold after the execution of a method, i.e. the method's *preconditions* and *postconditions* In the coming section 2 we will see particularely how these predicates are expressed in the specification language `JML` .
Let us have some class with a method named *m* and two predicates *P(x)* and *Q(y)* as its precondition and postcondition, and let the method *callm* does a call to the method *m* :

```
//@requires P(x);
//@modifies x;
//@ensures Q(y);
public void m() {
   ...
}

public void callm() {
   ...
   m();
}
```

When verifying a *method declaration* , the precondition is assumed and on this assumption the postcondition must be proven - i.e. assume that *P(x)* is true and prove that *Q(y)* holds after the execution of the method. When verifying a *method call* then the precondition *P(x)* of the called method - must hold just before the method call -i.e. must be proven to hold and the postcondition *Q(y)* of the called method is assumed to hold.
Then if a method call is done in a method implementation the caller should guarantee that the precondition of the called method holds. Then we can either *assume* its precondition to hold, or prove it. Informally, this means that either the precondition of the called method *callm* on the figure above is part of the precondition of the method *m*, or it must be established by the statements that are before the call of the method *m*.

Consider this code fragment where `JCSystem.beginTransaction` is called where the method that does the call has the default specification. Let suppose we have the property *no nested transactions* on 4.1.

```
   //@requires true;
   //@ensures true;
   public void m() {
        JCSystem.beginTransaction();
    }
```

For this example two observations can be made:

- Method `m` doesnot assume the precondition of

`JCSystem.beginTransaction()`(see 4.1), neither establishes it. The generated proof obligation of such an annotation is equivalent to : `TRANS == 0` which is not provable even if there is no transaction opened. This is not very satisfactory as we want that if an implementation respects the property "No Nested Transactions" the proof obligations should be provable.

- One can conclude from this example also that method `m` should "inform" all its possible callers that it opens a transaction and in this way, if a method calls it, the caller should guarantee that no other transactions are opened.

- What is the solution ?
  If method `m` requires as precondition that no other transactions are made when it is called, i.e. its precondition should be
  `TRANS == 0`. Proof obligations now can be established : for the method declaration of `m` they are equivalent to :
  `TRANS == 0` $\Rightarrow$ `TRANS == 0`.Also in this way anyone that calls `m` will be "informed" that there is a transaction opened.

This suggests that the generation of specification for modular programs can be seen as propagation of pre and postconditions of methods: first translate the specification in some method's pre and postconditions and then propagate it. The rest of this report will present this mechanism - formally and with some examples and will argue for the relation between it and some well known techniques as the *weakest precondition calculus.*

2.3 Programming Language

As explained above, we aim at generating appropriate specification for Java Card Programs. As the whole Java Card language is complex, for the theoritical foundations of our work we study a simple imperative language with exceptions and method calls. This language captures all the parts of Java Card that are relevant for our work. We use the defined simple language to reason about the weakest precondition calculus (see 5) and we use it to define the modifiable set of a statement and for the functions for calculating preconditions and postconditions.(see 2.4) However, we would like to emphasize that our approach works for the whole Java Card language.
The domains from where we will go to take our method names and variable names and exception names are defined :
`ProgramVars` - a set of variable names
`GhostVars` - a set of ghost variable names
`MethodNames` - a set of method names
`ExceptionNames` - a set of exception names.
For clarity, we assume that the three sets are disjoint.

We assume that a method can then return a value or it can be `void`.

`EXPR` = literal |
v,v $\in$ `ProgramVars` |
methodCall(m) , m $\in$ `MethodNames` |

x = `EXPR` , x *in* `ProgramVars`

In the definition of a statement that follows we use the notion of `bool_EXPR` which means an EXPR evaluates to a boolean value.
`STMT` =
skip |
try { STMT } *catch* (name_Exception) { STMT } |
throw `EXPR` |
`EXPR` |
if (bool_EXPR) *then* { `STMT` } *else* { `STMT` } |
while (bool_EXPR) `STMT` |
{ `STMT` } |
`STMT` ;`STMT`

Remark: For convenience when defining functions on statements and expressions below, we always define them with a single case disctinction, ignoring the case STMT = EXPR and `EXPR`.

2.4 The *modifies* clause in JML

The technique that we use for the specification generation uses the set of variables that can be modified by a statement: in particular this set of variables for a method declaration are the variables declared in the JML clause modifies of a method specification (see 2). As the generated specification involves only specification variables we define the modifiable set over the domain of ghost variables only.

2.4.1 Definition

If one looks at the JML specification one will find this definition of the modifiable set of a method.

Definition 1 (Modifiable) *The* modifiable clause *is the only set of locations which from the client's point of view, can be assigned to during the execution of the method.*

In our particular case as we are constraint to a subset of JML we don't have dependees, see[7]. Nor we will have in the modifiable set real Java variables. The modifiable set then will be defined taking into account these conditions as follows in the next section.

2.4.2 The Modifiable set of a statement for the case when only ghost static variables appear in the specification

For the general Java case a modifies set can contain either Java variables or JML model variables. As we are abstracting from the concrete case we will constraint the modifiable set of a statement only to ghost variables. We will point out after stating the definition that we use how it differs from the one that takes into account Java variables also.

$$modifies : \texttt{STMT} \rightarrow Set(\texttt{GhostVars}).$$

We overload the function *modifies* to receive as argument also objects `m` from the domain `MethodNames` and to return the set of variables from the JML annotation for `modifies` for the method declared with name `m`. We define *modifies* to return a set of ghost variables as we limit our specification to abstract variables only.

$$modifies : MethodNames \rightarrow\rightarrow Set(\texttt{GhostVars})$$

Intuitively the modifiable set for a statement is the union of the modifiable sets of its sub-statements. The definition of modifiable set for a statement follows the structure of a statement :

$$\begin{aligned}
modifies(skip) &= \varnothing \\
modifies(\texttt{stmt1}; \texttt{stmt2}) &= modifies(\texttt{stmt1}) \cup modifies(\texttt{stmt2}) \\
modifies(try\{\texttt{stmt1}\}catch(name\_Exception)\{\texttt{stmt2}\}) &= modifies(stmt1) \cup modifies(stmt2) \\
modifies(throw\ e) &= modifies(e) \\
modifies(if\ (cond)\ then\ \texttt{stmt1}\ else\ \texttt{stmt2}) &= modifies(\texttt{stmt1}) \cup modifies(\texttt{stmt2}) \cup modifies(\texttt{cond}) \\
modifies(while\ (cond)\ do\ \texttt{stmt1}) &= modifies(\texttt{stmt1}) \cup modifies(\texttt{cond}) \\
modifies(call(m)) &= modifies(m) \\
modifies(\{\ \texttt{s}\ \}) &= modifies(\texttt{s}) \\
modifies(x = \texttt{E}) &= modifies(\texttt{E})
\end{aligned}$$

Remark the difference between this abstract definition and the concrete definition of `modifies`. It is for the case of assignment and in the concrete case, we have:
$modifies(x = \texttt{E}) = x \cup modifies(\texttt{E})$

Example
Let us have this class definition with some method defined with its specification as in the figure:

```
   public class SomeClass {
    //@public static ghost int Z, X;
    ...
    //@requires Z==0;
    //@modifies Z;
    //@ensures true;
    public boolean cond() {
         ...
    }

    ...
}
```

Example
We have for example this statement and we want to know what its `modifiable` set is:

```
    SomeClass sc = new SomeClass();
ClassVerifyPin vf = new ClassVerifyPin();
...
if (sc.cond()) {
    JCSystem.beginTransaction();
} else {
    vf.verifyPin();
}
```

Looking at the specification of `JCSystem.beginTransaction()` at Figure 4.1 we see that :

$$modifies(JCSystem.beginTransaction()) = \{JCSYSTEM.TRANS, ClassVerifyPin.VERIFYPIN\}$$

From the specification of `verifyPin()` at Figure ?? we have :

$$\begin{array}{l} modifies(verifyPin()) = \\ \quad \{ClassVerifyPin.VERIFYPIN\} \end{array}$$

and the specification at Figure 2.4.2 for *(*sc.cond()) is :

$$modifies(sc.cond()) = \{SomeClass.Z\}$$

We apply then the rule (see definition of *modifies*):

$$modifies(if(sc.cond())\{JCSystem.beginTransaction();\}else\{vf.verifyPin();\}) =$$

$$\begin{array}{l} modifies(JCSystem.beginTransaction()) \cup \\ modifies(vf.verifyPin()) \cup \\ modifies(sc.cond) = \\ \\ \{JCSystem.TRANS, ClassVerifyPin.VERIFYPIN\} \cup \\ \{ClassVerifyPin.VERIFYPIN\} \cup \\ \{SomeClass.Z\} = \\ \\ \{JCSystem.TRANS, ClassVerifyPin.VERIFYPIN, SomeClass.Z\} \end{array}$$

end Example

3 Java Card Applications

Java Card is a standard defined by Sun Microsystems tailored to smart cards. It is a subset of the Java language. It is rather limited - it does not support concurrent programming, multidimensional arrays, floating point types, etc. There is a standard Java Card API specified and implemented by Sun Microsystems. It provides a framework of classes and interfaces for building and communicating with and working with Java Card applets. Java Card application has one instance of a class that implements the interface `javacard.framework.Applet` from the standard API [10]. A brief description to certain features of the Java Card Application and its interaction with the Java Card Runtime Environment(JCRE) system is given here after.

1. *Life Cycle*
   The abstract class `javacard.framework.Applet` from the standard Java Card packages has methods that serve that the JCRE control the applet. These methods are the so called entry points. Called by the JCRE the applet passes from one state to another [11], [9] :

   `install` - the JCRE calls this method of the applet when installing it. So after it is called the applet is in `installed` state.
   `personalized`- In [9] this phase is described as the stage where a registered applet instance receives personalization data and initializes." An applet is personalized only once, and only then becomes operational. There is no explicit support in the system concerning personalization. Although it is a standard logical phase in the life of an applet, it has to be implemented explicitly by the programmer. In particular, there is nothing in the system to prevent re-personalization or use of an un-personalized (hence not operational) applet; it is up to the programmer to set up an explicit flag." That is this stage is not documented in the `JCRE` specification, anyway it is of importance for our properties. `register` - once called by the JCRE the applet instance is considered to be `alive` and exists until deleted by the Applet manager on the card.
   `process` that is responsible for processing the external commands by dealing with the so called *application protocol data units* (APDUs)
   `select, deselect` - on a smart card it is possible that several applet instances exist, when `select` of an applet is called it becomes `selected` make the applet the one whose process method will process the received external commands until its `deselect` method is called.

2. *Firewall*
   Several Java Card applications can be installed on a smart card. In order that there is no leaking of sensitive data between applications the JCRE has a firewall system. It assures that a data belonging to an application can be accessed by another applet if the requesting one satisfies certain access rules. The objects that belong to an applet and that can be accessible by other applets must be of class that implements `javacard.framework.Shareable`, see [11].

3. *Transaction system*
   An applet might need to atomically update several different data.Then either all the updates take place correctly, or else all data is restored to their previous values. The Java Card platform supports a transactional model in which an applet can designate the beginning of an atomic set of updates with a call to the `javacard.framework.JCSystem.beginTransaction()` method. Each object / variable after this point is conditionally updated, anyway the new value is not yet committed.
   When the applet calls `javacard.framework.JCSystem.commitTransaction()` all conditional updates become the real new values of the objects that are updated. Like this if there is a failure - a power lost or other system failure occurs, then JCRE restores all the data modified in the transaction to its previous value, by calling `javacard.framework.JCSystem.abortTransaction()`

After this brief introduction of JCRE, certain properties of interest are discussed. They concern the above aspects of the Java Card : transactions, applet communication, applet life cycle.

4 Java Card Security Properties

The properties that we want to check an application for are presented in this section. Some of these properties do not have a relevant translation in JML, as they are syntactical - an example of such a property is: if an applet implements the interface Application then it cannot implement another Shareable interface. Such a statement can be checked by static analysis on the source code. Most of these properties can be seen as temporal ones - a method `x` cannot be invoked before the method `y` is invoked. The properties are divided in these groups as follows :

4.1 Interaction between an applet and JCRE

No nested transactions As we said earlier a transaction can be started only by the method `JCSystem.beginTransaction` and the transaction can be committed only by invoking the method `JCSystem.commitTransaction` (the methods are static) For this property a possible annotation is by adding a `ghost static variable TRANS` to the class JCSystem :

```
package javacard.framework;

   public class JCSystem {
  //@public ghost static int TRANS;
  //@requires TRANS == 0 ;
  //@modifies TRANS;
  //@ensures TRANS == 1 ;
  public void beginTransaction() {
     //@set TRANS = 1;
     ...
  }

  //@requires TRANS == 1 ;
  //@modifies TRANS;
  //@ensures TRANS == 0 ;
  public void commitTransaction() {
    //@set TRANS = 0;
    ...
  }
}
```

Example 1.

No checked exception can be thrown during the execution of the card We can specify this property by imposing the default predicate that must be proven after the exceptional termination of an applet state after an exception to be *False*.

Neither begin, nor commit can be done by invoking a method from some "shared" method For this we propose that every time when it is detected by the static analysis of Java code that a direct or transitive invokation to a *"transaction"* method is done to require that the class where this method call is done has type not smaller than the class `javacard.framework.Shareable`

No authentification can be done within a transaction Checking if the pin is correct is done by the method `check(byte, short,byte )` from interface `package javacard.framework`, see [11] For this property if the precondition of the method `check(...)` is that the ghost variable `TRANS` from `package javacard.framework.JCSystem` is equal to *0*.

In this way we have the code fragment for the method that does the verification of the pin number specified in this way:

```
package javacard.framework;
public interface PIN {
  //@requires javacard.framework.JCSystem.TRANS == 0;
  //@modifies nothing;
  check(byte, short,byte) {
      ...
  }
}
```
Example 2.

then for example a code that doesn't respect the property is :

```
...
JCSystem.beginTransaction();
verifyPin();
...
```
Example 3.

This code will fail to be proven correct because the the method *beginTransaction* ensures that the value of the variable *TRANS* is equal to 1 and the method `verifyPin` requires it to be equal to 0.

4.2 Life Cycle management

The life cycle can be modeled as a ghost variable also. Suppose a byte ghost static variable called `LIFESTATE` is introduced in applet . A possible applet state will be represented by ghost static final int variables :

```
//@ public static ghost byte BLOCKED = 0;
//@ public static final ghost byte PERSONALIZATION = 1;
```

A command can be executed iff it is authorised in this state every "user" command in an applet is represented in general by a separate method. Then for every command that can be executed only in state X, we will require that the method that executes it has as precondition

```
requires LIFESTATE== X;
```

If the application is blocked then no shareable object can be accessible The method that gives access to a Shareable object is from the `abstract class Applet` from the standard Java Card API :

```
abstract public class Applet {
   //@public static ghost byte LIFESTATE;
   ...
   //@ensures \result != null ⇒ (LIFESTATE != BLOCKED) &&
   (LIFESTATE != PERSONALIZATION )
   public Shareable getShareableInterfaceObject(...)  {
   }
}
```

Example 4.

In this way it will be guaranteed that the method will return an object iff the state is not blocked.

Only the command for personalization is permitted in the state of death For the method process that realizes some apdu command different from the personalization then we will have that the command should have as precondition:

```
abstract public class Applet {
   //@public static ghost byte LIFESTATE;
   //@public static ghost byte PERSONALIZATION;
   ...
   //@requires (LIFESTATE != PERSONALIZATION )
   private void appGetBalance(APDU apdu) {
      purse.appGetBalance(apdu);
   }
}
```

5 Weakest Precondition Calculus

This section deals with a technique which is a constructive way to find a precondition of a statement given some postcondition which is known as the weakest precondition calculus or *wp*. As we are interested in specification over predicates where program states are over the space of `ghost static` variables, an abstract version of the *wp* is defined, which is called *wp'*. We define *wp'* as a predicate transformer for predicates where variables are ghost variables, thus ignoring the Java variables and values. The correctness of *wp'* is proved.

5.1 Hoare Logic

Hoare Logic usually is used for proving program correctness see [6]. It provides proof rules to derive the program correctness of a complete program from the correctness of its constituents.Sentences in this logic have the form: *[P]s[Q]* for total correctness and $\{P\}s\{Q\}$ for partial correctness. These definitions involve assertions P and Q in some logic (usually Predicate logic) and a statement s from the language that we reason about. The total correctness sentence *[P]s[Q]* expresses that if in the program state x the assertion P holds, the execution of s that starts in x must terminate in a program state *x'* in which the assertion Q holds. The partial correctness sentence $\{P\}s\{Q\}$ expresses that if in the program state x the assertion P holds and if the execution of s that starts in x terminates in a program state *x'* then the assertion Q holds in state *x'*. In our work we deal with the total correctness of a Java program where assertions are first order predicates. The weakest precondition calculus (*wp*) is an algorithm to find a precondition *wp(s,Q)* for a program s given a postcondition Q for s such that the sentence *[wp(s,Q)]s[Q]* expresses the program correctness of statement *s*. Also the predicate calculated with *wp* is the weakest precondition among all possible preconditions *P'* of s that satisfy *[P']s[Q]*, i.e. $\forall P' : Predicate[P']s[Q].P' \Rightarrow wp(s,Q)$

5.2 *WP* calculus

Definition 2 (WP) *$wp(stmt, Post)$ is true for those initial states for which the stmt terminates in a program state that satisfies Post.*

The rules for *WP* with exceptions are formulated below see [12], [1].It should be pointed out that the postcondition from which the weakest precondition is calculated is a couple of two predicates - where the first one should be provable after all the executions of a statement that are exceptionally terminated and the second one should be provable for all the states in which the program is if a statement terminates its execution normally. Then the question is when a program terminates normally and when exceptionally : a program terminates abruptly on an exception if during its execution an exception is thrown and normally in all the other cases (there are another types of abrupt termination - when a break, return or continue cause the program to terminate. In this definition of the weakest precondition for Java these cases are not considered. In these definitions we neither treat Java subtyping. For example the exceptional postcondition must depend on the object with the smallest exceptional type that can manage the raised exception) Also for the rule of a method call functions $post^n \rightarrow Predicate$ and $post^e : MethodNames \rightarrow Predicate$ are used - they return the exceptional and normal postconditions respectively.

5.2.1 Definition of *WP* for a Simple Language

$$WP : STMT \rightarrow (Predicate, Predicate) \rightarrow Predicate$$

- $wp(skip, [Post^e, Post^n]) = Post^n$

- $wp(s1; s2, [Post^e, Post^n]) = wp(s1, [Post^e, wp(s2, [Post^e, Post^n])])$
- $wp(try\{s1\}catch(Exception)\{s2\}, [Post^e, Post^n]) = wp(s1, [wp(s2, [Post^e, Post^n]), Post^n])$
- $wp(throw\ Expr, [Post^e, Post^n]) = wp(Expr, [Post^e, Post^e])$
- $wp(call(m), [Post^e, Post^n]) =$
  $pre(m) \wedge \forall modifies(m)(post^n(m) \Rightarrow Post^n) \wedge (post^e(m) \Rightarrow Post^e))$
- $wp(x := Expr, [Post^e, Post^n]) = wp(Expr, [Post^e, Post^n[x \mid v]])$
  where v is the value of the expression *Expr*
- $wp(\ \{\ \text{s}\ \}\ , [Post^e, Post^n]) = wp(s, [Post^e, Post^n])$
- $wp(if\ (cond)\ s1\ else\ s2, [Post^e, Post^n]) =$

  $wp(cond, [P^e, (v == true) \Rightarrow wp(s1, [Post^e, Post^n])])$
  $\wedge$
  $wp(cond, [P^e, (v == false) \Rightarrow wp(s2, [Post^e, Post^n])])$
  , where v is the value of the expression *cond*
- $wp(while\ (cond)\ do\ s1, [Post^e, Post^n]) = X$

  where X is the weakest solution of the equation :
  $X = wp(cond, [P^e, v == true \ \Rightarrow\ wp(s1, X))$
  $\wedge$
  $v == false \ \Rightarrow\ Post^n])$
  or as *wp* distributes over conjunction :
  $X = wp(cond, [P^e, (v == true \ \Rightarrow\ wp(s1, X))])$
  $\wedge$
  $wp(cond, [P^e, v == false \ \Rightarrow\ Post^n])$

5.3 *WP'*: abstract *WP* calculus

As for our purposes *ghost static variables* will be used which neither depend on any Java variable we propose to abstract from the Java values and to construct a predicate transformer that do not take into account them.

5.3.1 Definition of *WP'* for a Simple Language

$$WP' : STMT \rightarrow (Predicate, Predicate) \rightarrow Predicate$$

- $wp'(x := E, [Post^e, Post^n]) = wp'(E, [Post^e, Post^n])$
  If we see once again the rule of *wp* for this case then we can see that a substitution is done in the predicate $Post^n$. Here as we limit the formulas to contain only ghost variables and as the variable x is a concrete Java variable we can omit this substitution.
- $wp'(skip, [Post^e, Post^n]) = Post^n$
- $wp'(s1; s2, [Post^e, Post^n]) = wp'(s1, [Post^e, wp'(s2, [Post^e, Post^n])])$
- $wp'(try\{s1\}catch(Exception)\{s2\}, [Post^e, Post^n]) = wp'(s1, [wp'(s2, [Post^e, Post^n]), Post^n])$

- $wp'(throw\ Expr, [Post^e, Post^n]) = wp'(Expr, [Post^e, Post^e])$

- $wp'(call(m), [Post^e, Post^n]) =$
  $pre(m) \wedge \forall modifies(m)(post^n(m) \Rightarrow Post^n) \wedge (post^e(m) \Rightarrow Post^e))$
- $wp'(\ \{\ \text{s}\ \}\ , [Post^e, Post^n]) = wp'(s, [Post^e, Post^n])$
- $wp'(if\ (cond)\ s1\ else\ s2, [Post^e, Post^n]) \ =$
  $wp'(cond, [P^e, wp'(s1, [Post^e, Post^n]) \wedge wp'(s2, [Post^e, Post^n])])$
  What can be noted for this rule is that it requires a precondition stronger than in the case *wp*. The motivation for this is the fact that the guards in the *if* statement are real Java values and we want to "get rid of " them.
- $wp'(while\ (cond)\ do\ s1, [Post^e, Post^n]) = X$
  where X is the weakest solution of the equation:
  $X = wp'(cond, [P^e, wp'(s1, X)\ \wedge\ Post^n])$

  As in the case of *if* the guard is not taken into account

5.3.2 Relation between *WP'* and *WP*

The relation that is seen between the standard precondition calculus and the precondition calculus *WP'* is that *WP'* is correct in relation with WP , but not complete. In terms of program verification this means that incorrect programs whose weakest precondition is calculated with the standard WP will be incorrect with weakest precondition calculated with WP' also. Still, there will be correct programs that will be rejected if their weakest precondition is calculated with WP' , but that will be correct in WP. Consider the example :

```
stmt =if (true) {
         JCSystem.beginTransaction()
} else {
         JCSystem.commitTransaction()
}
```

Applying the rule for the *wp'* for any couple of predicates Post we see that the precondition will be :
wp'(stmt, Post) = `JCSystem.TRANS == 0` $\wedge$ `JCSystem.TRANS == 1` $\wedge \forall (modifies(stmt) F)$
for some some F
which is an insatisfiable formula.
While in the case of *wp* we will have :
wp(stmt, Post) = true $\Rightarrow$ `JCSystem.TRANS == 0` $\wedge$
false $\Rightarrow$ `JCSystem.TRANS == 1` $\wedge \forall (modifies(stmt) F)$ for some
some F
which is not equivalent to the predicate $False$

5.3.3 Correctness

Here before dealing with the correctness of the *wp'* a property for monotoncity will be stated as it will participate in the proof for the correctness (For more details on predicate transformers and their properties see [3]):

Proposition 1 (Monotone) *wp' is monotone predicate transformer:*

$\forall \mathtt{P}, \mathtt{Q}, \mathtt{P}', \mathtt{Q}'$: Predicate $\forall \mathtt{s} : \mathtt{STMT}(\mathtt{P} \Rightarrow \mathtt{P}' \wedge \mathtt{Q} \Rightarrow \mathtt{Q}' \Longrightarrow \mathtt{wp}'(\mathtt{s}, [\mathtt{P}, \mathtt{Q}]) \Rightarrow \mathtt{wp}'(\mathtt{s}, [\mathtt{P}', \mathtt{Q}']))$

Proof : by induction over the structure of statement

Theorem 1 (Correctness)

$\forall stmt : STMT \forall Post : Predicate\ wp'(stmt, [Post^e, Post^n]) \Rightarrow wp(stmt, [Post^e, Post^n])$

Proof: The most interesting cases follow hereafter, the full proof is in the *Appendix* A

- stmt $= x := Expr$

  1. by definition $wp'(x := Expr,\ [Post^e,\ Post^n]) = wp'(Expr, [Post^e, Post^n])$
  2. by definition $wp(x := Expr, [Post^e,\ Post^n]) = wp(Expr, [Post^e, Post^n[x \mid v]])$
     where v is the value of *Expr* As $Post^n \in$ Predicate so it contains only ghostvariables
     $$\Rightarrow$$
     x $\notin FV(Post^n)$
     $$\Rightarrow$$
     $Post^n[x \mid v] = Post^n$
     $$\Rightarrow$$
     $wp(Expr, [Post^e, Post^n[x \mid v]] = wp(Expr, [Post^e, Post^n])$
  3. H.I.
     $$wp'(Expr, [Post^e, Post^n]) \Rightarrow wp(Expr, [Post^e, Post^n])$$
  4. from 1, 2, 3 the proposition holds

- stmt = s1;s2

  1. $wp'(s1; s2, [Post^e, Post^n]) =$
     $$= wp'(s1, [Post^e, [wp'(s2, [Post^e, Post^n])]])$$
  2. H.I. for s2:
     $$wp'(s2, [Post^e, Post^n]) \Rightarrow wp(s2, [Post^n, Post^e])$$
  3. As *wp'* is monotone and from the definition of a monotone predicate transformer :
     $wp'(s1, [Post^e, wp'(s2, [Post^e, Post^n])]]) \Rightarrow wp'(s1, [Post^e, wp(s2, [Post^e, Post^n]])$
  4. H.I. for s1 :
     $wp'(s1, [Post^e, wp'(s2, [Post^e, Post^n])]]) \Rightarrow wp(s1, [Post^e, wp(s2, [Post^e, Post^n]]))$
     which is equivalent to :
     $$wp'(s1; s2, [Post^n, Post^e]) \Rightarrow wp(s1; s2, [Post^n, Post^e]$$

End proof

6 Algorithms for generation of JML specification

6.1 Algorithm for generation of Preconditions

6.1.1 Definition

$$Pre : STMT \rightarrow Set(GhostVars) \rightarrow List(Formula)$$

The first argument is the statement whose precondition is calculated, the second argument is a set of variables that can be modified in the context where this statement is declared by the statements declared before this statement. The usual context here is a method body and the statements that are declared before a statement are all the statements that are executed before it when the method is called.

$$Pre(s, modif) = \begin{cases} Pre(s1, modif) & \text{if s = \{s1\}} \\ [true, nil] & \text{if s = skip} \\ List & \text{si (s=methCall(m))} \wedge \\ & \forall p : Predicate(\\ & member(p, Pre(m)) \wedge \\ & (\text{FV(p)} \cap \text{modif} = \varnothing) \\ & \Leftrightarrow member(p, List)) \\ @(Pre(cond, modif), & \\ \quad Pre(s1, modif \cup modif(cond)), & \\ \quad Pre(s2, modif \cup modif(cond))) & \text{if s=if(cond) then s1 else s2;} \\ @(Pre(cond, modif), & \\ \quad Pre(s1, modif \cup modif(cond))) & \text{if s = while(cond)s1;} \\ @(Pre(s1, modif), & \\ \quad Pre(s2, modif \cup modif(s1))) & \text{if s=s1;s2} \\ @(Pre(s1, modif), & \\ \quad Pre(s2, modif \cup modif(s1))) & \text{if s= try \{s1 \} catch (Exception)\{ s2\}} \\ Pre(s1, modif) & \text{if s= throw s1} \end{cases}$$

Finally the precondition of a method is defined as a function as follows :

$$Pre : MethodNames \rightarrow List(Formula)$$

$$Pre(methodName) = @(initPre(methodName), Pre(body(methodName), \varnothing))$$

where the function *body* returns for a method the statement that represents its body:

$$body : MethodNames \rightarrow STMT$$

and the function *initPre* returns for a method the precondition with which it is initialized(if it is initialized explicitly then the function will return the explicit initial expression else it will return the predicate `True`):

$$initPre : MethodNames \rightarrow STMT$$

6.1.2 Relation between the proposed precondition calculus and *WP'*

This section aims to find and prove a relation between the function named *Pre* (as it is defined in the section for the precondition) and *WP'*.

Before formulating the relation between the function for generation of preconditions and the *wp'* some properties that are needed for do this are stated:

The following proposition tells that the function `Pre(stmt, modif)` "filters" those predicates from the precondition of `stmt` whose free variables are not in the set `modif`

Lemma 1 $Pre(call(m), \varnothing) = Pre(m)$

Proof:
{by definition of function Pre}
$member(f, Pre(call(m), \varnothing)) \Longleftrightarrow f \in \{p \mid member(p, Pre(m)) \wedge FV(p) \cap \varnothing = \varnothing\}$
{for any set A is true: $A \cap \varnothing = \varnothing$}
$\Rightarrow$
$member(f, Pre(call(m), \varnothing)) \Longleftrightarrow f \in \{p \mid p \in Pre(m))\}$
{ which is the same as }
$Pre(call(m), \varnothing) = Pre(m)$
End proof

Proposition 2 $\forall modif \quad \forall stmt \quad member(f, Pre(stmt, modif)) \Longleftrightarrow f \in \{p \mid member(p, Pre(stmt, \varnothing)) \wedge FV(p) \cap modif = \varnothing\}$

Proof:
Here only part of the proof is written.(see *Appendix* B) *by structural induction over the structure of a stmt*

- stmt = call(m)

  { From the definition of function Pre for any set modif of variable names it is equal to: }
  $Pre(call(m), modif) = \{p \mid member(p, Pre(m)) \wedge FV(p) \cap modif = \varnothing\}$

  { by definition of Pre} $\Rightarrow$
  $member(f, Pre(call(m), modif)) \Longleftrightarrow f \in \{p \mid p \in Pre(m) \wedge (FV(p) \cap modif) = \varnothing)\}$
  { by the previous lemma } $\Rightarrow$
  $member(f, Pre(call(m), modif)) \Longleftrightarrow f \in \{p \mid p \in Pre(call(m), \varnothing) \wedge (FV(p) \cap modif) = \varnothing\}$

- stmt = s1;s2

1. by defintion of *Pre*:
   Pre(s1;s2, modif) =
   = @(Pre(s1, modif) , Pre(s2, modif $\cup$ modif(s1)))

2. for any set of variable names *modif* and any formula f it is true:
   $member(f, Pre(s1; s2, modif)) \iff member(f, Pre(s1, modif)) \vee member(f, Pre(s2, modif \cup modif(s1)))$

3. IH twice for any formula f:
   $member(f, Pre(s1, modif)) \iff f \in \{p \mid member(p, Pre(s1, \varnothing)) \wedge FV(p) \cap modif = \varnothing\}$

   $member(f, Pre(s2, modif \cup modif(s1))) \iff$
   $f \in \{p \mid member(p, Pre(s2, \varnothing)) \wedge$
   $(FV(p) \cap (modif \cup modif(s1))) = \varnothing\}$

4. from 1 and 2 follows:
   $member(f, Pre(s1; s2, modif))$
   $\iff$
   $f \in \{p \mid member(p, Pre(s1, \varnothing)) \wedge FV(p) \cap modif = \varnothing\} \vee$
   $f \in \{p \mid member(p, Pre(s2, \varnothing)) \wedge$
   $(FV(p) \cap (modif \cup modif(s1))) = \varnothing\}$
   $\iff$
   $f \in \{p \mid member(p, Pre(s1, \varnothing)) \wedge FV(p) \cap modif = \varnothing\} \cup$
   $\{p \mid member(p, Pre(s2, \varnothing)) \wedge$
   $(FV(p) \cap (modif \cup modif(s1))) = \varnothing\}$
   $\iff$
   $f \in \{p \mid (member(p, Pre(s1, \varnothing)) \wedge FV(p) \cap modif = \varnothing) \vee$
   $(member(p, Pre(s2, \varnothing)) \wedge$
   $(FV(p) \cap (modif \cup modif(s1))) = \varnothing)\}$
   $\iff$
   $f \in \{p \mid (member(p, Pre(s1, \varnothing)) \wedge FV(p) \cap modif = \varnothing) \vee$
   $(member(p, Pre(s2, \varnothing)) \wedge FV(p) \cap modif(s1) = \varnothing \wedge$
   $FV(p) \cap modif = \varnothing)\}$

5. apply the HI for $member(p, Pre(s2, \varnothing)) \wedge FV(p) \cap modif(s1) = \varnothing$
   $\iff$
   $f \in \{p \mid (member(p, Pre(s1, \varnothing)) \wedge FV(p) \cap modif = \varnothing) \vee$
   $(member(p, Pre(s1, modif(s1))) \wedge FV(p) \cap modif = \varnothing)\}$
   $\equiv$
   $f \in \{p \mid ((member(p, Pre(s1, \varnothing)) \vee member(p, Pre(s1, modif(s1)))) \wedge$
   $FV(p) \cap modif = \varnothing\}$

6. from 1:
   $f \in \{p \mid member(p, Pre(s1; s2, \varnothing)) \wedge FV(p) \cap modif = \varnothing\}$
   proven

End proof

Corrollarry 1 $FV(Pre(stmt, modif)) \cap modif = \varnothing$

We state that the function *Pre* is extracting the free part of the *wp'*.If a method is called in the precondition of which appears as program variable for which no other information is available before this method call. The standard way to this is then to place the predicate for this variable as a precondition of the caller and if the latter is called then to propagate it as a precondition of its caller and so on. From a practical point of view this is what a programmer should do when annotating a modular program. This has some formal point also which are formulated here.

Proposition 3 $wp(stmt, [P, Q_1 \wedge Q_2]) = wp(stmt, [P, Q_1]) \wedge Q_2$
$where FV(Q2) \cap modif(stmt) = \varnothing$ *and* $Post(stmt)! = False$

In this proof we will use the following predicate logic fact :
$(\exists x.P(x) \wedge (\forall x.P(x) \Rightarrow Z)\ ) \Rightarrow Z$

Proof:
$\exists x.P(x) \wedge \forall x.(P(x) \Rightarrow Z)$
$\equiv \exists x.P(x) \wedge \forall x.((\neg P(x)) \vee Z)$
{ predicate calculus }
$\equiv \exists x.P(x) \wedge ((\forall x.\neg P(x)) \vee Z)$
$\equiv \exists x.P(x) \wedge ((\neg \exists x.P(x)) \vee Z)$
$\equiv \exists x.P(x) \ \wedge \ \exists x.P(x) \Rightarrow Z$
$\equiv Z$
end of Proof

Now we go back to the prove of the proposition
Proof:
By structural induction over the structure of statement
The proof is done for *WP'*. The same result can be proven also for *WP'* The interesting part is given here after; the full proof is in the Appendix C

- stmt = call(m)
  $wp(stmt, [Post^e, Post^n \wedge Q]) =$
  { definition of wp for method call}
  $= Pre(m) \wedge \forall modif(m)((Post^n(m) \Rightarrow Post^n \wedge Q) \ \wedge (Post^e(m) \Rightarrow Post^e)) =$
  $= Pre(m) \wedge \forall modif(m)((Post^n(m) \Rightarrow Post^n) \ \wedge (Post^n(m) \Rightarrow Q) \ \wedge (Post^e(m) \Rightarrow Post^e)) =$
  $\forall$ { is distributive over conjunction }
  $= Pre(m) \wedge \forall modif(m)(Post^n(m) \Rightarrow Q) \ \wedge \forall modif(m)((Post^n(m) \Rightarrow Post^n) \ \wedge (Post^e(m) \Rightarrow Post^e)) =$
  { $Post^n(m) \neq \varnothing$, so the previous proposition holds }
  $Pre(m) \wedge Q \wedge \forall modif(m)((Post^n(m) \Rightarrow Post^n) \wedge (Post^e(m) \Rightarrow Post^e)) =$
  { definition of wp for method call}
  $wp(call(m), [Post^e, Post^n]) \wedge Q$
  the proposition holds

- stmt = s1;s2;

  $wp(s1; s2, [Post^e, Post^n \wedge Q])$
  $wp(stmt, [Post^e, Post^n \wedge Q]) =$
  { definition of wp for composition }
  $wp(s1, wp(s2, [Post^e, Post^n \wedge Q])) =$
  { I.H. for s2 }
  $wp(s1, wp(s2, [Post^e, Post^n]) \wedge Q) =$
  { I.H. for s1 }
  $wp(s1, wp(s2, [Post^e, Post^n]) \wedge Q =$
  { definition of wp for composition }
  $wp(s1; s2, [Post^e, Post^n]) \wedge Q$
  the proposition holds

Supposing that we have some modular program we will assume that the statements have their postconditions different from *false*.

From now on we will require that the postcondition will be satisfied for at least one possible value in the program space, i.e.it must be a predicate different from false.

Proposition 4 *Suppose that the formula Post in wp'(stmt,Post) is a formula containing only the connectors containing conjunction , negation, disjunction and universal quantification and every quantified universally subfomula is positive. Then wp'(stmt, Post) is a formula equivalent to a formula containing only the connectors conjunction , negation, disjunction and universal quantification. and every quantified universally subfomula is positive(positive here means that it is a formula not under negation).*

Proof: by structural induction over the `STMT`.
We want to state then that the function *Pre* calculates the predicate *Q2* from proposition 3 for *wp'*. Thus a relation between the existing algorithm and our approach is established. The next statement tells what is the logical information that we extract and propagate- this is the part which is free in a proof obligation formula generated with *wp'* containing only static ghost variables:

Theorem 2 $wp'(stmt, [Post^e, Post^n]) \ = \ \exists\, formula\, Q.Pre(stmt, \varnothing) \wedge \forall modif(stmt)(Q)$

Proof: By structural induction over the stmt structure. Only the cases for method call and composition are considered (other cases being similar):

{ by definition of wp' }
$wp'(call(m), [Post^e, Post^n]) =$
$Pre(m) \wedge \forall.modif(m)(Post^n(m) \Rightarrow Post^n \wedge Post^e(m) \Rightarrow Post^e)$
{ from Lemma 1 }
$Pre(call(m), \varnothing) \wedge \forall.modif(m)(Post^n(m) \Rightarrow Post^n \wedge Post^e(m) \Rightarrow Post^e) =$
{ if Q = $Post^n(m) \Rightarrow Post^n \wedge Post^e(m) \Rightarrow Post^e$ }
$Pre(call(m, \varnothing)) \wedge \forall.modif(m).Q$
the proposition holds

$wp'(s1; s2, [Post^e, Post^n]) =$
{ by definition of wp' }

$wp'(s1, wp'(s2, [Post^e, Post^n])) =$
{ I.H. for s2 }
$wp'(s1, Pre(s2, \varnothing) \wedge \forall modif(s2).Q^{s2}) =$
{ from proposition 2 the conjunctor of $Pre(s2, \varnothing)$ that doesnot contain free variables from modif(s2) is Pre(s2, modif(s1)) and let the other part of the formula be F}
$wp'(s1, Pre(s2, modif(s1)) \wedge F \wedge \forall modif(s2).Q^{s2}) =$
{ from proposition 3 }
$wp'(s1, F \wedge \forall modif(s2).Q^{s2}) \wedge Pre(s2, modif(s1)) =$
{ I.H. for s1 }
$Pre(s1, \varnothing) \wedge \forall modif(s1).Q^{s1} \wedge Pre(s2, modif(s1)) =$
{ the formula Q^{s1} has a subformula universally quantified over modif(s2) Q^{s2}. As Q^{s1} does not contain universal quantification under negation proposition 4, then this quantification can be pulled out by renaming where necessary}
$Pre(s1, \varnothing) \wedge \forall modif(s1) modif(s2).Q'^{s1} \wedge Pre(s2, modif(s1)) =$
{ by definition of function Pre}
$Pre(s1; s2, \varnothing) \wedge \forall modif(s1; s2).Q'^{s1}$ the proposition holds

End

6.2 Algorithm for generation of Normal Postconditions

6.2.1 Definition

As in the case of the precondition the first argument is the statement whose postcondition is calculated, the second argument is a set of variables that can be modified in the context where this statement is declared by the statements declared after this statement. The usual context here is a method body and the statements that are declared after a statement are all the statements that are executed before it when the method is called.

$$Post : STMT \rightarrow Set(GhostVars) \rightarrow List(Formula)$$

The function *Post* is defined as follows:

$$Post(s, modif) = \begin{cases} Post(s1, modif) & \text{if s = \{s1\}} \\ [true, nil] & \text{if s = skip} \\ List & \text{if (s=methCall(m))} \wedge \\ & \forall p : Predicate(\\ & member(p, Post(m)) \wedge \\ & \text{(FV(p)} \cap \text{modif} = \varnothing) \\ & \Leftrightarrow member(p, List)) \\ @(@(Post(s1, modif), Post(cond, modif \cup modif(s1)) \vee & \\ \quad @(Post(s2, modif), Post(cond, modif \cup modif(s2))) & \text{if s=if(cond) then s1} \\ & \text{else s2;} \\ @(Post(cond, modif \cup modif(s1)), & \\ \quad Post(s1, modif)) & \text{if s = while(cond)s1;} \\ @(Post(s2, modif), & \\ \quad Post(s1, modif \cup modif(s2))) & \text{if s=s1;s2} \\ @(Post(s2, modif), & \\ \quad Post(s1, modif \cup modif(s2))) & \text{if s=try \{s1\}} \\ & \text{catch(Exception) \{ s2 \}} \end{cases}$$

Reamark:For the case for an `if` statement we overload the disjunction symbol, by passing to it as argument lists.
The postcondition of a method is defined as a function as follows :

$$Post : MethodNames \rightarrow List(Formula)$$

$$Post(methodName) = @(initPost(methodName), Post(body(methodName), \varnothing))$$

and the function *initPost* returns for a method the postcondition with which it is initialised(if it is initialised explicitly then the function will return the explicit initial expression else it will return the predicate `True`):

$$initPost : MethodNames \rightarrow STMT$$

6.3 Algorithm for generation of Exceptional Postconditions

When talking about programming languages with exceptions, the standard definition of Hoare logic is not sufficient, see [?],[?]. To reason for correctness of Java programs, one should take into account all possible kinds of termination for a Java program - exceptional termination, normal termination, termination on continue, break, return. In order to do this, one should consider predicates imposed as postcondition when the program terminates normally and predicates that must be true when the program terminates abruptly - on continue, break, etc and predicates that must be true after an exception is thrown. In this section a similar technique for calculating the exceptional postcondition.

6.3.1 Definition

The exceptional postcondition is defined as a function not only of the set of the variables that can be modified by the statements that are executed after the statement but also of the set of exceptions that are handled. In the definition of the function that calculates the exceptional predicate for a given statement `stmt` we take into account all the thrown exceptions that are handled in order to exclude their postcondition from the exceptional postcondition of `stmt`. We will need to look at the types of the statements. So we will widen our language definition. A domain of class names will be used : `ClassNames`. A special subset of `ClassNames` will be of interest : the domain of the exception classes `ExceptionNames`. We need a subtype relation to define over the elements `ClassNames`, for which we will use the JML notation <:
So `name1` <: `name2` reads as
$name1$ is subtype of $name2$.
A function that given an expression will return its type named *type* is defined :
$type : \texttt{EXPR} \rightarrow \texttt{ClassNames}$

A function that takes as first argument a method name `m` and as second argument a name of exceptional type `e` returns the predicate that must be true when `m` returns exceptionally by throwing a `e`. In fact it returns the predicate `P(x)` specified in the JML clause `exsures (Exception ) P(x)` . If such a clause does not exist in the specification of the method the predicate `False` is returned:
$exsures : \texttt{MethodNames} \rightarrow \texttt{ExceptionNames} \rightarrow \texttt{Predicate}$

The signature of the exceptional postcondition is:
$ExcPost : \texttt{STMT} \rightarrow \texttt{List(ExceptionNames)} \rightarrow \texttt{Set(GhostVars)} \rightarrow$
$\texttt{List((exc : ExceptionNames, excpost : List(Formula)))}$

We consider that the default exceptional predicate is *false* - this means that if not specified explicitly the predicate which the program state should establish on an exceptional termination we consider that the program should not terminate exceptionally.

$$
ExcPost(s, excs, modif) = \begin{cases}
ExcPost(s1, excs, modif) & \text{if s = \{s1\}} \\[1ex]
[\langle java.lang.Exception, false\rangle, nil] & \text{if s = skip} \\[1ex]
\begin{array}{l} @(ExcPost(s1, excs, modif) \\ \langle o.exc, \\ \quad @(Post(s1, modif \cup modif(s2)), \\ \qquad o.excpost) \\ \rangle) \end{array} & \begin{array}{l} \\ \\ \\ \\ \text{if s = \{s1;s2\}} \\ \forall\ o\ in \\ ExcPost(s2, excs, modif) \end{array} \\[1ex]
\begin{array}{l} @(ExcPost(s1, [excs, e], modif), \\ \quad \langle o.exc, \\ \quad @(post, \\ \qquad o.excpost \\ \qquad \rangle \\ )) \end{array} & \begin{array}{l} \\ \\ \\ \\ \\ \text{if s = try \{s1\}} \\ \text{ catch (e)\{s2\}} \\ \forall\ o\ in \\ ExcPost(s2, excs, modif) \\ \wedge \\ \forall\ post\ in \\ CatchExc(\\ \quad s1, \\ \quad e, \\ \quad modif \cup modif(s2)) \end{array} \\[1ex]
List & \begin{array}{l} \text{if s = call(m)} \\ \wedge(Exc\_Name, exsures(m, Exc\_Name)) \\ in\ List \Leftrightarrow \\ \forall e:\ ExceptionNames : e\ in\ excs \\ \neg(Exc\_Name <: e) \end{array} \\[1ex]
List & \begin{array}{l} \text{if s = throw e;} \\ \langle (type\ e), Post(e)\rangle \\ in\ List \\ \Leftrightarrow \\ \forall e' : ExceptionNames : \\ e'\ in\ excs : \neg((type\ e) <: e') \end{array} \\[1ex]
ExcPost(cond; s1, excs, modif) & \text{if s = while } cond \text{ s1;} \\[1ex]
\begin{array}{l} @(ExcPost(cond; s1, excs, modif), \\ \quad ExcPost(cond; s2, excs, modif) \\ ) \end{array} & \begin{array}{l} \\ \\ \text{if s = if } cond \\ \text{ \{s1 \} else \{ s2 \};} \end{array}
\end{cases}
$$

The list `l = ExcPost(s, excs, modif)` returned must be transformed in a list

`l'` such that : $\forall exc : ExceptionNames\ \forall i : Nat(\ a_i\ in\ l) \wedge (a_i.exc == exc)$
$\Rightarrow$
$\exists\ c\ in\ l'\ c.exc == exc \wedge$
$c.excpost == \bigvee a_i.excpost$

In the definition for the case of `try { s1 }catch(Exc ) { s2 }` statement we treat `s1` with another function: *CatchExc*. It filters those exceptional post-conditions of the catch statement which are ensured to hold when an exception `Exc` (or an exception that is a subtype of `Exc`) is thrown on execution of `s1` and returns those of them which do not contain free variables that can be modified by `s2`.

$$CatchExc : STMT \rightarrow ExceptionNames \rightarrow List(Predicate)$$

$$CatchExc(s, e', modif) = \begin{cases} False & \text{if s = skip} \\ CatchExc(s1, e', modif) & \text{if s = \{s1\}} \\ List & \begin{array}{l}\text{if s = call(m)} \\ \wedge exsures(m, Exc\_Name) \\ \textit{in List} \Leftrightarrow \\ FV(exsures(m, Exc\_Name)) \cap modif = \varnothing \\ \wedge \\ (\textit{type } e) <: e' \end{array} \\ List & \begin{array}{l}\text{if s = throw e} \\ \wedge \\ p \textit{ in List} \\ \Leftrightarrow \\ p \textit{ in } Post(e) \\ \wedge \\ FV(p) \cap modif = \varnothing \\ \wedge \\ (\textit{type } e) <: e' \end{array} \\ \begin{array}{l} @(CatchExc(s1, e', modif), \\ \;Post(s1, modif \cup modif(s2)) \wedge \\ \;CatchExc(s2, e', modif) \\ ) \end{array} & \text{if s = s1;s2} \\ \begin{array}{l} @(CatchExc(cond; s1, e', modif), \\ CatchExc(cond; s2, e', modif)) \end{array} & \begin{array}{l}\text{if s = if } cond \\ \text{\{s1 \} else \{ s2 \};}\end{array} \\ List & \begin{array}{l}\text{if s = try \{s1\}} \\ \text{catch (e)\{s2\}} \\ \wedge \\ p \textit{ in List} \\ \Leftrightarrow \\ (o \textit{ in } ExcPost(s, [], []) \\ \wedge \\ o.exc <: e' \\ \wedge \\ p \textit{ in } o.excpost \\ \wedge \\ FV(p) \cap modif = \varnothing) \end{array} \\ CatchExc(cond; s1, e', modif) & \text{if s = while } cond \text{ s1;} \end{cases}$$

Then the method exceptional postcondition is defined as follows :

$$ExcPost : MethodNames \rightarrow List(Predicate)$$

$$ExcPost(meth\_Decl) = @(initExsPost(meth\_Decl), ExsPost(body(meth\_Decl, [], [])))$$

7 Conclusions

The result of our work is an implementation of the two techniques for calculating preconditions and postconditions which works for the JavaCard language. The specification that we generate is generated over the control flow graph of an application. As we stated before the specification that we generate is only static and abstract - as we allow only static ghost variables in the specification. For the kind of properties that we want to inspect an application one can obtain good results by using static fields for several reasons :

1. We propose a mechanism for automatic annotation which relieves users of the burden of annotating programs from scratch - a task few enjoy or are good at.
2. in a Java Card applet there is exactly one instance of the class that extends the `Applet` interface. So one can consider properties concerning the applet life cycle (see 3) as static properties as specifying its life cycle by using a static variable to model the application life state.
3. properties which involve invokation of static methods are susceptible to be specified over static fields: for example the transaction mechanism which is carried out by the static methods.
4. the Purse Applet - an applet provided by Gemplus(a smart card company) for research purposes was annotated with our tool for the transaction properties.

7.1 Future Work

1. *Generation of instance specification* The fact that the specification that we generate involves only static variables means that properties which are at instance level cannot be specified with our approach for example properties of the kind : the application cannot have access to the array outside the specified boundaries by the first byte and the length of the array. An appropriate objective for extending our approach is to generate instance specification :
2. An implementation of the exceptional postcondition calculation should be done which is neither a technical nor an algorithmic problem.
3. Due to the fact that we are using static analysis for the control flow graph that we construct the dynamic type of an object can be missed: consider this example :

```
public abstract A {
   public abstract void m();
   }
}

public class B extends A {
   public void m() {
     JCSystem.beginTransaction();
   }
}

public class C{
   public A a;
   public n() {
      a = new B();
      a.m();
   }
}
```

Here the correct precondition of method `n` in class `C` is `JCSystem.TRANS == 0`, (see 4). Our tool will ommit the dynamic type of the field `a` and will not inspect the method `m` in the class `B`, but the method `m` in class `A`. A good point to look at when extending the static analysis is to have more precise evaluation of the type of an object : for example constructing a set of possible types for an object which is a more refine approximation than that we are doing.

4. Another shortcoming is the fact that for guarded statements we generate very strong preconditions. Consider the example (see 4 for the example of transactions):

```
public class B extends A{
   public void m() {
      boolean c =JCSystem.transactionDepth();
     if (c == 0) {
         JCSystem.beginTransaction();
      } else {
         JCSystem.commitTransaction();
      }
   }
}
```

Standard rules f weakest precondition calculus tells us that
wp(body(m), true) =
$(c == \text{true}) \Rightarrow \text{TRANS} == 0 \wedge \forall\, \text{TRANS}(\ \text{TRANS} == 1 \Rightarrow \text{true})$
$\wedge$
$(c == \text{false}) \Rightarrow \text{TRANS} == 1 \wedge \forall\, \text{TRANS}(\ \text{TRANS} == 0 \Rightarrow \text{true}\ )$
but for the abstract weakest precondition (*wp'*) we obtain :
$$\text{wp'}(\text{body}(m), \text{true}) = \text{TRANS} == 0 \wedge \forall\, \text{TRANS}(\ \text{TRANS} == 1 \Rightarrow \text{true}) \wedge \text{TRANS} == 1 \wedge \forall\, \text{TRANS}(\ \text{TRANS} == 0 \Rightarrow \text{true})$$

The problem is that the formula obtained by *wp'* is unsatisfiable. An appropriate extension for our work is to generate *weaker* specification. This must surely involve instance variables and aliasing which is a task that need a careful treatment.

5. We would like to prove the soundness of our approach. Anyway we hope that it is correct. What one wants wants to hold is:
   Suppose we specify an application with our technque in order to verify if it satisfies a property `A`. If the proof obligations generated from this specification can be established this must imply that the application respects the property `A`.

6. Also a good direction is to generate specification for a block of statements. For example the block between every call of method `a()` and `b()` must not throw an exception. Thus specification will not be tied to method declarations, but can be at statement level.

A Proof of correctness of `wp'`

Proof:

- stmt = Empty

  1. by def wp'(Empty, [$Post^e, Post^n$]) = $Post^n$
  2. by def wp(Empty, [$Post^e, Post^n$]) = $Post^n$
  3. from 1 ,2

     $$wp'(Empty, [Post^e, Post^n]) \;\Rightarrow\; wp(Empty, [Post^e, Post^n])$$

- stmt = $x = Expr$

  1. by def $wp'(x = Expr,\; [Post^e,\; Post^n]) \;=\; wp'(Expr, [Post^e, Post^n])$
  2. by def $wp(x = Expr, [Post^e,\; Post^n]) = \; wp(Expr, [Post^e, Post^n[x \mid v]])$
     where v is the value of $Expr$ As $Post^n \in$ Predicate so it contains only ghostvariables

     $$\Rightarrow$$

     x $\notin FV(Post^n)$

     $$\Rightarrow$$

     $Post^n[x \mid v] = Post^n$

     $$\Rightarrow$$

     $wp(Expr, [Post^e, Post^n[x \mid v]] = wp(Expr, [Post^e, Post^n])$
  3. H.I.

     $$wp'(Expr, [Post^e, Post^n]) \Rightarrow wp(Expr, [Post^e, Post^n])$$

  4. from 1, 2, 3 the proposition holds

- stmt = if cond then s1 else s2

  1. $wp'(stmt, [Post^e, Post^n]) =$

     $$= wp'(cond, [P^e, wp'(s1, [Post^e, Post^n] \wedge wp'(s1, [Post^e, Post^n]])$$

  2. HI twice:

     $$wp'(s1, [Post^e, Post^n]) \;\Rightarrow\; wp(s1, [Post^e, Post^n])$$

     $$wp'(s2, [Post^e, Post^n]) \;\Rightarrow\; wp(s2, [Post^e, Post^n])$$

3. from 2 :

$$wp'(s1, [Post^e, Post^n]) \wedge v == true \ \Rightarrow \ wp(s1, [Post^e, Post^n])$$

$$wp'(s2, [Post^e, Post^n]) \wedge v == false \ \Rightarrow \ wp(s2, [Post^e, Post^n])$$

$$P^e \ \Rightarrow \ P^e$$

where v is the value of the expression $cond$

4. from 3 : $wp'(s1, [Post^e, Post^n]) \Rightarrow \ v == true \ \Rightarrow \ wp(s1, [Post^e, Post^n])$
   $wp'(s2, [Post^e, Post^n]) \Rightarrow \ v == false \ \Rightarrow \ wp(s2, [Post^e, Post^n])$

5. from 4
   $wp'(s1, [Post^e, Post^n]) \wedge wp'(s2, [Post^e, Post^n])$

   $$\Rightarrow$$

   $v == true \ \Rightarrow \ wp(s1, [Post^e, Post^n]) \wedge$
   $v == false \Rightarrow \ wp(s2, [Post^e, Post^n])$
   where v is the value of the expression $cond$

6. from monotonciity of WP'
   $wp'(cond, [P^e, wp'(s1, [Post^e, Post^n]) \ \wedge \ wp'(s2, [Post^e, Post^n])])$

   $$\Rightarrow$$

   $$\begin{aligned} wp'(cond, [P^e, (&v == true \ \Rightarrow \ wp(s1, [Post^e, Post^n]) \\ &\wedge \\ &(v == false \ \Rightarrow \ wp(s2, [Post^e, Post^n])])]) \end{aligned}$$

7. I.H.

   $$\begin{aligned} wp'(cond, [P^e, &v == true \ \Rightarrow \ wp(s1, [Post^e, Post^n]) \\ &\wedge \\ &v == false \ \Rightarrow \ wp(s2, [Post^e, Post^n])])]) \end{aligned}$$

   $$\Rightarrow$$

   $$\begin{aligned} wp(cond, [P^e, &v == true \ \Rightarrow \ wp(s1, [Post^e, Post^n]) \\ &\wedge \\ &v == false \ \Rightarrow \ wp(s2, [Post^e, Post^n])])]) \end{aligned}$$

8. from 6 and 7 from transitivity of the implication the property holds

- stmt = s1;s2

1. $wp'(s1; s2, [Post^e, Post^n]) =$

   $$= wp'(s1, [Post^e, [wp'(s2, [Post^e, Post^n])])$$

2. H.I. for s2:

$$wp'(s2, [Post^e, Post^n]) \Rightarrow wp(s2, [Post^n, Post^e])$$

3. As *wp'* is monotonne and from the definition of a monotonne predicate transformer :

$$wp'(s1, [Post^e, wp'(s2, [Post^e, Post^n])]) \Rightarrow wp'(s1, [Post^e, wp(s2, [Post^e, Post^n]])$$

4. H.I. for s1 :

$$wp'(s1, [Post^e, wp'(s2, [Post^e, Post^n])]) \Rightarrow wp(s1, [Post^e, wp(s2, [Post^e, Post^n]]))$$

which is equivalent to :

$$wp'(s1; s2, [Post^n, Post^e]) \Rightarrow wp(s1; s2, [Post^n, Post^e]$$

- stmt = try { s1 } catch { s2 }
  analogous to the previous case
- stmt = call(m)
  as $wp'(call(m), [Post^n, Post^e]) = wp(call(m), [Post^n, Post^e])$ we can conclude that the proposition holds

□

B Proof of Proposition 1 from the section for the relation between WP and WP'

Proposition
$\forall modif \;\; \forall stmt \;\; member(f, Pre(stmt, modif)) \Longleftrightarrow f \in \{p \mid member(p, Pre(stmt, \varnothing)) \wedge FV(p) \cap modif = \varnothing\}$

Lemma 2 $Pre(call(m), \varnothing) = Pre(m)$

Proof:
$Pre(call(m), \varnothing) = Pre(m)$

$member(f, Pre(call(m), \varnothing)) \Longleftrightarrow f \in \{p \mid member(p, Pre(m)) \wedge FV(p) \cap \varnothing = \varnothing\}$
{for any set A is true: $A \cap \varnothing = \varnothing$}
$\Rightarrow$
$member(f, Pre(call(m), \varnothing)) \Longleftrightarrow f \in \{p \mid p \in Pre(m))\}$ {we conclude }
$*$: $Pre(call(m), \varnothing) = Pre(m)$

by structural induction over the structure of a stmt

- stmt = skip

  $Pre(skip, modif) = @(true, nil)$ for any modif set

$\Rightarrow$ {true for $\varnothing$ }
$\Rightarrow$ $Pre(skip, \varnothing) = @(true, nil)$
{ for any set modif of variable names we have } $(FV(true) \cap modif) = \varnothing \cap A = \varnothing$
$\Rightarrow$
$(true \in Pre(skip, modif))\wedge$ there are no other formulas in the list Pre(skip, modif) $\wedge(member(true, Pre(skip, \varnothing))\wedge$ there are no other formulas in the list Pre(skip, $\varnothing$) $\wedge(FV(true) \cap modif) = \varnothing$ true for any modifies set
$\Rightarrow$
$member(true, Pre(skip, modif)) \iff f \in \{p \mid member(p, Pre(skip, \varnothing))\wedge FV(p) \cap modif = \varnothing\}$

- stmt = call(m)

  { From the definition of function Pre for any set modif of variable names it is equal to: }
  $Pre(call(m), modif) = \{p \mid member(p, Pre(m)) \wedge FV(p) \cap modif = \varnothing\}$

  { by definition of Pre} $\Rightarrow$
  $member(f, Pre(call(m), modif)) \iff$
  $f \in \{p \mid p \in Pre(m) \wedge FV(p) \cap modif = \varnothing)\}$
  $* \Rightarrow$
  $member(f, Pre(call(m), modif)) \iff f \in \{p \mid p \in Pre(call(m), \varnothing) \wedge FV(p) \cap modif = \varnothing\}$

- stmt = if (cond) then s1 else s2

  1. by defintion of *Pre*:
     Pre(if (cond) then s1 else s2, modif) =
     $@(Pre(cond, modif), Pre(s1, modif \cup modif(cond)), Pre(s2, modif \cup modif(cond)))$

  2. for any set of variable names *modif* and any formula f it is true:
     $member(f, Pre(if\ (cond)\ then\ s1\ else\ s2, modif))$
     $\iff$
     $member(f, Pre(cond, modif)) \vee member(f, Pre(s1, modif \cup modif(cond))) \vee member(f, Pre(s2, modif \cup modif(cond)))$

  3. IH three times for any formula f :
     $member(f, Pre(cond, modif))$
     $\iff$
     $f \in \{p \mid member(p, Pre(cond, \varnothing)) \wedge FV(p) \cap modif = \varnothing\}$

     $member(f, Pre(s1, modif \cup modif(cond)))$
     $\iff$
     $f \in \{p \mid member(p, Pre(s1, \varnothing)) \wedge FV(p) \cap (modif \cup modif(cond)) = \varnothing\}$

$member(f, Pre(s2, modif \cup modif(cond)))$
$\iff$
$f \in \{p \mid member(p, Pre(s2, \varnothing)) \wedge FV(p) \cap (modif \cup modif(cond)) = \varnothing\}$

4. from 2,3 follows:
   member(f, Pre(if (cond) then s1 else s2, modif)
   $\iff$
   $f \in \{p \mid member(p, Pre(cond, \varnothing)) \wedge FV(p) \cap modif = \varnothing\}$
   $\vee$
   $f \in \{p \mid member(p, Pre(s1, \varnothing)) \wedge FV(p) \cap (modif \cup modif(cond)) = \varnothing\}$
   $\vee$
   $f \in \{p \mid member(p, Pre(s2\varnothing)) \wedge FV(p) \cap (modif \cup modif(cond)) = \varnothing\}$

   $\iff$

   $f \in \{p \mid (member(p, Pre(cond, \varnothing)) \wedge (FV(p) \cap modif) = \varnothing)$
   $\vee$
   $(member(p, Pre(s1, \varnothing) \wedge FV(p) \cap (modif \cup modif(cond)) = \varnothing)$
   $\vee$
   $(member(p, Pre(s2, \varnothing) \wedge FV(p) \cap (modif \cup modif(cond)) = \varnothing)\}$

   $\iff$
   $f \in \{p \mid (member(p, Pre(cond, \varnothing)) \wedge (FV(p) \cap modif) = \varnothing)$
   $\vee$
   $(member(p, Pre(s1, \varnothing) \wedge (FV(p) \cap modif(cond)) = \varnothing \wedge (FV(p) \cap modif) = \varnothing)$
   $\vee$
   $(member(p, Pre(s2, \varnothing) \wedge (FV(p) \cap modif(cond)) = \varnothing \wedge (FV(p) \cap modif) = \varnothing)\}$

   $\iff$
   $f \in \{p \mid (member(p, Pre(cond, \varnothing))$
   $\vee\ member(p, Pre(s1, modif(cond)))$
   $\vee\ member(p, Pre(s2, modif(cond)))$
   $\wedge\ FV(p) \cap modif = \varnothing\}$
   {follows from 2 }
   $\equiv$
   $f \in \{p \mid$ member(p, Pre (if (cond) then s1 else s2, $\varnothing$)) $\wedge FV(p) \cap modif = \varnothing\}$

- stmt = s1;s2

1. by defintion of *Pre*:
   Pre(s1;s2, modif) =
   = @(Pre(s1, modif) , Pre(s2, modif ∪ modif(s1)))

2. for any set of variable names *modif* and any formula f it is true:
   $member(f, Pre(s1; s2, modif)) \iff member(f, Pre(s1, modif)) \vee member(f, Pre(s2, modif \cup modif(s1)))$

3. IH twice for any formula f:
   $member(f, Pre(s1, modif)) \iff f \in \{p \mid member(p, Pre(s1, \varnothing)) \wedge FV(p) \cap modif = \varnothing\}$

   $member(f, Pre(s2, modif \cup modif(s1))) \iff$
   $f \in \{p \mid member(p, Pre(s2, \varnothing)) \wedge$
   $(FV(p) \cap (modif \cup modif(s1))) = \varnothing\}$

4. from 1 and 2 follows:
   $member(f, Pre(s1; s2, modif))$
   $\iff$
   $f \in \{p \mid member(p, Pre(s1, \varnothing)) \wedge FV(p) \cap modif = \varnothing\} \vee$
   $f \in \{p \mid member(p, Pre(s2, \varnothing)) \wedge$
   $(FV(p) \cap (modif \cup modif(s1))) = \varnothing\}$
   $\iff$
   $f \in \{p \mid member(p, Pre(s1, \varnothing)) \wedge FV(p) \cap modif = \varnothing\} \cup$
   $\{p \mid member(p, Pre(s2, \varnothing)) \wedge$
   $(FV(p) \cap (modif \cup modif(s1))) = \varnothing\}$
   $\iff$
   $f \in \{p \mid (member(p, Pre(s1, \varnothing)) \wedge FV(p) \cap modif = \varnothing) \vee$
   $(member(p, Pre(s2, \varnothing)) \wedge$
   $(FV(p) \cap (modif \cup modif(s1))) = \varnothing)\}$
   $\iff$
   $f \in \{p \mid (member(p, Pre(s1, \varnothing)) \wedge FV(p) \cap modif = \varnothing) \vee$
   $(member(p, Pre(s2, \varnothing)) \wedge FV(p) \cap modif(s1) = \varnothing \wedge$
   $FV(p) \cap modif = \varnothing)\}$

5. apply the HI for $member(p, Pre(s2, \varnothing)) \wedge FV(p) \cap modif(s1) = \varnothing$
   $\iff$
   $f \in \{p \mid (member(p, Pre(s1, \varnothing)) \wedge FV(p) \cap modif = \varnothing) \vee$
   $(member(p, Pre(s1, modif(s1))) \wedge FV(p) \cap modif = \varnothing)\}$
   $\equiv$
   $f \in \{p \mid ((member(p, Pre(s1, \varnothing)) \vee member(p, Pre(s1, modif(s1)))) \wedge$
   $FV(p) \cap modif = \varnothing\}$

6. from 1:
   $f \in \{p \mid member(p, Pre(s1; s2, \varnothing)) \wedge FV(p) \cap modif = \varnothing\}$
   proven

End proof

C Proof of Proposition 2 from the section for the relation between WP and WP'

Proposition:
$wp(stmt, [P, Q_1 \wedge Q_2]) = wp(stmt, [P, Q_1]) \wedge Q_2$
$where FV(Q2) \cap modif(stmt) = \varnothing$ and $Post(stmt)! = False$ Proof : Here only the case for the normal postcondition is considered as the case where the conjunction being in the exceptional postcondition is identical

- stmt = call(m)
  $wp(stmt, [Post^e, Post^n \wedge Q]) =$
  { definition of wp for method call}
  $= Pre(m) \wedge \forall modif(m)((Post^n(m) \Rightarrow Post^n \wedge Q) \wedge (Post^e(m) \Rightarrow Post^e)) =$
  $= Pre(m) \wedge \forall modif(m)((Post^n(m) \Rightarrow Post^n) \wedge (Post^n(m) \Rightarrow Q) \wedge (Post^e(m) \Rightarrow Post^e)) =$
  $\forall$ { is distributive over conjunction }
  $= Pre(m) \wedge \forall modif(m)(Post^n(m) \Rightarrow Q) \wedge \forall modif(m)((Post^n(m) \Rightarrow Post^n) \wedge (Post^e(m) \Rightarrow Post^e)) =$
  { $Post^n(m) \neq \varnothing$, so the previous proposition holds }
  $Pre(m) \wedge Q \wedge \forall modif(m)((Post^n(m) \Rightarrow Post^n) \wedge (Post^e(m) \Rightarrow Post^e)) =$
  { definition of wp for method call}
  $wp(call(m), [Post^e, Post^n]) \wedge Q$
  the proposition holds

- stmt = s1;s2;
  $wp(s1; s2, [Post^e, Post^n \wedge Q])$
  $wp(stmt, [Post^e, Post^n \wedge Q]) =$
  { definition of wp for composition }
  $wp(s1, wp(s2, [Post^e, Post^n \wedge Q])) =$
  { I.H. for s2 }
  $wp(s1, wp(s2, [Post^e, Post^n]) \wedge Q) =$
  { I.H. for s1 }
  $wp(s1, wp(s2, [Post^e, Post^n]) \wedge Q =$
  { def. of wp for composition }
  $wp(s1; s2, [Post^e, Post^n]) \wedge Q$
  the proposition holds

- stmt = if cond then s1 else s2
  $wp(stmt, [Post^e, Post^n \wedge Q]) =$
  { definition of wp for if , where v is the value of the guard cond }
  $wp(cond, [Poste^e, v == t \Rightarrow wp(s1, [Post^e, Post^n \wedge Q])$
  $\wedge$
  $v == f \Rightarrow wp(s2, [Post^e, Post^n \wedge Q])]) =$
  $wp(cond, [Post^e, v == t \Rightarrow wp(s1, [Post^e, Post^n \wedge Q])])$
  $\wedge$
  $wp(cond, [Poste^e, v == f \Rightarrow wp(s2, [Post^e, Post^n \wedge Q])])$

{ H.I. twice }
$wp(cond, [Post^e, (v == t \Rightarrow (wp(s1, [Post^e, Post^n]) \wedge Q))\wedge$
$\quad (v == f \Rightarrow (wp(s2, [Post^e, Post^n]) \wedge Q))]) =$
{ logic }
$wp(cond, [Post^e, (v == t \Rightarrow (wp(s1, [Post^e, Post^n]))) \wedge$
$\quad (v == t \Rightarrow Q)\wedge$
$\quad (v == f \Rightarrow wp(s2, [Post^e, Post^n]))\wedge$
$\quad (v == f \Rightarrow Q)]) =$

$wp(cond, [Post^e, (v == t \Rightarrow wp(s1, [Post^e, Post^n])) \wedge$
$\quad (v == f \Rightarrow wp(s2, [Post^e, Post^n]))\wedge$
$\quad Q]) =$

{ H.I. }
$wp(cond, [Post^e, (v == t \Rightarrow wp(s1, [Post^e, Post^n])) \wedge$
$\quad (v == f \Rightarrow wp(s2, [Post^e, Post^n])) \wedge Q =$

{ definition of wp for if }
$wp(if\ cond\ then\ s1\ else\ s2, [Post^e, Post^n]) \wedge Q$
the proposition holds

- stmt = x = E
  { v is the value of E}
  $wp(x = E, [Post^e, Post^n \wedge Q]) = wp(E, [Post^e, (Post^n \wedge Q)[x \mid v]])$ see 2.4.2

  $wp(x = E, [Post^e, Post^n]) = wp(E, [Post^e, Post^n[x \mid v] \wedge Q])$
  { H.I. }

  $wp(x = E, [Post^e, Post^n]) = wp(E, [Post^e, Post^n[x \mid v]]) \wedge Q$
  { definition of wp }
  $wp(x = E, [Post^e, Post^n]) = wp(x = E, [Post^e, Post^n]) \wedge Q$

- stmt = while cond do s
  $wp(stmt, [P^e, P^n \wedge Q]) = X$
  { definition of wp X is the greatest fixpoint of the equation (Djikstra , "Predicate calculus and program semantics)": }
  $X = wp(cond, v == true \Rightarrow wp(s, X) \wedge v == false \Rightarrow (P^n \wedge Q))$
  { as the wp is monotonne function and continuous the solution exists and it can be calculated iteratively and is equal to the disjunction of } X_i

  where
  $X_0 = P^n \wedge Q$
  $X_{i+1} = wp(cond, v == true \Rightarrow (P^n \wedge Q) \wedge v == true \Rightarrow wp(s, X_i))$

  $X' = wp(while\ cond\ do\ s, [P^e, P^n])$
  and X' is the union of X'_i , where

$X'_0 = P$
$X'_{i+1} = wp(cond, v == true \Rightarrow P^n \wedge v == true \Rightarrow wp(s, X'_i))$

now we prove that $\forall i. \quad X_i = X'_i \wedge Q$

$\forall i. \quad X_i = X'_i \wedge Q$
$X_0 = P^n \wedge Q$
$X'_0 = P^n$
$\Rightarrow X_0 = X'_0 \wedge Q$

{ H.I. } $X_i = X'_i \wedge Q$
{ inductive case } $X_{i+1} = wp(cond, v == f \Rightarrow (P^n \wedge Q) \ \wedge \ cond == t \Rightarrow wp(s, X_i))$
{ apply H.I. } $X_{i+1} = wp(cond, v == f \Rightarrow (P^n \ \wedge Q) \ \wedge \ cond == t \Rightarrow wp(s, X'_i \wedge Q))$
{ apply H.I. } $X_{i+1} = wp(cond, v == f \Rightarrow (P^n \ \wedge Q) \ \wedge \ cond == t \Rightarrow (\ wp(s, X'_i) \wedge Q \ ))$
{ logic laws } $X_{i+1} = wp(cond, v == f \Rightarrow P^n \ \wedge$
$v == f \Rightarrow Q \ \wedge$
$v == t \Rightarrow Q \ \wedge$
$v == t \Rightarrow \ wp(s, X'_i)) =$
$= wp(cond, (cond == f \Rightarrow P^n \wedge v == t \Rightarrow \ wp(s, X'_i)) \wedge Q)$
{ apply definition of X' } $= wp(cond, (X'_{i+1}) \wedge Q)$
$X_{i+1} = wp(cond, (X'_{i+1}) \wedge Q)$

Then as we have that

1. by Knaster Tarski :
   $wp(while \ cond \ do \ s, P^n) = \bigvee X'_i$
2. $\bigvee X_i = \bigvee (X'_i \wedge Q) = (\bigvee X'_i) \wedge Q$
3. $wp(while \ cond \ do \ s, P^n \wedge Q) = X = \bigvee X_i$
4. we will be able to conclude that $X = wp(while \ cond \ do \ s, P^n) \wedge Q$

End proof

References

[1] Lilian Burdy. Jack specification.

[2] Lilian Burdy and Antoine Requet. Jack: Java applet correctness kit. Technical report, Gemplus Software Research Labs.

[3] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics.*

[4] Cormac Flanagan and K.Rustan M. Leino. Houdini, an annotation assistant for esc/java.

[5] Engelbert Hubbers, M.Oostdijk, and Erik Poll. From finite state machines to provably correct java card applets.

[6] Marieke Huisman. *Reasoning about Java programs in higher order logic using PVS and Isabelle.* PhD thesis, University of Nijmegen, 2001.

[7] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. A behavioral interface specification language for java. Technical report, Iowa State University, feb 2000.

[8] K. Rustan M. Leino, Greg Nelson, and James B. Saxe. *ESC/Java User's Manual.* Compaq Computer Corporation, 2000.

[9] Renaud Marlet and Daniel Le Metayer. Security properties and java card specifities to be studied in the secsafe project. Technical report, Trusted Logic, 2001.

[10] Sun Microsystems. *Java Card 2.2 Application Programming Interface.*

[11] Sun Microsystems. *Java Card 2.2 Runtime Environment(JCRE) Specification.*

[12] K.Rustan M.Leino. *Toward Reliable Modular programs.* PhD thesis, California Institute of Technology, jan 1995.

[13] Jeremy W. Nimmer and Michael D. Ernst. Static verification of dynamically detected program invariants:integrating daikon and esc/java.