Taking into account Java's Security Manager for static verification

Julien Charles

INRIA Sophia-Antipolis
julien.charles@inria.fr

Abstract. The verification of Java programs is a difficult task, especially with components like the Security Manager which modify the semantic of the Java Virtual Machine (JVM). To model this invasive behaviour the Security Manager can be implemented as an aspect component, using AspectJ.
In this paper we describe a framework for static verification of Java programs containing AspectJ advices specified with Pipa and we instanciate it over a case study, the Security Manager. The framework is built around a weakest precondition calculus over Java bytecode using guarded commands. It has an advice weaving semantic which is correct against AspectJ advice weaving semantic.

1 Introduction

Java is a language commonly used on small device, with specific flavour of it being used on heterogenous devices like cellphones or JavaCards. Thus taking into account their specific caracteristic each implementations is important for verification. Especially if specific system components change the overall behaviour of JVMs. Verification of system components is scattered through litterature: there has been verifications of bytecode verifiers, as well as access contoller. As far as our knowledge goes though [13], the Security Manager of the JVM has not been verified and its semantic is not taken into account when usually doing static verification of Java programs.

The main goal of this paper is to verify an implementation of a Security Manager, and also to give a framework to more generally verify programs taking into account the semantic differences that different component of the JVM can bring to executions of program inside a JVM.

1.1 The Security Manager

The Security Manager applies the security policies on two levels. First on the library level: each time a writing or reading operation is called for instance a call to the Security Manager is made and if the caller has not respected a given security policy, a `SecurityException` is thrown. Second on the JVM level: each time a class is loaded, it checks on a meta level with the use of the ClassLoader (which for this paper will be considered as a part of the JVM) if the currently

inspected class has the right to access a specified type from an outside package. The latter is hard to take into account for a program verification, because it changes the behavior of type resolution which is part of the language semantic. A practical way to model these changes would be to consider the Security Manager as an Aspect which would weave at the cutting points representing type resolution.

```
package a;
public class Main {
  static {
    System.setSecurityManager(new SecurityManager() {
      public void checkPackageAccess(String target) {
        if(target.equals("b"))
          throw new SecurityException("That is true");
        }});
    }

    public static void main(String[] args) {
      System.out.println("Nextline will throw an exception");
      b.A a = new b.A();
    }
  }
```

The output of the program:

```
Next line will throw an exception
Exception in thread "main" java.lang.SecurityException: That is true
       at a.b.Main$1.checkPackageAccess(Main.java:9)
       ...
```

Fig. 1. An invasive Security Manager

1.2 Modeling with AspectJ

Aspect Oriented Programming (AOP) is a paradigm that offers modularity though it is ortogonal to the usual Object Oriented Programming paradigm. AOP enables to weave code directly into a program, changing the behaviour of given language constructs. Two new notions have been introduced through AOP:

- the concept of cutting points, points in the program where code can be inserted, and
- advices, code to be inserted at a specified cutting point.

AspectJ is one of the most popular of the AOP languages. It is Java-based, so it is a natural choice for modelling the Security Manager through aspects.

Implementing the Security Manager as an invasive Aspect is easy. As we have seen in the above example, a minimal Security Manager could change the behaviour of the JVM at the type access step. The difficulty here would be

that the type is checked only on the first call, for this purpose a field has to be introduced, as shown in Figure 2.

```
public aspect SecurityManager {
  Set<String> s = new HashSet<String>();
  pointcut anyPublicMethod(Object o) : target(o) && !within(SecurityManager)
                                       && call( public *(*))
  before(Object o) : anyPublicMethod(o) {
    String pkg = o.getClass().getPackage().toString();
    if(!s.contains(pkg)) {
      s.add(pkg);
      if(pkg.equals("b"))
        throw new SecurityException("That is true");

    }
  }
}
```

Fig. 2. An Aspect implementation of the invasive SecurityManager

1.3 Verification framework

The verification framework we will use to handle the Security Manager concerns is an adaptation of static verification techniques based on weakest precondition calculus. It is inspired by Java extended static verification tools that use guarded commands language like ESC/Java2 [8] and the Mobius PVE [9].

The verification process is made of several steps: First the program and the aspects have to be fully specified:

1. the program and its aspects are annotated using an aspect specific behavioural specification language, Pipa,
2. the behavioural specifications are desugared
3. the aspects are abstracted to models

Then the proper compilation of aspects is done:

1. the program is compiled to bytecode with its specifications
2. the program is transformed into guarded commands
3. the model methods representing advices are weaved into the program

Finally a weakest precondition calculus is made on the transformed program, and verification conditions are generated in order to be solved automatically or interactively.

1.4 Related Work

The verification framework is adapted for the verification of AspectJ programs annotated with Pipa. The pointcut semantic of AspectJ has been properly defined [1] as well as the advice weaving semantic [14] though only parts have

been formalized [4]. Pipa [26] is an annotation language which was inspired by Clifton and Leavens' work [6]. There are some extensions to it like pointcuts annotations [24], or Moxa [25]. Our framework relies on verification based on an intermediate language, like what is done in ESC/Java2 [11], Boogie [2], or Krakatoa [20]. The work is made to be adapted on a BoogiePL-like guarded command which is coupled with a VCGen [3, 12].

Modular verification of Aspects Verification in Aspect Oriented Programming language is often linked to a modular approach. It is orthogonal to the aim of this paperh: we tailor verification of aspects which were not specially wanted by the user, and are most-likely imposed by the environment. Clifton and Leavens [6, 7, 5] propose a programming discipline to define and specify aspects, using an extension of JML as specification language. They define two kind of aspects, the *spectators* and *assistants* and propose to explicitly allows the weaving of these aspects. This approach allows efficient modular reasonning and easy implementations because they show a way of weaving specifications using a control-flow graph analysis. In their 2004 work [15] Krishnamurthi *et al* present a model-checking framework for verification of programs containing aspects. It is quite different from our work as it presents a generic framework for aspects. In an unpublished paper [16], Kunz presents a Hoare logic for modular verification of aspects. The paper is quite formal, but he does not add special annotations for method specifications as Clifton and Leavens. For this reason his approach remains less modular than Clifton's, but more expressive.

Outline: First, in Section 2 we will show how the program has to be properly annotated. Then in Section 3 we will transform the program to ease its verification: the aspects will be turned into model methods and the main program code into guarded commands. In section 4 we will explain how to verify the transformed program and finally give a conclusion in Section 5.

2 Specifying the program

The verification framework we are targetting relies on program and aspects specifications that are precise enough to make the verification conditions generated provable. The annotation language we use is Pipa (an extension of JML for AspectJ). The subset of the language is a JML level 0 [17] like subset of Pipa. Pointcuts specifications [24] and universe types will be ignored. Nevertheless some non level 0 constructs like model or pure methods are allowed. Behavioural specifications are allowed too because these specifications can be desugared [22].

2.1 JML

JML is a behavioural specification language for Java. It is made of numerous keyword: you can express methods pre and post conditions with it (the keywords `requires` and `ensures`), exceptional postcondition (`exsures`). There are notions of frame conditions (the `assignable` clauses). One can also express assertions

(`assert` construct), and loop invariants. Some data structures can be defined only on the specification level, the model classes, fields and methods.

Invasive model methods Model methods are specification methods that can be with or without side-effects, their effects being determined by their specifications. They can have a body, but in our framework we are interested by bodyless model methods. They are used to assume an effect over the program, and so state that if their requires are satisfied, the given effect (given by the ensure clause) will be satisfied. The model methods can be defined as invasive, because just like aspect they can have unexpected side effects on their environment.

```
ghost int i;
...
requires i > 0;
assignable i;
ensures i == 1;
model void m();
```

```
...
i++;
//@ m();
//@ assert(i == 1);
...
```

Fig. 3. Model method definition and use

Model methods call To be able to use model methods with the full expressivity desired, we add a new annotation to JML, the simple annotation method call. Its syntax would be simply the model method call in the middle of annotations as shown on Figure 3. It can be easily translated into guarded commands (like the one there [3]), but we will define it more precisely in Subsection 3.3.

2.2 Pipa

Pipa is a behavioural specification language based on JML. The specification of an advice is similar to the one of a method call. It has a pre- and post- condition, an exceptional postcondition and a frame condition. The specifications are not different from JML's method's specification, but they are used on advices. Pipa mainly adds two constructs specific to around advice.

Around advice specifications The difficulty of specifying an around advices is due to the presence of the proceed construct. It has to be stated in the method specification, and in JML no keywords exist to represent it. The solution commonly used was proposed by Clifton and Leavens [7]: we need a `proceed` and a `then` construct, to express what happens before and after the proceed call. Therefore the specification of an around advice is divided into two parts the part before the proceed (before the execution of the target instruction) and the part after the proceed. These two parts have both pre and post conditions as well as a frame condition. The first part has the construct proceed which take a boolean condition that determine if the instruction, target of the proceed, can be executed.

```
int f;
...
/*@   requires f > 0;
  @   ensures f == 1;
  @   proceeds true;
  @ then
  @   requires f == 2;
  @   ensures f == 3;
  @*/
void around() : call (void m()) {
     ...
}
```

Fig. 4. Around annotations

2.3 The annotated program

To continue with the Security Manager example, at this step we have to annotate the Security Manager and the base program. The Security Manager (Figure 5) has to be annotated, and especially its main advice are annotated. We specify that the advice modifies nothing, and that an exception will be thrown if the package name of the called method is `b`, and otherwise nothing will be done. To be able to use the field `s` in the specifications it is made `spec_public`. Please note that all the methods called in the specifications at this stage are side-effect free.

```
public aspect SecurityManager {
  //@ public invariant s != null;
  Set<String> /*@ spec_public @*/ s = new HashSet<String>();

  pointcut anyPublicMethod(Object o) : target(o) &&
    !within(SecurityManager) && call( public *(*))
  /*@ requires o != null && s != null;
    @ assignable \nothing;
    @ ensures s.contains(o.getClass().getPackage().toString())
    @         && !(o.getClass().getPackage().equals("b"));
    @ exsures (SecurityException) !s.contains("b")
    @                   && (o.getClass().getPackage().equals("b"));
    @*/
  before(Object o) : anyPublicMethod(o) {
    ...
  }
}
```

Fig. 5. The annotated Security Manager

The base program is annotated with normal JML specification (Figure 6). Though, if we want to manage to verify it, the specifications must take into account the aspect that will be weaved afterward. Therefore the specifications of the base program have to reflect the advice specifications. Here, an exception is thrown if there is a call on package `b` and this is the first try (otherwise the field `s` would already contain the package representation String).

```
/*@ assignable \nothing
  @ ensures s.contains("b");
  @ exsures !s.contains("b");
  @*/
public static void main(String[] args) {
   b.A a = new b.A();
}
```

Fig. 6. The annotated base program

2.4 Desugaring specifications

In the seminal paper about Pipa [26], the authors stated that annotations written with Pipa could be translated back to JML annotations, and they propose to weave Pipa annotations to JML annotations, being unclear how to do it. What we understand by translating Pipa annotation to JML annotations is removing most of the Pipa specific annotations. We turn the advices into methods so we can apply the behaviours desugaring described in Raghavan and Leavens paper [22].

proceed Proceed construct is transformed into the predicate `proceed(bool)` and conjuncted to the ensures specifications of the part before the `then` of the given case. If the proceed was omitted it is added in the ensures as `proceed(false)`.

then The `then` is the only Pipa construct that cannot be desugared into proper JML. It is used in the contract of around advices, and it modifies the semantic of the `proceed()` instruction in the code. Behaviours of around advices is desugared around the `then` construct: each part of each behaviour taking place before the `then` construct are desugared with each other as stated for standard JML behaviour in [7]. And the part after the `then` construct is given the same treatment.

3 Program transformations

3.1 Compilation to invasive model methods

The next step after the specifications are desugared is the compilation of aspects to their *effects* on the program. The approach presented here restricts itself to advices and fields introduction.

Aspects are first turned into model class, a specification only class. Fields are translated into their model versions. Advices are turned to model methods, which are bodyless. To respect the semantic of AspectJ, a static method `aspectOf()` has to be added. This method returns the current instance of the aspect. These transformations are direct, but the specifications of the advices have to be enriched with the pointcuts conditions that cannot be determined statically. In the weaving described in [14], the code of non static joint point are added to the program directly. Here we add this condition to the specification of the model method representing the advice, since we only manipulate the specifications and the signature of the method representing the advice.

Translation of the specifications The annotations are translated directly to the model methods. The method has the same signature of the corresponding advice, and the specifications are copied entirely to the model method. It works smoothly for most of the cases except for the around advice which is split into 2 model methods: the one corresponding to the first part, and the one corresponding to the part after the `then` construction. The `proceed` instruction has to be translated to JML as well: it becomes a global ghost variable of type bool. For an around advice the model method corresponding to the first part has to be modified consequently. The ensures has the equality `(proceed == proc_cond)` where `proc_cond` was the argument to the annotation `proceed`. The assignable clause is also modified: the `proceed` global variable is added to it.

Translation of the point cuts The pointcuts are not directly translated to the model methods, though the conditions over the pointcut which are not purely syntactical (the ones not taking only patterns as arguments) are added to the specification of the method. More precisely the algorithm is the following:

1. the point cuts are fully resolved and unfolded to simple boolean expressions (only a combination of ||, `&&`, or `!` with the point cuts specific keywords).
2. then the point cuts expression is ordered in two group: the pattern related (the one which take an argument of type pattern) and the others. Typically it will be separated by an `and` (`&&`), but it won't be always the case.
3. if the sepation between the pattern point cuts and the other point cuts is:
   (a) an `and`: the method `requires` clause is conjuncted with the expression of the non pattern point cuts and a new behaviour is added to the methods, where only the `requires` is precised and is the negation of the expression of the non pattern point cuts.

(b) a `or`: the method is duplicated, with one version with conjuncted to its `requires` clause the expression of the non pattern point cuts. This case should most likely be an error in the specification of the point cut.

On our example, the Security Manager aspect, the transformation is straightforward (Figure 7). The field is turned to an equivalent model field. The aspect invariant becomes a class invariant. The advice becomes a normal method. The pointcut was entirely static so the specifications do not have to be enriched with the dynamic elements of the pointcut. The method `aspectOf()` is added as well.

```
/*@ public model class SM {
  @   public invariant s != null;
  @   model Set<String> s;
  @
  @   requires o != null && s != null;
  @   assignable \nothing;
  @   ensures s.contains(o.getClass().getPackage().toString())
  @                    && !(o.getClass().getPackage().equals("b"));
  @   exsures (SecurityException) !s.contains("b")
  @                    && (o.getClass().getPackage().equals("b"));
  @   public model void beforeAnyPublicMethod (Object o);
  @
  @   assignable \nothing;
  @   ensures \result != null;
  @   public model static SM aspectOf();
  @ }
  @*/
```

Fig. 7. The model Security Manager

3.2 Compilation to bytecode

The only semantics that are properly defined for AspectJ are defined on Java bytecode [4, 14]. Therefore the only way to implement a verification which is correct against the semantic of AspectJ is by defining it on the bytecode level. The program as well as the annotations have to be compiled to Java bytecode and and bytecode annotations.

Compilation of source code The source code of the Java program can be compiled with `javac`. The source code of `before` and `after` advices does not have to be changed to be compiled to Java bytecode. The advice just have to be named with a unique method name. The around advices are more complex to compile as they contain the instruction `proceed()`. The proceed instruction is translated on bytecode as a method that does not need to have a body, and which is specific to a single around advice that is compiled.

Compilation of specifications We have chosen the Bytecode Modelling Language (BML) to annotate the bytecode. BML is a version of JML for bytecode, and there exists a simple transformation from fully desugared JML to BML as

presented M. Pavlova PhD. thesis [21]. Since our specifications have been turned into simple JML (in Subsection 2.4), we can translate all the JML annotations to BML's. The advices all keep their Pipa/JML specifications unchanged for the translation except in case of an `around` advice. The `ensures`, `proceeds` and `requires` that are the nearer to the `then` construct are removed from the `around` advice specifications. The `proceed()` specific method takes its specification from the one that were removed from the `around` advice. The `requires` of the `proceed()` equals the removed `ensures` and the ensures clause of the proceed() is the `proceed` clause implies the `requires` clause that were removed. The frame condition should be easily inferred from the advice's static joint point.

Following these rules, the Security Manager model is unmodified by this step.

3.3 Compilation to Guarded commands

To make the weaving lighter than the weaving done by `ajc`, the model of the aspect will be weaved. Therefore we need a form for the bytecode that can handle directly the weaving of the model. For this purpose we use a guarded command language. The language of choice is BoogiePL, because work has been done on the bytecode level for it. There is a formal definition of Java bytecode translation to BoogiePL [18] as well as existing implementations of the translation [19, 23]. The base program code is transformed, as well as the advices code.

Description of the language The version of the language we use is the same set of instruction described for weakest precondition on bytecode in [3]. The full language described in [10] is not necessary since our approach operates on bytecode. It has 5 commands: `assume`, `assert`, `assign`, `skip`, `havoc`, `goto` and label expressions. The first four commands are standard. `havoc` is used to assign an arbitrary to a variable, typically after a method call when a variable was modified by it. The `goto` command determines an unconditionnal jump to several labels.

BML inclusion The inclusion of BML annotations to BoogiePL has been described in [19], and is straightforward since BoogiePL has already assert and assume statements which are used in BML. BoogiePL also permits annotation of methods with requires, ensures as well as frame conditions. The two annotation constructs that are introduced for this paper and are not yet described are the model methods and the model method's calls and the aspect specific construct `proceed`.

The model methods are methods that are only defined with their specifications and on which we allow call within the specifications. Their effects on the program is only represented by their specifications. In BoogiePL one can generate procedure which does not contain implementation which would match exactly this behaviour. The model methods calls within specification would be replaced by a `call` command withing BoogiePL methods' body.

The proceed is a predicate that can be called at any time in the program. In BoogiePL it is easily represented by a variable `proceed` of type `bool`.

Translating the Security Manager For the Security Manager, the guarded command translation is easy, but the number of lines grow (5 lines at the byte-code version and 27 for guarded command version) as shown for the base program on Figure 8. Nevertheless the aspects would have to be transformed into guarded commands too. The model Security Manager is kept unchanged.

```
init:
   // initialization of the method
   old_heap := heap;
   reg0 := #0;
   assume requires(main, (old_heap, #0)); // true
start:
   // beginning of the program
   heap := add(heap, b.A); // new call
   stack[0] := new(heap, b.A);
   arg0 := stack[0]; // constructor call
   pre_heap := heap;
   assert arg0 != null;
   assert requires(b.A.<init>, pre_heap, arg0);
   havoc heap;
   goto b.A.<init>_normal, b.A.<init>_excp;
b.A.<init>_excp:
   havoc stack[0]
   assume alloc (stack[0], heap) ∧ typeof(stack[0]) <: Throwable;
   assume exsures(b.A.<init>, (pre_heap, arg0), (heap, stack[0]));
   goto handler;
b.A.<init>_normal:
   havoc stack[0];
   assume ensures(b.A.<init>, (pre_heap, arg0), (heap, stack[0]));
post: // post condition and exception handler
   assert ensures(main, (old_heap, #0), (heap, stack[0]));
                                   // s.contains("b")
   goto;
handler:
   assert exsures(main, (old_heap, #0), (heap, stack[0]));
                                   // !s.contains("b");
   goto;
```

Fig. 8. The guarded command version of the base program

4 Verification process

4.1 Weaving of the models

At this stage of the transformation, all the program source is turned into guarded commands. The complete code is composed of the guarded command version of the base program, the guarded command version of the advices, and the model abstraction of the aspects. What has to be done at this stage is the weaving of the models to the normal code.

The model abstraction is made of:

1. methods which represents the behaviour of before and after advices, model methods that represent the first part of around advices, model that represent the second part and,
2. methods representing the advices, and for around advices, proceeds model methods.

The weaving consists in adding these methods call in the guarded commands program. The methods are inserted at the same points as they would on bytecode. The semantic of the weaving we base our work upon has been presented in two papers [14, 4]. We use the rules which were defined in [4]. Since we operate on the guarded command version of the program, we need a correspondence between the bytecode program and the guarded command program. This will be assured by a correspondence table. This table has been computed during the translation from bytecode to guarded commands in the previous stage.

BEFORE

$$\frac{\begin{array}{c}\texttt{isShadow}(m, pc) \wedge ads \neq [] \\ \wedge\ \texttt{matchPcut}(\varepsilon, head(ads).pointcut, m, pc) \wedge \texttt{head}(ads).\texttt{kind} = \texttt{Before} \\ \wedge\ (\varepsilon', gcs') = \texttt{insertBeforeAdvice}(\varepsilon, m, pc, head(ads), gcs)\end{array}}{\langle \varepsilon, m, pc, ads, nextpc, gcs\rangle \rightarrow \langle \varepsilon', m, pc, tail(ads), nextpc, gcs'\rangle}$$

AFTER

$$\frac{\begin{array}{c}\texttt{isShadow}(m, pc) \wedge ads \neq [] \\ \wedge\ \texttt{matchPcut}(\varepsilon, head(ads).pointcut, m, pc) \wedge \texttt{head}(ads).\texttt{kind} = \texttt{After} \\ \wedge\ (\varepsilon', gcs') = \texttt{insertAfterAdvice}(\varepsilon, m, pc, head(ads), nextpc, gcs)\end{array}}{\langle \varepsilon, m, pc, ads, nextpc, gcs\rangle \rightarrow \langle \varepsilon', m, pc, tail(ads), nextpc, gcs'\rangle}$$

Fig. 9. The two weaving rules which are modified

The weaving we use is the one presented in [4]. We use the four rules defined in the paper, which takes as an entry an environment ε, the current method m, the code pointer in the method pc, the list of advices ads which is composed of the advices name with their syntactic pointcuts and $nextpc$, a pointer to the next instruction in the program. We add a new parameter, gcs, a tuple containing the guarded command program and the bytecode/guarded commands correspondence table (Figure 9). The main difference in the algorithm is for the BEFORE and AFTER rules, where gcs is modified and not the code. The the `insertBeforeAdvice` and `insertAfterAdvice` are modified in order to add the guarded commands translation of a simple static method call to the method `aspectOf()`, followed by a call to the advice model method. In the case of the SecurityManager, we only have to weave a before advice, as shown in Figure 10.

The around advice weaving is not properly treated in [4] or in [14]. In our framework the around advice was compiled in two methods. So we derive from the later a generic weaving method, adapted to the way they were specified. The key idea is to weave the around-before method like a normal around advice, and

add two guarded command at the end of the weaved method, a non deterministic goto that goes to the begining of the pointcut or after the pointcut, and at the beginning of the pointcut add the assertion `proceed = true`.

```
start:
   ...
   stack[0] := new(heap, b.A);
   arg0 := stack[0]; // advice method call
   pre_heap := heap;
   assert arg0 != null;
   assert requires(SM.beforeAnyMethod, pre_heap, arg0);
   havoc heap;
   goto SM.beforeAnyPublicMethod_normal, SM.beforeAnyMethod_excp;
SM.beforeAnyMethod_excp:
   havoc stack[0]
   assume alloc (stack[0], heap) ∧ typeof(stack[0]) <: Throwable;
   assume exsures(SM.beforeAnyMethod,
                  (pre_heap, arg0), (heap, stack[0]));
   goto handler;
SM.beforeAnyMethod_normal:
   havoc stack[0];
   assume ensures(SM.beforeAnyMethod,
                  (pre_heap, arg0), (heap, stack[0]));
   arg0 := stack[0]; // constructor call
   ...
```

Fig. 10. The weaved guarded command base program

4.2 The weakest precondition calculus

After all the previous steps, we have a guarded command representation of the weaved base program, a guarded command representation of the weaved advices, and a model representing the advices. What is left to do is to check the base program with a weakest precondition calculus, check the weaved advices, and check the weaved advices against the model. The calculus that we use is the one presented in [3]. For the base program which we showed before we obtain the following verification condition presented in Figure 11.

5 Conclusion

In this paper we have described a verification framework for taking into account meta-programming concern in the verification. These concerns were modelled using AspectJ aspects specified with Pipa. The verification was made using a standard guarded command language, and an abstraction of aspects as model classes. The concerns were by choice invasive, so no real modular verification was possible.

A great part of the framework is meant to exist as part of the Mobius PVE [9], like the translation of annotated bytecode to BoogiePL [19], or the weakest precondition calculus over a BoogiePL-like language. But there is lots of implementation to be done still: JML should be extended with the Pipa constructs (to

```
∀ heap, requires(main) →
   (new(heap, b.A) != null) ∧
   ((new(heap, b.A) != null) →
       requires(SM.beforeAnyMethod) ∧
       (requires(SM.beforeAnyMethod)
        ∀ heap,
          ∀ stack[0], alloc (stack[0], heap) ∧
          typeof(stack[0]) <: Throwable →
          exsures(SM.beforeAnyMethod) → !s.contains("b")
        ∨
        ∀stack[0], ensures(SM.beforeAnyMethod) →
         arg0 != null ∧
         (arg0 != null → requires(b.A.<init>) ∧
          (requires(b.A.<init>) →
          ∀ heap, (∀ stack[0], alloc (stack[0], heap)
            ∧ typeof(stack[0]) <: Throwable →
            exsures(b.A.<init>) → !s.contains("b")) ∨
            (∀ stack[0], ensures(b.A.<init>)→ s.contains("b")))))
```

Fig. 11. The verification condition generated for the base program

our knowledge no implementation of Pipa exists), the abstraction from advices to models has to be implemented, and the weaving too.

An interesting difficulty arises when using this framework: when a user first specify the base program, his specifications are not complete enough to enable verification. There should be a way to enrich automatically these specifications, maybe by using specification weaving like what is presented in Moxa [25].

References

1. Pavel Avgustinov, Elnar Hajiyev, Neil Ongkingco, Oege de Moor, Damien Sereni, Julian Tibble, and Mathieu Verbaere. Semantics of static pointcuts in aspectj. In *POPL*, pages 11–23, 2007.
2. M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K.R.M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *FMCO*, 2005.
3. M. Barnett and K.R.M. Leino. Weakest-precondition of unstructured programs. In *Program Analysis For Software Tools and Engineering*, pages 82–87, New York, NY, USA, 2005. Association of Computing Machinery Press.
4. Nadia Belblidia and Mourad Debbabi. Formalizing aspectj weaving for static pointcuts. In *SEFM '06: Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, pages 50–59, Washington, DC, USA, 2006. IEEE Computer Society.
5. C. Clifton. *A design discipline and language features for modular reasoning in aspect-oriented programs*. PhD thesis, Iowa State University, 2005.
6. C. Clifton and G. Leavens. Observers and assistants: A proposal for modular aspect-oriented reasoning, 2002.
7. C. Clifton and G. Leavens. Spectators and assistants: Enabling modular aspect-oriented reasoning, 2002.

8. D. Cok and J.R. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Construction and Analysis of Safe, Secure and Interoperable Smart Devices: Proceedings of the International Workshop CASSIS 2004*, volume 3362 of *Lecture Notes in Computer Science*, pages 108–128. Springer-Verlag, 2005.
9. Mobius Consortium. Mobius Program Verification Environment. A beta version is available online from http://mobius.inria.fr, 2007.
10. R. DeLine and K.R.M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.
11. C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Programming Languages Design and Implementation*, volume 37, pages 234–245, June 2002.
12. C. Flanagan and J.B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Principles of Programming Languages*, pages 193–205, New York, NY, USA, 2001. Association of Computing Machinery Press.
13. Pieter H. Hartel and Luc Moreau. Formalizing the safety of java, the java virtual machine, and java card. *ACM Comput. Surv.*, 33(4):517–558, 2001.
14. Erik Hilsdale and Jim Hugunin. Advice weaving in aspectj. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 26–35, New York, NY, USA, 2004. ACM Press.
15. Shriram Krishnamurthi, Kathi Fisler, and Michael Greenberg. Verifying aspect advice modularly. *SIGSOFT Softw. Eng. Notes*, 29(6):137–146, 2004.
16. Cesar Kunz. Modular verification and certificate translation for advice weaving.
17. Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David R. Cok, Peter Müller, Joseph Kiniry, and Patrice Chalin. JML Reference Manual. Department of Computer Science, Iowa State University. Available from http://www.jmlspecs.org, February 2007.
18. H. Lehner and P. Müller. Formal Translation of Bytecode into BoogiePL. In M. Huisman and F. Spoto, editors, *Bytecode Semantics, Verification, Analysis and Transformation*, Electronic Notes in Theoretical Computer Science, 2007.
19. Ovidio Mallo. A translator from bml annotated java bytecode to boogiepl. Master's thesis, ETH Zurich, 2007.
20. C. Marché, C. Paulin-Mohring, and X. Urbain. The Krakatoa tool for certification of Java/JavaCard programs annotated with JML annotations. *Journal of Logic and Algebraic Programming*, 58:89–106, 2004.
21. Mariela Pavlova. *Java bytecode verification and its applications.* Thése de doctorat, spécialité informatique, Université Nice Sophia Antipolis, France, January 2007.
22. A. D. Raghavan and G. T. Leavens. Desugaring JML method specifications. Technical Report TR #00-03e, Department of Computer Science, Iowa State University, 2000. Current revision from May 2005.
23. Alex Suzuki. Translating Java Bytecode to BoogiePL. Master's thesis, ETH Zurich, 2006.
24. Yi Wang and Jianjun Zhao. Specifying pointcuts in aspectj. In *3rd Asian Workshop on Aspect-Oriented Software Development (AOAsia 2007)*, Beijing, China, 2007.
25. Kiyoshi Yamada and Takuo Watanabe. Moxa: An aspect-oriented approach to modular behavioral specifications. 2005.
26. Jianjun Zhao and Martin C. Rinard. Pipa: A behavioral interface specification language for aspectj. In Mauro Pezzè, editor, *FASE*, volume 2621 of *Lecture Notes in Computer Science*, pages 150–165. Springer, 2003.