Extending supported language subset of Jive

David Steiger

Semester Project Report

Software Component Technology Group

Department of Computer Science ETH Zurich

http://sct.inf.ethz.ch/projects

SS07

Abstract

Jive, the Java Interactive Verification Environment, is a verification tool for JML annotated programs written in Diet Java Card, a subset of Java. The research groups working with JML defined a hierarchy of several language levels, which divide the different features according to their importance. One part of this report describes the extension of Jive with assume and assert statements, which brings it one step further to JML level 0 coverage. Another big part of this project was the definition and implementation of Hoare-rules for non-default constructors.

Contents

1. Introduction

1.1 Verification

Nowadays, as programs tend to get larger and more complex, the importance of correctness increases for programmers. One often applied method to find bugs in programs is testing, but as testing every possible input to a program usually would mean the need of practically infinitely much time, you can never be sure in the absence of bugs in a program only by testing it. The way to prove that programs are correct is verification, and there are several groups doing research in this field of computer science.

1.2 Specification

If a program is attempted to be verified, it is always necessary to have very clear requirements, because else it can not be told if the program behaves like intended. A good practice to get clear requirements is to formalize them and write them down as specifications. As verification is a process based on logic, the specification should either be provided as logic formulas or be transformed into formulas. As writing specifications as logic formulas is very difficult, especially if the writer is not familiar with it, the developers of Jive decided to use the second approach. This means that the programmer working with Jive can write the specification in a language with very similar syntax to usual programming languages. The difficult task of transforming the specifications into logic formulas is then automatically done by Jive.

1.3 Jive

Jive, the Java Interactive Verification Environment, is a cooperative work of ETH Zurich and TU Kaiserslautern. Jive is a verification tool for object-oriented programs based on Hoare logic. In the current implementation, Jive expects as input programs written in Java-KEx [1], a subset of sequential Java which provides all important object-oriented features

like inheritance and dynamic method binding. The specifications in the programs have to be annotations written in JML [2].

1.4 JML

JML, the Java Modeling Language, is a specification language with similar syntax as Java, providing additionally specification-specific expressions and statements. Most features and keywords are designed to be used in preconditions and postconditions of a method and in invariants. There are, however, some constructs which appear inside the body of a method implementation. These are the JML statements.

The research groups working on JML defined a hierarchy of several language levels. The level 0 features should be supported by all tools using JML. Jive already supports most of the JML level 0 features, but for full support it still needs to be extended.

1.5 Project Task

The first task in this project is to extend Java-KEx with two JML level 0 features, namely the JML statements assert and assume. The second task is to implement non-default constructors, as this is a better solution then the init methods used previously in Jive.

2. Jive in detail

In formal verification, the goal is to provide a formal proof on an abstract mathematical model of the program which is attempted to be verified. For Jive, the used model is Hoare logic, which in the context of formal semantics of programming languages belongs to axiomatic semantics.

2.1 Hoare Logic

The central element in Hoare logic is the so called Hoare triple. It consists of a precondition P, a command C, and a postcondition Q. The usual notation of a Hoare triple is like this:

$$\{P\}C\{Q\}$$

The command C represents a concrete piece of code. It can be a single program statement, but also a sequence of statements or a whole method. The precondition P is expected to be provided in the form of a logical formula, which the user of Hoare logic assumes to hold before execution of the command. The postcondition Q must be provided as a logical formula, too. It contains the assertions which have to hold after execution of the command.

If a verification tool like Jive can prove a Hoare triple, it means that for every program execution, such that the precondition is fulfilled and the command terminates, the postcondition is guaranteed to be fulfilled afterwards, too. In the Jive environment the command is also known as component reference (CompRef).

The fact, that the command (which can be a method with possible recursive method calls inside) terminates, might be difficult to prove. Thus termination of the command is often not required to be proven, this is known as partial correctness in computer science. Jive uses partial correctness Hoare logic. If termination must be proven, then we talk about total correctness verification.

To provide a formal proof as mentioned at the beginning of this chapter, a verification tool like Jive usually tries to derive logical formulas which can be sent to theorem provers. In Jive, the formulas are created during the construction of so called proof trees. In order to generate the proof trees out of given Hoare triples, several Hoare rules and axioms have to

be defined. They give the semantics to the different supported constructs of the programming language, which in case of Jive corresponds to a subset of Java. Hoare rules describe a valid transformation from one Hoare triple (or a set of Hoare triples) to another Hoare triple (or a set of Hoare triples) and can be applied in both directions. Hoare axioms define triples which are valid assertions. This means that the triples which match to axioms are assumed to be true by default and need no further processing. Thus Hoare axioms can be used as a stopping point in the resolution process of the construction of a proof tree. All the nodes of such a proof tree are Hoare triples. The root node is in Jive the Hoare triple of a method, the branches between the nodes correspond to the used Hoare rules, and the leaves of a complete proof tree are Hoare triples which match to axioms. Note that as long as the proof tree is not complete, there exist so called open leaves, these are Hoare triples that could not be matched to axioms so far.

In Jive, Hoare rules and axioms not always consist only of Hoare triples, but sometimes also contain first oder logic formulas which are required to be proven. These formulas are also called verification conditions or lemmas. They can either be sent directly to the automatic theorem prover Simplify or interactively be proven using the theorem prover Isabelle.

2.2 Functionality of Jive

Originally the development of Jive was divided into two separated parts, the group at ETH Zurich was developing the front end and the back end was assigned to the group at TU Kaiserslautern. For some time the two parts were also physically separated, this means the code was divided into two different folders and the parts were dependent on different versions of the Java compiler. Lately the two parts were merged, so that the whole project can be compiled by one compiler now.

2.2.1 Jive Front End

As mentioned earlier, the front end expects as input a program written in Java-KEx, annotated with JML specifications. The first step is to parse and type check the input, this is done using existing tools, namely the JML compiler and the Multijava compiler. The most important part of the result is an abstract syntax tree, which contains all the instructions of the program. It is stored in a data structure for further use.

The properties which the programmer defines and wants to hold, written as JML annotations, are converted into so called proof obligations in Jive. This is done in the proof obligation generator. The whole abstract syntax tree from the previous step is visited there and proof obligations in the form of Hoare triples are created out of the JML constructs. The proof obligations are then sent to the back end.

Another task performed by the front end is the generation of program-dependent theories

for the theorem provers. The theories contain context information about declared fields, types, subtype relationships etc.

There exist also program-independent theories, they contain information about the store model and the semantics of the DJC language which are the same for every valid DJC program, and thus need not be constructed every time Jive is run. All theories, be it program-dependent or -independent, are stored in files on the hard disk and thus can be inspected.

The whole processing of the front end activities is done automatically when Jive is started up. The first user interaction is possible when the GUI is available, which is constructed in the back end.

2.2.2 Jive Back End

The actual verification happens in the back end of Jive. As mentioned earlier, the goal is to transform the incoming Hoare triples into proof trees containing the verification conditions which remain to be proven.

Note that it is even theoretically impossible for the theorem provers to prove every correct program and disprove every incorrect program at the same time, because it is an undecidability problem. This means that we can be sure, that a proven method is really correct, but if the theorem prover fails to provide a proof, we do not know if the method is correct or incorrect, it could be either case.

The design decision was to make the verification interactive, this means that the user can specify which methods should be proven, and which Hoare rules respectively axioms should be applied in order to generate the proof tree. Thus the user can help Jive to prove a program correct step by step, although this is a rather tedious work and not very attractive. So there is also an automation of this process provided to the user, it is called tactic in Jive.

The starting point for a tactic in Jive is always an unproven Hoare triple, which corresponds to an open proof obligation. The precondition of the proof obligation was either generated in the front end or in a previous step by another tactic or the user in the back end. Similarly the command of the Hoare triple and the postcondition were generated, that is, either in the front end or in a previous step in the back end.

A tactic is then expected to apply Hoare rules and axioms in a reasonable way, aiming to traverse all statements and converting all Hoare triples with the Hoare rules such that they match to predefined axioms in the end. If a tactic finishes successfully, then only verification conditions remain to be proven. If it finishes with open Hoare triples, then for the proof these triples need further processing either by another tactic or by the user.

The probably most used and currently best supported tactic is practical weakest precondition (pwp). It works bottom up, this means that it first takes the last statement of the method. It calculates the weakest precondition that has to hold before the statement,

such that the postcondition is fulfilled after execution of the statement. This precondition is then the postcondition in the next step for the last statement but one, and iterating in this way the pwp tactic proceeds along until all statements of the method have been traversed.

After the pwp tactic traversed the topmost statement, it worked out the weakest precondition which has to hold, such that after execution of all statements of the method the postcondition is fulfilled. If we name this resulting precondition WP and the original precondition of the method P, then what remains for the pwp tactic is to generate a lemma requiring that WP can be derived from P. This is simply formalized as an implication from P to WP.

2.2.3 Abstract Syntax Trees (AST)

Concerning implementation, the main data structure used in Jive are trees. As mentioned earlier, already in the first step of the front end the input of Jive is translated into an abstract syntax tree by the JML compiler. As Jive only supports a small subset of JML and Java is supported, this usually huge tree is then transformed into a much smaller AST which contains only the supported language constructs. They must be previously defined in separate classes in Jive. If there appear unsupported language constructs in the program code, either a warning or an error message is sent to the console, depending on the severity that it would mean for the verification process.

Because of historical reasons mentioned at the beginning of this chapter, the back end only works with a different kind of AST and would not understand the AST created in the front end. So the front end AST is again converted into a new AST. Maybe the conversion will no longer be necessary in the future, when the two ASTs can be merged. The module in the front end which performs the above mentioned conversion from the front end AST to the back end AST is called program converter. The input of the conversion is the front end AST, or more precisely the sequence of objects corresponding to a traversal of the nodes in the AST. Each of these objects was created from the corresponding class of the language construct it represents. The output of the conversion is again a sequence of objects corresponding to a tree traversal, this time the objects are instances of classes that were previously generated by the Katja tool.

Katja was mainly implemented to make the handling with formal grammars easier. It allows the programmer to define a formal grammar, like the one that describes our supported language subset in Jive, in a short and readable way, namely written in a file with Katja syntax. The tool then automatically generates Java types for the symbols in the grammar, including subtype relationships. This makes changes to the grammar a lot easier, because the programmer does not need to work on whole Java classes, but can work directly on the strings of the symbols in the grammar.

3. Implementation of JML statements

3.1 JML statements

As mentioned earlier, the special thing about JML statements is, that they appear inside the body of a method implementation. The JML reference manual lists seven types of JML statements [3], but only two of them will be discussed here, namely assert and assume statements. They are the first JML statements to be implemented in Jive. Note that when semantics of such statements are discussed, one should always be aware if runtime behavior is looked at, or if the focus is on static verification. During the static verification process, the program is not executed. The goal is to verify certain properties statically. There is a difference between runtime semantics and static verification semantics.

3.1.1 Assert

The reason why it is preferable to have an assert statement is that it makes debugging a lot easier. With the assert statement it can be tested if specific variables have the expected values at a certain point in the program execution. Thus it is possible to discover bugs in an early phase, instead of having a strange program behavior because of bugs in a later phase, when it would be difficult to detect the reason for the behavior.

The runtime semantics of a JML assert statement is practically the same as of the assert statement introduced in Java 1.4. The Java version was not implemented in Jive, because the used Multijava compiler does not support assert statements and the additional effort to change it would have been beyond the goal of a semester thesis.

An assert statement expects as a first argument a boolean expression. The difference between the Java version and the JML version is, that in the Java assert statement the expression can have side effects, while for the JML assert statement the expression is a so called JML predicate. A predicate is a side-effect free expression which is allowed to use the various JML extensions. It is usually expected that the predicate of an assert statement evaluates to true. In this case nothing happens at runtime, that is, the assert statement is equivalent to an empty statement. If the predicate evaluates to false, the program throws a JmlAssertError (org.jmlspecs.jmlrac.runtime.JmlAssertError). JmlAssertError is a subclass of the Java class Error. It prints an error message to the console. The message can

optionally be substituted by providing a different message as the second argument of an assert statement. If this functionality is used, the second argument of an assert statement must be a String separated by a colon from the predicate. The just mentioned runtime behavior is of minor importance in the Jive environment, because the focus is on static verification.

For static verification of an assert statement, an intuitive goal is to verify that the predicate evaluates to true. This is also the approach that was implemented in Jive. Note however, that as JmlAssertError is indirectly a subclass of Throwable, it would be possible to catch the error (e.g. with a try-catch-statement) and recover from it, resulting in a correct program even when the predicate evaluates to false. Sometimes it is reasonable to have a semantics with exceptions. A similar approach could be used for the assert statement. There are however two counter-arguments to this approach, which convinced us not to implement a semantics with exceptions.

First, programmers usually use the assert statement for debugging, and for debugging it is important to recognize erroneous predicates. So it would make no sense to catch a JmlAssertError. Also the Java API states that a reasonable application should not try to catch an Error [4]. If the JmlAssertErrors in a program are supposed not to be caught, it is also useless to have a semantics with exceptions.

Second, in static verification we want to avoid introducing exceptions when it is possible, as this only complicates verification. It is also easier to implement a semantics without exceptions.

Thus our design decision in Jive was to verify that the predicate of the assert statement evaluates to true and to falsify the program proof when we reach a predicate evaluating to false. This approach was also used in other program verifiers, for example Boogie [5].

3.1.2 Assume

Unlike the assert statement, there exists no counterpart for the assume statement in Java. Syntactically an assume statement is very similar to an assert statement. It also has a JML predicate as first argument, optionally followed by a colon and a String. Like for the assert statement, at runtime an assume statement is equivalent to an empty statement if the predicate evaluates to true. When the predicate of an assume statement evaluates to false, a JmlAssumeError (org.jmlspecs.jmlrac.runtime.JmlAssumeError) is thrown. This behavior can not be seen in Jive, because as mentioned at the beginning of chapter 3, in static verification the program is not executed.

With the assume statement, the programmer can specify properties of which he knows that they are true. Thus he can help the theorem prover. For example if there is a boolean expression which the theorem prover is not able to prove, but the programmer knows without any doubt that it is correct, then he can use the expression as the predicate of an assume statement. The consequence is that the theorem prover no longer tries to prove or

disprove in case the expression evaluates to false, but only deals with the case when the expression evaluates to true. Thus the theorem prover might with the provided help of assume statements be able to prove a program, which it was not able to prove without this help. Another benefit is that introducing assume statements can accelerate the verification process, because we practically cut the path of the expression evaluating to false off the prove tree.

Sometimes it might even be easier for the programmer to specify a property, which he knows to hold, directly as the argument of an assumption. But this programming style is very risky and should not frequently be used, we will discuss the reason below.

The predicate of an assume statement might be impossible to prove, at least for a certain theorem prover. Note that if the predicate could be proven, then the assert statement always would be the better choice instead of the assume statement. Thus verification of the predicate is wanted to be avoided, and the implementation of the assume statement in Jive followed this reasoning. Similarly as we have seen for assert statements, the design decision in Jive was not to use a semantics with exceptions for the assume statement. If the predicate of an assume statement evaluates to false, then the programmer can notice this wrong assumption during runtime, when the error message is printed. The program code following a wrong assumption is in Jive always treated to be correct regardless of its content.

It is important to understand the consequences of this approach: an assume statement becomes a possible source of unsoundness. By introducing a wrong assumption anything can be proven. Thus a theorem prover could declare a program to be correct even when it is incorrect! Unsound verification is useless, because users must be able to rely on the output of the theorem prover. Thus it is strongly recommended to programmers to use the assume statement very carefully, because introduced assumptions have to be correct.

However the assume statement can be of great help in verification if used reasonably. It was decided that the benefits of the assume statement outbalance the risk of unsoundness discussed, and it was implemented in Jive. The responsibility to provide correct assumptions is intentionally passed from Jive to the programmer.

3.2 Hoare Axioms

This section shows the concrete formal Hoare axioms that were used for the implementation of assert and assume statements and provides some explanation. As discussed above, the design decision was not to use semantics with exceptions for assert and assume statements, this allowed to formulate Hoare axioms which are much easier to implement and use.

3.2.1 Assert Axiom

The used Hoare axiom for the assert statement is:

$$\{\sigma(\mathrm{E}) \wedge \mathbf{P}\} \ \text{assert E} \ \{\mathbf{P}\}$$

The σ is a translation function used to transform specification expressions to first order logical formulas (see also [6]). The axiom means, that if the transformed predicate E evaluates to true and the precondition P holds, then P will also hold in the post state, that is, the program state after execution of the assert statement.

From the point of view of the pwp tactic which works bottom up, the weakest precondition that has to hold, such that P can be proven to hold in the postcondition, is the conjunction of the value of E and P. In case the predicate evaluates to false, the precondition of the axiom which is $\{\sigma(\mathrm{E}) \wedge \mathrm{P}\}$ evaluates to false regardless of the value of P. Thus as expected in this case a theorem prover will either provide a disprove to the program or state that the program could not be proven, for example when the theorem prover couldn't evaluate the expression E.

3.2.2 Assume Axiom

The used Hoare axiom for the assume statement is:

$$\{\sigma(\mathrm{E}) \Rightarrow \mathbf{P}\} \ \text{assume E} \ \{\mathbf{P}\}$$

It might have been more intuitive to use the expression E in the postcondition, resulting in the formula $\{\mathbf{P}\}$ assume E $\{\sigma(\mathrm{E}) \wedge \mathbf{P}\}$. But the design decision was to use the first version. As the pwp tactic works bottom up, for the second version it would have been necessary to match a formula generated in a previous step to the postcondition $\{\sigma(\mathrm{E}) \wedge \mathbf{P}\}$, which is more complicate than to add an implication from the value of E like in the used axiom. So it was much easier to implement.

The semantics of the axiom is, that if the transformed predicate E implies the precondition P, then P will also hold in the the program state after execution of the assume statement. Implication means that if the predicate E is true, then P must also be true to make the whole precondition of the axiom true. Note that in case of E evaluating to false the precondition of the axiom becomes trivially true regardless of the value of P, as discussed earlier this is a possible source of unsoundness, because in this case any formula P could be proven.

From the point of view of the pwp tactic, the weakest precondition that has to hold, such that P can be proven to hold in the postcondition, is the implication from the value of E to P.

3.3. Implementation of JML Statements

As described earlier, the Jive tool contains two parts which where originally separated, the front end and the back end. For the JML assert and assume statements, we will first have a look at the front end implementation.

3.3.1 Implementation in the Front End

As was mentioned already in chapter 2.2.3, one functionality of the Jive front end is to transform the incoming abstract syntax tree from JML into another AST containing only the supported language constructs. The resulting AST is constructed by instantiating predefined classes. So when implementing assert and assume statements, the first task is to provide such classes for both statements. The implemented code can be found in the files jive/frontend/program/AssertStmt.java, resp. jive/frontend/program/AssumeStmt.java.

Interface of AssertStmt:

```
public class AssertStmt extends Statement {
        AssertStmt(stmt, impl, occurrence, parent)
        public Formula predicate()
        public String toString()
}
```

Like all statements, the assert statement inherits from the common super class Statement which is located in the same folder. The constructor is also standard, note however that only JML assert statements are supported. If in future an extension for Java assert statements should be implemented, a simple solution would be to just add another constructor with different signature, that is, stmt would be of type JAssertStatement instead of JmlAssertStatement.

Method predicate() transforms the JML predicate of the assert statement to a Formula object. The class Formula is generated by the Katja tool. It is Jives internal representation of first order logic formulas, as used in pre- and postconditions of Hoare triples.

Before the transformation starts we first set a flag, with the effect that the variables in the predicate are translated to so called ProgVar's instead of LogicalVar's. JML treats its bound variables (for example introduced by a \forall expression) like local variables. For specifications outside of a method body this causes no problems, because the local variables of the method are out of scope of the specification. But with the introduction of JML statements like assert and assume, the variables could also be local variables. The same problem already appeared for loop invariants, so it was natural to use the same flag, we renamed it to isInLoopInvOrSpecStmt to make its extended use obvious.

After the before mentioned flag is set, the actual transformation is delegated to JmlExpressionTransformer, which is a module for exactly this purpose of transforming JML expressions to Formulas. Finally, the flag is reset.

Method toString() is only for Jive internal use, it just prints the assert statement to the console.

As the syntax of assert and assume statements is very similar, also the code in class AssumeStmt is very similar to AssertStmt and does not need to be discussed.

Coming back to the transformation from the JML AST to the front end AST, for statements this transformation is done in Statement.getStatement(stmt, impl, occurrence, parent). We first find out which type the statement to be transformed belongs to by simple type matching, then we create a new object of the appropriate type. We do not want to have the exactly same statement twice, so we call method getUnique of the implementation impl to assign the appropriate reference if the statement already existed. For assert and assume statements this is implemented analogously as for other statements.

Now we have discussed everything to enable Jive to build the front end AST including assert and assume statements. As mentioned earlier, in the current Jive version this front end AST must be converted into another AST to be able to use it in the back end.

For this conversion we first have to modify the file jive/container/intf/PVCCompRefs.katja. We simply add the symbols PAssertStmt and PAssumeStmt in the PStatement block and define the signature of these symbols below. The Katja tool generates then the needed classes in Jive automatically (the result can be seen in folder jive/container/intf/pcomprefs).

The actual conversion from front end AST to back end AST is done in the ProgramConverter (jive/frontend/program/ProgramConverter.java). We add the cases for assert and assume statements in method getPStatement(stmt). PAssertStmt and PAssumeStmt expect the predicate as an argument of type Formula, to provide it we call our previously implemented method predicate() of AssertStmt and AssumeStmt objects.

There remains one last problem which arised during testing. Previously it was not possible that JML constructs appeared inside of a Java method body. With the introduction of JML statements this became possible. The problem is that for JML old constructs some logical variables and formulas need to be created, and this process is finished after proof obligation generation which happens before AST conversion. So when assert and assume statements are processed it is unfortunately too late to handle JML old constructs only with existing code.

Our solution is to add additional code in the proof obligation generator (jive/frontend/pog/ProofObligationGenerator.java) to detect JML statements using \old constructs already at this point of execution and thus be able to create the required variables and formulas using existing functionalism. The code can be found in method scanForOldInJavaCode in the proof obligation generator.

For each method in the program we test, if it has a method body containing a list of statements (else no JML statement can occur). Then follows an iteration through the statements and if an assert or assume statement is found, the formula of the predicate is created. This is done because during the creation of the formula, in case of a JML old

construct occurring in the predicate, the needed logical variables and formulas for the old program state will be created too. At the end we can call the already existing method generatePreCondSaveOld() to save the states for JML old constructs, and are finished with fixing the problem.

3.3.2 Implementation in the Back End

In Jive, the actual Hoare rules and axioms are implemented in the back end. For axioms usually two classes are implemented, a check class and an instantiation class. Both classes inherit from the abstract base class for all Hoare rules and axioms named Rule.java. Thus as mentioned in the description of class Rule, they have to override the call method and also should override the parametersOk method. Of course this also applies to implementations for assert and assume axioms. The check and instantiation classes same as their common super class Rule can be found in folder jive/container/impl. The implementation for JML assert statements is in files KEx_check_assert_axiom.java and KEx_inst_assert_axiom.java, for the names of the files implementing the assume axiom just replace “assert” by “assume”.

Interface of KEx_check_assert_axiom:

```
public class KEx_check_assert_axiom extends Rule {
        KEx_check_assert_axiom(c)
        boolean parametersOk(ptn)
        private void checkParameters(ptn)
        ProofTreeNodeIt call(ptn)
}
```

As its name suggests, the check class of an axiom is used to perform syntactical checks on an open proof tree node. It checks whether the proof tree node contains a Hoare triple matching the axiom.

Method checkParameters first checks if the proof tree node contains a program part which is an assert statement by calling the general method for this functionality named checkParametersBackward of the super class Rule.java. Then it makes sure that the proof tree node contains no assumptions and that the precondition is a conjunction like it was shown in the definition of the assert axiom in section 3.2.1. If a check fails an error message is sent to the GUI.

For more detailed checks the method call exists. It first calls checkParameters and thus also runs the before mentioned checks. Then follow two checks of the predicate (named E in section 3.2.1). First it is checked to be a correctly typed Formula and second if the E in the precondition is the same as the E in the component reference (middle part) of the Hoare triple. There remains a last check to be done, which is the comparison of the P in the precondition to the P in the postcondition. If all checks finish successfully, we think the

Hoare triple matches correctly the axiom and thus we close the proof tree node and return it.

The instantiation class of the assert axiom is used to create a Hoare triple when given the postcondition and the component reference of the triple. This time method checkParameters only checks if the supplied postcondition is not null and if the component reference is an assert statement. The precondition obviously can not be checked like in the check class, because it is not available.

Method call in the instantiation class starts similarly as in the check class, it first runs checkParameters and then checks if the predicate is a correctly typed Formula. The precondition is then created by constructing a conjunction of the predicate E and the postcondition P. Then the Hoare triple existing of the new precondition, the given component reference and the postcondition is constructed. Finally the sequent containing an empty assumption list and the triple is inserted into the current proof container and the proof tree node is returned.

The check and instantiation classes of the assume axiom are implemented analogously to the assert axiom, the only difference is the implication in the precondition instead of the conjunction. The implementation of these classes is thus not needed to be discussed further.

The last important step in the back end is the integration of the axioms in the pwp tactic (jive.tactics.KEx_pwp_TACTIC.java). The implementation can be done analogously to other statements. First a new instance of both the check and instantiation class is created in the constructor of the pwp tactic. Then in the substrategy method the cases for assert and assume axioms are implemented, printing a string to the log and just running the call method of the appropriate instance.

There are also some minor details which had to be implemented in the back end, they are mostly related to parsing for the GUI. As the implementation of these details is done analogously to existing code, it does not need to be discussed.

4. Non-default Constructors

So far Jive only supported the default constructor, that means the programmer was not allowed to implement own constructors. If a non-default constructor was found in a program, Jive would print an error message on the console. There was a work around to simulate the behavior of a constructor, the programmer could write a so called init method. The init method was expected to be invoked just after the creation of a new object of the corresponding class. However there were two drawbacks to this solution.

First, it makes the language more useful if it supports the standard Java constructors. Programmers are used to them and would be confused by this work around.

Second, the solution was not sound. For example, when creating a new object, we can usually assume that all fields of the object contain the default value of the appropriate type. For the init method this assumption can be wrong, because a field could already have changed its value when the init method is called.

During this project, non-default constructors where implemented, however there are a few things missing:

To create a Hoare triple for constructors, a special term for invariant checks should be added to the postcondition. This term has to be added to the theorie generators of Simplify and Isabelle. This was no done successfully during this project.

There should be created some examples and test cases to see if the implementation does really what it should do.

5. Conclusion

During this project, the JML statements assert and assume were successfully implemented extending the Jive tool and helping to support JML level 0 features. The extensions took effect through the whole system of Jive, from parsing in the front end to implementation of the Hoare rules in the back end and integration into the pwp tactic.

As another extension, constructors were implemented, replacing the init methods used formerly. To have its full functionality, the theories for Simplify and Isabelle should be revised in future. Another important note: As Jive does not support method overloading, also the current implementation of constructors does not support constructor overloading. This means that the programmer only should implement one single constructor per class, else the verification could go wrong. As programmers conveniently use constructor overloading, this would be a nice feature to add in future to Jive.

Bibliography

[1] A. Poetzsch-Heffter, J.-M. Gaillourdet and N. Rauch. Soundness and Relative Completeness of a Programming Logic for a Sequential Java Subset. Internal Report.

[2] G. Leavens, A. Baker, and C. Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. In *ACM SIGSOFT Software Engineering Notes, 31(3):1-38, 2006.*

[3] http://www.eecs.ucf.edu/~leavens/JML/jmlrefman/jmlrefman_12.html

[4] http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Error.html

[5] M. Barnett, B.-Y. Evan Chang, R. DeLino, B. Jacobs and K.R.M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In FMCO 2005.

[6] A. Darvas and P. Müller. Formal encoding of JML Level 0 specifications in JIVE. Technical Report, 2007.