Using ASF+SDF for the Verification of Annotated Java Programs

Robbert de Haan

UNIVERSITEIT VAN AMSTERDAM

Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Supervisors: Prof. dr. P. Klint and dr. F.S. de Boer
Afstudeerrichting: Programmatuur

Master′s thesis March 2003

Acknowledgments

I would like to thank Paul Klint and Frank de Boer for giving me the opportunity to work on this project. It has been an interesting experience to work with them.

Furthermore, I would like to thank Jurgen Vinju for his extreme enthusiasm and support in the last couple of months. Thanks also go out to him, Jørgen Iversen, Paul Klint and Frank de Boer for looking at preliminary versions of this thesis and giving me the valuable suggestions I needed to improve it.

My warmest appreciation goes out to my family. Especially to my parents, Linda and Arnold de Haan, who have always been there for me when I needed them and to Frank, Sonja and Kai, who keep reminding me that there is more to life than study.

Contents

Chapter 1

Introduction

1.1 Motivation

”In 1948 Alan Turing suggested a role for assertions in checking the correctness of large programs. In 1967, Robert Floyd suggested that a verifying compiler could check the correctness of such assertions by automatic theorem proving. Ever since, these ideas have provided a properly scientific basis for research into the problems of software engineering.” - C.A.R. Hoare

Program verification is about checking that programs are correct, or in other words, about checking that they do exactly what they are supposed to do. Ideally we would just throw every program written through some kind of magical machine that performs this check automatically, but unfortunately the actual situation doesn’t even approach this.

What programmers have to do when they want to verify a program is either picture it using some simpler model and prove that this model is correct or label parts of the program that are important and prove that the information in the labels correctly describes the program. Both approaches still involve a lot of manual work (part of the work can be automated), which is why programs are only verified when their correctness is of vital importance.

Program verification is typically applied in areas where flaws in programs either cost money or lives. Examples are programs that:

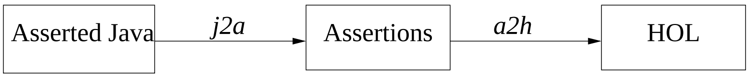
- control the subway system, a space shuttle or a car,
- provide secure communication through use of encryption,
- shield a system from hackers,
- do climate control for a greenhouse,
- go in mass production, making it very expensive to replace them with a corrected version.

1.2 About this project

This project is about program verification for object oriented programs and is an initiative of Frank de Boer. It is similar to the LOOP project worked on in Nijmegen (see section 1.3.1) and a follow-up of the project worked on by Cees Pierik (see section 1.3.2).

The idea of this project is to create a tool that allows one to attach a special kind of labels to a Java program that describe the behavior of the program. The tool will then convert this labeled program to a series of logical formulas, that should all be true if the contents of the labels and the behavior of the program match. So using these formulas one can verify that the behavior of the program is as it is expected to be, by proving their correctness.

This project resulted in the creation of two separate tools, called *j2a* and *a2h*, written in ASF+SDF (see section 1.4.1). The first tool generates the formulas from an asserted Java program, while the second one allows the user to translate these formulas to the syntax of the theorem prover[1] HOL (see section 1.4.3).



1.3 Related work

In this section we mention two related projects and their main similarities and differences compared to this one.

1.3.1 The LOOP project

LOOP stands for ‘Logic of Object-Oriented Programming’. The LOOP project (see [14]) is being worked on in Nijmegen and also involves the verification of Java modules using assertion labels. The assertion language used is JML (Java Modeling Language, see [7]) which labels entire methods and modules at once, instead of one separate statement at a time [2].

The LOOP tool translates the entire semantics of the module, combined with the assertion labels, into logical theories. This approach is radically different from ours, as we both do verification on a much smaller scale (one state transition at a time) and abstract from the semantics of the programming language in our resulting formulas.

Just like our formulas, the theories that the LOOP tool generates can be read by a theorem prover, which helps automate proving their correctness.

[1] A theorem prover is a tool that is designed to formally prove logical formulas, while providing automation that simplifies the proof process.

[2] As our assertion language does.

1.3.2 Verification of flowcharts

Cees Pierik built a tool for the generation of the resulting formulas, called *verification conditions*, from flowcharts (see section 2.3). The corresponding theory is described in [1] and more specifically in [2].

The idea of the tool is to have the flowcharts, diagrams that picture state-changes, describe the semantics of some object oriented program. Using assertion labels at control states, conditions can be generated that verify state-changes.

The tool *j2a* developed in this project basically generates the verification conditions by comparing the semantics of the asserted program with that of the equivalent flowchart. When the structure of the corresponding flowchart is known, the generation of the conditions follows quite naturally from the theory.

1.4 Technology used

1.4.1 ASF+SDF

ASF+SDF ([11]) is the combination of the formalisms ASF (Algebraic Specification Formalism) and SDF (Syntax Definition Formalism).

SDF is a formalism that enables the user to define the lexical and context-free syntax of, for example, a programming language. Because in the SDF-specification both the lexical and context-free syntax are given, it implicitly defines a mapping from a text string that matches the syntax to the corresponding syntax tree. A tool called *sglr* has been written to perform this translation.

ASF is a formalism that describes conditional transformation rules with respect to some signature containing the syntax on which to apply the rewriting. It can be used to describe the semantics of the syntax tree which, in practice, is about transforming the syntax tree. The tool *asfe* performs this transformation, using a syntax tree as input and generating another syntax tree as output.

The result of combining the two is that SDF-modules can be used to describe the syntax of the language and ASF-modules to describe the semantics, using the corresponding SDF-modules as their signature definitions. Using sglr, asfe and these modules, the rewriting of a text string that matches the syntax to an syntax tree that describes the semantics of the string is automatic. Another tool, called *unparsePT* has been written to convert this tree to the corresponding text string.

1.4.2 The ASF+SDF Meta-Environment

The ASF+SDF Meta-Environment ([11]) is an interactive development environment for the automatic generation of interactive systems for manipulating programs, specifications, or other texts written in a formal language.

It basically combines the above-mentioned three programs, making it a generic compiler that compiles a program in a manner that depends on the supplied ASF+SDF-specification. But it also provides extra features such as a user interface that gives an overview of the import graph, an editor with syntax-checking abilities and, in the newest version, an optional link with an integrated debug environment, making it a very convenient programming environment for the development and adjustment of specifications written in ASF+SDF.

For more information about the ASF+SDF Meta-Environment, see the home page ([11]) or the online manual ([12]).

1.4.3 HOL

HOL ([5]), which stands for "Higher Order Logic", is a system designed to support interactive theorem proving in higher order logic. To this end, the formal logic is interfaced to a general purpose programming language in which terms and theorems of the logic can be denoted, proof strategies expressed and applied, and logical theories developed. HOL contains many pre-proved mathematical theorems and a couple of automated theorem provers to reduce the amount of work needed.

A theorem is typically proven by inserting it into HOL as a goal, advancing or splitting the goal into subgoals using the theorems and automated theorem provers and by proving these simplified subgoals in the end. Unfortunately, as in ordinary mathematics, if you fail to prove a goal in HOL this does not necessarily mean it is wrong. It could be that the goal is correct, but that you just haven't found the way to prove it yet.

1.5 Outline of this thesis

Chapter 1 is meant to provide the reader with some basic information concerning this project.

Chapter 2 will provide the programming language independent basics that are used in the implementation of the assertion mechanism.

Chapter 3 will give the details of the implementation for Java. The chapter will provide, for each statement supported, enough information to deduce which verification conditions should be generated when implementing the assertion mechanism.

Chapter 4 will further explain how the assertion mechanism should be used and what specific steps are needed by giving an example that has been verified using this mechanism.

Chapter 5 will discuss some of the specific design choices made when implementing the assertion mechanism from Chapter 3.

Chapter 6 will list some suggestions for future work on this project.

Chapter 2

Theoretical background

2.1 Introduction

This chapter is meant to give an overview of the theoretical, programming language-independent basics that are used to build up the system for the generation of the verification conditions, the logical formulas that will be used to verify the behavior of the program.

We will start this chapter by giving the details of the assertion language. This is the language that has been constructed to annotate the programs. Although it could be used with any object-oriented programming language, the syntax is chosen to match that of Java. Because of this, it should be easier for Java programmers to get familiar with it.

Then we will explain what flowcharts are, what role they play in this project and the need for substitutions. The chapter will be concluded with an explanation of the substitutions used.

For further background information the reader should read [2].

2.2 Assertion language basics

2.2.1 Variables

The basic building blocks of the assertion language are variables. These entities refer to the objects and the fields of these objects. They are of the form u, z, **this** or $v.x$ where:

- u stands for a temporary variable. Temporary variables are the variables that have the most limited scope possible. In most programming languages these are the variables that are declared locally in a function (for C) or in a method (for Java). They are called temporary, because they disappear when the flow of control leaves the function or method.
- z stands for a logical variable. These are the variables used in ordinary logic, either as a free variable or as a variable bound by existential or

universal quantification. These logical variables do not exist in the programming language, although the free variables will be given a meaning in the scope of a method. They cannot change value during execution of a method, which makes them especially useful for comparing changed values of variables and expressions with their values at the beginning of the method.

- **this** refers to the current object, assuming there is such an object.
- $v.x$ stands for the instance variable x on the object referred to by the variable v.

2.2.2 Types

The variables just mentioned must have known types. The types in the assertion language are the basic types **Int** and **Bool**, the types C from some chosen set of class types and the types denoting a finite sequence of **Int**egers, **Bool**eans or objects of some class type C.

Some variables can also be of *undefined* (or *error*) type due to errors in the typing of sub-variables. $x.y$, for example, is syntactically valid as a variable if x refers to an object, but has error type if y is not an instance variable of x.

2.2.3 Expressions

The expressions used in the assertion language are the variables and terms $op(e_1, \cdots, e_n)$ (not necessarily in this notation), where the e_i stand for expressions and *op* stands for some operation on a fixed number of expressions. A selection of some popular expressions has been made for this tool, although it is relatively easy to add new ones.

A new type of variable of the form $z[e]$ is introduced as well, denoting the e^{th} member of the sequence referred to by z.

The expressions used are:

1. The expressions of type Int:
   - v, a variable of type Int.
   - $-e$, where e should be of type Int.
   - $|e|$, where e should be of type 'sequence of Int'. It stands for the length of sequence e.
   - $e_1 +/-/* e_2$, respectively the sum, difference and multiplication of the Int typed expressions e_1 and e_2.
   - An arbitrary sequence of digits, which stands for a constant of type Int.
2. The expressions of type Bool:

- **true**.
- **false**.
- v, a variable of type Bool.
- $e_0 \ \leq / \geq / < / > \ e_1$, where e_0 and e_1 are of type Int.

These symbols have the obvious meanings.

3. The expressions that can have any type:
   - $(\mathbf{nil} : t)$, denoting an undefined expression of type t.
4. The expressions that are of type Bool if they are defined, but can be of undefined type if any of their parameters is undefined:
   - $e_0 \ \wedge / \vee \ e_1$, where e_0 and e_1 are expressions of type Bool.
   - $\neg e$, where e is an expression of type Bool.
   - $e_0 \ = / \neq \ e_1$, where e_0 and e_1 are expressions of the same type.

2.2.4 Assertions

Assertions are basically logical formulas, which are always of type Bool. The idea is to choose the variables and expressions that best match your programming language and allow inclusions into the assertion language. This way the assertion language can be used to express facts about the program using similar syntax.

In our case the variables and expressions match our programming language Java. The assertions are of the following forms, where the first two forms arrange the inclusion of Java syntax:

- $\{v\}$, where v is a variable of type Bool.
- $\{e\}$, where e is an expression of type Bool.
- $A_0 \Rightarrow A_1$, where A_1 and A_2 are assertions.
- $A_0 \vee / \wedge A_1$, where A_1 and A_2 are assertions.
- $\neg A$, where A is an assertion.
- $\forall z(A)$, where A is an assertion.
- $\exists z(A)$, where A is an assertion.

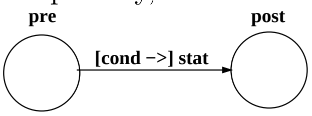
2.3 Flowcharts

The translation of annotated Java statements to the formulas that will enable us to verify the behavior of the program is based on a comparison with flowcharts. Flowcharts are simple diagrams that consist of three parts:

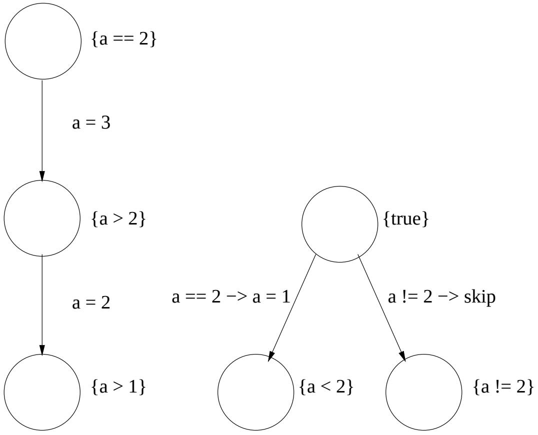
- **States** that are depicted as orbs. These model certain points in the program.

- **Transitions** that are depicted as directed lines between states. They are annotated with the statement that causes the change of state and can optionally have a condition that must be valid before the transition will occur.
- **Assertions** as described in the previous section, which add information to states. When looking at a transition, we call the assertion at the state before the transition the *precondition* and the assertion at the state after the transition the *post-condition* of the statement attached to the transition.

Graphically, a flowchart that consists of a single transition looks like this:



The brackets [] denote that the conditional part is optional. The concrete examples below show how you can build bigger flowcharts by attaching more states to each other (on the left) and how you can use the condition at the transition to get a case split (on the right). As the example on the right shows we also have a statement called 'skip' for flowcharts that are used to model transitions that don't have any effect.



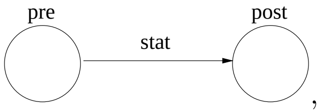
2.4 Assertions and verification conditions

The idea is now to view the annotated program as a very large flowchart and to generate verification conditions for the transitions. The transitions are between different states, in which variables can have different values. We would like to have all the necessary information about each transition in the state before the transition, so that we can generate the verification condition in that state.

We accomplish this using statement-specific[1] substitutions that calcu-

[1] Depending on the statement given at the transition.

late the *weakest precondition* of the post-condition. This is the most general condition that makes the post-condition valid after execution of the statement. We then make the implication of the precondition of the statement to the weakest precondition of the post-condition our verification condition.

So in the case of the following abstract flowchart 

, the verification condition generated would be $\boxed{pre \Rightarrow post[..]}$, where the substitutions [..] depend on *stat*.

If we also had condition *cond* at the transition we would include this information in the implication, making it $\boxed{pre \wedge cond \Rightarrow post[..]}$.

2.5 Substitutions

Substitutions are an essential part of the assertion mechanism. Therefore they should be described extensively before explaining which substitutions are needed for specific statements.

2.5.1 Normal substitutions

Normal substitutions are used to replace free variables and expressions. In general they will traverse the structure of an assertion and replace free variables and expressions as they are encountered. The notation used for them is $[e_2/e_1]$, which denotes that we should replace expression[2] e_1 by expression e_2. We can also use the notation $[e_{n+1}/e_1, \cdots, e_{2n}/e_n]$, which denotes the simultaneous substitution of e_i by e_{i+n} for $1 <= i <= n$.

Because normal substitution replaces free variables and expressions and not references to instance variables, the formula $x.y[z/y]$ is equivalent to $x.y$ and not to $x.z$. If we do want to replace the reference to $x.y$ with a reference to $x.z$ we should explicitly state so, using the substitution $[x.z/x.y]$.

There's another important problem with substituting references to instance variables. Even if u_1 and u_2 are two simple temporary variables of the same class type, in most cases it is impossible to check statically (without executing any code) whether they refer to the same object. When doing a simple variable-name substitution (for example $[u_2/u_1]$) this is not important, but when substituting references it is. If u_1 and u_2 refer to the same object, then $u_1.y$ and $u_2.y$ refer to the same instance variable and both $u_1.y[x/u_1.y]$ and $u_2.y[x/u_1.y]$ are equivalent to x. While if they refer to different objects, $u_2.y[x/u_1.y]$ is not equivalent with x.

So we need some conditional expression that can perform a comparison check 'dynamically', which means that we get a formula that enables us to do a case split that performs the right substitution whether the variables

[2] Note that free variables are also expressions.

are equal or not. For this purpose, an expression **if** e_1 **then** e_2 **else** e_3 **fi** is added to the assertion language. This conditional expression is relevant for the following special cases of the substitution, which are needed to properly deal with substitution of references to instance variables ($\equiv$ denotes syntactic equivalence):

$$
\begin{aligned}
(v_1.x)[e/v.x] &\equiv \textbf{if } v_1[e/v.x]{=}\, v \textbf{ then } e \textbf{ else } (v_1[e/v.x]).x \textbf{ fi}\\
(v_1.y)[e/v.x] &\equiv (v_1[e/v.x]).y
\end{aligned}
$$

where, as in the section describing the assertion language, v and v_1 denote variables and e is some expression.

Example

Assume x and a are variables. Then

$$
\begin{aligned}
(x.b.c > 3)[a/a.b] &\equiv x.b.c[a/a.b] > 3[a/a.b]\\
&\equiv (x.b[a/a.b]).c > 3\\
&\equiv (\textbf{if } x[a/a.b] = a \textbf{ then } a \textbf{ else } (x[a/a.b]).b \textbf{ fi}).c > 3\\
&\equiv (\textbf{if } x = a \textbf{ then } a \textbf{ else } x.b \textbf{ fi}).c > 3
\end{aligned}
$$

2.5.2 *New* substitutions

New substitutions are a special kind of substitution, denoted as $[\textbf{new}/u]$, used to model the creation of a new object referred to by some temporary variable u.

Creating an object means that a new object is added to the collection of existing objects. So when moving an assertion that states something about a new object to the previous state (see sections 2.3 and 2.4) we actually move it to a situation were the new object doesn't exist yet. The *new* substitution is designed to deal with this.

For most expressions, the substitution just does a top-down traversal over the structure. However, there are some special cases that define its special behavior. First, for a reference to an instance variable of the newly created object:

$$(u.x)[\textbf{new}/u] \equiv (\textbf{nil}\text{: } t)$$

where t is the type of the instance variable $u.x$.

u can also occur in (in)equalities. If neither e_1 nor e_2 is a conditional expression or u they cannot refer to the newly created object and we have ordinary traversal:

$$
\begin{aligned}
(e_1 = e_2)[\textbf{new}/u] &\equiv (e_1[\textbf{new}/u]) = (e_2[\textbf{new}/u])\\
(e_1 \neq e_2)[\textbf{new}/u] &\equiv (e_1[\textbf{new}/u]) \neq (e_2[\textbf{new}/u])
\end{aligned}
$$

If e_1 is u and e_2 is neither u nor a conditional expression (or vice versa) they cannot refer to the same object (because one of them refers to the newly created object and the other does not), so we get:

$$(e_1 = e_2)[\mathbf{new}/u] \equiv \mathbf{false}$$
$$(e_1 \neq e_2)[\mathbf{new}/u] \equiv \mathbf{true}$$

And of course, $(u = u)[\mathbf{new}/u] \equiv \mathbf{true}$

If e_1 is a conditional expression of the form **if** e_3 **then** e_4 **else** e_5 **fi** then:

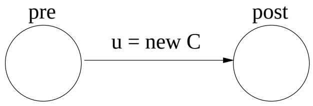
$$(e_1 = e_2)[\mathbf{new}/u] \equiv$$
$$\mathbf{if}\ e_3[\mathbf{new}/u]\ \mathbf{then}\ (e_4 = e_2)[\mathbf{new}/u]\ \mathbf{else}$$
$$(e_5{=}e_2)[\mathbf{new}/u]\ \mathbf{fi}$$

As with expressions, for most assertions the substitution just traverses inductively over the structure. When a new object is created however, as mentioned before, the collection of existing objects changes, which has consequences for universal and existential quantification. The following equations capture this change of scope when z ranges over objects:

$$(\exists z\ (A))[\mathbf{new}/u] \equiv (\exists z\ (A[\mathbf{new}/u])) \vee (A[u/z][\mathbf{new}/u])$$
$$(\forall z\ (A))[\mathbf{new}/u] \equiv (\forall z\ (A[\mathbf{new}/u])) \wedge (A[u/z][\mathbf{new}/u])$$

Example

Suppose that we have a transition with a statement that creates a new object and that we know in the starting state that only one object exists, namely the current one. In logic, this assumption *pre* could be written as $\boxed{\forall z\ (z = \mathbf{this})}$.



The transition adds a new object to the collection of existing objects. Because of this, precisely two objects exist at the second state, namely the object referred to by u and the current object. We denote this fact with the formula $\boxed{\forall z\ (z = \mathbf{this} \vee z = u)}$ for *post*.

In this situation, we would use the substitution $[\mathbf{new}/u]$ to calculate the weakest precondition of *post* (more about this in the next chapter), which would become:

$$(\forall z\ (z = \mathbf{this} \vee z = u))[\mathbf{new}/u] \equiv$$
$$\forall z\ ((z = \mathbf{this} \vee z = u)[\mathbf{new}/u]) \wedge (z = \mathbf{this} \vee z = u)[u/z][\mathbf{new}/u] \equiv$$
$$(\forall z\ (z = \mathbf{this} \vee \mathbf{false}) \wedge (u = \mathbf{this} \vee u = u)[\mathbf{new}/u] \equiv$$
$$(\forall z\ (z = \mathbf{this} \vee \mathbf{false}) \wedge (\mathbf{false} \vee \mathbf{true})$$

It is clear that the result is equivalent to the precondition *pre*, showing us that our post-condition *post* follows from precondition *pre* under the transition given.

Existential quantification over sequences of objects

Let z be a logical variable ranging over sequences of objects and z' a logical variable ranging over sequences of boolean values. The variables z and z' together will, in a post-condition after object creation, code a sequence of objects possibly including the newly created object. At the places where z' yields **true** the value of the coded sequence is the newly created object, where z' yields **false** the value of the coded sequence is the same as the value of z. This encoding is described by a special substitution $[z‘, u/z]$, where u again refers to the newly created object.

In our assertion language, the only operations on sequences are $|z|$, which denotes the length of sequence z, and $z[n]$, which gives us the n^{th} element of sequence z (only defined if $1 <= n <= |z|$). So we will describe the substitution $[z‘, u/z]$ by stating its effects on these operations:

$$\begin{aligned} z[z', u/z] &\equiv \text{undefined} \\ (|z|)[z', u/z] &\equiv |z| \\ (z[e])[z', u/z] &\equiv \textbf{if } z'[e'] \textbf{ then } u \textbf{ else } z[e'] \textbf{ fi} \end{aligned}$$

where $e' = e[z', u/z]$

For the remaining expressions and assertions the substitution just does a top-down traversal over the structure (for example, $(|z| \wedge |z'|)[z‘, u/z] \equiv (|z|)[z‘, u/z] \wedge (|z'|)[z‘, u/z]$).

Given this encoding we can now define:

$$(\exists z\ (A))[\mathbf{new}/u] \equiv \exists z\ (\exists z'\ (|z| = |z'| \wedge (A[z', u/z][\mathbf{new}/u])))$$

where z ranges over sequences of objects.

For an example on using this substitution, the reader is referred to [2].

Chapter 3

Tool support for verification

3.1 Introduction

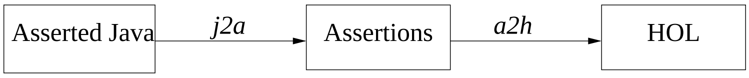
This chapter is about the details specific to the implementation. The language that our theory is applied to is Java, so the generation of the verification conditions is also discussed with respect to this language. One important aspect that has not (yet) been implemented is class inheritance. Because of this we can simplify matters a little bit.

Basically, this chapter contains every interesting implementation aspect related to this project. The next section gives a rough overview of the actual implementation. Then a detailed overview will be given of the statements currently supported and their interpretation within the tool. And the last section will dive into the machinery used for interpreting exception handling.

This chapter does not contain actual code fragments of the implemented system itself, although it will demonstrate what the translation of the simple statements would look like in ASF+SDF. The actual code will be available via the web later on.

3.2 The building blocks of this project

The translation of the asserted statements to HOL-statements has been split up into two parts. First the asserted statements are translated to assertions in the assertion language and then these assertions are translated to HOL-syntax.



Five ingredients are needed to establish this process: The syntax of the assertion language, the syntax of the annotated language (in this case Java) enriched with information about where to insert assertions, the syntax of

HOL, rewrite rules that state how to translate asserted Java to assertions and rewrite rules that state how to translate assertions to HOL-formulas. In my implementation I abbreviated these pieces respectively as *assert*, *java*, *hol*, *j2a* (which also specifies where to insert the assertions) and *a2h*.

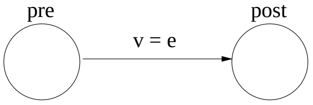
It is clear that the first translation only requires *assert*, *java* and *j2a* while the second one only requires *assert*, *hol* and *a2h*. As *assert*, *java* and *hol* are only concerned with syntax, the corresponding parts of the ASF+SDF specification consist of SDF-modules only. The most interesting parts, *j2a* and *a2h*, consist of ASF-modules that describe the rewrite process and, because we use special syntax for pieces of code that are still being rewritten, SDF-modules that describe the additionally needed syntax.

3.3 Conversion of simple statements

As mentioned before, we first model each statement as a flowchart and then describe the generated verification conditions. Because the flowcharts and the conditions are themselves based on Hoare's logic (see [3]), we will also show the corresponding rules from this theory. Additionally, to show how easy it is to implement the generation process using ASF+SDF we will sketch what the corresponding rules would look like in ASF.

3.3.1 Assignments

When dealing with a simple assignment of the form $\boxed{pre\ v = e;\ post}$, where *pre* is the precondition, *post* is the post-condition, v is a variable and e is an expression in Java which is also in the assertion language, we get the following corresponding flowchart



We now calculate the weakest precondition of *post* using the substitution $[e/v]$ and get the verification condition $\boxed{pre \Rightarrow post[e/v]}$ as announced at the end of the previous chapter. This condition is actually based on the rule $\boxed{\{P[e/v]\}\ v := e\ \{P\}}$ from Hoare's logic that says that $P[e/v]$ is the weakest precondition of the post-condition when executing an assignment.

The corresponding rewrite rule in ASF could look like the one below, which gives a direct mapping of the statement to the generated verification condition:

```
[cSA] convertSimpleAssignment(
          Assertion1 Variable = Expression; Assertion2
                                ) =
      Assertion1 ==> Assertion2[Expression/Variable]
```

Assignments with object creation

We also allow assignments where v is a temporary variable and e is of the form *new C()*, where C is the name of an object. But here we cannot use the simple substitution $[b/a]$ because the collection of existing objects before and after the assignment is different. Instead we have to use the substitution $[\mathbf{new}/v]$, giving the verification condition $\boxed{pre \Rightarrow post[\mathbf{new}/v]}$. In this special case, the corresponding rule would be $\boxed{\{P[new/u]\}\ u := new\ C\ \{P\}}$ and a matching ASF-rule could be of the form

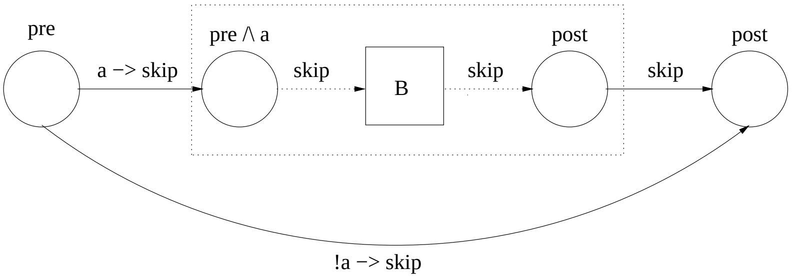
```
[cNA] convertNewAssignment(
          Assertion1 Variable = new ClassName(); Assertion2
                              ) =
      Assertion1 ==> Assertion2[new/Variable]
```

3.3.2 If-then and if-then-else

If then

If-then statements are a little bit trickier to convert, because they typically contain a whole block of statements themselves. If-then statements are of the form $\boxed{pre \text{ if } (a) \text{ then } \mathbf{B}\ post,}$ where **B** is a block of (possibly asserted) statements. a is either a boolean Java expression that is also valid in the assertion language or a boolean Java equation of the form 'v == **null**' or 'v != **null**', where v is a variable. The reason for these last two forms is that this way we can easily deduct the type of the **null**-statement when translating it into assertion syntax.

The if-then statement corresponds with the following flow diagram:



The block with the B in it stands for the flow chart of the block of statements **B**. Above the left arrow the annotation '$a \rightarrow skip$' is found, meaning that this path is only taken if a is valid and that the *skip*-statement (which doesn't do anything) is executed during the transition. This is also how the other annotated transition on the left, where ! is Java-syntax for $\neg$, should be read.

For the if-then statement we have the rule

$$\boxed{\frac{\{P\wedge a\}\ B\ \{Q\}\ and\ P\wedge\neg a\rightarrow Q}{\{P\}\ if\ (a)\ then\ B\ \{Q\}}}$$

Its first assumption is that we can deduce the post-condition of the statement from its precondition, when condition a is valid, by executing block B. The second assumption says that we can, when a is not valid, deduce the post-condition directly from its precondition. If these assumptions are both valid, then we may conclude that P and Q are valid as pre- and post-condition of the statement.

So first we have to generate the implication $\boxed{(pre \wedge \neg a) \Rightarrow post}$ for the transition that skips the if-then statement if a is false. Then we have the obvious implications near the beginning between *pre* and $pre \wedge a$ and at the end between *post* and *post*. As described in section 2.4, this would give us implications $pre \wedge a \Rightarrow pre \wedge a$ and $post \Rightarrow post$, but since these implications are trivially true we do not generate them.

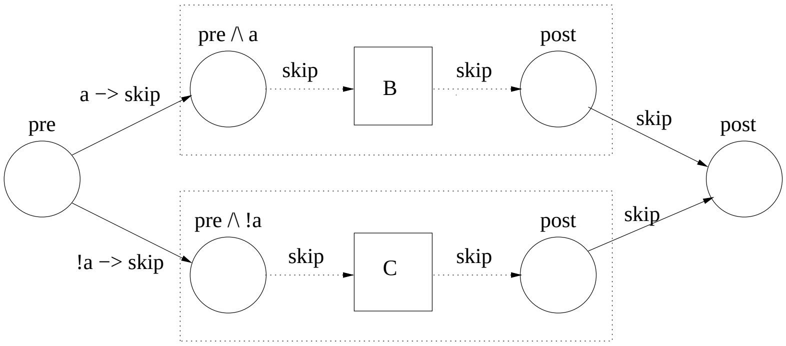
Which other verification conditions should now be generated? Clearly this depends on the structure of the block of statements **B**. To show this, the part that depends on block **B** has been placed in a dotted rectangle, which can only be interpreted when looking at the flowchart of block **B** as well.

Assuming that the verification conditions generated are separated by commas, we could express the conversion of the if-then statement in ASF using

```
[cIT] convertIfThen(Assertion1
                    if (Expression) then Block
                    Assertion2) =
      Assertion1 && (!Expression) ==> Assertion2,
      convertBlock((Assertion1 && Expression) Block Assertion2)
```

If then else

If-then-else statements are very similar to if-then statements. They are of the form *pre* if (a) then **B** else **C** *post* and correspond with the following flow chart:



The way the verification conditions are generated is as with if-then statements, with the exception that both transitions from the initial state pass through a block of statements. The rule for this statement is

$$\boxed{\dfrac{\{P \wedge a\}\ B\ \{Q\}\ \textit{and}\ \{P \wedge \neg a\}\ C\ \{Q\}}{\{P\}\ \textit{if}\ (a)\ \textit{then}\ B\ \textit{else}\ C\ \{Q\}}}$$

with a similar meaning as the previous one. The corresponding ASF-rule would be

```
[cITE] convertIfThenElse(Assertion1
                         if (Expression) then Block1 else Block2
                         Assertion2) =
       convertBlock((Assertion1 &&   Expression)  Block1 Assertion2),
       convertBlock((Assertion1 && (!Expression)) Block2 Assertion2)
```

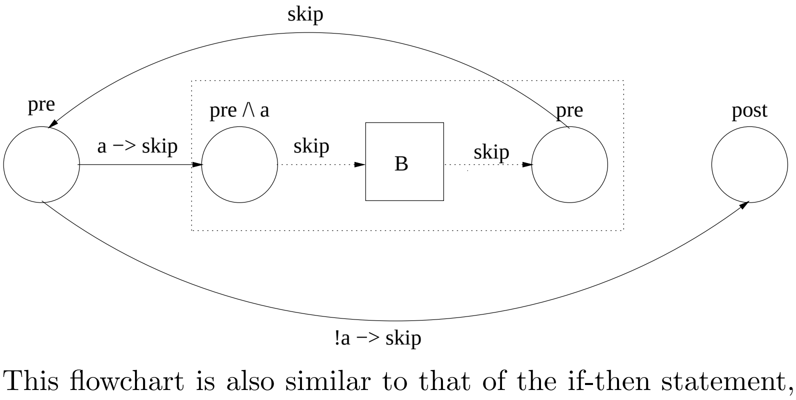
3.3.3 While

While loops are of the form $\boxed{\textit{pre}\ \text{while}\ (a)\ \mathbf{B}\ \textit{post}}$ and have the following rule

$$\boxed{\dfrac{\{P \wedge a\}\ B\ \{Q\}}{\{P\}\ \textit{while}\ (a)\ B\ \{P \wedge \neg a\}}}$$

which basically states that if the precondition remains valid after executing the statements in the loop once, it remains valid after any number of executions of the statements in the loop. Or in other words, that in this case the precondition is still valid after the execution of the while-statement has finished.

The flowchart for this statement is:



This flowchart is also similar to that of the if-then statement, as is the generation of the verification conditions. The important difference is the addition of a backward transition at the end of block **B**. As this picture clearly shows, *pre* is considered to be the invariant of the loop.

We don't need to generate an additional condition for the backward transition as it would give us the trivially true implication $pre \Rightarrow pre$.

We could implement the while-statement in ASF using

```
[cW] convertWhile(Assertion1
                  while (Expression) Block
                  Assertion2) =
     Assertion1 && (!Expression) ==> Assertion2,
     convertBlock((Assertion1 && Expression) Block Assertion1)
```

3.4 Method invocation

3.4.1 The context switch

Method definitions can be annotated with a pre- and a post-condition, which makes it possible to describe the effects that placing a call to a method has. Typical points of interest are the effects caused by assignments to instance variables and information about the expression the method returns [1].

An important problem with annotating method invocations is that (especially temporary) variables in the method are not accessible or even visible from the outside and vice versa and that the reference value **this** refers to a different object in the environment of the caller than it does in the callee. To solve this problem we use the concept of the *context switch*, a combination of substitutions that solves the change of environment.

3.4.2 The rule for method invocation

We only look at calls to methods that return an expression combined with an assignment that assigns this expression to a temporary variable. So we look at assignments of the form $\boxed{pre\ x = e_0.m(e_1, \cdots, e_n);\ post}$ for which we use the notation $O\ x := e_0.m(E)\ R$.

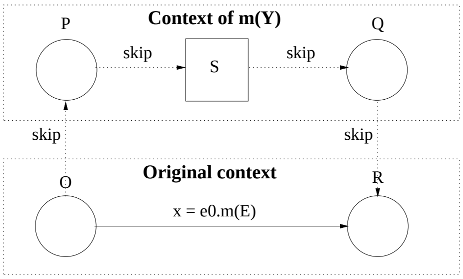
We assume $m()$ has formal parameters $y_1, \ldots, y_n$ for which we use shorthand notation Y. Name the body of $m()$ S, its precondition P and its post-condition Q. The expression this method returns is e and r is an abstract reference to the return-value of the method.[2]

The picture below shows the different contexts that play a role during the method invocation. Instead of proving the transition from O to R directly we will take a detour via the method that is being called. We assume that the path from P to Q has been verified when we encountered the method m() in the class specification and will verify the transitions from O

[1] We only allow methods to have one return-expression, because this is an assumption made in the corresponding theory.

[2] Such an abstract reference is useful, because it enables the user to make statements about what a method returns.

to P and from Q to R in the 'original' context where we invoked the method.



We have the following rule for method invocations that helps us:

$$\boxed{\frac{\{P\}S\{Q[e/r]\}\ \mathit{and}\ Q[e_0/\mathit{this}][E/Y]{\rightarrow}R[r/x]}{\{P[e_0/\mathit{this}][E/Y]\}\ x{:=}e_0.m(E)\ \{R\}}}$$

The parts of this rule should be explained:

- The first assumption, $\{P\}S\{Q[e/r]\}$, states what we already assumed, namely that P and Q should be correct pre- and post-conditions of $m()$. Because r, the reference to the result, has no meaning in the body S of $m()$ it should be substituted by the expression it refers to (which *does* have meaning in the body).
- The second assumption, $Q[e_0/\mathit{this}][E/Y] \rightarrow R[r/x]$, forms the implication from the post-condition of the method to the post-condition of the method call in the context of caller. $[E/Y]$ is shorthand for the simultaneous substitution $[e_1/y_1, \cdots, e_n/y_n]$.
- And finally, the conclusion $\{P[e_0/\mathit{this}][E/Y]\}\ x := e_0.m(E)\ \{R\}$ of the goal states that, if the assumptions are met, we have an implication of the precondition of the method to the post-condition of the method call in the context of the caller.

Using the formula we need to generate verification conditions for the assumptions and for the implication $\boxed{O \Rightarrow P[e_0/\mathit{this}][E/Y]}$ in order to verify the implication $O \Rightarrow R$ (in the context of the caller).

3.4.3 Invariance

The approach mentioned in the previous subsection does not deal with logical variables that possibly appear in the pre- and post-condition of the method. Logical variables are used as invariant expressions, typically to compare the old and the new value of a variable or expression. It is therefore very common for a comparison between such logical variable and an expression to appear in the precondition. Some 'machinery' should be added to correct the relationship between logical variables in the precondition of the method and expressions in the precondition of the method call.

The problem is corrected with the following rule that can cope with invariance:

$$\boxed{\dfrac{\{P\}S\{Q[e/r]\}\ \mathit{and}\ Q[e_0/\mathit{this}][E/Y][F/Z]{\rightarrow}R[r/x]}{\{P[e_0/\mathit{this}][E/Y][F/Z]\}\ x{:=}e_0.m(E)\ \{R\}}}$$

New in this rule is the substitution $[F/Z]$ that stands for the simultaneous substitution of all logical variables in the assertion with their matching expressions in the context of the caller.

3.4.4 Notes on using the assertion mechanism for method invocation

It is important to note that the pre- and post-condition given for a method will typically depend on the type of calls that we expect to be made to this method, either from another part of our program or by some unknown program. In the next chapter, for example, a method is given that is expected to be called with some appropriate parameter of type integer (some assumption about it is stated in the precondition), although in principle it could be called with any integer value as actual parameter. Of course we could just rewrite the method to make it work on any input, but you usually want to *check* something about a program using the assertions and not modify the program to match the assertions (although this could be useful when writing a method for the first time).

A more important thing to note, about the context switch, is that it can only be used under certain conditions. This was noticed when discussing its implementation using ASF+SDF. Of course the sort of expressions that can be used as actual parameters is already limited to the ones accepted by the assertion language, but there's another problem. When supplying a method with an expression containing an instance variable as an actual parameter, we may get a faulty condition when using the context switch on the post-condition. What happens is that we again substitute a formal parameter for its actual parameter, but that this one may suddenly have a different value because the value of the instance variable was changed during execution of the method. Therefore we can only correctly use the context switch if we don't use instance variables in the actual parameters [3].

3.5 Constructor invocation

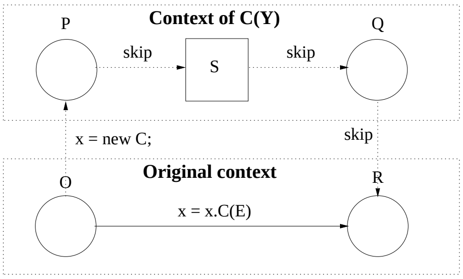
In most cases, the substitution mechanism described for assignments with object creation is not sufficient. The reason for this is that in Java all objects

[3] Or if we are certain that the instance variables supplied are not changed during execution of the method, but this would typically be something that you would verify using the assertion mechanism.

have constructors, methods that are called when a new object is created. The effect of this constructor call at object creation time should also be considered. In this section we assume the constructor that is being called has been annotated with a pre- and a post-condition, in a way similar to annotating an ordinary method.

We will deal with object creation coupled with a constructor call by modeling these two actions as a special kind of method invocation. If we have a statement of the form $\boxed{O\ x = new\ C(e_1, \cdots, e_n);\ R}$ we can model it as $O\ x = new\ C;\ x = x.C(e_1, \cdots, e_n);\ R$, assuming for the moment that you can actually use a constructor as a callable method on an object returning the object itself.

The second statement is a method call, and we already have a rule to verify method calls. The main difference is that we also need the information that x is a newly created object. We will use the following diagram, calling the precondition of the constructor P, its post-condition Q and its body S:



This picture differs from the one used with ordinary method invocation with respect to the transition between O and P, which is no longer trivial. What we will do is apply the context switch first, which will move assertion P to the original context. Then we will use the statement-specific substitution $[\mathbf{new}/x]$ to move the assertion to the state of O and then we will form the implication between O and P, which gives us implication $\boxed{O \Rightarrow P[x/this][E/Y][F/Z][\mathbf{new}/x]}$. The rest of the verification conditions that are generated are similar to those for method invocation.

This choice of interpretation has consequences for the format of the pre- and post-condition of the constructor. If you want, for example, to say that after the assignment the new object is unique (which was a trivial consequence when using the simple form of object creation) this must already be stated in the precondition of the constructor and repeated up to the post-condition of the constructor.

3.6 Exception handling

3.6.1 Introduction

It would be very convenient to be able to deal with exception handling using our assertion mechanism. Inheritance is not supported (yet), which simplifies the process a little bit and allows us to focus on the principles involved without having to worry about types of exceptions that are not known at compile time.

Exceptions are modeled as special boolean variables, distinguishable from ordinary boolean variables, that are only **true** when an exception of the matching type is thrown. In Java only one exception can be thrown at any given time during the execution of the program, which implies that at most one 'exception variable' is **true** at any given time.

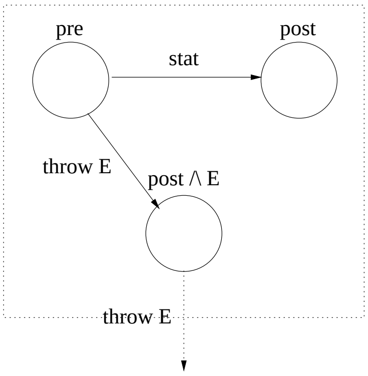
So far we have begun modeling each type of statement as a flowchart and derived its intended interpretation using this flowchart. Because the flow of control was more or less sequential this was a very easy and natural way to do this. When an exception is thrown however, the flow of control is interrupted and will continue, if at all, at the first embracing statement that handles the exception thrown. This behavior should be reflected in the way verification conditions are generated from the assertions given, but it does require additional machinery to build the proper corresponding flowcharts.

3.6.2 Exception mechanism principles

There are basically two ways in which exceptions can be thrown in Java. They can be thrown *explicitly* using the throw-statement or *implicitly* when executing a statement or some expression that violates the normal semantics of Java, for example when encountering a division by zero. Although implicit exceptions appear to be thrown very unexpectedly, in Java most of them are only thrown at a point where they are specified as a possible result of an expression evaluation or statement execution.

When executing a statement we need to know which exceptions can be thrown, either explicitly or implicitly, in order to construct the proper flowchart. Exceptions only affect the state after the interrupted statement and not the states before it, so the place to describe the effect of an exception appears to be the post-condition of the statement that caused it. Since we already have distinguishable boolean variables for the exceptions we will use these same variables to specify, with their occurrence in the post-condition, the exceptions that can be thrown during execution of a statement.

We define $\boxed{\textit{throw } E}$ as an alias for $\boxed{E \rightarrow \ \textit{skip}}$ as annotation for transitions in a flowchart, where E is a boolean variable corresponding with some exception class E. Within a block of statements (pictured with a dotted rectangle) we have the following generic translation for a statement *pre stat post* that can result in a throw of exception E. If exception booleans are not explicitly given a value in a part of the flowchart then this means that they are **false** in this part.



As the picture shows the transition from the exception state (the one at the bottom) leaves the block where the exception was thrown. It may be caught later on, but this depends on the surrounding program.

From the picture shown we can derive two verification conditions, one for the normal transition and one for the transition caused by the exception thrown. Assuming that the *pre* contains boolean exception variables $E, F_1, \cdots, F_n$ and *post* contains $E, G_1, \cdots, G_m$ we would generate ([..] being the substitution depending on *stat*):

- $\mathit{pre} \wedge (\neg E \wedge \neg F_1 \wedge \cdots \wedge \neg F_n) \Rightarrow \mathit{post}[..] \wedge (\neg E \wedge \neg G_1 \wedge \cdots \wedge \neg G_m)$
- $\mathit{pre} \wedge E \wedge (\neg F_1 \wedge \cdots \wedge \neg F_n) \Rightarrow (\mathit{post} \wedge E) \wedge (\neg G_1 \wedge \cdots \wedge \neg G_m)$

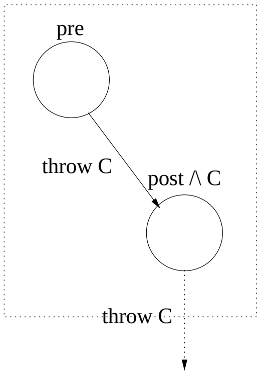
Note that we will also need to generate exception conditions similar to the second one for the exceptions $G_1, \cdots, G_m$ (since they occur in *post*!), but to keep the picture simple the corresponding transitions have been left out.

3.6.3 Exception related statements

If an exception occurs while executing a statement, it may have side-effects. For conditional and compositional statements (such as **if** and **while**) it just breaks the flow of control, but the effect usually depends on the type of statement. The most important ones will be mentioned.

Throw

The **throw**-statement is the only statement that explicitly throws an exception. It is currently only supported in the simple form $\boxed{\text{throw new C();}}$, where C is the class type of the exception thrown. Its translation to a flowchart is:



In this case we don't have a normal transition and the only condition generated for the exception transition would be $pre \wedge C \wedge (\neg F_1 \wedge \cdots \wedge \neg F_n) \Rightarrow (post \wedge C) \wedge (\neg G_1 \wedge \cdots \wedge \neg G_m)$, as implicit exceptions cannot be thrown at a throw-statement.

Try catch

The **try-catch**-statement is the first variant of the more general **try-catch-finally**-statement, which will be treated later on. Its syntax is

$$\boxed{try\ A\ catch(Exception_1\ v_1)\ B_1 \cdots catch(Exception_n\ v_n)\ B_n}$$

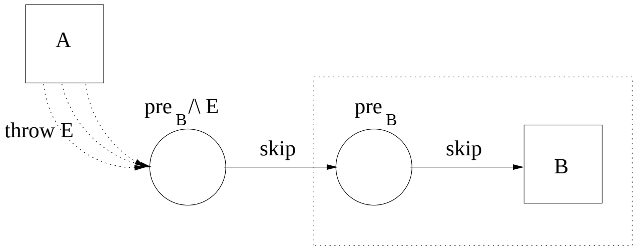
where A and the B_i are blocks of statements, the $Exception_i$ are types of exception classes and the v_i are variable names.

If an uncaught exception of type E is thrown in block A, then the present catch-clauses will be inspected (in order) to see if one of the types $Exception_i$ matches E. [4] If one of them matches then the exception is caught, normal execution will continue in the corresponding block and when execution leaves this block it will also leave the **try-catch**-statement.

So what we need to do is to generate, for every exception thrown, the implication from the post-condition of every statement where the exception is thrown (either implicit or explicit) to the precondition (if present) of the first statement in the catch-block that handles the exception, which can be seen as the precondition of the entire catch-block.

For a statement of the form $try\ A\ catch(E\ v)\ B$, where B has precondition pre_B, we would get the chart below. The dotted transitions coming from block A are coming from the throws that occur within the block and leave it, as can be seen in the charts of the previous sections.

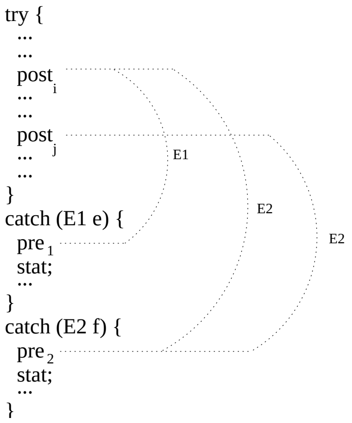
[4] Or, when using inheritance, to see if one of the $Exception_i$ is a superclass of E.



We don't need to generate the implication $pre_B \wedge E \Rightarrow pre_B$ for these flowcharts, since it is always true. Observe the situation shown in the next picture, which models a possible series of thrown exceptions in some try-block. Assuming for the moment that all statements apply to the generic case of section 3.6.2, so the precondition of the throw in the chart above will be of the form $post \wedge E$ for some post-condition $post$ and some boolean exception variable E, we would generate the following additional verification conditions for the jumps in the flow of control:

- $(post_i \wedge b_1) \wedge (\neg b_2 \wedge \cdots \wedge \neg b_n) \Rightarrow$
  $(pre_1 \wedge b_1) \wedge (\neg b_2 \wedge \cdots \wedge \neg b_n)$
- $(post_i \wedge b_2) \wedge (\neg b_1 \wedge \neg b_3 \wedge \cdots \wedge \neg b_n) \Rightarrow$
  $(pre_2 \wedge b_2) \wedge (\neg b_1 \wedge \neg b_3 \wedge \cdots \wedge \neg b_n)$
- $(post_j \wedge b_2) \wedge (\neg b_1 \wedge \neg b_3 \wedge \cdots \wedge \neg b_n) \Rightarrow$
  $(pre_2 \wedge b_2) \wedge (\neg b_1 \wedge \neg b_3 \wedge \cdots \wedge \neg b_n)$

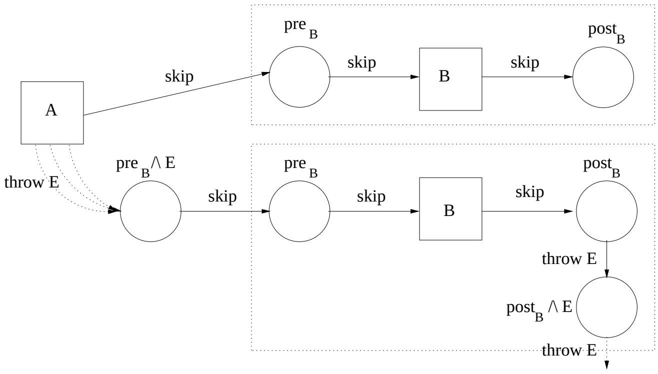
where we also assume that the exceptions that can be thrown are $E_1,\cdots,E_n$ with boolean exception variables $b_1,\cdots,b_n$ and that all of these variables occur in every assertion.


```
try {
  ...
  ...
  post_i
  ...
  ...
  post_j
  ...
  ...
}
catch (E1 e) {
  pre_1
  stat;
  ...
}
catch (E2 f) {
  pre_2
  stat;
  ...
}
```

Try finally

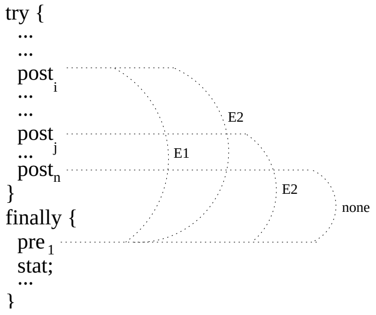
The **try-finally**-statement is of the form $\boxed{try\ A\ finally\ B}$, where A and B are again blocks of statements. If block A terminates normally, execution continues in block B. If an uncaught exception is thrown in A, execution continues normally in B, but when block B terminates normally the exception is thrown again. Should another uncaught exception be thrown in block B in the last case, then the previous exception is forgotten and the new exception leaves block B.

The flowchart of the statement would look like the one below. As can be seen block B will be executed whether block A terminates normally or abnormally, but the way in which the termination of B is handled will differ.



For the example situation of the next picture, under similar assumptions as in the previous section we would generate the following additional verification conditions for the jumps in the flow of control:

- $(post_i \wedge b_1) \wedge (\neg b_2 \wedge \cdots \wedge \neg b_n) \Rightarrow$ $(pre_1 \wedge b_1) \wedge (\neg b_2 \wedge \cdots \wedge \neg b_n)$
- $(post_i \wedge b_2) \wedge (\neg b_1 \wedge \neg b_3 \wedge \cdots \wedge \neg b_n) \Rightarrow$ $(pre_1 \wedge b_2) \wedge (\neg b_1 \wedge \neg b_3 \wedge \cdots \wedge \neg b_n)$
- $(post_j \wedge b_2) \wedge (\neg b_1 \wedge \neg b_3 \wedge \cdots \wedge \neg b_n) \Rightarrow$ $(pre_1 \wedge b_2) \wedge (\neg b_1 \wedge \neg b_3 \wedge \cdots \wedge \neg b_n)$
- $post_n \wedge (\neg b_1 \wedge \cdots \wedge \neg b_n) \Rightarrow$ $pre_1 \wedge (\neg b_1 \wedge \cdots \wedge \neg b_n)$



```
try {
 ...
 ...
 post_i
 ...
 ...
 post_j
 ...
 post_n
}
finally {
 pre_1
 stat;
 ...
}
```

Try catch finally

We are now ready to describe the **try-catch-finally**-statement, which is a complex combination of the principles just mentioned. It's syntax is

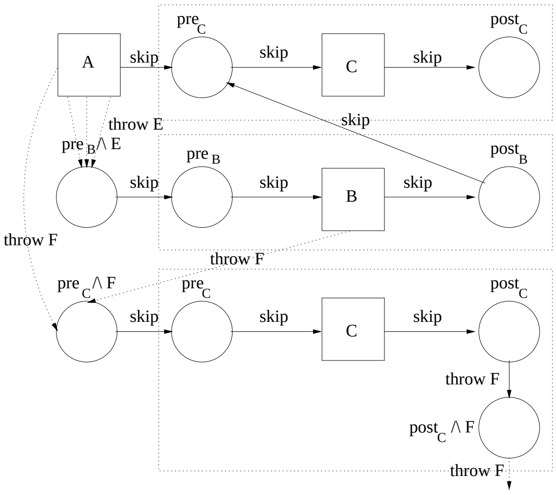
$$\boxed{try\ A\ catch(Exception_1\ v_1)\ B_1 \cdots catch(Exception_n\ v_n)\ B_n\ finally\ C}$$

where the $Exception_i$ are types of exceptions, the v_i are variables and A, the B_i and C are blocks of statements.

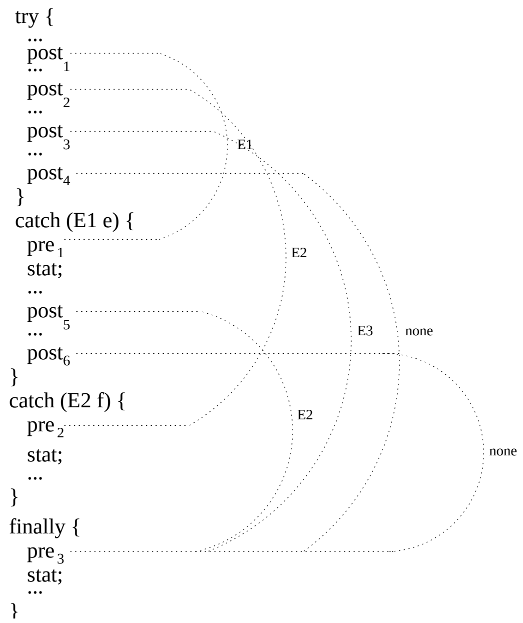
Its basic properties are these:

- If an uncaught exception is thrown in block A it is again caught in the matching catch-block, if available, as described for the **try-catch-statement**.
- When we have a normal exit from block A or one of the B_i, block C is executed.
- If we exit one of the B_i because of an exception, or if block A throws an exception that is not caught in one of the catch-clauses, then block C will also be executed. But if block C finishes normally after this, the exception is thrown again at the end of C.

The next picture will give the general idea of the flowchart constructed for a statement of the form $try\ A\ catch(E\ v)\ B\ finally\ C$, where F is some exception that is not caught by the catch-clause:



Again assuming that the exceptions that can be thrown are $E_1,\cdots,E_n$, their matching boolean exception variables are $b_1,\cdots,b_n$ and that these booleans occur in every assertion, we would generate the following additional verification conditions for the jumps in the situation of the picture below:



```
try {
  ...
  post_1
  ...
  post_2
  ...
  post_3
  ...
  post_4
}
catch (E1 e) {
  pre_1
  stat;
  ...
  post_5
  ...
  post_6
}
catch (E2 f) {
  pre_2
  stat;
  ...
}
finally {
  pre_3
  stat;
  ...
}
```

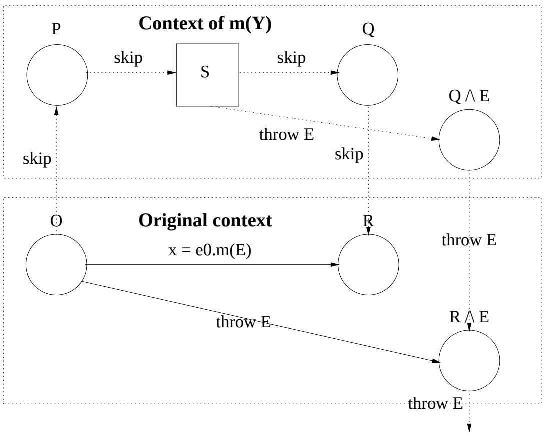
- $(post_1 \wedge b_1) \wedge (\neg b_2 \wedge \cdots \wedge \neg b_n) \Rightarrow$
  $(pre_1 \wedge b_1) \wedge (\neg b_2 \wedge \cdots \wedge \neg b_n)$
- $(post_2 \wedge b_2) \wedge (\neg b_1 \wedge \neg b_3 \wedge \cdots \wedge \neg b_n) \Rightarrow$
  $(pre_2 \wedge b_2) \wedge (\neg b_1 \wedge \neg b_3 \wedge \cdots \wedge \neg b_n)$
- $(post_3 \wedge b_3) \wedge (\neg b_1 \wedge \neg b_2 \wedge \neg b_4 \wedge \cdots \wedge \neg b_n) \Rightarrow$
  $(pre_3 \wedge b_3) \wedge (\neg b_1 \wedge \neg b_2 \wedge \neg b_4 \wedge \cdots \wedge \neg b_n)$
- $post_4 \wedge (\neg b_1 \wedge \cdots \wedge \neg b_n) \Rightarrow$
  $pre_3 \wedge (\neg b_1 \wedge \cdots \wedge \neg b_n)$
- $(post_5 \wedge b_2) \wedge (\neg b_1 \wedge \neg b_3 \wedge \cdots \wedge \neg b_n) \Rightarrow$
  $(pre_3 \wedge b_2) \wedge (\neg b_1 \wedge \neg b_3 \wedge \cdots \wedge \neg b_n)$
- $post_6 \wedge (\neg b_1 \wedge \cdots \wedge \neg b_n) \Rightarrow$
  $pre_3 \wedge (\neg b_1 \wedge \cdots \wedge \neg b_n)$

3.6.4 Exception handling and method invocation

The changes in the flow of control that occur when an exception is thrown are currently being handled by generating suitable implications from the postcondition of the statement that threw the exception to the precondition of the statement that is first executed when the exception is caught. When the flow of control leaves a method, however, we have a similar problem as we had with method invocation. This problem is that the scope of the variables used in the assertions is limited. This suggests that we need a mechanism similar to the context switch to cope with this problem.

When you look somewhat closer, you will see that the problem is not only similar to method invocation, but actually caused by it! The flow of control can only leave a method after it has been called using the method invocation.

Since we mention the exceptions that can be thrown during the execution of a statement in the post-condition of this statement, it seems useful to state the exceptions that can occur within and leave a method (and their effects) in the post-condition of the corresponding method. This way the context switch, the basics of which we already have, can take care of the difficulties caused by the change of scope. The next picture (compare with the one from section 3.4.2) shows how we can use the context switch combined with exception handling.



What we need beside the verification conditions generated for a context switch with normal execution, are the ones for the exceptions that can occur within the method. Each exception that leaves the method is 'caught' in the post-condition and generates an implication to the matching[5] exception state in the original context. So basically, we use a similar detour to the context of the method to verify the throwing of exceptions in the original context as we used for normal execution of the invocation.

In the case that an exception is thrown, we neither perform the usual substitution that replaces the generic *result*-variable of the method with its return-expression nor the usual substitution that assigns the result of the method call to x. Namely, when the flow of control leaves the method because of an exception the method does not return a value and the assignment in the original context will be interrupted before x can get a new value.

[5] We implicitly assume, although it is not checked at the moment, that the same boolean exception variables are used in the post-condition of the method call as in the post-condition of the method.

Chapter 4

Case study: A verified example

4.1 Introduction

In this chapter an example will be given to give an idea of the actual implementation of the assertion mechanism and the work involved with using it. During the treatment of the example some extra information will be given that is needed when annotating a program and when using the rewrite system.

The example given will focus on inserting a new Node in a sorted linked list of Nodes, assuming that the new Node should not be inserted as the new head of the list in order to simplify the example. It is actually an advanced version of the example mentioned in [2].

4.2 Step one: Annotating the program

4.2.1 The program

We have the following definition of a Node:

```
class Node {
  Node next;
  int key;

  Node insert(int n) {
    Node cur;
    Node tmp;

    cur = this;
    while(cur.next != null && cur.next.key < n)
      cur = cur.next;

    tmp = new Node();
    tmp.key = n;
    tmp.next = cur.next;
    cur.next = tmp;

    return this;
  }
}
```

So a Node contains an instance variable *key* that contains an integer value and an instance variable *next* that points to the next Node in the list or is set to **null** if it is the last Node in the list. The method *insert* searches for the correct place to insert a new Node with value n and then inserts a newly created Node. The idea is that the *insert*-method is called on the head of the list, but this is not necessary.

We want to verify the following things:

- That the method *insert* inserts the new Node in the right place
- That the list remains sorted
- That the values of the elements of the list are not changed during the execution of the *insert*-method

To make the verification of the method easier these properties are verified separately. In this example only the first step will be treated, but the source of the other parts can be found in Appendix A.

4.2.2 The annotation

Programs using the assertion mechanism need to start with the header

/+AssertedJava+*/*

as a signal to the rewrite system that this is an annotated program that should be translated into verification conditions. For the assertion labels we use the syntax

/*+ [*label*] *assertion* +*/

The label is needed to distinguish the assertions generated in later steps and to enable better error reporting. Because of the notation, any ordinary Java compiler will consider the assertions comments and will just ignore them.

To make it easier to write and read the assertions, it is possible to define macros before declaring or defining classes. These macros can be used where any other assertion can be used and are defined as one big block of definitions of format

/*+
$\$name_1(par_{11},\cdots,par_{1n}) := assertion_1$
$\$name_2(par_{21},\cdots,par_{2m}) := assertion_2$
...
+*/

Each definition has a name that begins with the $-sign and can optionally have a list of untyped variable names, which can have the format of either temporary or logical variables. These will be substituted by their actual values when using the macro, via the ordinary substitution mechanism. So they will not be substituted by their textual occurrence in the corresponding assertion, but rather by their functional occurrence. The parameters can be left out altogether in the definition, in which case the parenthesis are also left out.

When we enter the *insert*-method we should be in a linked list and the key that is given should be larger than the key in the current object if we don't want the new object to form a new head of the list. So we need a precondition of the method, namely

```
/*+ [s] $linkedList(@z) && {n > this.key} +*/
```

Note that we have to make references to instance variables explicit using the prefix '**this.**' and that in this implementation logical variables have a '*@*' in front of them. In this example we have a logical variable '*@z*' that is of type 'sequence of objects' whose elements point to the elements of the list. *$linkedList* is a macro that states that the logical variable argument is a sequence of objects that refers to a linked list. It is defined as

```
  $linkedList(@z) := (@forall int @i (
                        ({1 <= @i} && {@i <= |@z|})
                        ==> {@z[@i] != (nil : Node)}
                             && ({@i < |@z|}
                                ==> {@z[@i].next == @z[@i+1]})
                      ))
                      && {@z[|@z|].next == (nil : Node)} && {this == @z[1]}
                      && {|@z| >= 1}
```

which states that every object in the sequence that is not the last should point to the next object in the sequence, no object in the sequence should be undefined (**null**), the last object in the sequence should be the last object in the list, the first object in the sequence should refer to the current object and that the sequence should contain at least one object. In the post-condition of the method we want to state that a Node with key n has been inserted in the right place (and that the method returns the current object, although this has nothing to do with the linked list). We use the assertion

```
/*+ [t] $addedKey(@z, n) && {RESULT == this} +*/
```

`RESULT` is a variable that is used in the precondition of a method that refers to the result of the method. Remember that logical variables are constants, so *@z* cannot have an index that refers to the new Node. The macro *$addedKey* is defined as

```
  $addedKey(@z, n) := @exists Node @Nd (
                        $addedToList(@z, @Nd)
                        && $notElem(@Nd, @z)
                        && {@Nd.key == n}
                      )
```

and states that a new Node with key *n* has been added to the original list in the correct place. The macros it uses are defined as

```
$notElem(elem, @z) := @forall int @i (
                        ({1 <= @i} && {@i <= |@z|})
                        ==> {elem != @z[@i]}
                      )

$addedToList(@z, tmp) := @exists int @i (
                           ({1 <= @i} && {@i <= |@z|})
                           && (({@z[@i].next == tmp})
                           && ({@i > 1} ==> {@z[@i].key < tmp.key})
                           && (({@i < |@z|}) ==> ({@z[@i+1].key >= tmp.key}
                                                  && {tmp.next == @z[@i+1]}))
                           && (({@i == |@z|}) ==> ({tmp.next == (nil : Node)}))
                         )
```

More specifically, *$addedToList* states that its second argument *tmp* is inserted in the list referred to by *@z* just after a Node with a smaller key and just before a Node with a key that is greater than or equal to *n* (if it is not appended to the end of the list). If it is appended it should become the new end of the list.

The rest of the assertions could be considered ‘filling’ that shows the path from the precondition of the method to the post-condition. First we present the rest of the assertions:

```
/*+ [s] $linkedList(@z) && {n > this.key} +*/
Node insert(int n /*+ Node [] @z +*/) {
  Node cur;
  Node tmp;

  /*+ [s] $linkedList(@z) && {n > this.key} +*/
  cur = this;
  /*+ [l1] $linkedList(@z) && $currentPos(cur, @z) +*/
  while(cur.next != null && cur.next.key < n)
    cur = cur.next;
  /*+ [l2] $linkedList(@z) && $correctPos(@z, cur) +*/
  tmp = new Node();
  /*+ [l3] $linkedList(@z) &&
          $correctPos(@z, cur) &&
          $notElem(tmp, @z) && {tmp != (nil : Node)}
  +*/
  tmp.key = n;
  /*+ [l4] $linkedList(@z) &&
          $correctPos(@z, cur) &&
          $notElem(tmp, @z) && {tmp.key == n} && {tmp != (nil : Node)}
  +*/
  tmp.next = cur.next;
  /*+ [l5] $linkedList(@z) && $correctPos(@z, cur) &&
          $notElem(tmp, @z) && {tmp.key == n} &&
          {tmp.next == cur.next} && {tmp != (nil : Node)}
  +*/
  cur.next = tmp;
  /*+ [l6] $addedToList(@z, tmp) && $notElem(tmp, @z)
          && {tmp.key == n} && {tmp != (nil : Node)} +*/
  return this;
}
/*+ [t] $addedKey(@z, n) && {RESULT == this} +*/
```

Notice that the logical variable *@z* is declared in the header of the method. Every free logical variable that is used in the method must be declared this way, using a comma-separated list, to make sure the types are known in advance. When calling the method, the variable *@z* will be given a value, just as *n* gets a value when the method is called.

The definitions of *$currentPos* and *$correctPos* are as follows

```
$currentPos(cur, @z) := @exists int @i (
                          ({1 <= @i} && {@i <= |@z|})
                          && ({cur == @z[@i]} &&
                              ({@i > 1} ==> {cur.key < n}))
                        )

$correctPos(@z, cur) := @exists int @i (
                          ({1 <= @i} && {@i <= |@z|})
                          && ({cur == @z[@i]} &&
                              ({@i > 1} ==> {cur.key < n})
                              && ({@i < |@z|} ==> {cur.next.key >= n}))
                        )
```

Of these macros *$currentPos* states that *cur* points to a position in the list that has a key smaller than *n* and *$correctPos* states that *cur* either points to the end of the list or to the Node after which the new Node should be inserted.

The meaning of the rest of the assertions is similar to the others that have been mentioned. See Appendix A for an overview of the asserted program as a whole. The source can also be found in the file **j2a/Split1.java**.

4.3 Step two: Generating the verification conditions

Now that we have written the annotations for the method we can continue the process by generating the verification conditions. The easiest way of doing this using this implementation is by using the Meta-Environment. Detailed information about its usage can be found in [12], but I will mention the steps needed to rewrite the annotated program.

First, enter the sub-directory **j2a** (Java to Assertion) and start the Meta-Environment by running '`meta`'. Open module **Main.sdf** using the *File→ Open Module*-menu. The panel will now show the import graph of module **Main**. Click on the box **j2a/Main** using the right button of your mouse, select *Edit term* in the menu that pops up and open file **Split1.java** (this file is not shown in the file list, because it does not have extension **.trm**).

In the XEmacs window containing **Split1.java** we can now parse the program using the menu *Term-actions→Parse*. Of course our program does not contain syntax-errors so after the parsing has been completed we can generate the intermediate verification conditions using the menu *Term-*

actions→Reduce, which will probably take about 5 to 10 minutes the first time we do this [1].

After a while, a window containing the result will pop up. If there are errors, the first word of the result will be `ErrorList` and the remainder will be a list of error messages. For us everything went as planned and the first word is `InstrList`, indicating that we generated the verification conditions.

The resulting list is of the form $[Decls, Instr_1, \cdots, Instr_n]$, where each $Instr_i$ is either of this form too or a labeled assertion. The first assertion, for example, is labeled *[s→l1]* which indicates that it is some kind of implication between the assertion labeled *[s]* and the assertion labeled *[l1]*. The nesting of lists matches the nesting of blocks in the program, which enables the next translation step to determine the scope of the declarations that can be found in the *Decls*-lists.

Now save the result in the directory **a2h** under the name **Split1.trm** and start a second version of the Meta-Environment from the directory **a2h** (Assertion to HOL). Now repeat the steps mentioned above, again opening module **Main.sdf** (but this time the one from directory **a2h**) and parsing and reducing term file **Split1.trm**. Save the result as **Split1.hol**.

4.4 Step three: Verifying the assertions in HOL

The file **Split1.hol** should be put in the directory that contains the files **OOScript.sml** and **OO_PRELUDE.sml**. If this is the first time you use these files you should run `Holmake` without any arguments in the directory to compile the theory *OO* contained in **OOScript.sml**. After this start the interpreter using the command '`hol.unquote Split1.hol`'

With the interpreter I use I get the following result:

```
-------------------------------------------------------------------
       HOL [Kananaskis 1 (built Thu Sep  5 14:11:06 2002)]

       For introductory HOL help, type: help "hol";
-------------------------------------------------------------------
<<HOL message: intLib loaded.  Use intLib.deprecate_int() to turn off integer
  parsing>>
<<HOL message: Created theory "Assertions">>
<<HOL message: Defined type: "NodeRec">>
<<HOL warning: Theory.new_constant: "[s->l1]" is not a standard constant name>>
<<HOL warning: Theory.new_constant: "[l1->l2]" is not a standard constant name>>
<<HOL warning: Theory.new_constant: "[l1->l1]" is not a standard constant name>>
<<HOL warning: Theory.new_constant: "[l2->l3]" is not a standard constant name>>
<<HOL warning: Theory.new_constant: "[l3->l4]" is not a standard constant name>>
<<HOL warning: Theory.new_constant: "[l4->l5]" is not a standard constant name>>
<<HOL warning: Theory.new_constant: "[l5->l6]" is not a standard constant name>>
<<HOL warning: Theory.new_constant: "[l6->t]" is not a standard constant name>>
[loading HOL power tools ************* ]
[closing file "/ufs/robbert/hol/tools/end-init-boss.sml"]
```

[1] On later runs this will take much shorter, but on the first run after starting the environment the rewrite rules have to be compiled.

This is hardly the place for an in-depth description of or a tutorial on using HOL, for which I can recommend [6], but I will give a *very* short description and explanation of the steps involved using one of the verification conditions.

Using the command `p();` we can request to see the first assertion, which shows us the following formula

```
- p();
> val it =
    Initial goal:

    ASSERTION [16->t]
      ((((?Li.
            (((SOME 1 ?<=? SOME Li = SOME T) /\
              (SOME Li ?<=? LEN (SOME Lz) = SOME T)) /\
             ((deref_ref (obj_at (SOME Lz) (SOME Li))
                 (NodeState (obj_at (SOME Lz) (SOME Li))).next =
               tmp) /\
              ((SOME Li ?>? SOME 1 = SOME T) ==>
               (deref_b (obj_at (SOME Lz) (SOME Li))
                  (NodeState (obj_at (SOME Lz) (SOME Li))).key ?<?
                deref_b tmp (NodeState tmp).key =
                SOME T))) /\
             ((SOME Li ?<? LEN (SOME Lz) = SOME T) ==>
              (deref_b (obj_at (SOME Lz) (SOME Li <+> SOME 1))
                 (NodeState (obj_at (SOME Lz) (SOME Li <+> SOME 1))).
                 key ?>=? deref_b tmp (NodeState tmp).key =
               SOME T) /\
              (deref_ref tmp (NodeState tmp).next =
               obj_at (SOME Lz) (SOME Li <+> SOME 1)))) /\
            ((SOME Li = LEN (SOME Lz)) ==>
             (deref_ref tmp (NodeState tmp).next = null))) /\
        !Li.
           (SOME 1 ?<=? SOME Li = SOME T) /\
           (SOME Li ?<=? LEN (SOME Lz) = SOME T) ==>
           ~(tmp = obj_at (SOME Lz) (SOME Li))) /\
        (deref_b tmp (NodeState tmp).key = SOME n)) /\ ~(tmp = null) ==>
       (?LNd.
          ~(LNd = null) /\
          ((?Li.
              (((SOME 1 ?<=? SOME Li = SOME T) /\
                (SOME Li ?<=? LEN (SOME Lz) = SOME T)) /\
               ((deref_ref (obj_at (SOME Lz) (SOME Li))
                   (NodeState (obj_at (SOME Lz) (SOME Li))).next =
                 LNd) /\
                ((SOME Li ?>? SOME 1 = SOME T) ==>
                 (deref_b (obj_at (SOME Lz) (SOME Li))
                    (NodeState (obj_at (SOME Lz) (SOME Li))).key ?<?
                  deref_b LNd (NodeState LNd).key =
                  SOME T))) /\
               ((SOME Li ?<? LEN (SOME Lz) = SOME T) ==>
                (deref_b (obj_at (SOME Lz) (SOME Li <+> SOME 1))
                   (NodeState (obj_at (SOME Lz) (SOME Li <+> SOME 1))).
                   key ?>=? deref_b LNd (NodeState LNd).key =
                 SOME T) /\
                (deref_ref LNd (NodeState LNd).next =
                 obj_at (SOME Lz) (SOME Li <+> SOME 1)))) /\
              ((SOME Li = LEN (SOME Lz)) ==>
               (deref_ref LNd (NodeState LNd).next = null))) /\
           !Li.
             (SOME 1 ?<=? SOME Li = SOME T) /\
             (SOME Li ?<=? LEN (SOME Lz) = SOME T) ==>
             ~(LNd = obj_at (SOME Lz) (SOME Li))) /\
          (deref_b LNd (NodeState LNd).key = SOME n)) /\ (this = this))

     : goalstack
-
```

which is the result of the implication from assertion *l6* to assertion *t* under the substitution [**this**/`RESULT`]. It looks somewhat complicated, because we use special HOL definitions to deal with some semantic properties. The assertions are placed in order onto the goal-stack of HOL, which is why we encounter the last implication first.

Look more closely at the formula. It is a very large implication of the form $\exists Li\ (A) \Rightarrow \exists LNd\ (\neg(LNd = (nil : Node)) \wedge \exists Li\ (B))$, where we almost have $A = B[temp/LNd]$. This suggests that if we tell HOL to fill in *temp* for *LNd* in the existential quantifier and that it should consider the same *Li* in the other existential quantifiers, the system may be able to verify the easy remains for us.

How can we do this? First we strip the label `ASSERTION [l6->t]` from the formula using the command '`e (REWRITE_TAC [ASSERTION_def]);`', supplying HOL with just enough information to perform this task. Then we tell HOL to break up the implication using '`e STRIP_TAC;`', which gives us direct access to the first existential quantifier in the right-hand side of the implication. Now we can specify that we want to instantiate *LNd* with the value *tmp*, providing the type of *tmp* as well, using '`e (EXISTS_TAC (Term ‘tmp :Node object‘));`' and can split up the resulting conjunction using '`e STRIP_TAC;`'.

The formula `~(tmp = null)` is trivial to prove because at this stage it is already in the assumption list, so HOL can prove it using its built-in prover *RW_TAC* which we can call using '`e (RW_TAC int_ss []);`'. The other subgoal we split up again, and its first result again to get direct access to the other quantification. Now we tell HOL that it should use the *Li* that we had on the left-hand side of the original implication using '`e (EXISTS_TAC (Term ‘Li : int‘));`'.

All the remaining subgoals are trivial enough for HOL, so we can solve them using '`e (OO_RW_TAC int_ss []);`' once for each subgoal [2].

Now we finally get the highly desirable message '`Initial goal proved.`' on top of the goal, which means we are done verifying this transition. We can now `drop();` the goal and continue with the next one. We repeat this process until we are done proving all goals and have thus verified the program.

[2] When we call *OO_RW_TAC*, we actually call *RW_TAC*, but supply it with all the information from theory *OO*.

Chapter 5

Discussion

5.1 Introduction

This chapter discusses some of the aspects of the implementation. The first section discusses the use of ASF+SDF for this project and the second one gives the advantages and disadvantages of using the assertion language itself for the description of the verification conditions. Then something will be told about our experience with HOL, after which the chapter will be concluded with a summary of the contributions that have been made while working on this project.

5.2 Advantages and limitations of using ASF+SDF

Using ASF+SDF is very similar to using a functional programming language, but its level of abstraction is even higher. Functional programming languages abstract from imperative languages by emphasizing evaluation of expressions over execution of commands. ASF+SDF abstracts from functional programming languages by putting the emphasis on subterm replacement, which could be seen as a special form of evaluation.

Just like functional languages, ASF+SDF has the advantage that definitions look very clear and natural. In ASF+SDF it is even easier to make the look of the definitions match the corresponding theory, as you can choose your own syntax. In my implementation for example, I chose to make the syntax look like that of an imperative language with function calls, but it would be very well possible to make the substitution mechanism look like the mathematical definitions given in Chapter 2. Furthermore, as with functional programming, the programmer can totally neglect memory usage when writing a specification.

It is important to note that, despite the similarities, ASF+SDF is *not* a functional programming language. In fact, it is not even a programming language, but just a formalism used for specifying syntactic and semantic transformations. It is exactly this property that makes it so extremely useful for specifying program transformations (which are all about syntactic and semantic transformation). However, the support for exception handling is nearly absent in ASF+SDF.

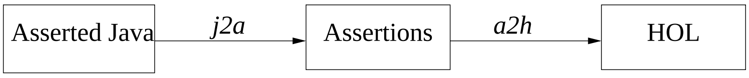
For instance, interrupting the rewrite process to give an appropriate error message is not possible. Generating user-defined error messages can only be done by rewriting the program to the text of the messages, which is not always a very easy thing to do. It usually involves adding extra rules to extract the text describing the errors found from the rest of the result.

Another example of what is not supported by ASF+SDF is the premature termination of the rewrite process in case of the detection of a fatal error.

5.3 Choice of intermediate language

5.3.1 Introduction

As already mentioned in the introduction, the rewrite process that has been implemented during this project consists of two steps:



In the middle of the process, we store the assertions using the assertion language. This section will discuss the pros and cons of using the assertion language for this purpose.

5.3.2 The intermediate language

At least three (probably different) languages play a role in the rewrite process:

1. **The *assertion* language**
   The language used to express assumptions about the program.

2. **The object-oriented *source* language**
   The language used to describe the program. It contains the assertion labels.

3. **The *target* language**
   The language that is readable by the chosen theorem prover. It is used to express the assumptions made in (hopefully) provable formulas.

Because a translation from the assertion language to the target language is usually known, the natural way to do the translation is to first translate a 'precondition - instruction - post-condition' combination (an *assumption* about the program) to a single assertion and then do a translation to the target language.

In the original tool this approach has not been taken. First the assertions are recursively translated into Java objects. Then the combining substitutions are applied to the objects, possibly creating new objects, and then the resulting objects are translated into the target language. This basically adds another language to the rewrite process, the *intermediate* language.

In our approach, the process consists of the before-mentioned two steps. We also use an intermediate language, but in this implementation it is the assertion language itself. One of the reasons for this is that it is assumed that the assertion language should be sufficiently expressive in order to state interesting assumptions about the program. If some assumptions can not be translated into the assertion language itself, it is very likely that the language used is too weak anyway and that an extension (or a completely different language) should be used instead.

5.3.3 Advantages of using the assertion language as intermediate language

There are also several more practical reasons for doing a translation to the assertion language in the first stage, instead of to another chosen language. These are the following:

- The substitutions used, as described in section 2.5, are defined in the context of the assertion language. Translating directly to another intermediate language in the first stage would require redefining the substitutions for this new language, while translating indirectly to another intermediate language would just introduce an extra translation stage without any additional advantages.
- We already have a translation of annotated Java to the assertion language and an easy translation of the assertion language to HOL-syntax. Using a new intermediate language would require defining a new mapping of annotated Java to this language, as well as defining the mapping from the new intermediate language to HOL-syntax.

5.3.4 Disadvantages of using the assertion language as intermediate language

Doing a direct translation to the assertion language when the assertion language is still under development has a strong disadvantage.

When this is the case, the assertion language will change syntax very often. As described in the previous section ASF+SDF is a formalism for describing syntax transformations and because of this the specifications written in it depend heavily on the precise format of the syntax used. When the structure of the assertion language is changed, this requires that the corresponding parts in the substitution mechanism, the translation of annotated Java to the assertion language and the translation to HOL-syntax are changed accordingly. This can be a lot of work, depending on how much the structure of the assertion language has been changed.

It is not problematic to add constructs to the assertion language in the current situation. This does require adding new rules for the translation to HOL-syntax and may require adding new rules to the substitution mechanism and the translation of annotated Java, but adding new rules is pretty straightforward in most cases.

5.4 Choice of theorem prover

Although so far relatively little has been said about HOL, the theorem prover used, it plays an essential role in the verification mechanism. After all it is in HOL where is decided whether the annotated transitions can be proven correct! We will also briefly introduce PVS, as we expect to achieve a higher level of automation using the latter.

5.4.1 About PVS

PVS ([24]) stands for "Prototype Verification System". It consists of a specification language integrated with support tools and a theorem prover. PVS tries to provide the mechanization needed to apply formal methods both rigorously and productively.

PVS works with specifications, rather than directly inserted goals, which have a closer resemblance to the functional way of programming. It also provides very convenient techniques for verifying the consistency of specifications. This is especially useful if at some point we decide to do some verification on the underlying logical definitions that we use in the theorem prover to express our assertions.

5.4.2 Discussion

My personal experience is that it takes a lot of time to learn how to maintain the right level of abstraction if you use your own definitions in HOL. Without the right level of abstraction subgoals can get cluttered up with unnecessary lower-level information. Such subgoals are usually longer formulas, unintuitive and typically have a different notation than the formula you entered in the first place [1].

Despite the simplicity of the assertions and the annotation method used, most resulting formulas, especially longer ones, are rather difficult to prove

[1] Pieces of which typically end up in the subgoals' assumptions.

in HOL. This is mainly because the proofs have to be 'generated' in a very formal and explicit way, as to ensure the correctness of the proof. It is very likely that using a different theorem prover, for example PVS, enables us to make (most of) the verification process automatic instead of manual.

When describing the case study for example, I only provided the proof for one of the verification conditions.[2] Compared with the other proofs I had this was a very short one, although I feel it should have been much shorter. It shouldn't even take an extremely smart theorem prover to figure out that if only one free variable of type Node is used in a formula that also has an existential clause with a bound variable of type Node that it would be probably be a good idea to try filling in this free variable. HOL does not do this for you, although there may be a way to program this behavior in a user-defined tactic.

5.5 Contributions

To emphasize what the contributions of this project have been, I will give a summary in this section. My contributions are:

- *Implementing the interpretation of annotated Java.* My work roughly involved building a mechanism that extracts type-information from a Java class, building the substitution mechanism for the assertion language and specifying the translation process of basic annotated Java to the assertion language. New in the last translation process was the specific interpretation of compound statements such as if-then, if-then-else and while.

- *Implementing method invocation.* There were some ideas on how to handle the assertions in the context of method invocation, but they had not been implemented yet. I did this for Java.

- *Constructor invocation.*
  After discussing constructor invocation with Frank de Boer we came up with a way to interpret constructor invocation in Java using the principles of method invocation. This has been included in the implementation as well, in the situations where non-default constructors are invoked during object creation.

- *Exception handling.*
  Frank de Boer had the suggestion of doing exception handling using boolean exception variables. I worked this idea out and implemented it for Java.

- *Macro mechanism implementation for Java.*

[2] Although proofs *are* available for all of the verification conditions.

Cees Pierik already had a macro mechanism for use with flowcharts, but I decided to implement my own version for use with Java classes.

- *Bug fixes in translation to HOL, improvement HOL-definitions.*
  Most of the translation to HOL-syntax came from Cees Pierik's tool. There were some important bugs in the translation however which have been corrected. Furthermore, I have adjusted some of the definitions used in HOL to make the verification process in HOL more logical. Because this implementation generates all verification conditions at once instead of for one selected transition at a time, I have added a definition to the theory that enabled the tool to label the assertions in HOL.

- *Reporting some bugs in the ASF+SDF Meta-Environment.*
  While working with the ASF+SDF Meta-Environment I encountered some bugs, which have been reported to and fixed by the ASF+SDF Meta-Environment support team.

- *Correcting the linked list example, improving it, proving it in HOL.*
  The example worked out in Chapter 4 is based on the example from [2]. This example required some corrections in order to be used in practice and was rather limited in what it proved about the linked list. The corrected version of the example has been improved and implemented as a Node-object in Java that has an insert-method. This insert-method has been verified using this tool, which took me two to three months as I needed to learn how to work with HOL as well.

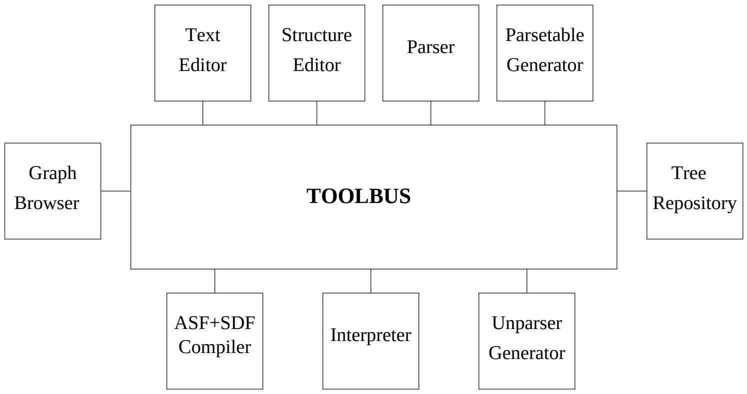
Chapter 6

Future work

6.1 Integration with existing tools/programming environments

6.1.1 Overview

In Chapter 4 it was shown how the Meta-Environment can be used to generate the verification conditions using the ASF+SDF-modules written during this project. As is briefly described in section 1.4.1 and 1.4.2, the Meta-Environment is actually a collection of tools connected using a tool called ToolBus ([8]).



To use the ASF+SDF-modules we only need a couple of these tools. These are the ones mentioned in section 1.4.1 (*sglr*, *asfe* and *unparsePT*).

In the Meta-Environment a special script, called *ToolBus script*, describes how the interaction between the tools should go. The best way to integrate the tools needed with an existing tool would be to add a ToolBus interface to the existing tool and write a script that describes how the communication between the tools should be. For specific information on how to write these scripts see [8].

6.1.2 About sglr, asfe and unparsePT

In this section we will describe the tools that we need for the rewrite process, in the order in which they are usually invoked.

sglr is a parser tool that uses a parse table to parse an input file. We can generate the parse tables for the tools of this project using the tool *pt-dump* with the Main-modules. The tool *pt-dump* is provided with the Meta-Environment.

After a successful parse, the resulting parse tree is written to an output file.

asfe is a tool that rewrites an input tree according to a parsed ASF specification. We can parse the ASF specifications from this tool by using the tool *eqs-dump* with the Main-modules and will usually input the parse tree resulting from the application of *sglr*.

After the rewrite, the result will be written to an output file formatted as a tree.

unparsePT is a tool that rewrites an input tree to the corresponding text string. It is used to write the output tree resulting from the application of *asfe* to an ordinary text file.

6.1.3 Notes

In order to connect the tools using a ToolBus script, the ToolBus itself is needed as well. Unfortunately, the tools mentioned above and the ToolBus are currently only available for Unix/Linux-based systems. So in order to use these tools on a machine running one of the Windows versions, it will be necessary to install Cygwin ([19]).

Instead of using the ToolBus and a ToolBus script to connect the tools from this project to an existing tool, there is also the option of executing the tools described in the previous section using system calls and redirecting the input and the output. This works fine if the program does not contain syntax errors or semantic errors, but makes it harder to handle the cases when it does contain them properly.

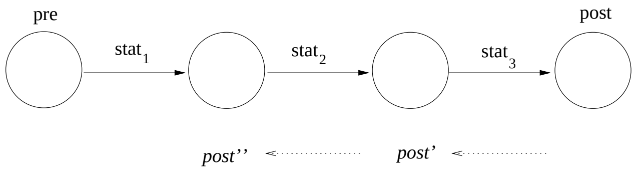
6.2 "Filling in the blanks"

Our tool more or less requires all statements in a block or a method to be annotated. In the theory of flowcharts, a non-annotated Java-statement would correspond with a flowchart that has a number of non-annotated states.

As a lot of the transitions in a block are trivial, as are many of the generated verification conditions, it would be nice of we could leave trivial parts of the block without annotation and let some tool insert the missing ones. Because the missing annotations would be fitted in by a tool we wouldn't need to generate verification conditions for them. This would save work when proving the conditions in the end, while we could still verify whether the precondition of the block implies the post-condition after executing the statements in the block.

We could fill in the gaps calculating weakest preconditions. We have already developed the machinery for this, as we already used weakest preconditions to move post-conditions to the state before the corresponding transition.

The idea is illustrated using the picture below, in which we have depicted a block of statements using a flowchart.



The first state has the precondition of the block, while the last one has its post-condition. Calculating the weakest precondition of *post* we get *post′*, which can be used to annotate the third state. Repeating this process with *post′* we get annotation *post″* for the second state. This would just leave us with the verification of the first transition, as we know for sure that the other ones are correct.

A tool that does the insertion could be used as a pre-processor on the program text before the tool *j2a* is applied to it or the insertion process could be included as a preliminary phase in the tool itself.

6.3 Extending the subset of Java

The tool only supports a very limited part of Java. A natural way to improve the tool would be to provide additional transformation rules for control structures or expressions.

Depending on the nature of these structures or expressions it could be necessary to extend the assertion language as well.

Bibliography

[1] W.-P. de Roever, F. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge University Press, 2001.

[2] F.S. de Boer and C. Pierik. *Towards an environment for the verification of annotated object-oriented programs*. Technical report UU-CS-2003-002, Utrecht University, 2003.

[3] Krzystof R. Apt. *Ten Years of Hoare's Logic: A Survey - Part I.* ACM Transactions on Programming Languages and Systems, Vol.3, No.4, October 1981, Pages 431-483.

[4] J. Gosling, B. Joy and G. Steele. *The Java language specification.* Available at `http://java.sun.com/`.

[5] The HOL theorem-proving system homepage. Available at `http://sourceforge.net/projects/hol/`.

[6] Cambridge Research Centre of SRI International. *The HOL System: Description* and *The HOL System: Tutorial.* Available from [5].

[7] The Java Modeling Language (JML) homepage. Available at `http://www.cs.iastate.edu/~leavens/JML/`.

[8] The ToolBus project homepage. Available at `http://www.cwi.nl/htbin/sen1/twiki/bin/view/SEN1/ToolBus`.

[9] P. Klint. *A Guide to ToolBus Programming.* April 22, 2002. Available from [8].

[10] J.A. Bergstra and P. Klint. *The Discrete Time ToolBus - a software coordination architecture.* Science of Computer Programming 31(2-3):205-229, 1998.

[11] The ASF+SDF Meta-Environment homepage. Available at `http://www.cwi.nl/projects/MetaEnv`.

[12] The ASF+SDF Meta-Environment manual. Available from [11].

[13] M.G.J. van den Brand, P. Klint and J.J Vinju. *Term Rewriting with Type-safe Traversal Functions.* In: B. Gramlich and S. Lucas (eds), Second International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2002), Electronic Notes in Theoretical Computer Science, Volume 70, issue 6, Elsevier Science Publishers, 2002.

[14] The LOOP project homepage. Available at `http://www.cs.kun.nl/~bart/LOOP/`.

[15] M. Huisman, B. Jacobs. *Java Program Verification via a Hoare Logic with Abrupt Termination.* In: T. Maibaum (ed), Fundamental Approaches to Software Engineering (FASE'00), Springer LNCS 1783, p.284-303, 2000.

[16] B. Jacobs. *A Formalisation of Java's Exception Mechanism.* In: D. Sands (ed.), Programming Languages and Systems (ESOP), (Springer LNCS 2028, 2001), p.284-301.

[17] M. van den Brand and M. de Jonge. *Pretty-Printing within the ASF+SDF Meta-Environment: A Generic Approach.* Report SEN-9904 March 1999, CWI.

[18] M. de Jonge. *Pretty-printing for software reengineering.* Report SEN-R0214 August 2002, CWI.

[19] The Cygwin homepage. Available at `http://www.cygwin.com/`.

[20] J.A. Bergstra, J. Heering and P. Klint (eds). *Algebraic Specification.* ACM Press/Addison-Wesley, 1989.

[21] M.G.J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. *The ASF+SDF Meta-Environment: a Component-Based Language Development Environment.* In: R. Wilhelm (ed). Proceedings of Compiler Construction (CC'01), LNCS 2027, 365–370, 2001.

[22] P. Klint. *A Meta-Environment for Generating Programming Environments.* ACM Transactions on Software Engineering and Methodology, 2(2):176-201,1993.

[23] A. van Deursen, J. Heering and P. Klint. *Language Prototyping: An Algebraic Specification Approach.* Volume 5 of AMAST Series in Computing, World Scientific, Singapore, 1996.

[24] The PVS Specification and Verification System homepage. Available at `http://pvs.csl.sri.com/`.

Appendix A

The example: source code

This chapter gives the full annotated source code of the example program described in Chapter 4. As the example has been split up in three parts in order to simplify the proof process, this chapter has been divided in three sections as well, each describing a different part of the annotation of the example.

A.1 Part one

A.1.1 Description

The first version of the annotation checks whether a new node with key n has been inserted in the right place, assuming that the keys and order of original linked list have not been altered and that the original part of the linked list (the part referred to by logical sequence @z) remains ordered during execution of the method.

A.1.2 Source code

```
/*+AssertedJava+*/

/*+
  $linkedList(@z) := (@forall int @i (
                       ({1 <= @i} && {@i <= |@z|})
                       ==> {@z[@i] != (nil : Node)}
                           && ({@i < |@z|}
                              ==> {@z[@i].next == @z[@i+1]})
                     ))
                     && {@z[|@z|].next == (nil : Node)} && {this == @z[1]}
                     && {|@z| >= 1}

  $currentPos(cur, @z) := @exists int @i (
                            ({1 <= @i} && {@i <= |@z|})
                            && ({cur == @z[@i]} &&
                                ({@i > 1} ==> {cur.key < n}))
                          )

  $correctPos(@z, cur) := @exists int @i (
                            ({1 <= @i} && {@i <= |@z|})
                            && ({cur == @z[@i]} &&
                                ({@i > 1} ==> {cur.key < n})
                                && ({@i < |@z|} ==> {cur.next.key >= n}))
                          )

  $notElem(elem, @z) := @forall int @i (
                          ({1 <= @i} && {@i <= |@z|})
                          ==> {elem != @z[@i]}
                        )

  $addedToList(@z, tmp) := @exists int @i (
                             ({1 <= @i} && {@i <= |@z|})
                             && (({@z[@i].next == tmp})
                             && ({@i > 1} ==> {@z[@i].key < tmp.key})
                             && (({@i < |@z|}) ==> ({@z[@i+1].key >= tmp.key}
                                                   && {tmp.next == @z[@i+1]}))
                             && (({@i == |@z|}) ==> ({tmp.next == (nil : Node)}))
                           )

  $addedKey(@z, n) := @exists Node @Nd (
                        $addedToList(@z, @Nd)
                        && $notElem(@Nd, @z)
                        && {@Nd.key == n}
                      )
  +*/

class Node {
  Node next;
  int key;

  /*+ [s] $linkedList(@z) && {n > this.key} +*/
  Node insert(int n /*+ Node [] @z, int [] @n +*/) {
    Node cur;
    Node tmp;

    /*+ [s] $linkedList(@z) && {n > this.key} +*/
    cur = this;
    /*+ [l1] $linkedList(@z) && $currentPos(cur, @z) +*/
    while(cur.next != null && cur.next.key < n)
      cur = cur.next;
    /*+ [l2] $linkedList(@z) && $correctPos(@z, cur) +*/
    tmp = new Node();
    /*+ [l3] $linkedList(@z) &&
             $correctPos(@z, cur) &&
             $notElem(tmp, @z) && {tmp != (nil : Node)}
    +*/
    tmp.key = n;
    /*+ [l4] $linkedList(@z) &&
             $correctPos(@z, cur) &&
             $notElem(tmp, @z) && {tmp.key == n} && {tmp != (nil : Node)}
    +*/
    tmp.next = cur.next;
    /*+ [l5] $linkedList(@z) && $correctPos(@z, cur) &&
             $notElem(tmp, @z) && {tmp.key == n} &&
             {tmp.next == cur.next} && {tmp != (nil : Node)}
    +*/
    cur.next = tmp;
    /*+ [l6] $addedToList(@z, tmp) && $notElem(tmp, @z)
             && {tmp.key == n} && {tmp != (nil : Node)} +*/
    return this;
  }
  /*+ [t] $addedKey(@z, n) && {RESULT == this} +*/
}
```

A.2 Part two

A.2.1 Description

The second version of the annotation checks whether the original part of the linked list remains ordered during execution of the method.

A.2.2 Source code

```
/*+AssertedJava+*/

/*+
  $sortedList(@z) := @forall int @i (
                      ({1 <= @i} && {@i <= |@z|})
                      ==> {@z[@i] != (nil : Node)}
                          && ({@i < |@z|}
                             ==> {@z[@i].key <= @z[@i+1].key})
                     )

  $notElem(elem, @z) := @forall int @i (
                          ({1 <= @i} && {@i <= |@z|})
                          ==> {elem != @z[@i]}
                        )
  +*/

class Node {
  Node next;
  int key;

  /*+ [s] $sortedList(@z) && {n > this.key} +*/
  Node insert(int n /*+ Node [] @z, int [] @n +*/) {
    Node cur;
    Node tmp;

    /*+ [s] $sortedList(@z) +*/
    cur = this;
    /*+ [l1] $sortedList(@z) +*/
    while(cur.next != null && cur.next.key < n)
      cur = cur.next;
    /*+ [l2] $sortedList(@z) +*/
    tmp = new Node();
    /*+ [l3] $sortedList(@z) && $notElem(tmp, @z)
    +*/
    tmp.key = n;
    /*+ [l4] $sortedList(@z) && $notElem(tmp, @z) +*/
    tmp.next = cur.next;
    /*+ [l5] $sortedList(@z) && $notElem(tmp, @z) +*/
    cur.next = tmp;
    /*+ [t] $sortedList(@z) && $notElem(tmp, @z) +*/
    return this;
  }
  /*+ [t] $sortedList(@z) && {RESULT == this} +*/
}
```

A.3 Part three

A.3.1 Description

The third version of the annotation checks whether the keys of the nodes of the original linked list remain the same during execution of the method.

A.3.2 Source code

```
/*+AssertedJava+*/

/*+
  $containsIntsFrom(@z, @n) := {|@z| == |@n|} &&
                               @forall int @i (
                                 ({1 <= @i} && {@i <= |@n|})
                                 ==> {@n[@i] == @z[@i].key}
                               )

  $notElem(elem, @z) := @forall int @i (
                          ({1 <= @i} && {@i <= |@z|})
                          ==> {elem != @z[@i]}
                        )
  +*/

class Node {
  Node next;
  int key;

  /*+ [s] $containsIntsFrom(@z, @n) && {n > this.key} +*/
  Node insert(int n /*+ Node [] @z, int [] @n +*/) {
    Node cur;
    Node tmp;

    /*+ [s] $containsIntsFrom(@z, @n) +*/
    cur = this;
    /*+ [l1] $containsIntsFrom(@z, @n) +*/
    while(cur.next != null && cur.next.key < n)
      cur = cur.next;
    /*+ [l2] $containsIntsFrom(@z, @n) +*/
    tmp = new Node();
    /*+ [l3] $containsIntsFrom(@z, @n) &&
             $notElem(tmp, @z)
    +*/
    tmp.key = n;
    /*+ [l4] $containsIntsFrom(@z, @n) &&
             $notElem(tmp, @z)
    +*/
    tmp.next = cur.next;
    /*+ [l5] $containsIntsFrom(@z, @n) &&
             $notElem(tmp, @z)
    +*/
    cur.next = tmp;
    /*+ [t] $containsIntsFrom(@z, @n) +*/

    return this;
  }
  /*+ [t] $containsIntsFrom(@z, @n) && {RESULT == this} +*/
}
```