Towards an environment for the verification of annotated object-oriented programs

Frank S. de Boer

Cees Pierik

institute of information and computing sciences, utrecht university

technical report UU-CS-2003-002

www.cs.uu.nl

Towards an environment for the verification of annotated object-oriented programs

Frank S. de Boer and Cees Pierik
Institute of Information and Computing Sciences
Utrecht University, The Netherlands
{frankb, cees}@cs.uu.nl

version: 1.0b

29 January 2003

Abstract

The main contribution of this paper consists of a description and formal justification of a tool which supports the specification and verification of a class of flowcharts that captures the basic dynamics of object-oriented programs. The computer-aided specification and verification involves the annotation of a flowchart with assertions and the automatic generation of the corresponding verification conditions. As such it forms a front-end tool of a theorem prover which is used to check the verification conditions interactively. To use the front-end tool for a specific theorem prover, one only needs to translate the semantics of the assertion languages. In this paper such a translation is given for the HOL theorem prover. The semantics of the flowcharts is axiomatized by the verification conditions which are formulated in terms of a weakest precondition calculus.

Contents

Chapter 1

Introduction

In recent years object-oriented languages have been widely used for many purposes including distributed and network programming. Object-oriented technology facilitates the development of complex systems and is already extensively supported by advanced tools that developers use to write, compile, run and debug programs.

Formal methods, that is, mathematically founded techniques for precise specification and rigorous verification, are necessary to obtain *reliable* software. The actual application of formal methods however is seriously hampered because of the many tedious calculations it involves. Therefore, to apply formal methods we need tools which support the *computer-aided* specification and verification of software. More specifically, this means using a computer, for increased speed and reliability, to carry out the tedious steps of formal verification.

The main contribution of this paper consists of a description and formal justification of a tool which supports the computer-aided specification and verification of a certain class of flowcharts that captures the basic dynamics of object-oriented programs. Characteristic of the execution of an object-oriented program is that it gives rise to complicated and dynamically evolving structures of references between objects because (1) objects can be created at arbitrary points during the execution of a program, and (2) references to objects (pointers), can be stored in variables and passed around.

Our tool is based on an implementation of a *weakest precondition* calculus for reasoning about the semantics of basic assignments in object-oriented programs. The calculus itself is formulated in terms of an assertion language which allows the description of properties of dynamic configurations of objects at an *abstraction level* that coincides with that of the programming language. This means that the only operations on pointers are testing for equality and dereferencing (looking at the value of an instance variable of the referenced object). Furthermore, it is only possible to mention the objects that exist in that configuration. Objects that do not (yet) exist never play a role. The abstraction level of the assertion language requires an explicit account of *aliasing* in structures of references between objects. Furthermore, in the case of the creation of a new object we have to solve the problem that in the state before its creation we cannot refer to the new object because it does not exist yet. We show that this problem can be solved by a contextual analysis of occurrences of references to the new object in the postcondition. The creation of objects also requires an explicit account of the changing scope of quantifiers: since the range of a quantifier is limited to the existing objects it is affected by the creation of a new object.

The input of the tool is a flowchart together with a mapping which associates an assertion to every location (an annotated flowchart). An assertion associated with a location is intended to describe certain invariant properties of the set of object-configurations which are reachable at that location by a computation of the flowchart ([17]). To prove that the assertions of an annotated flowchart are indeed satisfied in the sense described above, it suffices to check the logical validity of a (finite) set of assertions which are automatically generated by an application of the weakest precondition calculus to the annotated flowchart. The validity of these assertions then can be interactively verified by the theorem prover HOL ([11]) in terms of an internal representation of

the assertion language.

This paper is organized as follows. In the following chapter we describe the architecture of the tool and the way it is used. In Chapter 3 we describe the formalism and formal semantics of flowcharts. In Chapter 4 we introduce the assertion language and its semantics. It is shown how to describe and reason about aliasing and object-creation in terms of a weakest precondition calculus. Chapter 5 describes the representation of the assertion language in the logic of the HOL system. Chapter 6 discusses a correctness proof of the insert operation in a (sorted) linked list. The last chapter contains some concluding remarks.

Chapter 2

The verification tool

In this chapter, we describe the architecture of the tool and the way it is used. Figure 2.1 displays a graphic description of this architecture. This description shows the flow of data through the system. The input of the tool consists of a flowchart, its specification, a class library, and a library of macro definitions. These components are described below.

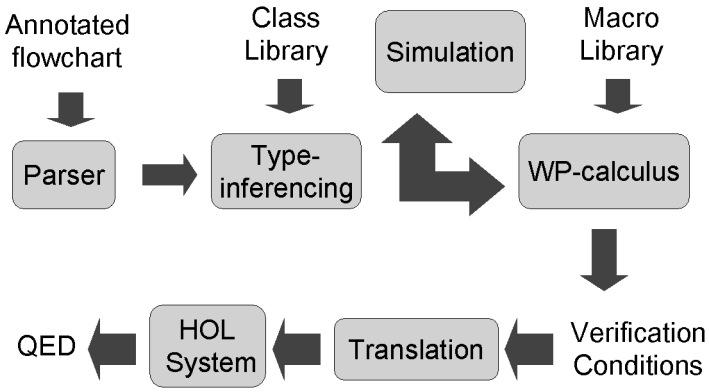

Figure 2.1: The architecture of the tool

The verification tool is implemented in Java. The tool contains lexical analyzers and parsers for annotated flowcharts, macros and class descriptions, which are obtained by means of the lexical analyzer generator JLex [12] and the parser generator CUP [6].

The tool uses a library of class descriptions. A class description consists of the name of the class, a list of instance variables and their types. The class descriptions are used to infer the types of instance variables that occur in the flowchart. Moreover, they provide the information needed to define the types that enable the representation of objects in the HOL logic. Chapter 5 gives a description of these type definitions.

Flowcharts themselves are drawn in the user interface of the tool and are shown graphically. They can be modified by mouse movements. Conditions and assignments can be entered for every transition in the flowchart. A simple form of type-inferencing is used to infer the type of every variable that is not explicitly typed. The user is expected to include type information for every temporary variable that occurs in a condition or assignment, if necessary. It suffices to state the type of every variable once.

Assertions can be assigned to every location of the flowchart. The use of (parameterized) macros from a library in these assertions is supported. In Chapter 6 we give an example of the use of (parameterized) macros.

The verification tool is then used to automatically generate the verification conditions and export them to a file. This requires the tool to compute the substitution operations that are defined in Chapter 4 and translate the resulting verification conditions to the HOL syntax. These two phases are completely automated. The resulting file can be loaded into HOL. The final proof of correctness consists of proving the correctness of the verification conditions one-by-one and is constructed during an interactive proof-session with the HOL system. The user is also enabled to view the simulation of a flowchart at a separate panel.

Chapter 3

Flowcharts

In this chapter, we describe the formalism and semantics of the class of strongly typed flowcharts which is currently supported by the verification tool. We assume a set $\mathcal{C}$ of *class names*. The set of basic types $\mathcal{B}$ is obtained by extending this set of class names with the types Int and Bool. For each basic type $B \in \mathcal{B}$ we denote by B^* the type of all finite sequences of elements of type B. The set of all types $\mathcal{T}$, with typical element t, is defined as $\mathcal{B} \cup \{B^* \mid B \in \mathcal{B}\}$.

For each of the types in $\mathcal{T}$ we assume a set of *instance variables* $x, y, \ldots$ and a set of *temporary variables* $u, v, \ldots$, i.e. the contents of such a variable will be a value of that type. For notational convenience only, we will often leave the typing information implicit. Every object belongs to a (unique) class, which contains data and procedures (*methods*) acting on these data. The data of an object is stored in instance variables, whose lifetime is the same as that of the object, and in temporary variables, which are local to a method and last as long as the method is active.

A flowchart describes the control flow of the methods of an object. In this paper, we consider flowcharts which consist of basic *guarded assignments*. These guarded assignments are constructed from the following set of expressions without *side-effects*, with typical element e:

$$e ::= u \mid e.x \mid \mathsf{op}(e_1, \ldots, e_n).$$

Here, u is a temporary variable, and x is an instance variable, and op is an operation (e.g. equality, multiplication, append, ...). If $n = 0$, op is a constant. In particular, we assume the presence of the constants self and nil, which denote the active object and the value $\perp$ respectively. The latter value stands for 'undefined' or 'uninitialised'.

Expressions of the form $e.x$ are used to refer to the value of the instance variable x of the object denoted by e. In the expression $e.x$ the type of the expression e is assumed to be C, for some class $C \in \mathcal{C}$. However, the evaluation of such an expression might also result in the value $\perp$. This occurs, for example, when we are dealing with a pointer which does not refer to an object, i.e., which has the value $\perp$. It is worthwhile to observe that indeed we have to consider this case because the instance variables of a newly created object that are used as pointers are initialised to the value of nil. Dereferencing such a nil-pointer will also result in the value $\perp$. The only other operation on objects is testing for equality.

A guarded assignment consists of a boolean expression e followed by an assignment. We have the following assignments $e.x := e'$ to an instance variable x and assignments $u := e$ to a temporary variable u. The execution of an assignment $e.x := e'$ consists of assigning the value of e' to the instance variable x of the object denoted by e. On the other hand, the execution of an assignment $u := e$ consists of assigning the value of e' to the temporary variable u.

It is worthwhile to observe that an assignment of the form $e.x[i] := e'$, with x being an array variable, can be modelled by an assignment $e.x := \mathsf{op}(e.x, i, e')$, where op is an operation which produces an array obtained from the array $e.x$ by assigning to the ith element the value of e' (see also [8]). We do not enforce such operators to preserve the length of an array (as one would expect of an assignment $e.x[i] := e'$). A single array variable thus can denote arrays of different length.

Object creation is realised by assignments of the form $u := \mathsf{new}$. The execution of an assignment $u := \mathsf{new}$ consists of the creation of a new object and assigning a reference to this object to the temporary variable u (of the object executing the assignment). Note that an assignment $e.x := \mathsf{new}$ can be simulated by the sequence of assignments $u := \mathsf{new}; e.x := u$, where u is a 'fresh' temporary variable.

To define the semantics of guarded assignments formally, we first introduce for each class $C \in \mathcal{C}$ an arbitrary *infinite* set O_C of *object identities*, with typical element o. We assume that the sets O_C are mutually disjoint. A (global) configuration σ associates with each class C a *partial* function $\sigma(C)$ on O_c with a finite domain such that $\sigma(C)(o)$, if defined, denotes the internal state of object o. The internal state of an object is a function that assigns a value (of a corresponding type) to each of its instance variables. For notational convenience we will also simply write $\sigma(o)$ to denote the internal state of the object o in σ. Similarly, the value of an instance variable x of an object o in the configuration σ we denote by $\sigma(o)(x)$.

A local context τ specifies the active object and the values of the temporary variables. Formally, τ is a pair $\langle o, f \rangle$, with o the identity of the object and f a function which assigns to every temporary variable u its value $f(u)$. In the sequel, however, we will denote the first component o of a local context $\tau = \langle o, f \rangle$ by $\tau(\mathsf{self})$ and $f(u)$ by $\tau(u)$, for every temporary variable.

Expressions are evaluated in a configuration σ and a local context τ. The result of the evaluation of an expression e is denoted by $\mathcal{E}(e)(\sigma, \tau)$. It is defined by induction on the structure of e. Table 3.1 displays the formal definition of the evaluation. Note that all operators except equality are strict (with respect to the value $\perp$) in all their arguments.

$$\mathcal{E}(u)(\sigma,\tau) = \tau(u)$$

$$\mathcal{E}(e.x)(\sigma,\tau) = \begin{cases} \perp & \text{if } \mathcal{E}(e)(\sigma,\tau) = \perp \\ \sigma(\mathcal{E}(e)(\sigma,\tau))(x) & \text{otherwise} \end{cases}$$

$$\mathcal{E}(\mathsf{op}(e_1,\ldots,e_n)) = \begin{cases} \perp & \text{if } \mathcal{E}(e_i)(\sigma,\tau) = \perp \text{ for some } i \in [1..n] \\ \mathsf{op}(\mathcal{E}(e_1)(\sigma,\tau),\ldots,\mathcal{E}(e_n)(\sigma,\tau)) & \text{otherwise} \end{cases}$$

$$\mathcal{E}(e_1 = e_2)(\sigma,\tau) = \begin{cases} \mathsf{true} & \text{if } \mathcal{E}(e_1)(\sigma,\tau) = \mathcal{E}(e_2)(\sigma,\tau) \\ \mathsf{false} & \text{otherwise} \end{cases}$$

$$\mathcal{E}(\mathsf{self})(\sigma,\tau) = \tau(\mathsf{self})$$

$$\mathcal{E}(\mathsf{nil})(\sigma,\tau) = \perp$$

Table 3.1: Semantics of the programming expressions

Given a configuration σ and a local context τ, the resulting configuration σ' of the execution of an assignment $e.x := e'$ by the object $\tau(\mathsf{self})$ is defined by: $\sigma'(o)(x) = \mathcal{E}(e')(\sigma,\tau)$, for $o = \mathcal{E}(e)(\sigma,\tau)$, and in all other cases σ equals σ'. Note that the execution of an assignment $e.x := e'$ does not affect the values of the temporary variables. On the other hand, the local context τ' resulting from the execution of an assignment $u := e$ is obtained from τ by assigning $\mathcal{E}(e)(\sigma,\tau)$ to u. Note that the execution of an assignment $u := e$ does not affect the (global) configuration of objects.

Given a configuration σ and a local context τ, the resulting configuration σ' of the execution of an assignment $u := \mathsf{new}$ by the object $\tau(\mathsf{self})$ is obtained from σ by extending the domain of σ with a new object o and initializing its instance variables to nil. Furthermore, the resulting local context τ' is obtained from τ by assigning o to the variable u.

Figure 3.1 shows an example flowchart that models the insertion method of a sorted linked list. We assume a class Node that contains an instance variable *next* that is used to point to the next node in the list and an instance variable *key* that contains the integer value stored in the node.

The temporary variables n (of type Int), *cur* and *tmp* (both of type Node) are local to the insert operation, with n being its formal parameter. The instance variable *hd* which refers to an object of class Node points to the head of the list. It is important to observe that the first node, i.e., its head, in the list is a dummy node (a sentinel) that is stored in the list to simplify boundary conditions.

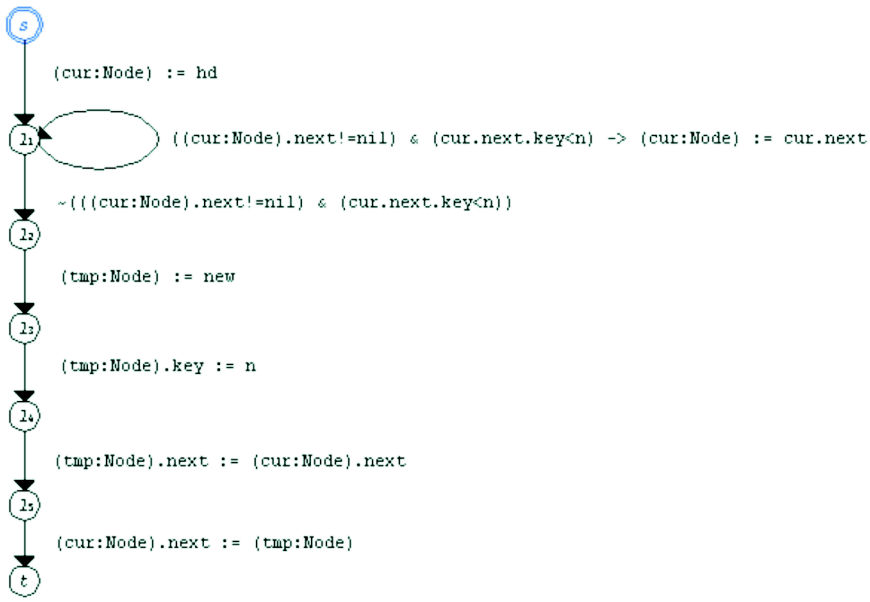

Figure 3.1: The flowchart of the insert operation

Chapter 4

The assertion language

The tool supports an assertion language that is designed to describe object structures. This language and its semantics are described in this chapter. One element of the assertion language will be the introduction of *logical variables*. These variables never occur in the expressions of the given programming language. Therefore, we are always sure that the value of a logical variable is not changed by a statement. Logical variables are used to express the constancy of certain expressions (for example in a proof rule for message passing, see [4]). They also serve as bound variables for quantifiers and have fixed types from the same set of programming language types $\mathcal{T}$ (see the previous chapter.

The set of expressions in the assertion language is larger than the set of programming language expressions, not only because it contains logical variables, but also because we include conditional expressions in the assertion language. These conditional expressions will be used for the analysis of the aliasing phenomenon, which arises due to the presence of the dereferencing operator.

In two respects the assertion language differs from the usual first-order predicate logic: Firstly, the range of quantifiers is limited to the *existing* objects in the configuration under consideration. For the classes different from the predefined ones, like that of the integers and booleans, this restriction means that we cannot talk about objects that have not yet been created, even if they could be created in the future. This is done to satisfy the requirements stated in the introduction. This implies that the range of the quantifiers can be different for different states. More in particular, a programming statement can change the truth of an assertion even if none of the program variables accessed by the statement occurs in the assertion, simply by creating an object and thereby changing the range of a quantifier.

Secondly, in order to strengthen the expressiveness of the logic, it is augmented with quantification over finite sequences of objects. It is quite clear that this is necessary, because simple first-order logic is not able to express certain interesting properties. The set of expressions of the language is obtained by the following grammar:

$$l ::= z \mid u \mid l.x \mid \mathsf{if}\ l_0\ \mathsf{then}\ l_1\ \mathsf{else}\ l_2\ \mathsf{fi} \mid \mathsf{op}(l_1, \ldots, l_n).$$

Here z denotes an logical variable. For conditional expressions $\mathsf{if}\ l_0\ \mathsf{then}\ l_1\ \mathsf{else}\ l_2\ \mathsf{fi}$, we assume that l_0 has type Bool and that l_1 and l_2 have the same type. Note that the evaluation of the expressions has no side effects, but might cause the throwing of an exception because of the included dereference operator. We have to take this into account when defining the semantics of the expressions.

The value of an expression is evaluated in a global configuration of objects σ, a local context τ, and a logical environment ω which assigns values to the logical variables. The result of the evaluation of an expression l is denoted by $\mathcal{L}(l)(\omega, \sigma, \tau)$. It is defined by induction on the structure of l. In Table 4.1 we list the formal definition of the evaluation of constructs that are not included in the programming language. The other definitions are similar to the ones given in Table 3.1 for the evaluation of programming language expressions and are therefore omitted.

$$\mathcal{L}(z)(\omega,\sigma,\tau) \equiv \omega(z)$$

$$\mathcal{L}(\textsf{if } l_0 \textsf{ then } l_1 \textsf{ else } l_2 \textsf{ fi})(\omega,\sigma,\tau) = \begin{cases} \bot & \text{if } \mathcal{L}(l_0)(\omega,\sigma,\tau) = \bot \\ \mathcal{L}(l_1)(\omega,\sigma,\tau) & \text{if } \mathcal{L}(l_0)(\omega,\sigma,\tau) = \textsf{true} \\ \mathcal{L}(l_2)(\omega,\sigma,\tau) & \text{if } \mathcal{L}(l_0)(\omega,\sigma,\tau) = \textsf{false} \end{cases}$$

Table 4.1: Semantics of the logical expressions

To reason about sequences we assume the presence of notations to express the length of a sequence (denoted by $|l|$) and the selection of an element of a sequence (denoted by $l[n]$, where n is an integer expression). More precisely, we assume in this paper that the elements of a sequence are indexed by $1,\ldots,n$, for some integer value $n \geq 0$ (the sequence is of zero length, i.e., empty, if $n = 0$). Accessing a sequence with an index which is out of its bounds will result in the value of nil.

The set of assertions, with typical element P, is defined by:

$$P ::= l_1 = l_2 \mid P \wedge Q \mid \neg P \mid \exists z P$$

Note that only equations are allowed as basic assertions. General boolean expressions are not allowed because they may be undefined and we want to remain within the realm of standard two-valued logics. For example, the evaluation of an inequality $l_1 \leq l_2$ will be undefined, e.g., result in the value $\bot$, if the evaluation of l_1 or l_2 results in $\bot$. However, we do allow the assertion $(l_1 \leq l_2) = \textsf{true}$ which evaluates to the boolean value true if the inequality holds for the *integer* values of l_1 and l_2. In all the other cases this assertion simply evaluates to the boolean value false. On the other hand, the assertion $(l_1 \leq l_2) = \textsf{nil}$ states that the evaluation of $l_1 \leq l_2$ is undefined, e.g., the evaluation of l_1 or l_2 gives rise to a null-pointer exception. In practice, we allow the user to simply write assertions like $l_1 \leq l_2$, which are interpreted as a shorthand for $(l_1 \leq l_2) = \textsf{true}$.

As already explained above, a formula $\exists z P$, with z a logical variable ranging over objects, states that P holds for an *existing* object. A formula $\exists z P$, with z of a sequence type, states the existence of a sequence of existing objects.

Formally, an assertion P is also evaluated in a configuration σ, a local context τ, and a logical environment ω. The result of the evaluation of an assertion P always yields a boolean value which is denoted by $\mathcal{A}(P)(\omega,\sigma,\tau)$. It is defined by an induction on the structure of P. The main interesting cases can be found in Table 4.2 where t denotes the type of the bound variable z. In general, the domain of quantification of a logical variable z depends on its type and the configuration σ. It is fixed for variables that range over integers and booleans: $dom(\textsf{Int},\sigma) = \mathbb{Z}$ and $dom(\textsf{Bool},\sigma) = \{\textsf{true},\textsf{false}\}$ in every configuration σ. On the other hand, we define $dom(C,\sigma) = dom(\sigma(C))$, for any class C. Finally, $dom(B^*,\sigma)$, for any basic type B, denotes the set of all (possibly empty) finite sequences of objects in $dom(B,\sigma)$. Observe that $\bot$ is not included in any domain.

$$\mathcal{A}(l_1 = l_2)(\sigma,\tau,\omega) = \begin{cases} \textsf{true} & \text{if } \mathcal{L}(l_1)(\sigma,\tau,\omega) = \mathcal{L}(l_2)(\sigma,\tau,\omega) \\ \textsf{false} & \text{otherwise} \end{cases}$$

$$\mathcal{A}(\exists z P)(\omega,\sigma,\tau) = \begin{cases} \textsf{true} & \text{if there exists an } \alpha \in dom(t,\sigma) \text{ such} \\ & \text{that } \mathcal{A}(P)(\omega\{\alpha/z\},\sigma,\tau) = \textsf{true}. \\ \textsf{false} & \text{otherwise} \end{cases}$$

Table 4.2: Semantics of assertions

The notation $\omega,\sigma,\tau \models P$ is also used to designate that P is true in the logical environment ω, the configuration σ, and the local context τ. Moreover, we define $\models P$, i.e., the assertion P is *valid*, by $\omega,\sigma,\tau \models P$, for every logical environment ω, configuration σ, and local context τ such that ω, σ, and τ only involve objects existing in σ. Note that given this restriction an assertion

like $\forall z(z \neq u) \wedge u \neq \mathsf{nil}$ does not make sense, i.e., is inconsistent (here u is assumed to denote an object). Indeed, semantically we want to rule out such an assertion because it states that the object referred to by u does not exist.

It is worthwhile to note that the assertion $\exists z\mathsf{true}$, where z ranges over objects (of an arbitrary class) is true if and only if there exists an object of that class (in the current configuration). In general, however, quantification is characterized by the usual validities like $\exists z P \leftrightarrow \neg\forall z\neg P$.

The following example assertion is part of the specification of the flowchart in Figure 3.1. It states that the sequence of nodes denoted by the logical variable z are linked by the instance variable *next*:

$$\forall n\ (1 \leq n \wedge n \leq |z| \rightarrow z[n].next = z[n+1]).$$

Here n is a logical integer variable. Note that by convention $z[|z|+1] = \mathsf{nil}$.

4.1 Aliasing

In this section we show how we can model assignments involving aliasing in the assertion language by means of substitutions. The next section discusses object-creation. The basic underlying idea as originally introduced in [10] and [7] and further developed in [2] is that the assertion resulting from the application of a substitution has the same meaning in the state before the assignment as the unsubstituted assertion has after the assignment. In other words, the substituted assertion describes the *weakest precondition*.
First we observe that given an assignment $u := e$, with u a temporary variable, and a *postcondition* P, the assertion $P[e/u]$ obtained from P by replacing every occurrence of u by e has the same meaning as the unsubstituted assertion P has after the assignment. This is formalized by the following substitution theorem.

Theorem 1 We have

$$\omega, \sigma, \tau \models P[e/u] \text{ if and only if } \omega, \sigma, \tau' \models P,$$

where τ' results from τ by assigning $\mathcal{E}(e)(\sigma, \tau)$ to u.

Proof
Standard induction on the complexity of P. □

On the other hand, the usual notion of substitution does not suffice for an assignment $e.x := e'$ because of possible aliases of the expression $e.x$, namely, expressions of the form $l.x$: it is possible that, after substitution, l refers to the object denoted by e, so that $l.x$ denotes the same 'memory cell' as $e.x$ and should be substituted by e'. It is also possible that, after substitution, l does not refer to the object e, and in this case no substitution should take place. Since we cannot decide between these possibilities by the form of the expression only, a conditional expression is constructed which decides "dynamically".

We have the following definition of the substitution operation $[e'/e.x]$ (syntactic identity is denoted by $\equiv$):

$$\begin{array}{lcl}
l[e'/e.x] & \equiv & l, \text{ for } l \equiv z, u \\
(l.x)[e'/e.x] & \equiv & \mathsf{if}\ l[e'/e.x] = e\ \mathsf{then}\ e'\ \mathsf{else}\ (l[e'/e.x]).x\ \mathsf{fi} \\
(l.y)[e'/e.x] & \equiv & (l[e'/e.x]).y \\
(\mathsf{op}(e_1, \ldots, e_n))[e'/e.x] & \equiv & \mathsf{op}(e_1[e'/e.x]), \ldots, e_n[e'/e.x]) \\
(\mathsf{if}\ l_1\ \mathsf{then}\ l_2\ \mathsf{else}\ l_3\ \mathsf{fi})[e'/e.x] & \equiv & \mathsf{if}\ l_1[e'/e.x]\ \mathsf{then}\ l_2[e'/e.x]\ \mathsf{else}\ l_3[e'/e.x]\ \mathsf{fi}
\end{array}$$

The first clause deals with the base cases of a logical variable z and a temporary variable u. In the second clause the expressions l and e are assumed to be of the same type. In the third clause either the types of the expressions l and e are distinct or the instance variables x and y are distinct. In the fourth clause we have that $\mathsf{op}[e'/e.x] \equiv \mathsf{op}$, in case $n = 0$ (i.e., in case of a constant). The definition is extended to assertions other than logical expressions in the standard way.

As a simple example, we consider the assignment $\mathsf{self}.x := 0$ and the postcondition $u.y.x = 1$, where x and y are instance variables and u is a temporary variable. Applying the corresponding substitution $[0/\mathsf{self}.x]$ to the assertion $u.y.x = 1$ results in the assertion

$$\mathsf{if}\ u.y = \mathsf{self}\ \mathsf{then}\ 0\ \mathsf{else}\ u.y.x\ \mathsf{fi} = 1.$$

This assertion clearly is logically equivalent to $u.y \neq \mathsf{self} \wedge u.y.x = 1$.

The following theorem states that $P[e'/e.x]$ is indeed the weakest precondition of the assertion P (with respect to the assignment $e.x := e'$).

Theorem 2 We have that

$$\omega, \sigma, \tau \models P[e'/e.x] \text{ if and only if } \omega, \sigma', \tau \models P,$$

where $\sigma'(o)(x) = \mathcal{E}(e')(\sigma, \tau)$, for $o = \mathcal{E}(e)(\sigma, \tau)$, and in all other cases σ agrees with σ'.

Proof
It suffices to prove by induction on the complexity of l that

$$\mathcal{L}(l[e'/e.x])(\omega, \sigma, \tau) = \mathcal{L}(l)(\omega, \sigma', \tau).$$

We shall only deal with the most interesting case: $l \equiv l'.x$. We have to show that

$$\begin{array}{ll} \mathcal{L}(\mathsf{if}\ l'[e'/e.x] = e\ \mathsf{then}\ e'\ \mathsf{else}\ (l'[e'/e.x]).x\ \mathsf{fi})(\omega, \sigma, \tau) & = \\ \text{if } \mathcal{L}(l'[e'/e.x])(\omega, \sigma, \tau) = \mathcal{L}(e)(\sigma, \tau) \text{ then } \mathcal{L}(e')(\sigma, \tau) \text{ else } \mathcal{L}(l'[e'/e.x]).x)(\omega, \sigma, \tau) & = \\ \mathcal{L}(l'.x)(\omega, \sigma', \tau), & \end{array}$$

where $\sigma'(o)(x) = \mathcal{L}(e')(\sigma, \tau)$, for $o = \mathcal{L}(e)(\sigma, \tau)$, and in all other cases σ agrees with σ'.

By the induction hypothesis we have that

$$\mathcal{L}(l'[e'/e.x])(\omega, \sigma, \tau) = \mathcal{L}(l')(\omega, \sigma', \tau),$$

We distinguish the following two cases. First let

$$\mathcal{L}(l'[e'/e.x])(\omega, \sigma, \tau) = \mathcal{L}(e)(\sigma, \tau).$$

We then calculate as follows:

$$\begin{array}{ll} \mathcal{L}(\mathsf{if}\ l'[e'/e.x] = e\ \mathsf{then}\ e'\ \mathsf{else}\ (l'[e'/e.x]).x\ \mathsf{fi})(\omega, \sigma, \tau) & = \\ \mathcal{L}(e')(\sigma, \tau) & = \\ \sigma'(o)(x) & = \\ \sigma'(\mathcal{L}(e)(\sigma, \tau))(x) & = \\ \sigma'(\mathcal{L}(l'[e'/e.x])(\omega, \sigma, \tau))(x) & = \\ \sigma'(\mathcal{L}(l')(\omega, \sigma', \tau))(x) & = \\ \mathcal{L}(l'.x)(\omega, \sigma', \tau). & \end{array}$$

Next, let

$$\mathcal{L}(l'[e'/e.x])(\omega, \sigma, \tau) \neq \mathcal{L}(e)(\sigma, \tau).$$

By construction of σ' it follows that $\sigma(\mathcal{L}(l'[e'/e.x])(\omega, \sigma, \tau))(x) = \sigma'(\mathcal{L}(l'[e'/e.x])(\omega, \sigma, \tau))(x)$. We then calculate as follows:

$$\begin{array}{ll} \mathcal{L}(\mathsf{if}\ l'[e'/e.x] = e\ \mathsf{then}\ e'\ \mathsf{else}\ (l'[e'/e.x]).x\ \mathsf{fi})(\omega, \sigma, \tau) & = \\ \mathcal{L}((l'[e'/e.x]).x)(\omega, \sigma, \tau) & = \\ \sigma(\mathcal{L}(l'[e'/e.x])(\omega, \sigma, \tau))(x) & = \\ \sigma'(\mathcal{L}(l'[e'/e.x])(\omega, \sigma, \tau))(x) & = \\ \sigma'(\mathcal{L}(l')(\omega, \sigma', \tau))(x) & = \\ \mathcal{L}(l'.x)(\omega, \sigma', \tau). & \end{array}$$

□

4.2 Object creation

Next we consider the creation of objects. We want to define the substitution $[\mathsf{new}/u]$ which models the creation of a new object referred to by the temporary variable u. This substitution should model logically the assignment $u := \mathsf{new}$. Execution of an assignment $u := \mathsf{new}$ consists of the creation of a new object and assigning a reference to this object to u. Note that an assignment $e.x := \mathsf{new}$ can be simulated by the sequence of assignments $u := \mathsf{new}; e.x := u$, where u is a 'fresh' temporary variable. For an assignment $e.x := \mathsf{new}$ we therefore can compute the weakest precondition of a postcondition P by $P[u/e.x][\mathsf{new}/u]$, where u is a fresh temporary variable which does not occur in P and e.

As with the usual notions of substitution we want the expression after substitution to have the same meaning before the assignment as the unsubstituted expression has after the assignment. However, in the case of the creation of a new object, there are expressions for which this is not possible, because they refer to the new object and there is no expression that could refer to that object before its creation, because it does not exist yet. Therefore the result of the substitution must be left undefined in some cases.

However we *are* able to carry out the substitution in case of assertions because a temporary variable u referring to the new object can essentially occur only in a context where either one of its instance variables is referenced, or it is compared for equality with another expression. In both of these cases we can predict the outcome without having to refer to the new object.

Here are the main cases of the formal definition of the substitution $[\mathsf{new}/u]$, with u a temporary variable, for logical expressions. As already explained above the result of the substitution $[\mathsf{new}/u]$ is undefined for the expression u. We have

$$l[\mathsf{new}/u] \equiv l, \text{ for } l \equiv \mathsf{self}, \mathsf{nil}, z, x, v,$$

where z is a logical variable, x is an instance variable, and v is a temporary variable distinct from u.

Since the (instance) variables of a newly created object are initialized to nil we have

$$(u.x)[\mathsf{new}/u] \equiv \mathsf{nil}.$$

The other possible context u may occur is that of an equality. If neither l nor l' is u or a conditional expression they cannot refer to the newly created object and we have

$$\Big(l = l'\Big)[\mathsf{new}/u] \quad \equiv \quad \Big(l[\mathsf{new}/u]\Big) = \Big(l'[\mathsf{new}/u]\Big).$$

If either l is u and l' is neither u nor a conditional expression (or vice versa) we have that after the substitution operation l and l' cannot denote the same object (because one of them refers to the newly created object while the other one refers to an already existing object):

$$\Big(l = l'\Big)[\mathsf{new}/u] \equiv \mathsf{false}.$$

On the other hand if both the expressions l and l' equal u we obviously have

$$\Big(l = l'\Big)[\mathsf{new}/u] \equiv \mathsf{true}.$$

For l a conditional expression of the form $\mathsf{if}\ l_0\ \mathsf{then}\ l_1\ \mathsf{else}\ l_2\ \mathsf{fi}$ we define

$$\Big(l = l'\Big)[\mathsf{new}/u] \equiv \mathsf{if}\ l_0[\mathsf{new}/u]\ \mathsf{then}\ (l_1 = l')[\mathsf{new}/u]\ \mathsf{else}\ (l_2 = l')[\mathsf{new}/u]\ \mathsf{fi}.$$

Finally, if $l_i[\mathsf{new}/u]$, for $i = 1, \ldots, n$, is defined, then

$$(\mathsf{op}(l_1, \ldots, l_n))[\mathsf{new}/u] \equiv \mathsf{op}(l_1[\mathsf{new}/u], \ldots, l_n[\mathsf{new}/u]),$$

for any other operator op of the language.

Since we assume that the only operations on 'pointers' are testing for equality and dereferencing, it is easy to see that $l[\mathsf{new}/u]$ is defined for boolean expressions l. The following lemma states that the value of $l[\mathsf{new}/u]$ before the creation of the new object, if defined, equals that of l after its creation.

Lemma 1 Let l be such that $l[\mathsf{new}/u]$ is defined. We have

$$\mathcal{L}(l[\mathsf{new}/u])(\omega, \sigma, \tau) = \mathcal{L}(l)(\omega, \sigma', \tau'),$$

where σ' is obtained from σ by extending the domain of σ with a new object o and initializing its instance variables to nil. Furthermore the resulting local context τ' is obtained from τ by assigning o to the variable u.

Proof
This lemma is proved by a straightforward induction on the complexity of l. Let us deal with one representative case: $l \equiv z.x$ (note that x is thus an instance variable and therefore distinct from the temporary variable u). Then $(z.x)[\mathsf{new}/u] \equiv z.x$. So we have to show that

$$\mathcal{L}(z.x)(\omega, \sigma, \tau) = \mathcal{L}(z.x)(\omega, \sigma', \tau'),$$

where σ' is obtained from σ by extending the domain of σ with a new object o and initializing its instance variables to nil. The resulting local context τ' is obtained from τ by assigning o to the variable u. Since $\omega(z)$ denotes an object existing in σ, we have that $\omega(z) \neq o$. It follows that $\sigma'(\omega(z))(x) = \sigma(\omega(z))(x)$. □

Next we consider lifting this substitution operation $[\mathsf{new}/u]$ to assertions. We define

$$(P \wedge Q)[\mathsf{new}/u] \equiv P[\mathsf{new}/u] \wedge Q[\mathsf{new}/u] \text{ and } (\neg P)[\mathsf{new}/u] \equiv \neg(P[\mathsf{new}/u]).$$

The changing scope of a bound occurrence of a variable z ranging over objects which is induced by the creation of a new object is captured as follows.

$$(\exists z\ P)[\mathsf{new}/u] = (\exists z(P[\mathsf{new}/u])) \vee (P[u/z][\mathsf{new}/u]).$$

The idea of the application of $[\mathsf{new}/u]$ to $(\exists z\ P)$ is that the first disjunct $\exists z(P[\mathsf{new}/u])$ represents the case that P holds for an 'old' object (i.e. which exists already before the creation of the new object) whereas the second disjunct $P[u/z][\mathsf{new}/u]$ represents the case that the new object itself satisfies P. Since a logical variable does not have aliases, the substitution $[u/z]$ consists of simply replacing every occurrence of z by u. It is worthwhile to observe that we can derive the following clause for universal quantification.

$$(\forall z\ P)[\mathsf{new}/u] = (\forall z(P[\mathsf{new}/u])) \wedge (P[u/z][\mathsf{new}/u]).$$

As a simple example, we consider applying $[\mathsf{new}/u]$ to the assertion $\forall z(u = z \vee \mathsf{self} = z)$ which states that the set of existing objects consist only of the object denoted by the temporary variable u and the object itself.

$$\begin{array}{ll} \Big(\forall z(u = z \vee \mathsf{self} = z)\Big)[\mathsf{new}/u] & \equiv \\ \forall z\Big((u = z \vee \mathsf{self} = z)[\mathsf{new}/u]\Big) \wedge (u = u \vee \mathsf{self} = u)[\mathsf{new}/u] & \equiv \\ \forall z(\mathsf{false} \vee \mathsf{self} = z) \wedge (\mathsf{true} \vee \mathsf{false}) & \end{array}$$

where the last assertion obviously reduces to $\forall z(\mathsf{self} = z)$. This assertion states that self is the only object which exists, which indeed is the weakest precondition of the assertion $\forall z(u = z \vee \mathsf{self} = z)$ with respect to $u := \mathsf{new}$.

Next we consider the case of an occurrence of a bound variable z which ranges over *sequences* of objects. First we observe that, without loss of expressiveness, we may assume that in the assertion

language the operations on sequences are limited to $|l|$, i.e. the length of the sequence l, and $l[n]$, i.e. the operation which yields the nth element of l. So we do not have, for example, equality on sequences as a primitive operation in the assertion language. Given this assumption, let z' be a (fresh) logical variable ranging over sequences of boolean values. The variables z and z' together will 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 the substitution operation $[z', u/z]$, the main characteristic cases of which are:

$$\begin{array}{ll} z[z', u/z] & \text{is undefined} \\ \Big(|z|\Big)[z', u/z] & \equiv |z| \\ \Big(z[l]\Big)[z', u/z] & \equiv \mathsf{if}\ z'(l')\ \mathsf{then}\ u\ \mathsf{else}\ z(l')\ \mathsf{fi}, \quad \text{where } l' = l[z', u/z]. \end{array}$$

This substitution operation $[z', u/z]$ is defined for the remaining expressions and extended to assertions in the standard way (its application to a compound expression is defined only if its application to its constituents is defined). Given the above restriction on the kind of operations on sequences, it is easy to see that this substitution is defined for boolean expressions and assertions.

The following lemma states the correctness of this substitution operation.

Lemma 2 Given a configuration σ, let ω be a logical environment with $\omega(z)$ a sequence of objects (in σ) and $\omega(z')$ a sequence of boolean values. Let α be a sequence of objects in σ such that the sequences $\omega(z)$ and α have equal length. Furthermore, for some object o in σ, we have, for all i, if $\omega(z')[i] = \mathsf{true}$ then $\alpha[i] = o$ else $\alpha[i] = \omega(z)[i]$. We have

$$\omega, \sigma, \tau\{o/u\} \models P[z', u/z] \text{ if and only if } \omega\{\alpha/z\}, \sigma, \tau\{o/u\} \models P$$

($\tau\{o/u\}$ results from τ by assigning o to u).

Proof
Straightforward induction on the complexity of l and P. We treat the only (slightly) non-trivial case of an expression of the form $z[l]$. Let $\omega' = \omega\{\alpha/z\}$, where α is a sequence of objects such that for all i, if $\omega(z')[i] = \mathsf{true}$ then $\alpha[i] = o$ else $\alpha[i] = \omega(z)[i]$. Moreover, let $\tau' = \tau\{o/u\}$.

By the induction hypothesis we have

$$\mathcal{L}(l[z', u/z])(\omega, \sigma, \tau') = \mathcal{L}(l)(\omega', \sigma, \tau').$$

Let $l' \equiv l[z', u/z]$. We then calculate as follows.

$$\begin{array}{ll} \mathcal{L}((z[l])[z', u/z])(\omega, \sigma, \tau') & = \\ \mathcal{L}(\mathsf{if}\ z'(l')\ \mathsf{then}\ u\ \mathsf{else}\ z(l')\ \mathsf{fi})(\omega, \sigma, \tau) & = \\ \text{if } \omega(z')[\mathcal{L}(l')(\omega, \sigma, \tau')] = \mathsf{true} \text{ then } o \text{ else } \omega(z)[\mathcal{L}(l')(\omega, \sigma, \tau')] & = \\ \text{if } \omega(z')[\mathcal{L}(l)(\omega', \sigma, \tau')] = \mathsf{true} \text{ then } o \text{ else } \omega(z)[\mathcal{L}(l)(\omega', \sigma, \tau')] & = \\ \omega'(z)[\mathcal{L}(l)(\omega', \sigma, \tau')] & = \\ \mathcal{L}(z[l])(\omega', \sigma, \tau'). & \end{array}$$

□

Given this encoding we can now define

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

where z ranges over sequences of objects.

As an example, consider the following assertion

$$\exists z_1 \Big(|z_1| = n \wedge \forall z_2 \exists i\ (z_1[i] = z_2) \Big),$$

where the logical variable z_1 ranges over sequences of objects and the logical variable z_2 ranges over objects themselves. This assertion states that there exist at most n objects. An application of the substitution $[z', u/z]$ to the assertion $|z_1| = n \wedge \forall z_2 \exists i\ (z_1[i] = z_2)$ results in the assertion

$$|z_1| = n \wedge \forall z_2 \exists i \Big(\mathsf{if}\ z'[i]\ \mathsf{then}\ u\ \mathsf{else}\ z_1[i]\ \mathsf{fi} = z_2\Big).$$

For technical convenience only, in order to apply the substitution $[\mathsf{new}/u]$ to this assertion, we first eliminate the conditional expression. We obtain

$$|z_1| = n \wedge \forall z_2 \exists i \Big(z'[i] \rightarrow u = z_2 \wedge \neg z'[i] \rightarrow z_1[i] = z_2\Big)$$

(assuming that $\neg, \rightarrow, \wedge$ lists these operators in decreasing binding priority). An application of $[\mathsf{new}/u]$ to this latter assertion results in the following:

$$|z_1| = n \wedge \forall z_2 \exists i \Big(z'[i] \rightarrow \mathsf{false} \wedge \neg z'[i] \rightarrow z_1[i] = z_2\Big) \wedge \exists i \Big(z'[i] \rightarrow \mathsf{true} \wedge \neg z'[i] \rightarrow \mathsf{false}\Big)$$

This assertion is clearly logically equivalent to the assertion

$$|z_1| = n \wedge \forall z_2 \exists i \Big(\neg z'[i] \wedge z_1[i] = z_2\Big) \wedge \exists i\, z'[i]$$

Summarizing the above we obtain as final result the assertion

$$\exists z_1 \exists z' \Big(|z_1| = |z'| \wedge |z_1| = n \wedge \forall z_2 \exists i\ (\neg z'[i] \wedge z_1[i] = z_2) \wedge \exists i\, z'[i]\Big)$$

This latter assertion clearly is logically equivalent to the assertion

$$\exists z_1 \Big(|z_1| = n - 1 \wedge \forall z_2 \exists i\ (z_1[i] = z_2)\Big)$$

which indeed corresponds with our intuition of the weakest precondition of the assertion which states that there exists at most n objects after the creation of a new object.

The following theorem states that $P[\mathsf{new}/u]$ indeed calculates the weakest precondition of P (with respect to the assignment $u := \mathsf{new}$).

Theorem 3 We have

$$\omega, \sigma, \tau \models P[\mathsf{new}/u] \text{ if and only if } \omega, \sigma', \tau' \models P,$$

where σ' is obtained from σ by extending the domain of σ with a new object o and initializing its instance variables to nil. Furthermore the resulting local context τ' is obtained from τ by assigning o to the variable u.

Proof
The proof proceeds by induction on the complexity of P. Again, we treat only the most interesting case of an assertion $\exists z P$, where z is a logical variable ranging over sequences of objects. We calculate as follows. By definition of the substitution operation $[\mathsf{new}/u]$ we have

$$\omega, \sigma, \tau \models (\exists z P)[\mathsf{new}/u] \text{ iff } \omega, \sigma, \tau \models \exists z \exists z' (|z| = |z'| \wedge P[z', u/z][\mathsf{new}/u]).$$

So, assuming that $\omega, \sigma, \tau \models (\exists z P)[\mathsf{new}/u]$, there exists a sequence α of objects in σ and a sequence β of boolean values, with α and β of equal length, such that for $\omega' = \omega\{\alpha/z, \beta/z'\}$ we have

$$\omega', \sigma, \tau \models P[z', u/z][\mathsf{new}/u].$$

By the induction hypothesis (measuring the complexity in terms of the number of quantifiers and propositional connectives) we next derive that

$$\omega', \sigma, \tau \models P[z', u/z][\mathsf{new}/u] \text{ iff } \omega', \sigma', \tau' \models P[z', u/z],$$

where σ' is obtained from σ by extending the domain of σ with a new object o and initializing its instance variables to nil. Furthermore the local context τ' is obtained from τ by assigning o to the variable u.

Let α' be a sequence of objects existing in σ' of the same length as α such that for all i, if $\beta[i] = \mathsf{true}$ then $\alpha'(z)[i] = \tau'(u) = o$ else $\alpha'(z)[i] = \alpha[i]$. Let $\omega'' = \omega'\{\alpha'/z\}$. It follows from lemma 2 that

$$\omega', \sigma', \tau' \models P[z', u/z] \text{ iff } \omega'', \sigma', \tau' \models P.$$

Finally, we observe that $\omega'', \sigma', \tau' \models P$ implies $\omega, \sigma', \tau' \models \exists z P$ (the logical variable z' is assumed not to occur in P).

Conversely, let ω, σ and τ be such that ω and τ only involve objects existing in σ and

$$\omega, \sigma', \tau' \models \exists z P,$$

where σ' and τ' are defined as above. So there exists a sequence α of objects existing in σ' such that

$$\omega\{\alpha/z\}, \sigma', \tau' \models P.$$

Let β be a a sequence of boolean values, with α and β of equal length, and for all i, if $\beta[i] = \mathsf{true}$ then $\alpha[i] = \tau'(u)$ else $\alpha[i]$ exists in σ. It follows that

$$\omega'', \sigma', \tau' \models P,$$

where $\omega'' = \omega\{\alpha/z, \beta/z'\}$ (the logical variable z' is assumed not to occur in P). Now let α' be a sequence of objects existing in σ of the same length as α such that $\alpha'[i] = \alpha[i]$, if $\beta[i] = \mathsf{false}$. Let $\omega' = \omega\{\alpha'/z\}$. By lemma 2 it then follows that

$$\omega'', \sigma', \tau' \models P \text{ iff } \omega', \sigma', \tau' \models P[z', u/z].$$

By the induction hypothesis we have that

$$\omega', \sigma', \tau' \models P[z', u/z] \text{ iff } \omega', \sigma, \tau \models P[z', u/z][\mathsf{new}/u].$$

By construction of ω' we have that

$$\omega', \sigma, \tau \models (|z| = |z'| \wedge P[z', u/z][\mathsf{new}/u]).$$

We conclude that

$$\omega, \sigma, \tau \models (\exists z P)[\mathsf{new}/u].$$

□

4.3 The verification conditions

Given the substitution operations defined above, the definition of the verification conditions is largely standard (see [17] for a detailed description of the general theory of Floyd's inductive assertion method). We sketch the general idea here and adapt it to our flowcharts.

Let L be a set of locations, and let T be a set of transitions between locations in L. Transitions from a location l to a location l' will be denoted by $(l, b \to a, l')$, where b is a boolean guard of the assignment a as defined in Chapter 3. A flowchart can be represented formally by a tuple (L, T, s, t), where $s \in L$ is the start location of the flowchart and $t \in L$ is the exit location. We assume that every computation of the flowchart starts in s. We say that a computation terminates successfully if it arrives at exit location t (for simplicity we assume that t has no outgoing transitions).

We want to prove that a flowchart $F = (L, T, s, t)$ satisfies a correctness specification $\{P\}F\{Q\}$, for some precondition P and postcondition Q. For partial correctness, this requires verifying that every successfully terminated computation that started in a state that satisfies P, terminates in a state that satisfies Q. By Floyd's inductive assertion method this can be verified by assigning to

each location l an assertion P_l and checking that (1) the precondition P implies the assertion P_s, (2) the assertion P_t implies the postcondition Q, and (3) checking for each transition $(l, b \to a, l') \in T$ the validity of the corresponding verification condition $(P_l \wedge b) \to P'_{l'}$, where the assertion $P'_{l'}$ results from $P_{l'}$ by an application of the substitution operation corresponding with the assignment a, that is,

- $P'_{l'}$ equals $P_{l'}[e/u]$, in case a is of the form $u := e$;
- $P'_{l'}$ equals $P_{l'}[e'/e.x]$, in case a is of the form $e.x := e'$;
- $P'_{l'}$ equals $P_{l'}[\mathsf{new}/u]$, in case a is of the form $u := \mathsf{new}$

The substitution operations $[e'/e.x]$ and $[\mathsf{new}/u]$ are defined above, whereas $[e/u]$ denotes the standard notion of substitution. The tool computes a verification condition for every transition in the flowchart according to the above definitions.

Chapter 5

Translation of the assertion language into HOL

In this chapter we describe the translation of the semantics of the previously introduced assertion language into the HOL logic (a typed higher order logic). Although the result of the translation will be given in the syntax of the HOL system, in many cases it is straightforward to adapt the translation to logics which are supported by other theorem provers.

The main issue of the translation is the representation of the types that play a role in the assertion language. If these types have been clarified, the translation becomes straightforward by following the definition of the assertion language semantics as given in Chapter 4. We will therefore start by describing the necessary type declarations. It is important to bear in mind that those declarations are automatically generated by the compiler from the class descriptions in the editor.

Before we discuss the types of objects, we extend the basic types Int and Bool to deal with exceptions. The types Int and Bool are present as basic types in the HOL logic, but we use the built-in polymorphic unary type operator `option` to include nil in these types. The polymorphic option data type has one (type) parameter and two constructors (SOME and NONE). Its definition is stored in the predefined theory `optionTheory`, but for clarity we give its definition here also (though in an informal notation):

$$\begin{aligned} \alpha \text{ option} &= \text{NONE} \\ &\mid \text{SOME } \alpha \end{aligned}$$

The types $\text{Bool}^{\perp}$ and $\text{Int}^{\perp}$ can be represented by the HOL types `bool option` and `int option`, respectively. Notice that type operators are written with suffix notation in HOL. The `NONE` constants in these types represent the value $\perp$, whereas the `SOME` constructors encapsulate the original values of these types. The operators in the assertion language need to be redefined in the logic to handle these option types. Function definition in the current version of HOL is handled by the `Define` function, which takes as input a number of equations separated by the conjunction sign (`/\`). Consider, for example, the following adapted definition of the less than relation (`LT`):

```
(LT NONE NONE = NONE) /\ (LT (SOME a) NONE = NONE) /\
(LT NONE (SOME a) = NONE) /\ (LT (SOME a) (SOME b) = SOME (a<b))
```

In the above definition, the variables `a` and `b` have type Int. Observe that the evaluation order in the example corresponds to that of the operator evaluation definition in Table 3.1.

We will now explain why we cannot tackle the representation of objects similarly. We first have to decide on the type of objects identitities of a certain class. Recall that we introduced *infinite* sets Id_C of object identities in Chapter 3 because this ensures the existence of new C type object identities in every configuration. Types in the HOL logic denote sets in the universe of the logic. We therefore want to declare a type for the set of existing objects O_C because this allows us to

quantify over the objects in the set. There is, however, one drawback of this natural approach - types in the HOL logic always denote non-empty sets. But the set of class C objects is possibly empty! Our solution to this problem is to include the constant nil in every set of objects of a certain class. This requires nil to have a polymorphic type.

Here are the details of this solution. Every class name from the class library is used to declare a new atomic type. A finite number of classes exist in a given verification context. Identifiers that start with a capital letter like `Node`, for example, are used to denote class names. Type definitions are handled by the function `new_type`, which takes as arguments the number of parameters of the new type and the name of the type (a string). For instance, the basic type `Node` is declared as follows:

```
new_type 0 "Node"
```

Throughout the remainder of this chapter, we will use the class `Node`, with instance variables $\texttt{next}_{\texttt{Node}}$ and $\texttt{key}_{\texttt{Int}}$, to illustrate the type declarations that are specific for a single class.

Subsequently, we introduce a polymorphic unary type operator `Object` and the constant `nil:'an Object` by means of the HOL functions `new_type` and `new_constant`, respectively. The expression `'an` denotes a type variable. The following two HOL expressions do the job:

```
new_type 1 "Object"
new_constant ("nil", Type `:'an Object`)
```

Its suffices to know that the latter expression adds a new constant `nil` of type `'an object` to the current theory. Since nil has this polymorphic type, it is, for example, an inhabitant of the set corresponding to the type `Node Object`. This set is intended to represent the *existing* objects of class `Node` extended with nil. An axiom is introduced that implies that this set is finite. The axiom states that there is a finite set `s` of which all inhabitants of type `Node Object` are a member:

```
?s. !(n:Node Object). (n IN s) /\ (FINITE s)
```

The above definition assumes the presence of the predicates `IN` and `FINITE`, which are pre-defined in the HOL logic. We will not discuss their definitions here. The question mark and the exclamation mark denote existential and universal quantification, respectively.

Next, we discuss the representation of internal states. Such states assign values to the instance variables of objects. We naturally represent these states by means of records: Each record field corresponds to an instance variable of an object. Fortunately, the notation used for field selection in HOL happens to coincide with the standard notation used for dereferencing. If `o` is a HOL record that contains a field `f`, then selecting this field is denoted by `o.f`. This improves the readability of the assertions in the logic. Below, we list the type definition that corresponds to the internal states of `Node` objects.

```
NodeRec = <| next: Node Object; key: int option |>
```

Giving this equation to the function `Hol_datatype` of the `bossLib` library results in a record type `NodeRec` with two fields: a field `next` of type `Node Object` and a field `key` of type `int`. Thus a record type is declared for each class. In general, the type of internal states of a certain class is the name of the class appended with `"Rec"`.

Another issue is the translation of the global configuration of objects σ into the type theory. Recall that σ is used to obtain the internal state of an existing object and is 'queried' in the semantics of the assertion language only in the definitions of $l.x$ and $\exists z P$. When evaluating an assertion, we only deal with one particular configuration and therefore we simply declare a constant that represents the partial configuration σ_C for each class C. For class `Node` we have, for example, the constant `NodeState : Node Object -> NodeRec`. In general, we represent σ_C by the function `CState`.

Our final examples illustrate the representation of arrays (sequences). We will first declare types for actual sequences, and then extend these types to take exceptions into account. A sequence consists of a list of values and its length. The declared type of a sequence therefore combines

these two components. It includes a function of integers (indices of the sequence) to values and an integer that denotes the length of the sequence. We use a Cartesian product type to combine these two. Below, we list some abbreviations of sequence types.

```
intArray = ((int->int option)#int)
boolArray = ((int->bool option)#int)
NodeArray = ((int->(Node Object))#int)
```

Observe that the tuple of two types a and b is written as `(a#b)`. To refer to the components of sequence types inhabitants, we use the projection functions `fst` and `snd`. The function `LEN` (for length) is defined similar to `snd`, but it takes the possibility of exceptions into account. A pointer to a sequence might have the value nil. Therefore, we give such a variable a type that is extended similarly as Int and Bool. For example, a variable to a sequence of type `Node` receives the type `NodeArray option`. The function `LEN` maps each value of this type to its corresponding length:

```
LEN NONE = NONE) /\ (LEN (SOME (a,b)) = (SOME b))
```

Recall that an array that is indexed out of bounds yields nil. We define functions for accessing arrays that reflect this behavior. The first argument of these functions is a pointer to an array, and the second argument is the index. Both arguments can yield exceptions, which complicates their definitions. Due to the size of these definitions, we only give one example.

```
(int_at NONE b = NONE) /\ (int_at (SOME Z)) NONE = NONE) /\
(int_at (SOME Z) (SOME i) = if (i < 1) \/ (i > (length Z))
                            then NONE else (SOME (fst Z i)))
```

The above defined function `int_at` is used to access arrays. In the equations, `b` is a value of type `int option` and `Z` is of type `intArray`. Similar functions `bool_at` and `obj_at` encode accessing sequences of booleans and objects, respectively.

Table 5.1 lists all import cases of the translation Tr of assertion language expressions into the HOL logic. We have not included the translation of temporary variables, but that case is similar to the one given for logical variables. This translation in fact expresses in HOL the semantics of the assertion language as defined in chapter 4.

In the given translation, Tr(op) is the translation of operator op. Above, we have given an example of such a translation for the less than operator. The expression $type(x)$ denotes the type of variable x. The translation comes with two further remarks. Firstly, we point out that we do not represent the local context τ and the logical environment ω explicitly to simplify the formulas. This simplification, however, requires that one ensures that the set of temporary variables and the set of logical variables are disjunct. The second remark concerns the translation of self. After translation, self becomes a variable that ranges over a set of object identities that also includes nil. The possibility that self equals nil is excluded by means of an axiom.

The definition of the translation for formulas is straightforward. Note that since `nil` is included in every type `C object`, we have to exclude it from the domain of quantification.

$$
\begin{array}{rcl}
Tr(z) & = & \begin{cases}
\texttt{(z:int option)} & \text{if } type(z) = \mathsf{Int} \\
\texttt{(z:bool option)} & \text{if } type(z) = \mathsf{Bool} \\
\texttt{(z:C Object)} & \text{if } type(z) = C \in \mathcal{C} \\
\texttt{(z:\^{}BArray option)} & \text{if } type(z) = B^* \text{ for some } B \in \mathcal{B}
\end{cases} \\
Tr(l.x) & = & \begin{cases}
\texttt{if } Tr(l) = \texttt{nil then nil} & \\
\quad \texttt{else (CState (}Tr(l)\texttt{)).x} & \text{if } type(x) \in \mathcal{C} \\
\texttt{if } Tr(l) = \texttt{nil then NONE} & \\
\quad \texttt{else (CState (}Tr(l)\texttt{)).x} & \text{otherwise}
\end{cases} \\
 & & \text{where } \mathtt{C} = type(l) \\
Tr(\mathsf{op}(l_1, \ldots, l_n)) & = & (Tr(\mathsf{op})\ Tr(l_1)\ \ldots\ Tr(l_n)) \\
Tr(l_1 = l_2) & = & Tr(l_1)\texttt{=}Tr(l_2) \\
Tr(\mathsf{self}) & = & \texttt{(self:C Object)} \\
 & & \text{where } \mathtt{C} = type(\mathsf{self}) \\
Tr(z[i]) & = & \begin{cases}
\texttt{(int\_at (}Tr(z)\texttt{) (}Tr(i)\texttt{))} & \text{if } type(z) = \mathsf{Int}^* \\
\texttt{(bool\_at (}Tr(z)\texttt{) (}Tr(i)\texttt{))} & \text{if } type(z) = \mathsf{Bool}^* \\
\texttt{(obj\_at (}Tr(z)\texttt{) (}Tr(i)\texttt{))} & \text{if } type(z) = C^* \text{ for some } C \in \mathcal{C}
\end{cases} \\
Tr(|z|) & = & \texttt{(LEN } Tr(z)\texttt{)} \\
Tr(\forall z\ P) & = & \texttt{!(z:int option).\~{}(z=NONE) ==> } Tr(P) \text{ if } type(z) = \mathsf{Int} \\
Tr(\forall z\ P) & = & \texttt{!(z:bool option).\~{}(z=NONE) ==> } Tr(P) \text{ if } type(z) = \mathsf{Bool} \\
Tr(\forall z\ P) & = & \texttt{!(z:C Object).\~{}(z=nil) ==> } Tr(P) \text{ if } type(z) = C, \text{ for some } C \\
Tr(\forall z\ P) & = & \texttt{!(z:BArray option). (z=NONE) ==> } Tr(P) \text{ if } type(z) = B^*, \text{ for some } B
\end{array}
$$

Table 5.1: The translation of assertion language

Chapter 6

An example: inserting into a sorted linked list

In this chapter, we briefly discuss an application of the tool to the verification of the correctness of the insert operation described in figure 3.1. We first describe the annotation of the flowchart with assertions containing parameterized macros and end with some remarks on the level of automation of the construction of the proof.

We want to specify in the postcondition of the insert operation the correct addition of the inserted node. We did so by introducing a logical variable z which denotes the *initial* list of linked nodes. The following assertion

$$\Pi_{i=1}^{|z|}(z[i].next = z[i+1] \wedge z[i] \neq \mathsf{nil}) \wedge hd = z[1] \wedge |z| \geq 1$$

(here and in the sequel we use the notation $\Sigma_{i=e}^{e'}P$ and $\Pi_{i=e}^{e'}P$ as an abbreviation of the bounded quantification $\exists i(e \leq i \wedge i \leq e' \wedge P)$ and $\forall i(e \leq i \wedge i \leq e' \rightarrow P)$) states, among others, that two consecutive elements of z are linked by the instance variable *next* (by convention $z[|z|+1] = \mathsf{nil}$) and that its first element is denoted by the variable *hd*. For this assertion we introduce the (parameterized) macro $linkedlist(z, next)$.

The following assertion describes the correct addition of a node *tmp* in the initial list z.

$$\sum_{i=1}^{|z|} \Big((z[i].next{=}tmp) \wedge (i > 1 \rightarrow z[i].key < tmp.key) \\ \wedge (i < |z| \rightarrow z[i+1].key \geq tmp.key) \wedge (tmp.next = z[i+1])\Big)$$

For this assertion we introduce the parameterized macro $addtolist(z, tmp)$.

We want to prove that the flowchart F of figure 3.1 satisfies the pre- and postcondition specification

$$\{linkedlist(z, next)\}F\{addtolist(z, tmp) \wedge tmp.key = n\}$$

by annotating it with assertions and checking in HOL the corresponding verification conditions which are generated automatically by our tool. We have the following annotations.

s**:** $linkedlist(z, next)$.

l_1**:** $linkedlist(z, next) \wedge currentpos(cur, z)$,

where the (parameterized) macro $currentpos(cur, z)$ stands for the assertion

$$\Sigma_{i=1}^{|z|}(cur = z[i] \wedge (i > 1 \rightarrow cur.key < n))$$

l_2: $linkedlist(z, next) \wedge correctpos(z, cur)$,

where the macro $correctpos(z, cur)$ stands for the assertion

$$\Sigma_{i=1}^{|z|}(cur = z[i] \wedge (i > 1 \rightarrow cur.key < n) \wedge (i < |z| \rightarrow cur.next.key \geq n)$$

l_3: $linkedlist(z, next) \wedge correctpos(z, cur) \wedge tmp \notin z$,

where $tmp \notin z$ is a macro for the assertion $\neg\Sigma_{i=1}^{|z|}(tmp = z[i])$ (also used below).

l_4: $linkedlist(z, next) \wedge correctpos(z, cur) \wedge tmp \notin z \wedge tmp.key = n$.

l_5: $linkedlist(z, next) \wedge correctpos(z, cur) \wedge tmp \notin z \wedge tmp.key = n \wedge tmp.next = cur.next$.

t: $addtolist(z, tmp) \wedge tmp.key = n$.

The flowchart annotated with these assertions was compiled into a number of verification conditions that were translated into the HOL logic by the tool and afterwards proven valid in the theorem-proving system of HOL. Three out of seven verification conditions were proven almost automatically by basic automatic-rewriting rules and two only required additionally the introduction of a witness to reduce an existentially quantified goal. The two verification conditions of the transitions departing from location l_1 required a bit more effort. This additional effort was mainly due to the required reasoning about the underlying data type of the integers. The typical reasoning about pointers consists only of some basic equational logic. The arithmetic involved consists only of simple Presburger arithmetic of array indices. This arithmetic is implemented in HOL in a separate proof tactic (`COOPER_TAC` from the `intLib` library). This tactic functions well on the domain it is written for, however it requires some effort to use it in combination with proof tactics for other domains. Our conclusion is that a fully automated correctness proof can be obtained by an appropriate integration of the proof tactics involved.

Chapter 7

Related work and future research

The main contribution of this paper consists of a description and formal justification a tool which supports the computer-aided specification and verification of a class of flowcharts that captures the basic dynamics of object-oriented programs. It forms a front-end to the theorem prover HOL.

Currently very interesting and promising work is being carried out in the field of computer-aided specification and verification of object-oriented programs at various places. Here the only projects we mention are Loop of the University of Nijmegen ([13]), Bali of the Technical University of Munich([3]), and Bandera of the Kansas State University ([5]).

The specific emphasis of our approach is, first of all, on the automated verification of programs annotated with assertions that allow one to specify properties in terms of the source code instead of some particular model of its semantics. In fact, the abstraction level of our assertion language corresponds with the Object Constraint Language (OCL) [19]. One of the main differences is that navigation in OCL is also an operation defined on sets of objects, whereas in our assertion language it is just a dereference operator on objects (as it is in the programming language). This difference stems from the intended use of OCL for describing class diagrams in the Unified Modeling Language (UML) ([18]) while our assertion language is specifically tailored to object-structures as they arise during the computation of an object-oriented program.

Another distinguishing feature of our approach is the automatic generation of the verification conditions by an implementation of a *calculus* for computing the weakest preconditions of assignments involving aliasing and object creation. This calculus has been extended in [16] for OCL, whereas in [15] a different Hoare logic for object-oriented programs is given based on an explicit representation of the global store model. A theorem prover can be used to check the validity of the verification conditions by simply encoding the formally defined semantics of the assertion language. Our front-end tool thus describes the program semantics *axiomatically* in terms of the weakest precondition calculus. This calculus provides some preprocessing of information about aliases and object creation which is made available to the theorem prover. The theorem prover only 'knows' about the semantics of the assertion language and is used solely to check simple verification conditions. In contrast, most existing approaches are based on a direct logical description of the program semantics in the theorem prover ([9]). One of the advantages of our appoach is its flexibility and scalability: the use of another theorem prover only requires a translation of the semantics of the assertion language and the incorporation of new programming constructs only requires the definition of new verification conditions in terms of the substitution operators introduced in this paper.

Currently, we are extending the system to the widely used programming language Java by, first of all, implementing message passing ([4]) and and the basics of the multi-threaded control flow of Java ([1]). We are also incorporating the Java mechanism of inheritance. As already remarked above, the verification conditions corresponding to these Java programming constructs are defined in terms of the substitution operators introduced in this paper.

A more long-term goal consists of a further development of the tool towards an interactive, *user-friendly* specification and verification environment which will provide support for the syntax,

pretty printing, and type checking of the assertion languages. Moreover, it will provide graphical user interfaces to a theorem prover and to the programming environment, and include the compiler that generates the verification conditions automatically.

Bibliography

[1] E. Abraham-Mumm and F.S. de Boer. Proof-outlines for threads in Java. Proceedings of CONCUR 2000, Lecture Notes in Computer Science, Vol. 1877, 2000.

[2] J.W. de Bakker. Mathematical theory of program correctness. Prentice-Hall.

[3] URL: http://www4.informatik.tu-muenchen.de/~isabelle/bali/.

[4] F.S. de Boer. A WP-calculus for OO. Proceedings of Foundations of Software Science and Computation Structures (FOSSACS), Lecture Notes in Computer Science, Vol. 1578, 1999.

[5] J. Hatcliff and M. Dwyer. Using the Bandera tool set to model-check properties of concurrent Java software. Proceedings of CONCUR 2001, Lecture Notes in Computer Science, 2001.

[6] The CUP parser generator. URL: http://www.cs.princeton.edu/~appel/modern/java/CUP/.

[7] E.W. Dijkstra. A discipline of programming. Prentice-Hall, 1976.

[8] D. Gries. The Science of Programming. Springer-Verlag Berlin Heidelberg, 1981.

[9] M. Huisman. Reasoning about Java programs in higher order logic with PVS and Isabelle. IPA Dissertation Series 2001-03. ISBN 90-9014440-4.

[10] C.A.R. Hoare. An axiomatic basis for computer programming. Communications ACM, Vol. 12, 1969.

[11] The HOL system. URL: http://www.cl.cam.ac.uk/Research/HVG/HOL/.

[12] JLex: A Lexical Analyzer Generator for Java.
URL: http://www.cs.princeton.edu/~appel/modern/java/JLex/.

[13] The LOOP project. URL: http://www.cs.kun.nl/~bart/LOOP/.

[14] S. Owre, J. Rushby and N. Shankar. PVS: A prototype verification system. Proceedings of the 1th Conference on Automated Deduction, Lecture Notes in Artificial Intelligence, Vol. 617, 1992.

[15] A. Poetzsch-Heffter and P. Mueller. Logical foundations for typed object-oriented languages. Proceedings of the IFIP Working Conference on Programming Concepts and Methods (PROCOMET98).

[16] B. Reus, M. Wirsing, R. Hennicker. A Hoare Calculus for Verifying Java Realizations of OCL-Constrained Design Models. Proceedings of FASE 2001, Lecture Notes in Computer Science, Vol. 2029, 2001.

[17] W.P. de Roever, F.S. de Boer, U. Hanneman, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. Concurrency Verification. Cambridge University Press 2001.

[18] J. Rumbaugh, I. Jacobson, G. Booch. The Unified Modeling Language Reference Manual. Addison-Wesley Object Technology Series.

[19] J.B. Warmer and A.G. Kleppe. The object constraint language: precise modeling with UML. Addison-Wesley Object Technology Series.