COMPUTER-AIDED SPECIFICATION AND VERIFICATION OF ANNOTATED OBJECT-ORIENTED PROGRAMS

F.S. de Boer
Utrecht University,
Utrecht, The Netherlands
frankb@cs.uu.nl

C. Pierik
Utrecht University,
Utrecht, The Netherlands
cpierik@cs.uu.nl

Abstract The main contribution of this paper consists of a description of a front-end tool which supports the computer-aided specification and verification of a class of flowcharts which capture the basic dynamics of object-oriented programs. The specific emphasis of our approach is on the automated verification of flowcharts annotated with assertions which allow one to specify properties directly in terms of the source code itself instead of some particular model of its semantics.

1. Introduction

In this paper we describe a software system which supports the computer-aided specification and verification of a certain class of flowcharts which capture the basic dynamics of object-oriented programs. The execution of an object-oriented program written in, for example, Java gives rise to a dynamic configuration of objects. Characteristic of such a configuration is that objects can be created at arbitrary points during the execution of a program, and references to objects, 'pointers', can be stored in variables and passed around. This implies that complicated and dynamically evolving structures of references between objects can occur.

The verification tool presented in this paper implements a *weakest precondition* calculus (introduced in (De Boer, 1999)) for reasoning about the semantics of basic assignments in object-oriented programming languages. The calculus itself is based on 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. In more detail, this means that the only operations on "pointers" (references to objects) are testing for equality and dereferencing (looking at the value of an instance variable of the referenced object). Furthermore, in a given configuration, it is only possible to mention the objects that exist in that configuration. Objects that do not (yet) exist never play a role.

The input of the verification tool is a flowchart together with a mapping which associates an assertion to every location, a so-called 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 (De Roever et al., 2001). In order 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 the (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 are interactively verified by the theorem prover HOL (HOL URL) in terms of an internal representation of the assertion language.

2. The Verification Tool

In this section we describe the global architecture of the tool and the way it is used. See figure 1 for a graphic description of the architecture.

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 JLex (JLex URL) and the parser generator CUP (CUP URL).

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 during a session to infer the types of instance variables that occur in the flowchart. Moreover, they are the basis of the type definitions that are used to reason over objects in the HOL logic. In section 5 we give 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.

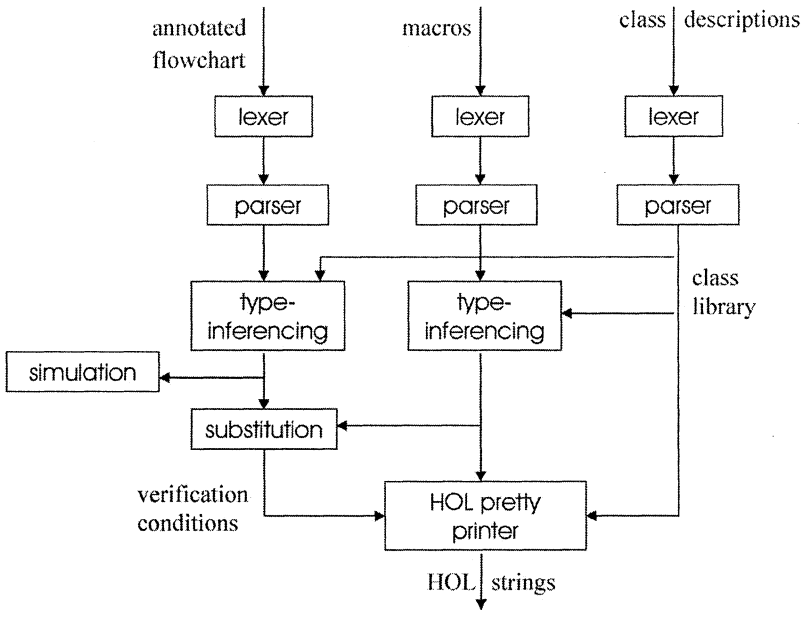

Figure 1. The tool architecture

Assertions can be assigned to every location of the flowchart. Parameterized macros from a library can be used in these assertions. In section 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 section 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.

3. Flowcharts

In this section we describe in more detail the formalism and semantics of the class of strongly typed flowcharts which are 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 objects of type B. For each of these types we assume a set of *instance variables* and a set of *temporary variables*, i.e. the contents of such a variable will be a value of that type.

We have 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 of some type and op is some operation of one of the data-types (like that of the integers and boolean values). (In case $n = 0$, op is a constant.) In particular we assume the presence of the constants self and nil which denote the active object and the value 'undefined', respectively. In the expression $e.x$ the type of the expression e is assumed to be some class name. The only other operation on objects is testing for equality.

Next we introduce the basic assignments which specify the operations which may occur in the flowcharts considered in this paper. We have the following assignments $e.x := e'$ to an instance variable x and assignments $u := e$ to a temporary variable u. More specifically, 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 $u := e$ by an object consists of assigning the value of e to its temporary variable u.

It is worthwhile to observe that an assignment of the form $e.x[i] := e'$, with x 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'.

Moreover, we consider in this paper assignments of the form $u :=$ new. The execution of an assignment $u :=$ 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 $e.x :=$ new can be simulated by the sequence of assignments $u :=$ new; $e.x := u$, where u is a 'fresh' temporary variable.

As an example of a flowchart we refer to figure 2 where a flowchart is given for inserting an integer value into a sorted linked list. We assume given a class Node that contains an instance variable *next* that will be 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 so called sentinel) that is stored in the list to simplify boundary conditions.

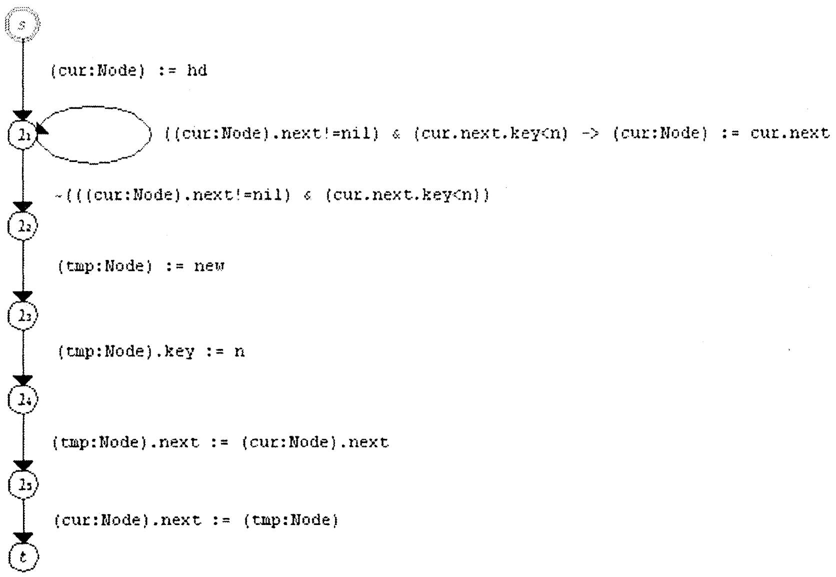
s
(cur:Node) := hd
l1
((cur:Node).next!=nil) & (cur.next.key<n) -> (cur:Node) := cur.next
~(((cur:Node).next!=nil) & (cur.next.key<n))
l2
(tmp:Node) := new
l3
(tmp:Node).key := n
l4
(tmp:Node).next := (cur:Node).next
l5
(cur:Node).next := (tmp:Node)
t

Figure 2. The flowchart of the insert operation

4. The Assertion Language

In this section a logical formalism is introduced for expressing certain properties of a configuration of objects.

One element of the assertion language will be the introduction of *logical variables*. These variables may not occur in the expressions of the given programming language, but only in the assertion language. Therefore we are always sure that the value of a logical variable can never be 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 (De Boer, 1999)). Logical variables also serve as bound variables for quantifiers.

In general, the set of expressions in the assertion language will be 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 phenomenon of aliasing which arises because of 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 in order to satisfy the requirements stated in the introduction. Because of this 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.

Formally, the set of *logical* expressions, with typical element l, is generated by extending the above grammar of expressions with logical variables z, and conditional expressions of the form $\textsf{if } l_0 \textsf{ then } l_1 \textsf{ else } l_2 \textsf{ fi}$ (here l_0 is a boolean expression and the expressions l_1 and l_2 are of the same type):

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

Here z denotes a logical variable.

In order 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, in case $n = 0$). Accessing a sequence with an index which is out of its bounds will result in the value of $\textsf{nil}$.

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

$$P ::= l \mid P \wedge Q \mid \neg P \mid \exists z P$$

Here l denotes a boolean expression.

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

It is worthwhile to note that the assertion $\exists z \textsf{true}$, where z ranges over objects (of some 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 $\exists z P \leftrightarrow \neg \forall z \neg P$.

As an example, consider the flowchart of figure 2 of the previous section. The following assertion 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 and Object Creation

In this section we show how we can model assignments involving aliasing and object-creation in the assertion language by means of substitutions. These substitutions are used by the tool to generate automatically the verification conditions of an annotated flowchart. The basic underlying idea 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.*

Aliasing

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.

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 main cases of the substitution operation $[e'/e.x]$ (syntactic identity is denoted by $\equiv$):

$$\begin{array}{lcll} (l.x)[e'/e.x] & \equiv & \left\{ \begin{array}{l} \mathsf{if}\ l[e'/e.x] = e\ \mathsf{then}\ e'\ \mathsf{else}\ (l[e'/e.x]).x\ \mathsf{fi} \\ (l[e'/e.x]).x \end{array} \right. & \begin{array}{l} (1) \\ (2) \end{array} \\ (l.y)[e'/e.x] & \equiv & (l[e'/e.x]).y & \end{array}$$

The (instance) variables x and y here are assumed to be distinct. The first case of the first clause should be applied if e and l have equal types. The second case defines the substitution if the types of e and l differ. The above definitions are extended in the standard way.

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

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

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 if l_0 then l_1 else l_2 fi we define

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

Since we assume that the only operations on 'pointers' are testing for equality and dereferencing, we have that $l[\mathsf{new}/u]$ is defined for boolean expressions l.

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

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

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.

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

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 $u := \mathsf{new}$.

The case of an occurrence of a bound variable z which ranges over *sequences* of objects is discussed in the full paper. We omit it here.

5. HOL

In this section we describe the translation of the previously introduced assertion language into the HOL logic. All variables in our assertion language range over integers, objects or arrays of some type. We use the standard HOL theory (`IntegerTheory`) to represent integers. Reasoning about the other types requires some type declarations, which will be described below. It is important to bear in mind that those declarations are automatically generated by the compiler from the class descriptions in the editor.

In the computational model described in the full paper the presence of an arbitrary *infinite* set O of object identities is assumed because this ensures in every global configuration the existence of new object identities. In addition we assume that each of these identities is typed. That is, each identity belongs to a certain *class* of objects. There are a finite number of classes in a given verification context. Identifiers that start with a capital letter are used to denote class names, for instance `Node`. For each class that is defined, a set of objects is assumed. A declaration of a type for such a set in the HOL logic, allows us, also in HOL, to quantify over the objects in the set. Types in the HOL logic denote sets in the universe of the logic. Declaring a type therefore implies the specification of a set of objects. There is however one drawback of this natural approach. Types in the HOL logic necessary denote non-empty sets. But the set of objects of a class is possibly empty! Our solution to this problem is to include `nil` as an object in every set of objects of a certain class. Note that this requires `nil` to have a polymorphic type.

Here are the details of this solution. Every class name from the class library is used in HOL to declare new atomic types. The following example statement shows such a type declaration for the class `Node`.

```
val _ = new_type 0 "Node";
```

The example statement declares `Node` to be a new 0-ary type operator. It is actually a valid statement in the meta-language of HOL, the functional programming language `ML`. This statement binds the result of the function `new_type` to the name that is placed after the) `val`-keyword. In this case we simply place _ instead of an identifier, to indicate that the expression does not have to be bound to a specific name.

Next, we introduce a polymorphic unary type operator Object and the constant `nil`:

```
val _ = new_type 1 "Object";
val _ = new_constant ("nil", Type `:'a Object`);
```

The latter statement adds nil to the current HOL theory as a constant with the polymorphic type `'a Object`. The expression `'a` denotes a type variable.

Note that type operators are written with suffix notation. Since `nil` has this polymorphic type, it is an inhabitant of the set represented by the type `Node Object`. This set is intended to denote the *existing* objects of class `Node` extended with `nil`. A HOL-axiom is generated that states that this set is finite.

Next we discuss how to represent the internal states of objects. The internal state of an object assigns values to each of its instance variables. We have chosen to represent internal states by means of records: Each field of a record corresponds to an instance variable of an object. Moreover, the notation used for field selection in HOL happens to coincide with the standard notation used for dereferencing in object-oriented programs also used in this paper. If `o` is of a record type that contains some field `f`, the selection of field `f` of object `o` is denoted by `o.f`. This improves the readability of the assertions in HOL. The `Hol_datatype` function from the `bossLib` library facilitates the definition of record types. Consider for instance the following statement:

```
val _ = Hol_datatype `NodeRec =
  <| next: Node Object
   ; key: int |>`;
```

This statement declares a record type `NodeRec` with two fields: a field `next` of type `Node Object` and another field `key` of type `int`. Not surprisingly, this definition is exactly the translation of an internal state of an object of class `Node`. For each class, a record type is declared. The name of the record type is the name of the class, appended with `"Rec"`.

Finally, we describe the translation of a global configuration of objects into the type theory. Such a global configuration specifies the internal state, i.e., the values of the instance variables, of every existing object and is 'queried' in the semantics of the assertion language only in the definitions of $l.x$ and $\exists zP$. Because the types of the internal states differ for each class, we introduce the notion *class state*. A class state is a function that maps object identities of a specific class to their internal states. Again, we illustrate this for the class `Node`:

```
val _ = new_constant ("NodeState"
                     , Type `:Node Object -> NodeRec`);
```

This statement introduces a constant `NodeState` of type `Node Object -> NodeRec`. In general, given a finite number of class names C we thus represent a global configuration by the corresponding constants `CState` of type `C Object -> CRec`.

As a last example of the necessary type declarations we will describe the representation of arrays of objects. An array is simply a function of integers to objects of a certain class. The types of these functions are declared for each class. A constant `length` of type `'array -> int` is introduced to refer to the length of every arbitrary array, independent of its type (`'array` is a type variable). Recall that an array that is indexed out of bounds equals nil. For each type of array, the compiler produces two axioms that handle these upper and lower bound respectively. These axioms can be used if necessary during a proof session and can be included in automatic rewriting tools.

The following table lists some examples of the translation Tr of the assertion language into the HOL logic. This translation in fact expresses in a natural and simple way in HOL the semantics of the assertion language which is given in the full paper. Note that in the translation of $\forall(z : C)P$, since `nil` is included in every type `C object`, we have to exclude it from the domain of quantification. On the other hand, we do allow sequences which contain nil.

Assertion Language		HOL
$Tr(z : b)$	=	`z:B` (where $B \in$ {Int, Bool})
$Tr(z : C)$	=	`(z:C Object)`
$Tr(z : B^*)$	=	`(z:^BArray)`
$Tr(u : B)$	=	`u:B` (where $B \in$ {Int, Bool})
$Tr(u : C)$	=	`(u:C Object)`
$Tr((l : C).x)$	=	`(CState` $(Tr(l))$`).x`
Tr(if l_0 then l_1 else l_2 fi)	=	`(if` $Tr(l_0)$ `then` $Tr(l_1)$ `else` $Tr(l_2)$`)`
Tr(self : C)	=	`(self:C Object)`
Tr(nil)	=	`nil`
$Tr(z[i])$	=	`((`$Tr(z)$`) (`$Tr(i)$`))`
$Tr(\|z\|)$	=	`(length` $Tr(z)$`)`
$Tr(\forall(z : C)P)$	=	`!(z:C Object).~(z=nil) ==>` $Tr(P)$
$Tr(\forall(z : B^*)P)$	=	`!(z:^BArray).`$Tr(P)$

Here C denotes an arbitrary class name and B denotes an arbitrary basic type. Many operators on e.g. integers are already present in HOL and can be used.

6. An Example: Inserting into a Sorted Linked List

In this section we briefly discuss an application of the tool to the verification of the correctness of the insert operation described in figure 2. 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 do so by introducing a logical variable z which denotes the *initial* list of linked nodes. The following assertion

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

(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 2 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

$$\Sigma_{i=1}^{|z|} \quad \begin{array}{l} cur = z[i] \wedge \\ i > 1 \rightarrow cur.key < n \wedge \\ i < |z| \rightarrow cur.next.key \geq n \end{array}$$

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 is compiled into a number of verification conditions that are 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 tools 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.

7. Related Work and Future Research

The main contribution of this paper consists of a description of a front-end tool which supports the computer-aided specification and verification of a class of flowcharts which capture the basic dynamics of object-oriented programs.

Currently there is much interesting work on computer-aided verification of object-oriented programs being carried out at various places. Here we mention only the projects Loop of the University of Nijmegen (LOOP URL), Bali of the Technical University of Munich (Bali URL), and Bandera of the Kansas State University (Hatcliff and Dwyer 2001).

The specific emphasis of our project is, first of all, on the automated verification of programs annotated with assertions which allow one to specify properties in terms of the source code itself instead of some particular model of its semantics. In fact, the abstraction level of our assertion language corresponds with that of the Object Constraint Language (OCL) (Warmer and Kleppe, 1998). One of the main differences is that in OCL 'navigation' is an operation defined on sets of objects whereas in our assertion language it simply is a dereference operator on objects (as it is in the programming language).

Moreover, for generating the verification conditions the tool implements a *calculus* for computing weakest preconditions. The formal semantics of this calculus is given in the full paper. This calculus has been extended in (Reus et al., 2001) for OCL, whereas in (Poetzsch-Heffter and Mueller 1998) a different Hoare logic for object-oriented programs is given based on an explicit representation of the global store model. The validity of the verification conditions are interactively verified by the theorem prover HOL in terms of an internal representation of the semantics of the assertion language. Our front-end tool thus describes the program semantics *axiomatically* in terms of a weakest precondition calculus. In fact, this calculus provides some preprocessing of information about aliasing and object creation which is made available to the theorem prover. The theorem prover itself only knows about the semantics of the assertion language and is used only to verify the validity of simple verification conditions. In contrast, most existing approaches are based on a direct logical description of the program semantics in the theorem prover (Huisman, 2001). Finally, the specific emphasis in this paper is on reasoning about aliasing and object creation.

Currently we are extending the system with an implementation of message passing (De Boer, 1999) and the basics of the multi-threaded flow of control of Java (Abraham-Mumm and De Boer, 2000).

References

Abraham-Mumm, E. and De Boer, F. S. (2000). Proof-outlines for threads in Java. Proceedings of CONCUR 2000, Lecture Notes in Computer Science, Vol. 1877.

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

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

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

URL: http://www.cs.princeton.edu/~appel/modern/java/CUP/.

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

URL: http://www.cl.cam.ac.uk/Research/HVG/HOL/.

URL: http://www.cs.princeton.edu/~appel/modern/java/JLex/.

URL: http://www.cs.kun.nl/~bart/LOOP/.

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

Poetzsch-Heffter, A. and Mueller, P. (1998). Logical foundations for typed object-oriented languages. Proceedings of the IFIP Working Conference on Programming Concepts and Methods (PROCOMET98).

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

De Roever, W.-P., De Boer, F. S., Hanneman, U., Hooman, J., Lakhnech, Y., Poel, M. and Zwiers, J. (2001). Concurrency Verification. Cambridge University Press.

Warmer, J. B. and Kleppe, A. G. (1998). The object constraint language: precise modeling with UML. Addison-Wesley Object Technology Series.