Towards Verifiable Specifications of Object-oriented Frameworks
A Case Study

Jörg Meyer, Arnd Poetzsch-Heffter

FernUniversität Hagen

Abstract. The specification of object-oriented frameworks has to fulfill several requirements. It should document the behavior of the classes in an abstract, implementation independent way. It should be formally founded so that the correctness of implementations can be proved w.r.t. it. Last but not least, it should provide an appropriate basis for the verification of programs using the framework. In this paper, we present a case study that focuses on the first two requirements. It shows an abstract specification of a linked list implementation with shared objects, sketches the underlying formal framework, and explains the necessary proof steps.

1 Introduction

The specification of an object-oriented program framework has to fulfill several requirements. It should document the classes of the framework in an abstract, implementation independent way. Implementation independency is necessary to hide implementation aspects that should not be exploited by a user of the framework. Abstraction is needed to raise the level of specification. As a second requirement, a specification should be verifiable, i.e. a specification technique should be embedded into a formal setting and complemented with verification rules and proof techniques. Formalization helps to clarify the semantics of the specification language. Verification is needed to establish the correctness of code. As a third requirement, the specification of a module M should provide an appropriate basis to verify the correctness of a module N that uses M.

Most of the work that was done about specification and verification of OO-programs has focussed either on specification or on verification. In the area of specification, the main goals were the development of easy to use and expressible specification languages; the precise formal relationship between specifications and programs – a prerequisite of formal verification – was of minor interest. In the area of verification, the focus was on the formal proof rules and techniques. The remaining interesting challenge is to apply the verification techniques to prove the correctness not only of some isolated properties, but of complete interface specifications.

Contents. This paper investigates the relation between specification and verification. It presents the interesting parts of a case study, in which we verified the specification of a list implementation consisting of three classes (cf. [LMMPH00] for a complete report). The goal of the presentation is to describe important aspects and problems that have to be dealt with when verification techniques are applied to realistic specifications. In particular, we illustrate how abstraction can be handled and demonstrate the complexity that results from aliasing. To our knowledge, it is the first time that an OO-program with complex aliasing is proved correct. In Section 2, we present and discuss the specification. Section 3, sketches the proof technique.

Relation to Other Work. The work is related to interface specification techniques for OO-programs. In particular, the used technique builds on the two tiered Larch approach (cf. [GH93,Lea97]). That is we use a general specification language to express abstract properties – in our case the specification language of PVS – and an interface specification language to describe program interfaces. In this presentation, only some constructs of the interface specification language will be illustrated.

For verification, we use a Hoare logic (see [PHM99]) and an interactive program prover that communicates with PVS ([MPH00]). Recently a number of different approaches to the verification of Java-like OO-programs have been investigated. The LOOP group developed a translator that generates PVS theories from a given Java program ([JvdBH+98]). The theories capture the program behavior and can be used to verify program properties in PVS ([Hui00]). In his thesis, David von Oheimb formalized a Java subset and a corresponding Hoare logic in Isabelle. His work focuses on meta-theory, in particular type safety and correctness and completeness properties of the logic (see [vO01]). A dynamic logic for a Java subset is presented in [Bec00].

For brevity, we can't treat modularity properties in this extended abstract, although we consider them very important. The interested reader is refered to [Mül01] for this topic.

2 Specifying Object-oriented Programs

In this section, we describe the specification technique along with parts of an implementation for doubly linked lists. We start with the abstract interface specification. Then we show how specification and implementation are related.

2.1 The Specification of Class DList

A class specification consists of a list of invariants (keyword `inv`) and a list of method specifications. A method specification consists of an optional requires clause (keyword `req`) and a list of pre-post-pairs. The meaning of such a specification is given by desugaring it into Hoare triples. Each invariant has to be maintained by all public methods of the program[1]. The condition stated in the requires clause, if any, may be assumed in the prestate. In addition, each pre-post-pair constitutes a Hoare triple; again the requires clause, if any, is conjoined to the precondition. To keep things simple, we do not use more elaborate specification constructs like e.g. old-expressions.

The example we consider here is centered around the class `DList`. The specification is given along with the external visible parts of the class declaration. We present only those parts that are needed in the following. The syntax of formulas follows the syntax of the PVS language (see [COR+95]). Capital letters denote logical variables holding values of PVS builtin or user defined types. Logical variables are used to relate variable values in preconditions to those in postconditions. `result` is used as a special variable representing the value returned by a method:

[1] We use modularity techniques to decrease the number of methods for which the invariant has to be shown (see [Mül01]), but this is beyond the scope of this abstract.

```
class DList
{
 inv X: wfDList(X,$);

 public static DList empty()
 pre  TRUE;
 post ADList(result,$) = null;
 pre  alive(X,$) AND $=S;
 post ($==S)(X);

 public DList rest()
 req  ADList(this,$) /= null;
 pre  ADList(this,$) = L;
 post ADList(result,$) = cdr(L);
 pre  alive(X,$) AND $=S;
 post ($==S)(X);

 public int first()
 req  ADList(this,$) /= null;
 pre  ADList(this,$)=L;
 post aI(result) = car(L);
 pre  $=S; post $=S;

 public boolean isempty()
 pre  ADList(this,$) = L;
 post aB(result) = null?(L);
 pre  $=S; post $=S;

 public void app(int i)
 pre  ADList(this,$)=L AND
      I=aI(i) AND T=this;
 post ADList(T,$)
   = append(L,cons(I,null));
}
```

To express interface properties of OO-programs in an abstract way, three ingredients are necessary. (1.) We have to be able to refer to the object store. (2.) We need a functional vocabulary to express the behavior of the methods. (3.) Abstraction functions are necessary to relate the implementation to the abstract behavior.

To refer to the object store, we use the global program variable \$ of type `Store`. As we will show later, `Store` is an abstract data type. The predicate `alive` checks for an object whether it is allocated in a given store. As shown by the specification above, the methods `first` and `isempty` do not modify the object store. The methods `empty` and `rest` do not modify objects that are alive in the prestate; `(S1==S2)(X)` is a derived predicate on stores saying that every location (= instance field) reachable from object `X` holds the same value in stores `S1` and `S2`. The frame behavior of method `append` is not specified.

To express the functional method behavior in the example, we used the data type `list[int]` of PVS with constant `null`, functions `car`, `cdr`, `cons`, `append`, and predicate `null?`. In other examples, specific data types have to be designed to describe the abstract behavior of a class or framework. The relation between the abstract level and the implementation level is captured by abstraction functions and predicates. The predicate `wfDList` expresses the fact that the link structure of a list is well formed in a given store. The function `ADList` maps the given object and objects referenced by it in a given store to a PVS list. For technical reasons, we use abstraction functions as well for the basic data types boolean and int to map Java values to PVS values (`aB` and `aI`). For instance, the specification of `append` reads as follows: Abstracting the this-object in the poststate yields a list with new last element `I`. Again, we like to point out that the specification does not refer to any implementation detail.

2.2 The DList Implementation

In the following, we relate the above specification to the implementation of class `DList` and two auxiliary classes. The subsection focuses on those aspects that are needed to make an interface specification verifiable.

In the example, doubly linked lists are implemented by a `DList`-object as list header and a sequence of `NodeL`-objects as shown in Figure 1. The class `NodeL` is a subclass of a class `Node`. This separation into two classes was done to illustrate some aspects of inheritance.

```
class DList
{
 protected NodeL firstNode;
 protected NodeL lastNode;

 public static DList empty() {...}
 public DList rest() { ... }
 public int first() {
   Node f = this.firstNode;
   int k = f.getElem();
   return k;
 }
 public boolean isempty() { ... }
 public void app(int i) { ... }
}
```

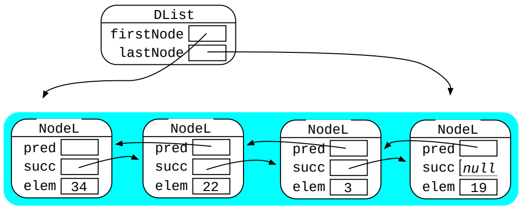

Fig. 1. Store Layout of a Doubly Linked List

Objects of class `Node` can be used in a very general way to create linked data structures, where each node holds a value of type `int`. The abstraction function `ANode` yields the `int`-value of the instance variable `elem`.

```
class Node
{
 protected int  elem;
 protected Node pred, succ;

 public int getElem()
 pre  $=S AND T=this;
 post aI(result) = ANode(T,S);
 pre  $=S; post $=S;

 public Node getPred()
 pre  $=S AND T=this;
 post result =
      S@@loc(T,Node?pred);
 pre  $=S;  post $=S;

 public Node getSucc()
 pre  $=S AND T=this;
 post result =
      S@@loc(T,Node?succ);
 pre  $=S;  post $=S;
}
```

The specification of method `getPred` shows how the object store can be accessed: `S@@loc(T,Node?pred)` denotes the value held by instance variable `Node?elem` of object `T` in store `S`. All functions and predicates concerning the object store are formalized in PVS. In particular, `Node?elem` is a constant. Type `Store` represents the abstract data type of object stores, `Location` is the set of instance variables, `Value` is a common type for all values and references occurring in the programming language. The following functions are provided: Let `l` be a location, `s` be a value of type store, `id` be a type identifier of a non-abstract class, and `v` be of type `Value`:

- `update(s,l,v)` returns the store, where `l` in `s` is updated with `v`.
- `new(s,id)` returns a reference to a fresh object allocated in `s`.
- `s##id` returns the store after allocating an object of type `id` in store `s`.

– `s@@l` returns the value stored in location `l` in store `s`.
– `alive(v,s)` returns *true* if the object (value) refered by `v` is allocated in `s`, *false* otherwise.

The semantics of the object store is specified by fourteen axioms. We present two of them to give an impression:

```
store1:  AXIOM L1 /= L2 => update(S, L1, X)@@L2 = S@@L2
store10: AXIOM NOT alive(new(S, T), S)
```

Axiom store1 describes that an update of some location does not affect other locations. Axiom store10 describes that some newly allocated object in a certain store was not alive in that store before allocation.

Based on class `Node`, we define a class `NodeL`. It contains an additional method and has a specification that guarantees the specific structure of the `NodeL`-objects implementing doubly linked lists (cf. the grey shaded area in Fig. 1).

```
class NodeL extends Node
inv  X: wfNodeL(X, $);
{
 public static NodeL initNodeL(int i)
 pre  I=aI(i);
 post ANodeL(result,$) = cons(I,null);
 pre  $=S;
 post result = new(S,NodeL) AND
      $=update(S##NodeL, loc(result,Node?elem), i);

 public int appback(NodeL n)
 req  n/=null AND n/=this AND lstNode?(this,$)
        AND fstNode?(n,$) AND lstNode?(n,$);
 pre  succn(X,$,N)=this AND ANodeL(X,$)=L AND ANodeL(n,$) = M;
 post ANodeL(X,$) = append(L,M);
 pre  $=S AND T=this AND X=n;
 post $=update(update(S, loc(T,Node?succ), X), loc(X,Node?pred), T);

 public NodeL getLast()
 pre  T=this AND $=S;
 post $=S AND result/=null AND
      (EXISTS (N:nat): succn(T,S,N)=result AND succn(T,S,N+1)=null);
}
```

To formalize the link structure for `NodeL`-objects, the following declarations and predicates are used. They should give an impression of what has to be available in a verification framework. For a detailed understanding, we assume that the reader is familiar with PVS syntax. By `<=` we denote the subtype relation on Java types which are represented by the constructor `ct` applied to their name:

```
NodeLObj?(X): bool = typeof(X) <= ct(NodeL)
NodeLObj?(X): TYPE = (NodeObj?)

fstNode?(X,S):bool = NodeLObj?(X) AND X/=null => S@@loc(X,Node?pred)=null
lstNode?(X,S):bool = NodeLObj?(X) AND X/=null => S@@loc(X,Node?succ)=null

predn((X: NodeObj), S, (n: nat)): RECURSIVE NodeLObj =
  IF X=null OR n=0 THEN X ELSE predn(S@@loc(X, Node?pred), S, n-1)
```

```
  ENDIF MEASURE n

succn((X: NodeObj), S, (n: nat)): RECURSIVE NodeLObj =
  IF X=null OR n=0 THEN X ELSE succn(S@@loc(X, Node?succ), S, n-1)
  ENDIF MEASURE n

wfNodeL((X: NodeLObj), S): bool =
  (EXISTS (i: nat): predn(X, S, i) = null)                         AND %(1)
  (EXISTS (k: nat): succn(X, S, k) = null)                         AND %(2)
  (NOT fstNode?(X,S) => typeof(S@@loc(X,Node?pred)) <= ct(NodeL) AND %(3)
   S@@loc(S@@loc(X,Node?pred), Node?succ) = X)                     AND %(4)
  (NOT lstNode?(X,S) => typeof(S@@loc(X,Node?succ)) <= ct(NodeL) AND %(5)
   S@@loc(S@@loc(X,Node?succ), Node?pred) = X)                         %(6)
```

The predicate `fstNode?`(`lstNode?`) holds for a `NodeL`-object X, if X has no predecessor (successor). `predn`(`succn`) yields the n-th predecessor (successor) of a `NodeL`-object. This allows us to formulate a wellformed condition for X using the predicate `wfNodeL`: X is well formed, if there exists a predecessor(successor)-object, whose `pred`(`succ`)-location is `null` (1+2). This guarantees `NodeL`-structures to be non cyclic. (3+5) describe that if X has a predecessor(successor) Y then Y is of type `NodeL`. (4+6) guarantee that inner objects of a `NodeL`-structure are correctly linked, by reaching itself via its predecessor(successor).

Wellformedness of `NodeL`-structures is a prerequisite for a correct method execution and abstraction. The wellformed condition `wfNodeL` constitutes the invariant for objects of the `NodeL` class. As expressed by the invariant clause of the interface specification of class `NodeL`, each method has to preserve this invariant.

To specify the functional behavior of `NodeL`'s methods an abstraction is needed to capture functional list properties. Therefore we use the following abstraction function with signature `Value, Store -> list[int]`, which abstracts `NodeL`-structures to the generic PVS list data-type list with type parameter `int`:

```
ANodeLn((X: NodeObj), (S: Store), (n: nat)):
 RECURSIVE list[int] = IF X=null OR n=0 THEN null
 ELSE cons(aI(S@@loc(X,Node?elem)),
           ANodeLn(S@@loc(X,Node?succ), S, n-1))
 ENDIF MEASURE n
```

Class `NodeL` extends class `Node` and adds a method `public int appback (NodeL n)`, which concatenates a `NodeL`-structure and a `NodeL`-object, referred by `this` and `n`. By this method we demonstrate the implementation dependency of a specification in contrast to the specification of class `DList`. The implementation dependency enables verification but is therefore not suitable for documentation and reuse. A basic requirement for a successful execution is that `n` refers a single `NodeL`-object and that `this/=n` holds. Furthermore appending is only allowed at the last object of a `NodeL`-structure. These requirements are summarized in the req-clause of method `appback`:

```
req n/=null AND n/=this AND lstNode?(this,$)
            AND fstNode?(n,$) AND lstNode?(n,$);
```

Furthermore `appback` has (1) a functional and (2) an environment behavior specification. (1) guarantees that `appback` does in fact let `n` become the new successor of `this` and all possible tails of the `NodeL`-structure with `this` at the end stay unchanged. (2) specifies that the store is changed by two location updates.

The specification techniques shown above are as well applicable to Java's interface types. To specify interface types, i.e. types without implementations, abstraction techniques can be exploited. Another interesting specification aspect occurs together with inheritance. A class S inherits a method m from class T without overriding the implementation. Nevertheless, a refined specification can be needed in the subclass. In our example, class `NodeL` inherits method `getSucc` from class `Node` and refines its specification. In addition to the specification given above class `NodeL` requires from its implementation that `pre ANodeL(this,$) = L; post ANodeL(result,$) = cdr(L);` holds.

Abstraction Functions for Class DList. Equipped with the datatypes and specification primitives above, we now present the rest of the specification of class `DList`, i.e. the wellformed condition and the abstraction. Both are needed to verify the implementation of `DList`. The wellformed condition used in the invariant looks as follows. A list is considered to be empty, if `firstNode` refers `null`. The abstraction `ADList: [Value, Store -> list[int]]` reuses the abstraction of `NodeL` and is equal to the abstraction of the `NodeL` object referred by `firstNode`.

```
wfDList((X: DListRef), S): bool =
 S@@loc(X,DList?firstNode)=null OR
  S@@loc(X,DList?firstNode)/=null AND
  cS@@loc(X,DList?lastNode)/=null AND
   EXISTS (n:nat): succn(S@@loc(X,DList?firstNode),S,n)=
                   S@@loc(X,DList?lastNode)

ADList_ax: AXIOM DListRef?(X) =>
           ADList(X, S)=ANodeL(S@@loc(X,DList?firstNode),S)
```

3 A logical Framework for Proof Construction

Within this section we give an overview of the formal framework used for specification and verification. As shown in the specification we have to express program independent and program dependent properties. Using PVS allows us to use the following technique: Theories containing formalizations of type identifiers, attribute-identifiers, and lemmata containing the subtype hierarchy are generated for all used classes. Program independent theories are generic w.r.t. the generated theories. Both parts together provide the formal background for specification and verification.

The used verification technique is based on a Hoare logic for the programming language we use. Partial correctness of programs w.r.t. their specification is shown by translating interface specifications into Hoare triples and proving them using the programming logic. Remaining implications which arise from the use of strengthening or weakening rules are proved by using PVS.

Figure 2 shows the result of the Hoare logic proof of the method `first` of class `DList`, where the functional property of `first` is proven. The use of some of the Hoare rules is displayed (formula parts touched by rules are underlayed grey): The var-rule allows to replace logical variables by local program variables, if they do not occur on the left hand side of an assignment; the inv-rule allows to conjoin formulas F to the pre- and postcondition of a triple, if F does not contain program variables or the variable `$` for the object store; the ex-rule allows to add existential quantifiers for logical variables, which do not occur in the postcondition. A proof outline embeds the information of a proof tree into the program text, which allows a flat representation of the proof tree. It can be read as follows. At the line containing the invocation of `getElem` the invocation statement is instantiated with the functional specification triple of that method, i.e. formal parameters within the specification are replaced by actual parameters. Above the method invocation, the axiom for location reads is instantiated. f, which is assigned a new value, is replaced by the term reading the location `this.firstNode` in store `$` in the postcondition and used as new precondition. The use of Hoare logic rules is displayed using the horizontal lines. Arrows point to the antecedent triple. The triple of the consequence is displayed outside the line-brackets. Strengthening and weakening steps are simply denoted by $\Longrightarrow$.

In the proof outline example, the program proof part is complete. It remains to show the implications, marked by $\Longrightarrow$. The proof that the implications hold is obvious in this example. This results directly from expanding the definitions of the abstraction functions and the axioms of the store formalization. The example shows, that in addition to the specification part, abstraction plays an important role during verification. Because of this degree of complexity, a theorem prover supporting these data type mechanisms is indispensable.

The complete proof for the example used in the case study was constructed with the JIVE environment. The JIVE-prover combines an interactive program prover with the general purpose theorem prover PVS to perform the program independent proof tasks.

4 Conclusion

In this extended abstract we showed some hot spots of a case study concerning the specification and verification of object-oriented programs. We demonstrated, how properties of object oriented programs can be described in a program independent abstract way. This allows for (1.) precise documentation of object-oriented programs and frameworks for reuse, and (2.) specifications, which can be directly used to prove the specified program properties. We presented specifications of non trivial program properties for the used list example, which are difficult to express precisely with operational specification techniques. Furthermore we gave a short sketch of the proof techniques, which are implemented within the JIVE proof environment. The JIVE system was used to construct the complete program proof of the case study coupled with the PVS prover.

Acknowledgments Marcel Labeth contributed to this work by constructing the program proof with the JIVE-System and proving all program independent proof obligations with PVS. We also thank Peter Müller for his important contributions to this work.

References

[Bec00] Bernhard Beckert. A dynamic logic for java card. In *Proc. 2nd ECOOP Workshop on Formal Techniques for Java Programs, Cannes, France*. TR 269, Fernuniversität Hagen, 2000. Available from `www.informatik.fernuni-hagen.de/pi5/publications.html`.

[COR+95] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. *A Tutorial Introduction to PVS*, April 1995.

[GH93] J. V. Guttag and J. J. Horning. *Larch: Languages and Tools for Formal Specification.* Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[Hui00] M. Huisman. *Reasoning about Java programs in higher order logic Using PVS and Isabelle.* PhD thesis, University of Nijmegen, 2000.

[JvdBH+98] B. Jacobs, J. van den Berg, M. Huisman, M. van Berkum, U. Hensel, and H. Tews. Reasoning about Java classes. In *Proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 1998. Also available as TR CSI-R9812, University of Nijmegen.

[Lea97] G. T. Leavens. Larch/C++ reference manual. Available from `http://www.cs.iastate.edu/~leavens/larchc++manual/lcpp_toc.html`, July 1997.

[LMMPH00] M. Labeth, J. Meyer, P. Müller, and A. Poetzsch-Heffter. Formal Verification of a Doubly Linked list implementation: A case study using the JIVE system. Technical Report 270, FernUniversität Hagen, 2000.

[MPH00] J. Meyer and A. Poetzsch-Heffter. An architecture for interactive program provers. In S. Graf and M. Schwartzbach, editors, *TACAS00, Tools and Algorithms for the Construction and Analysis of Software*, volume 276 of *Lecture Notes in Computer Science*, pages 63–77, 2000.

[Mül01] P. Müller. *Modular Specification and Verification of Object-Oriented Programs.* PhD thesis, FernUniversität Hagen, 2001.

[PHM99] A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In D. Swierstra, editor, *ESOP '99*, LNCS 1576. Springer-Verlag, 1999.

[vO01] D. von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic.* PhD thesis, Technische Universität München, 2001.

$$\{\ \mathit{this} \neq \mathit{null} \wedge \mathit{ADList}(\mathit{this}, \$) \neq \mathit{null} \wedge \mathit{ADList}(\mathit{this}, \$) = L\ \}$$

```
public int first() {
```

$$\{\ \mathit{this} \neq \mathit{null} \wedge \mathit{ADList}(\mathit{this}, \$) \neq \mathit{null} \wedge \mathit{ADList}(\mathit{this}, \$) = L\ \}$$

$$\Longrightarrow$$

$$\{\ \mathit{this} \neq \mathit{null} \wedge \$@@\mathit{loc}(\mathit{this}, \mathit{DList?firstNode}) \neq \mathit{null} \wedge \mathit{ADList}(\mathit{this}, \$) = L\ \}$$

$$\Longrightarrow$$

$$\left\{\begin{array}{l} \exists S, T : S = \$ \wedge T = \$@@\mathit{loc}(\mathit{this}, \mathit{DList?firstNode}) \wedge \mathit{this} \neq \mathit{null} \wedge \\ \$@@\mathit{loc}(\mathit{this}, \mathit{DList?firstNode}) \neq \mathit{null} \wedge \mathit{ADList}(\mathit{this}, S) = L \wedge T \neq \mathit{null} \end{array}\right\}$$

$$\overline{\hspace{30em}}\ \downarrow\text{–[ex-rule]}$$

$$\left\{\begin{array}{l} \mathit{this} \neq \mathit{null} \wedge \$@@\mathit{loc}(\mathit{this}, \mathit{DList?firstNode}) \neq \mathit{null} \wedge T \neq \mathit{null} \quad \wedge \\ \$ = S \wedge T = \$@@\mathit{loc}(\mathit{this}, \mathit{DList?firstNode}) \wedge \mathit{ADList}(\mathit{this}, S) = L \end{array}\right\}$$

$$\Longrightarrow$$

$$\left\{\begin{array}{l} \mathit{this} \neq \mathit{null} \wedge \$@@\mathit{loc}(\mathit{this}, \mathit{DList?firstNode}) \neq \mathit{null} \wedge \$ = S \quad \wedge \\ T = \$@@\mathit{loc}(\mathit{this}, \mathit{DList?firstNode}) \quad \wedge \\ \mathit{ADList}(\mathit{this}, S) = L \wedge T = S@@\mathit{loc}(\mathit{this}, \mathit{DList?firstNode}) \wedge T \neq \mathit{null} \end{array}\right\}$$

$$\overline{\hspace{30em}}\ \downarrow\text{–[var-rule]}$$

$$\left\{\begin{array}{l} \mathit{this} \neq \mathit{null} \wedge \$@@\mathit{loc}(\mathit{this}, \mathit{DList?firstNode}) \neq \mathit{null} \quad \wedge \\ \$ = S \wedge T = \$@@\mathit{loc}(\mathit{this}, \mathit{DList?firstNode}) \wedge T_1 \neq \mathit{null} \quad \wedge \\ \mathit{ADList}(T_1, S) = L \wedge T = S@@\mathit{loc}(T_1, \mathit{DList?firstNode}) \wedge T \neq \mathit{null} \end{array}\right\}$$

$$\overline{\hspace{30em}}\ \downarrow\text{–[inv-rule]}$$

$$\left\{\begin{array}{l} \mathit{this} \neq \mathit{null} \wedge \$@@\mathit{loc}(\mathit{this}, \mathit{DList?firstNode}) \neq \mathit{null} \quad \wedge \\ \$ = S \wedge T = \$@@\mathit{loc}(\mathit{this}, \mathit{DList?firstNode}) \end{array}\right\}$$

```
Node f; f = this.firstNode;
```

$$\{\ f \neq \mathit{null} \wedge \$ = S \wedge T = f\ \}$$

```
int k; k = f.getElem();
```

$$\{\ \mathit{aI}(k) = \mathit{ANode}(T, S)\ \}$$

$$\overline{\hspace{30em}}\ \uparrow\text{–[inv-rule]}$$

$$\left\{\begin{array}{l} \mathit{aI}(k) = \mathit{ANode}(T, S) \wedge T_1 \neq \mathit{null} \wedge \mathit{ADList}(T_1, S) = L \quad \wedge \\ T = S@@\mathit{loc}(T_1, \mathit{DList?firstNode}) \wedge T \neq \mathit{null} \end{array}\right\}$$

$$\overline{\hspace{30em}}\ \uparrow\text{–[var-rule]}$$

$$\left\{\begin{array}{l} \mathit{aI}(k) = \mathit{ANode}(T, S) \wedge \mathit{this} \neq \mathit{null} \wedge \mathit{ADList}(\mathit{this}, S) = L \quad \wedge \\ T = S@@\mathit{loc}(\mathit{this}, \mathit{DList?firstNode}) \wedge T \neq \mathit{null} \end{array}\right\}$$

$$\Longrightarrow$$

$$\{\ \mathit{aI}(k) = \mathit{car}(L)\ \}$$

```
return k;
```

$$\{\ \mathit{aI}(\mathit{result}) = \mathit{car}(L)\ \}$$

$$\overline{\hspace{30em}}\ \uparrow\text{–[ex-rule]}$$

$$\{\ \mathit{aI}(\mathit{result}) = \mathit{car}(L)\ \}$$

```
}
```

$$\{\ \mathit{aI}(\mathit{result}) = \mathit{car}(L)\ \}$$

Fig. 2. Example for a Proof Outline of a Hoare-logic Proof