Making Executable Interface Specifications More Expressive

Peter Müller, Jörg Meyer, and Arnd Poetzsch-Heffter

Fernuniversität Hagen, D-58084 Hagen, Germany
[Peter.Mueller, Joerg.Meyer, Arnd.Poetzsch-Heffter]@Fernuni-Hagen.de

Abstract. Executable interface specification languages allow for expressive documentation and efficient testing and debugging. Since they are based on expressions of the underlying programming language, they can easily be applied by programmers without requiring mathematical skills. In this paper, we present the core of an executable interface specification language for Java. Its main contributions are an extensive coverage of side-effects on object structures, and a clean semantics. The presented techniques can be implemented without modifications to the Java compiler or the virtual machine.

1 Introduction

Language support for the specification of methods and classes is one of the most wanted extensions to Java (cf. [Sun], Bug Id 4071460). This paper provides an overview over an interface specification language for Java. The presented techniques improve comparable approaches by making four contributions: (1) clarification of semantical aspects, (2) specification methodology for executable specifications, (3) specification techniques for expressing properties of linked object structures, and (4) a simple implementation technique for the developed features.

Interface specifications of object-oriented programs typically express method properties by pre- and postconditions and class properties by so-called invariants. Interface specifications are a precise means for documentation. They describe the contract between the user and provider of a software component (cf. [Mey92a]).

We distinguish between declarative and executable specifications (cf. [Luc90] for an approach to combine these techniques). Declarative specifications are based on an extended logical framework relating the operational world of programs to the declarative world of theorem provers (cf. [GH93], [PH97]). They are very expressive (e.g., universal and existential quantification over objects, explicit abstraction from the program level) and appropriate for mechanical theorem proving, but in general non-executable. In executable specifications, program properties are usually formulated based on the constructs of the underlying programming language, in particular by boolean expressions (cf. [FM98], [Mey92b]). In an object-oriented context, executable specifications have three important advantages over their declarative counterparts: (1) As direct extension of the underlying programming language, they are easier to learn and simpler

to use. (2) They provide a powerful support for testing and debugging. (3) The OO-features of the programming language (in particular dynamic binding) can be exploited in specifications.

Executable techniques have two central drawbacks for the specification of OO-programs: 1. Abstraction cannot be expressed in the canonical way by functions from data objects of the programming language to values in an abstract domain. 2. Side-effects on and modifications of linked object structures are more difficult to handle. In this paper, we present new executable specification techniques to overcome these drawbacks without sacrificing the advantages of executable specifications.

Overview Section 2 explains the basic specification aspects. Section 3 concentrates on techniques for the specification of linked object structures. Section 4 describes the implementation method.

2 Specification Technique

This section explains our specification method, provides an overview over the **J**ava **i**nterface **s**pecification **l**anguage JISL, and illustrates the techniques by an example.

2.1 Specifying Interfaces

Executable interface specification languages typically use an extended expression syntax of the underlying programming language to specify method behavior and class invariants. Sometimes sophisticated additional constructs are provided for this purpose, e.g., to handle bounded quantification, object creation, reachability of objects, etc. (cf. [LBR99]). We make only use of quantification over finite integer ranges and so-called old-expressions of the form `old(e)` where `e` is an expression. `old` expressions may only occur in postconditions. The value of `old(e)` is the value of `e` evaluated in the corresponding prestate.

In this paper, we concentrate on aspects that we consider improvements compared to existing approaches, namely methodological issues, semantical aspects, and specification of side-effects on linked object structures. We illustrate our techniques by the following two class fragments taken from the Java AWT[1]:

```
class Component {
  Container parent;
  int x; int y;
  Object placeHolder;
  ... }

class Container extends Component {
  int ncomponents;
  Component component[] = new Component[4];
```

[1] The field `placeHolder` is used to represent all omitted fields.

```
public int getComponentCount()              { return ncomponents; }
public Component getComponent(int i)        { return component[i]; }
public Component add(Component comp)        { ... }
... }
```

Methodology. An interface specification language should support a specification methodology to provide guidance for developing specifications. In particular, the methodology should support data abstraction for expressing properties of classes without referring to the actual, possibly private implementation parts. In declarative specifications, abstraction (1) is implicitly assumed (cf. [GH93]) or (2) has to be described within the specification framework (cf. [PH97]). Both solutions are inappropriate for executable specification frameworks: The first one is incomplete and would destroy executability. The second cannot be used, because executable specifications based on Java expressions do not provide a sufficiently abstract language layer.

Therefore, we exploit the basic idea of observability in abstract data type theory: Instead of mapping object structures to complex values, abstraction of objects and object structures is expressed by so-called *observer methods* that allow one to inspect the states of objects and object structures without modifying them. The methodology is as follows: Each class has a set of observer methods (observers for short). E.g., a list has two observers: the length and the ith element with i less or equal to the length. Observers can be existing methods of a class or they can be introduced to characterize the abstract properties of a class. Observers are usually simple methods that correspond to informal properties of the abstract type implemented by a class. Observers abstract from the concrete implementation and allow one to change the implementation without affecting the interface specification by adapting the observers to the new implementation. Non-observer methods of a class are specified using the observers.

We add the following observers to the above classes.[2] The methods `getComponentCount` and `getComponent` of class `Container` are observers. The following specification of `getComponent` illustrates the application of observers:

```
public observer Component getComponent(int i)
  PRE  0 <= i  &&  i < getComponentCount()
  POST true
```

Furthermore, we add an observer `contains` that yields whether a `Container` object contains a given component:

```
public observer boolean contains(Component comp) { ... }
```

The well-formedness of components is expressed by `wf` which yields whether the component is correctly linked to its container:

```
public observer boolean wf()
  { return (parent == null || parent.contains(this)); }
```

[2] For reasons that are described later, we mark observer methods with the keyword `observer`.

A well-formed container must fulfill additional constraints which can be specified by overriding `wf` in `Container`:

```
public observer boolean wf() {
  for (int i = 0; i < ncomponents; i++)
    if (!(component[i] != null  &&  component[i].wf()  &&
          component[i].parent == this))          return false;
  return super.wf();         }
```

This example demonstrates another aspect of the abstraction provided by the observer technique: By overriding, observers can easily be adapted to the requirements of subclasses. Due to dynamic binding, a specification requiring e.g. the well-formedness of a parameter of static type `Component` will always refer to the appropriate definition of `wf`.

Clarification of Semantics. Executable interface specifications are checked at runtime. To be useful for testing and debugging, it is crucial that evaluating specifications does not affect program behavior (except for performance). Thus, specifications must not produce side-effects on the specified program. Although this rule seems almost trivial, it is not enforced by other executable specification languages. To guarantee the absence of side-effects during execution of specifications, we require that specifications do not contain writing field accesses. Since all observers are marked with a keyword, this property can easily be checked: Observers, pre- and postconditions must neither contain writing field accesses nor invocations of non-observer methods. This fairly strict but simple rule caused no problems in the examples we considered so far. A more elaborate technique would be to use data flow analysis to check that a specification does not affect the program behavior.

2.2 Example

To demonstrate the expressivity of our interface specification language, we specify the behavior of method `add` in class `Container`. This method is supposed to behave as follows (slightly simplifying the original AWT method):

1. The method guarantees the following properties under the precondition that both the container and the parameter component are well-formed upon invocation of `add`, and the component is not contained in any container.
2. The number of contained components is increased by one.
3. `comp` is added as component with highest index to `this`.
4. `add` returns `comp`.
5. Components contained before execution of `add` stay unchanged.
6. Except for the `parent` field, the parameter component is not modified.

Properties 1–4 are expressed by the following pre-post-pair (note that well-formedness of the container includes well-formedness of all associated components, in particular the `parent` field of `comp` has to be updated):

```
public Component add(Component comp)
 PRE  wf() && comp.wf() && comp.getParent() == null
 POST wf() && getComponentCount() == old(getComponentCount()) + 1  &&
      this.getComponent( old(getComponentCount())+1 ) == old(comp) &&
      result == old(comp)
```

The identifier `result` refers to the value returned by a method. Properties 5 and 6 cannot be expressed by the language features described so far since they require to compare whole object structures in two different execution states whereas `old` allows only for the comparison of values. We address this important aspect in the next section.

3 Specification of Frame Properties

In this section, we present a flexible technique for specifying the absence of side-effects on object structures. The basic idea is to explicitly mark those parts of the heap memory that are supposed to be left unchanged by method execution. Dynamic checks are used to detect modifications of marked objects.

3.1 Specifying the Absence of Side Effects

In this subsection, we argue that specification languages should allow one to specify the absence of side-effects and sketch two basic approaches to this task.

Malevolent Side-Effects. A side-effect is a modification of a field instance (a so-called *location*). In Java, side-effects are caused by field updates. The use of side-effects is very common in OO-programming since they allow for efficient implementations. However, many program errors are due to unwanted or overlooked side-effects. In contrast to errors in the functional behavior of methods, unwanted side-effects are very hard to detect since their malevolence often becomes evident long after the side-effect happened.

Therefore, an interface specification language should allow one to specify the absence of side-effects on certain object structures (so-called *frame properties*). Violations of these specifications should be detected as early as possible to inform the programmer or tester which field update caused the unwanted side-effect.

Approach. There are two basic approaches to the specification of frame properties: (1) One can explicitly mention those locations in the specification, that must not be modified by a method. (2) One can enumerate the locations that may be modified by a method in a so-called *modifies-clause*. All locations not contained in the modifies-clause have to remain unchanged. This technique is very common in declarative interface specification languages (cf. e.g. [MPH99]), but for OO-programs, it requires a rather complex specification framework. Furthermore, in the context of observer-based specifications, the first approach is more natural, since it uses observers that allow one to compare the value of a location in two different execution states (the pre- and poststate of a method). Therefore, we will elaborate on the first approach in this paper.

3.2 A Specification Technique for Frame Properties

Checking frame properties requires the ability to compare an object structure in two execution states. To do that, we introduce a specification primitive `unchanged(`S`)` where S denotes an object structure. In this subsection, we explain how object structures can be described in a flexible way, and how specifications containing `unchanged` can be dynamically checked.

Specification of Object Structures. An object structure is a set of objects linked by references. Since objects consist of locations which hold either values or references to other objects, object structures can be modeled as sets of locations and their values. Figure 1 illustrates the object structure of a `Container` which contains one `Component` (boxes denote locations, arrows denote references; the shaded areas depict objects).

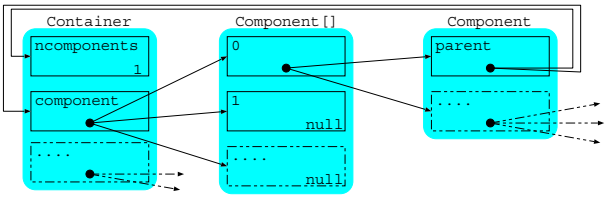

Fig. 1. Object Structure of a Container Object

The illustrated structure consists of all locations that are reached from a given `Container` object by following chains of references. Such structures are useful to specify that a whole object structure is not modified by a method. However, to specify that a method performs some updates on a structure but leaves the bigger part unchanged, means are required to describe restricted object structures. E.g., for method `setLocation` of class `Component` one would like to specify that the object structure of `this` is left unchanged except for locations `x` and `y`.

To allow programmers to specify arbitrary object structures in a flexible way, we introduce a primitive type `structure` with three operations: The empty structure is denoted by `<>`. `<e.f>` describes the structure consisting only of the location `e.f` where `e` is an expression and `f` is a field name. The join operation of two structures is denoted by `+`. Object structures are described in observer methods that consist only of a single return statement with a `structure`-valued expression (see below for an example). Restricting the body of such observers simplifies the implementation of JISL without seriously limiting the expressivity (the conditional operator `?` and method invocations can be used instead of conditional statements and loops). For a more convenient notation, we assume a public observer `getReachLocs()` to be predefined for every class. It describes the structure of all locations that are reachable from `this`.

To enable efficient checking of specifications, we do not really create values of type `structure` (cf. next section). For this reason, it is not allowed to use `structure` as type of local variables, formal parameters, fields, or anywhere else except as return type of observer methods.

The `unchanged` Expression. We express the fact that a structure `S` is left unchanged by a method `m` by adding `unchanged(S)` to `m`'s postcondition. `unchanged` is an operator of the specification language which takes an argument of type `structure` and yields a boolean value. Since `unchanged` compares a structure in the prestate to the corresponding structure in the poststate, two restrictions apply: (1) `unchanged` may only appear in postconditions. (2) The argument of `unchanged` must be defined in both states. I.e., it must not contain `result`, and the ranges of quantified variables must stay unchanged during method execution. This can be achieved by using constants and `old` expressions to denote the ranges (see below for an example).

There are two possibilities for evaluating the `unchanged` operator:

1. Upon entry of a method the postcondition of which contains an `unchanged(X)` expression, the structure `X` is copied and stored. When the postcondition is checked, the stored structure is compared to the current structure of `X`. `unchanged(X)` yields whether both structures are identical.
2. When a method `m` with an `unchanged(X)` expression in its postcondition is entered, all locations of `X` are marked to be unmodifiable. Every writing field access is dynamically checked not to modify marked locations. Otherwise, an exception is thrown. When `m`'s postcondition is evaluated, the marks are removed. This behavior guarantees that `m` does not modify the object structure `X`. Thus, `unchanged(X)` always yields true.

The above possibilities differ in two important aspects:
Semantics: Solution 1 allows one to temporary modify an object structure and re-establish its initial value before the postcondition containing `unchanged` is evaluated. This seems natural, but defers detection of unwanted side-effects until the postcondition is checked. That makes debugging difficult.
Performance: Solution 1 requires to store object structures for every incarnation of a method containing `unchanged` expressions in its postcondition. This is very time-consuming and leads to an extreme waste of memory, in particular for recursive methods. In contrast, solution 2 can be implemented with far less overhead: Each field has to be supplemented by a mark. Writing field access has to check for a mark before updating a location. That leads to a time and memory overhead of about 100%, which is an acceptable value for debug runs.

Therefore, we favor solution 2. Its implementation is described in Section 4.

3.3 Example

In this subsection, we revisit the example introduced in Section 2. In the following, we will specify the frame properties of `Container`'s `add` method: (1) The parameter component stays unchanged except for the `parent` field. (2) Components that are already part of the container are not modified.

For the first property, we specify a substructure for `Component` which contains the whole object structure except the locations reachable via the `parent` field:

```
public observer structure woParent()
  { return this.placeHolder.getReachLocs() + <this.x> + <this.y>; }
```

This observer method allows us to specify property (1) by conjoining `unchanged(comp.woParent())` to `add`'s postcondition.

Execution of `add` modifies the container in several ways (see Section 2). However, the object structures of components that are referenced by the container before `add` is invoked stay almost unchanged: only the enclosing container (reachable via `parent`) is modified. Thus, we can again use the `woParent` substructure to specify this behavior:

```
FORALL (i: 1..old(getComponentCount())):
              unchanged(getComponent(i).woParent())
```

This example demonstrates that the proposed specification technique is expressive enough to describe the absence of side-effects in an intuitive and flexible way, even for mutually recursive object structures. The next section will describe how our techniques can be implemented in Java.

4 Implementation Aspects

In this section, we describe the implementation of JISL. In particular, we present a purely Java-based technique to realize the `unchanged` operation with an acceptable space and runtime overhead.

Execution of Interface Specifications. To enable a purely Java-based implementation of JISL, a preprocessor translates specifications into Java code that is inserted into the specified program. The generated code checks the pre- and postconditions and throws a `RuntimeException` if a specification is violated.

In the following, we focus on the `unchanged` expression which is the most interesting specification primitive. For details on the implementation of the other constructs, the reader is referred to [Mül95].

Realization of the `unchanged` Operation. An `unchanged(X)` expression in a postcondition describes that the structure `X` has not been changed since the control flow has passed the corresponding precondition. As described in Section 3, this property can be checked by (1) marking all locations of `X` in the corresponding precondition, (2) allowing updates for unmarked locations only, and (3) unmark the locations of `X` when evaluating the postcondition. We explain the realization of these steps in the following.

Markers for Locations. To mark a location `x.v`, we associate each field `v` of a class with a mark `marker$v`, which has the same access modifier as `v`. Recursive or concurrent invocations of a method can lead to multiple marking of one location. Therefore, markers have to be realized as counters. We use marker fields of type `long`. Markers are initialized to zero. A location `x.v` is marked, if `x.marker$v` > 0 holds. E.g., for the `Component` class, the following markers are introduced:

```
long marker$parent=0;  long marker$placeHolder=0;
long marker$x=0;       long marker$y=0;
```

If the control flow of a thread reaches the precondition of a method whose postcondition contains `unchanged(X)`, `X` is evaluated and the markers of all locations of `X` are incremented by one. When the corresponding postcondition is evaluated, all markers of structure `X`[3] are decremented. The code for the increment and decrement operations is derived from the structure specification (see below).

Controlling Write Access to Locations. Location updates must consider the value of markers of their target locations. This can be achieved by replacing all writing field accesses by invocations of appropriate access methods. These methods perform a normal location update for unmarked locations and throw an exception if the target location is marked. E.g., for `Component`'s `parent` field, the following access method is introduced:[4]

```
static Container set$parent(Compoent obj, Container value)      {
  if (obj.marker$parent > 0) throw new UnchangedSpecException();
  else                       return obj.parent = value;         }
```

A field access `c.parent = e` is replaced by `Component.set$parent(c, e)`.

Mark and Unmark Operations. As stated in Section 3, object structures are described by `structure`-valued expressions (SE for short). Structures are marked and unmarked as follows: Each SE is translated into a $\mathcal{M}$ark and an $\mathcal{U}$nmark statement. For every `structure`-valued observer `m` the methods `mark$m` and `unmark$m` are introduced. Essentially, the body of these methods is obtained by applying $\mathcal{M}$ and $\mathcal{U}$ to the expression returned by `m`. The following table shows the translation of an SE into the mark operation $\mathcal{M}$ ($\texttt{se}_i$ denotes an SE and `jexpr` a Java expression; `m` is a `structure`-valued observer):

`<>`	→ `;`	`// empty structure`
`<jexpr.a>`	→ `jexpr.marker$a++`	`// instance field`
`jexpr.m(`$\texttt{p}_1$`, ..., `$\texttt{p}_n$`)`	→ `jexpr.mark$m(`$\texttt{p}_1$`, ..., `$\texttt{p}_n$`)`	`// method invocation`
$\texttt{se}_1$ `+` $\texttt{se}_2$	→ $\mathcal{M}$`(`$\texttt{se}_1$`) ; `$\mathcal{M}$`(`$\texttt{se}_2$`)`	`// structure union`

For the observer `woParent`, the following mark statement is generated:

```
{ placeHolder.mark$getReachLocs(); marker$x++; marker$y++; }
```

Besides executing such statements, mark methods have to perform two tasks: (1) Marking and unmarking has to be synchronized to prevent concurrent threads from invalidation marking information. (2) The operations have to take care that cyclic object structures do not lead to non-terminating mark/unmark operations. This can be achieved by using standard techniques for graph traversal.

For each `unchanged(X)`, $\mathcal{M}$`(X)` and $\mathcal{U}$`(X)` are executed when evaluating the precondition and the postcondition, resp. The expression itself always evaluates to `true` (see Section 3).

[3] Note that the postcondition refers to the structure computed in the prestate. I.e., `unchanged(X)` is equivalent to `unchanged(old(X))`.

[4] Access methods have to be static to simulate static binding of attributes in Java.

Problems and Workarounds. By the technique described above, JISL can be implemented as preprocessor without modifying the Java compiler or virtual machine. However, the simplicity of this solution entails some problems: (1) By means of native methods or reflection, programmers can update locations without using the access methods. To observe location updates by native methods, the virtual maschine has to be modified. For using reflection, the `Field` class has to be adapted such that it pays attention to marked locations. (2) The generation of marker fields and methods produces overhead for even those classes the fields of which are never marked to be unmodifyable (e.g., event objects). Enhanced static analysis of programs can reduce this overhead.

5 Conclusions

We presented an executable interface specification language for Java. Compared to existing languages, our proposal has three important advantages: (1) Our methodology provides abstraction from implementation details by using observer methods. (2) Interface specifications have a clean semantics since they are guaranteed not to have side-effects. (3) We have developed a new technique for specifying and checking frame properties of methods by describing and marking object structures. These features allow for expressive specifications and powerful testing and debugging support. An implementation of the interface specification language as described in Section 4 is considered further work.

References

[FM98] C. Fischer and D. Meemken. JaWa: Java with assertions. In C. H. Cap, editor, *JIT '98 Java-Informations-Tage 1998*. Springer-Verlag, 1998.

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

[LBR99] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06c, Iowa State University, Department of Computer Science, January 1999.

[Luc90] D. C. Luckham. *Programming with Specifications: An Introduction to Anna. A Language for Specifying Ada Programs*. Springer-Verlag, 1990.

[Mey92a] B. Meyer. Design by contract. In D. Mandrioli and B. Meyer, editors, *Advances in object-oriented software engineering*. Prentice Hall, 1992.

[Mey92b] B. Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[MPH99] P. Müller and A. Poetzsch-Heffter. Modular specification and verification techniques for object-oriented software components. In G. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*. Cambridge University Press, 1999. (to appear).

[Mül95] P. Müller. Specification and implementation of an annotation language for an object-oriented programming language. Master's thesis, Technische Universität München, 1995. (In German).

[PH97] A. Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitation thesis, Technical University of Munich, Jan. 1997. URL: `www.informatik.fernuni-hagen.de/pi5/publications.html`.

[Sun] Sun. Java developer connection. Available from `http://java.sun.com/jdc`.