Programming and Interface Specification Language of JIVE — Specification and Design Rationale

Peter Müller, Jörg Meyer, Arnd Poetzsch-Heffter
Email: [Peter.Mueller, Joerg.Meyer, poetzsch]@Fernuni-Hagen.de
Fachbereich Informatik
Fernuniversität Hagen
D-58084 Hagen

Abstract

This report describes the programming and interface specification language of the Java Interactive Verification Environment JIVE. The JIVE system is a prototype implementation of a logic-based programming-environment for an object-oriented programming language. Logic-based programming-environments are language-dependent software development tools that support formal specification and verification.

We summarize the properties of an ideal programming language for the prototype and argue that Java is a good candidate. The design of the supported Java subset is discussed and a formal definition of the abstract syntax is presented.

Program specifications are denoted in an interface specification language. This report discusses the design of the JIVE interface specification language and presents its abstract syntax. An example program illustrates the application of the programming and the interface specification language.

Contents

Chapter 1

Introduction

The **J**ava **I**nteractive **V**erification **E**nvironment JIVE is a logic-based programming environment developed in the Lopex research project[1] at Fernuniversität Hagen and Technische Universität München[2]. Lopex stands for **lo**gic-based **p**rogramming **e**nvironments **c**onstructed from formal language **s**pecifications. Logic-based programming environments are software development tools which support formal specification and verification of programs. We aim at generating such programming environments from formal specifications of typical procedural or object-oriented programming languages. In the first phase of the project, we build a prototype (called JIVE) for a subset of Java. This report gives a rationale for the selection of Java and defines the subset used in JIVE. Furthermore, the interface specification language is presented.

General Approach Our approach to specification and verification builds as much as possible on well-known techniques. The specification technique used in Lopex is based upon the two-tiered Larch approach (cf. [GH93]). Program specifications consist of two major parts: (a) A program-independent specification which provides the mathematical vocabulary (e.g., definitions of abstract data types) and (b) a program dependent part that relates the implementation to universal specifications. An interface specification of a class C consists of (a) a specification for each public method of C, and (b) a class invariant. Method specifications are given by pre- and postconditions. Class invariants describe properties that have to hold for each object of a class in any state where the object is accessible from outside.

For verification, we use a Hoare-style programming logic (cf. [Hoa69]) as presented in [PH97]. This logic is a formalization of the axiomatic semantics of the underlying programming language. Thus, correctness of a program is showed by translating its specification into Hoare triples and proving these triples in the logic.

System Architecture Some of the design decisions described in this report are motivated by the tools used in our prototype. Thus, we have to take a quick look at the system architecture of a logic-based programming environment. The architecture of JIVE consists of three major components: (1) the **p**rogram **v**erification **c**omponent PVC, (2) the **v**erification **m**anagement **c**omponent VMC, and (3) the **t**heorem **p**rover **c**omponent TPC.

[1] www.informatik.fernuni-hagen.de/pi5/forschung/lopex_en.html

[2] The Lopex project is sponsored by Deutsche Forschungsgemeinschaft (DFG).

Specifications and proofs can be split into program dependent and program independent parts. In general, all program dependent parts are carried out in the PVC. I.e., the PVC provides support for editing programs and interface specifications and for proving triples in the programming logic. To build the PVC, we use the Synthesizer Generator, a tool for generating language-based editing environments (cf. [RT89]).

Program independent proof obligations (i.e., first order formulas) that result from certain rule applications of the programming logic are proved in the TPC. Thus, we use one of the elaborated theorem provers/proof checkers like PVS ([COR+95]) or Isabelle ([Pau94]) for that purpose. The specification language of the TPC is as well used to formalize the program independent parts of specifications (see above).

The VMC manages the proofs for a program component and keeps track of the remaining proof obligations. In particular, the VMC decides whether a program is completely verified.

Overview The rest of this report is structured as follows: Chapter 2 provides a rationale for the selection of Java for the Lopex prototype and defines the language subset supported by JIVE. Interface specifications are given as so-called annotations which are denoted in an interface specification language. The interface specification language of JIVE is presented in chapter 3. Chapter 4 discusses the programming and interface specification language by an example program. Our conclusions and a summary of further work are contained in chapter 5.

Chapter 2

Programming Language

In this chapter, we describe the demands a programming language for the Lopex prototype has to meet. We argue why Java is a good choice for a starting point. The main part of this chapter defines a subset of Java, which is suitable for the purpose of verification, and specifies its abstract syntax.

2.1 Design Concerns

The Lopex project aims at the generation of logic-based programming environments for procedural and object-oriented programs. This implies requirements for the programming language used in the prototype:

1. The programming language should be object-oriented. This has essentially three reasons: (a) Most procedural languages are subsets of the common object-oriented languages. Thus, programming environments for object-oriented languages are more general in a sense that they allow the treatment of procedural programming languages as well. (b) The most interesting problems for specification and verification are caused by object-oriented language features, namely subtyping, inheritance, and dynamic binding. (c) Object-oriented languages provide encapsulation. Encapsulation eases verification as it guarantees the absence of certain side-effects.

2. The language of the prototype should be representative for a large class of object-oriented and procedural languages (containing C++, Java, Eiffel, Sather, Oberon, Modula-3, Simula, BETA, and Ada95). I.e., it has to provide all language features that can be found in common object-oriented languages, e.g. a class concept, strong typing, encapsulation, subtyping, dynamic binding, inheritance, and exception handling. Furthermore, this features should be provided in a way that is typical for the whole language class.

3. To enable a stepwise extension of the prototype, the features of the programming language should be orthogonal. This allows to start with a subset of a language and to enhance the prototype by adding further language features during progress of the project.

4. Verification heavily relies on an elaborated module concept. Modules provide an additional level of encapsulation which eases the verification of invariants (cf. [MPH97a]

for the notion of module concepts supporting semantically private types). Thus, the language of the prototype should provide a module concept we can build on.

In the next subsection, we summarize the reasons for the selection of Java. We compare Java to other object-oriented language w.r.t. the above criteria.

2.2 Rationale for the Selection of Java

The Lopex prototype JIVE uses Java as programming language. Java fulfills the requirements described above:

1. Java is a modern object-oriented language.

2. Java provides all important object-oriented language features: a class concept, strong typing, encapsulation, multiple subtyping, dynamic binding, and single inheritance. These features are supported in a way that is typical for the language class mentioned above.

3. Most of the Java language features are orthogonal. E.g., exception handling, threads and monitors, the package concept, and arrays can be omitted in the subset without need for semantical changes to the rest of the language. However, subtyping and inheritance are closely connected as classes can only inherit from direct supertypes. In this context, this is not regarded as a disadvantage because (a) connecting subtyping and inheritance is typical for the relevant language class and (b) both features are supported by the subset used in JIVE anyway.

4. Java provides a package concept which enables encapsulation of types and a kind of friend mechanism for classes of one package. This is a good basis to build on with more elaborated module concepts.

Beside these requirements, Java has several advantages that influenced our decision:

1. Java is a very modern language. Although the Java-fever seems quite exaggerated, it causes some positive effects for our project: (a) Research projects from all over the world deal with Java. In particular, Nipkow and von Oheimb formally proved a Java subset type correct (cf. [NvO98]). (b) Java is used in commercial software development more and more often. Thus, cooperations with industrial partners become more likely. (c) The immense interest of students in Java eases sourcing out parts of the project as diploma theses.

2. Java comes with a small class library our system can be applied to. More extensive evaluation can be done by verifying larger class libraries (e.g., the Java Algorithm Library, cf. [AS96]) or by proving the correctness of program components based on JavaBeans (cf. [Ham97]).

3. Java allows the development of concurrent and distributed programs, e.g. by providing threads, monitors, and remote method invocation. Thus, extending JIVE to concurrent and distributed programs is possible without switching to another programming language.

Of course, Java is not the only programming language that would be suitable for the Lopex prototype. In the following, we discuss the advantages and disadvantages of some other candidates.

Sather and Eiffel provide interesting concepts for inheritance and genericity. Both languages fulfill most of the requirements presented in section 2.1 (except module concepts). In the end, we ruled out Eiffel and Sather because their approach to inheritance is very unusual and not typical for the language class we aim at.

C++ has several important disadvantages compared to Java: (1) Pointer arithmetics enables the violation of data encapsulation. (2) C++ is weakly typed as it allows a very liberal use of casts. (3) C++ has a very complex and in many cases unclear semantics. Thus, a correct programming logic can hardly be found. Of course, most of this drawbacks can be overcome be defining an appropriate subset of the language. But this subset would essentially look like Java.

Ada95 can be regarded as an object-oriented extension to Ada. I.e., object-oriented features are added by combining, modifying, or generalizing the constructs of Ada (e.g., the class concept is realized by a combination of types, procedures, and packages). As a consequence, it is nearly impossible to define a small subset of Ada95 which can be regarded as the core of object-oriented languages.

Smalltalk programs are purely structured due to the syntax of statements. This complicates verification. Furthermore, Smalltalk is an untyped language. Type correctness of programs is a prerequisite for verification. Thus, typing properties of Smalltalk programs would have to be proved separately which causes additional effort.

BETA was not chosen for the Lopex prototype because of its unusual program structure. BETA generalizes classes and methods to a universal pattern construct. Although our technique could be applied to patterns as well, the results would not carry over to other languages naturally.

To sum up, among the practical important object-oriented languages, Java is our favorite because it fulfills all requirements of section 2.1 and provides some additional benefits as pointed out above. Due to limited time and money of the Lopex project, we can not handle full Java. Thus, we define a subset of Java which provides all important features of object-oriented languages and can thus be regarded as a kernel of this language class. The definition of this subset is presented in the next subsection.

2.3 Specification of a Java Subset

The Java subset used by JIVE is called SVENJA (**s**mall **v**erification **en**abled **Ja**va). Is is based on Java version 1.0 (cf. [GJS96]).

Design Concerns The design of SVENJA was influenced by five goals:

1. SVENJA has to be a subset of Java in a sense that SVENJA programs can easily be transformed into semantically equivalent Java programs. All features of SVENJA should have the same semantics as the corresponding Java features. Furthermore, the syntax of SVENJA should be as close to the Java syntax as possible.

2. SVENJA has to provide all important object-oriented language features such as a class concept, strong typing, encapsulation, subtyping, dynamic binding, and inheritance.

3. SVENJA has to provide the main features of imperative programming languages (e.g., recursion, iteration, basic data types, etc.)

4. SVENJA should be as small as possible (w.r.t. goals 2 and 3) to focus on the central problems.

5. Wherever it is reasonable, SVENJA should be as close as possible to BALI (cf. [NvO98]). This eases our cooperation with Nipkow and von Oheimb.

Notation The syntax of SVENJA is presented in the Synthesizer Generator notation (see introduction). The form of a production declaration is

```
phylum :  operator-name_1  ( phylum_1^1 phylum_1^2 ... phylum_1^k1 )
       |  operator-name_2  ( phylum_2^1 phylum_2^2 ... phylum_2^k2 )
          ...
       |  operator-name_i  ( phylum_i^1 phylum_i^2 ... phylum_i^ki )
       ;
```

In the context of this report, a *phylum* can be regarded as a terminal or non-terminal of the grammar. (The left-hand-side phylum of each production has to be a non-terminal.) Each line of a declaration defines a possible right-hand-side for the left-hand-side phylum. Furthermore, it specifies the name of a constructor (called operator) that builds terms of the left-hand-side phylum from terms of the right-hand-side phyla.

For SVENJA, we assume three terminal symbols *IDENTIFIER*, *INTLITERAL*, and *BOOLLITERAL* which are defined in the scanner specification. To enable editing of incomplete programs, there exists a placeholder for each left-hand-side phylum. A placeholder consists of an operator without parameters.

Conventions All right-hand-sides for a phylum are grouped together as shown above. The operator names are prefixed by "*op_*". Each declaration starts with the placeholder production. The operators of placeholders are denoted by *op_phylumNil* where *phylum* is the name of the left-hand-side phylum. Wherever it is possible, we use the non-terminal symbols of the Java LALR(1) grammar (cf. [GJS96], chap. 19) as phylum names.

Structure The presentation of the syntax follows the structure of the Java Language Specification ([GJS96]). I.e., we use the same non-terminal names and the same arrangement of productions wherever it is reasonable. In some cases, this results in a grammar that is slightly more complex that it would be if we developed a completely new grammar. But sticking to the structure of the Java grammar allows extending the subset without major changes of the existing parts.

In the following, *subsections* of this report correspond to *chapters* of [GJS96] and *subsubsections* of this report correspond to *sections* of [GJS96] wherever this is reasonable. Furthermore, we adopt the segment headings of [GJS96].

2.3.1 Lexical Structure

Most parts of the lexical structure of a programming language belong to the scanner specifications rather than to the syntax. The scanner specification of SVENJA is not presented

here. It is very similar to Java, i.e., SVENJA uses the same keywords and the same lexical structure of literals and identifiers. In contrast to Java, SVENJA identifiers may not contain the $ character to avoid ambiguities with the interface specification language (see chapter 3).

2.3.1.1 Identifiers

The structure of identifiers is described in the scanner specification. The production below is needed to introduce the terminal *IDENTIFIER* into the syntax.

```
Identifier : op_IdentifierNil()
           | op_IDENTIFIER( IDENTIFIER )
           ;
```

2.3.1.2 Literals

SVENJA provides three kinds of literals: integer literals, boolean literals, and the `null` literal. There are no literals for floats, characters, and strings as these primitive types are not supported in SVENJA (see section 2.3.2).

```
Literal : op_LiteralNil()
        | op_IntegerLiteral( INTLITERAL )
        | op_BooleanLiteral( BOOLLITERAL )
        | op_NullLiteral()
        ;
```

2.3.2 Types, Values, Variables

SVENJA provides reference types and a restricted set of primitive types, namely `int` and `boolean`.

```
Type          : op_TypeNil()
              | op_PrimitiveType( PrimitiveType )
              | op_ReferenceType( ReferenceType )
              ;

PrimitiveType : op_PrimitiveTypeNil()
              | op_NumericType( NumericType )
              | op_boolean()
              ;

NumericType   : op_NumericTypeNil()
              | op_IntegralType( IntegralType )
              ;

IntegralType  : op_IntegralTypeNil()
              | op_int()
              ;
```

A reference type is a class or interface type. In contrast to Java, SVENJA does not support array types. This simplifies the data model of the language.

```
ReferenceType        : op_ReferenceTypeNil()
                     | op_ClassOrInterfaceType( ClassOrInterfaceType )
                     ;

ClassOrInterfaceType : op_ClassOrInterfaceTypeNil()
                     | op_Name( Name )
                     ;

ClassType            : op_ClassTypeNil()
                     | op_ClassOrInterfaceType1( ClassOrInterfaceType )
                     ;

InterfaceType        : op_InterfaceTypeNil()
                     | op_ClassOrInterfaceType2( ClassOrInterfaceType )
                     ;
```

To ease the mapping of concrete syntax into abstract syntax, the grammar of abstract SVENJA syntax is very similar to the grammar of concrete Java syntax. Thus, we have to use the complicated productions above to avoid the ambiguities pointed out in chapter 19.1.1 of [GJS96].

2.3.3 Names

Java uses two kinds of names: simple names and qualified names. Qualified names are used in four cases: (1) to denote package names, (2) to address names in other packages, (3) to address fields, and (4) to call static methods. Cases (1) and (2) cannot occur in SVENJA as packages are not provided (see section 2.3.4). Cases (3) and (4) are avoided in SVENJA by changing the syntax of field access (see sections 2.3.8.9 and 2.3.8.10) and method invocation (see section 2.3.8.11). Thus, SVENJA provides only simple names.

```
Name       : op_NameNil()
           | op_SimpleName( SimpleName )
           ;

SimpleName : op_SimpleNameNil()
           | op_Identifier( Identifier )
           ;
```

Again, the above productions reflect certain grammar problems described in chapter 19.1.1 of [GJS96].

2.3.4 Packages

In its first version, JIVE does not support modular verification. Thus, SVENJA does not provide packages. The design of a module concept which supports modular verification by semantical encapsulation is one of our current research topics (see [MPH97a] for an introduction). As packages are not supported, a non-terminal *CompilationUnit* (cf. [GJS96], chap. 7.4) is dispensable. Thus, phylum *TypeDeclarations* is the root of the grammar[1].

[1] Technically, *CompilationUnit* can be used to define a transformation which introduces the predefined types of each SVENJA program (see section 2.3.5.1).

```
TypeDeclarations : op_TypeDeclarationsNil()
                 | op_TypeDeclarationsPair(TypeDeclaration TypeDeclarations)
                 ;

TypeDeclaration  : op_TypeDeclarationNil()
                 | op_ClassDeclaration( ClassDeclaration )
                 | op_InterfaceDeclaration( InterfaceDeclaration )
                 ;
```

The Synthesizer Generator enforces list productions to be right-recursive. As a naming convention, we denote the operator of the tuple production of a list *List* by *op_ListPair*.

Transforming SVENJA Files into Java Files In Java, packages are important to determine whether a field or method of a class is accessible from another class or not. Java provides four access modes: public, protected, private, and default access. Private class members are accessible only from inside the class they are declared in. In addition to that, default access provides access for all classes of the same package. Protected access allows default access plus access from all subclasses of the class[2]. Public members can be accessed from all other classes of a program. In particular, there is no access mode that allows access only from inside the class and its subclasses (like the protected mode in C++).

Verification heavily relies on data encapsulation. Therefore, SVENJA has to enforce encapsulation of all fields of a class. I.e., fields should be accessible from inside the class and its subclasses only. Unfortunately, this mode is not provided by Java. There are two possibilities to circumvent the absence of packages:

1. All classes of a SVENJA program are considered to be part of one package. Thus, the public, protected, and default access of Java are equivalent. In particular, fields are either private (and thus not accessible from subclasses) or completely public. This solution makes verification very complicated.

2. A separate package is assumed for each class of a SVENJA program, where each package imports all other packages of that program. This solution allows to transform each SVENJA program into a semantically equivalent Java program.

To ease verification, we chose the latter way for SVENJA. So protected access provides the desired access mode for fields.

2.3.5 Classes

In this section, we present the syntax of class declarations and declarations of class members (fields, methods, constructors, and static initializers).

2.3.5.1 Modifiers

To receive a LALR(1) grammar, modifiers have to be covered in a separate production (cf. [GJS96], chap. 19.1.2). Java has ten different access modifiers (`public`, `protected`, `private`, `static`, `abstract`, `final`, `native`, `synchronized`, `transient`, and `volatile`). SVENJA provides five of them: `protected` for fields and methods, `public`, `static`, and `native` for

[2] I.e., packages provide a kind of friend mechanism for class members with protected or default access mode.

methods, and `abstract` to denote abstract classes and methods. `private` is not supported to keep the number of rules in the programming logic small. Furthermore, this simplifies the context conditions for inheritance. `final` is used in Java to state that a class must not have any subtypes. This is an important aspect for modular verification and will thus be added together with a module concept. As threads are not supported in SVENJA, `synchronized` and `volatile` can be omitted. `transient` declares fields not to be part of the persistent state of an object. This is only important when objects are saved to persistent storage. Therefore, it is not supported in SVENJA.

Predefined Reference Types SVENJA assumes the existence of three predefined reference types: the classes `Object` and `Operator` and the interface `Interface`. `Object` is used as root of the subtyping hierarchy. `Operator` provides methods for unary and binary operations (see section 2.3.9.3). `Interface` is used as a default if a class does not implement any other interface (see section 2.3.5.2). The definitions of these types are given in appendix B.1.

As SVENJA does not provide packages or `import`-clauses, all reference types of a program have to be present, in particular the three predefined types. To perform e.g. type-checking, the signatures of all fields and methods of the predefined types have to be accessible. But the method implementations of `Object` and `Operator` cannot be expressed in SVENJA syntax as some methods of `Object` in Java are `native` and the methods of `Operator` require Java code to evaluate unary and binary expressions (see section 2.3.9.3).

For those cases, Java provides the modifier `native` to denote that the body of a method is implemented in another language. In SVENJA, `native` indicates that the body of the method will be replaced by a Java method when the SVENJA program is transformed into Java. `native` may only be used for predefined methods.

```
Modifiers : op_ModifiersNil()
          | op_ModifiersPair( Modifier Modifiers )
          ;

Modifier  : op_ModifierNil()
          | op_public()
          | op_protected()
          | op_static()
          | op_abstract()
          | op_native()
          ;
```

2.3.5.2 Class Declaration

In SVENJA, we use a simplified version of class declarations. As SVENJA does not support packages, only the modifier `abstract` is permitted for classes. Recall that a SVENJA program can be regarded as a set of Java packages each containing exactly one class. This requires each class to be public. To improve readability, the modifier `public` is omitted in SVENJA programs. It is inserted when the program is transformed into a Java program. Furthermore, we enforce each class declaration to have an extends clause (`extends Object` can be used as default). To simplify context conditions, each class has to implement exactly one interface (similar to BALI, where each class implements at most one interface, cf. [NvO98]). If a class C needs to implement more than one interface (say, $I_1 \ldots I_n$), a new interface I extending

$I_1 \ldots I_n$ has to be introduced and C implements I. We assume an empty default interface `Interface` for classes which do not implement any other interface.

```
ClassDeclaration : op_ClassDeclarationNil()
           | op_ClassDecl( Modifiers Identifier Super Interfaces ClassBody )
           ;

Super      : op_SuperNil()
           | op_ClassType( ClassType )
           ;

Interfaces : op_InterfacesNil()
           | op_InterfaceType( InterfaceType )
           ;

ClassBody  : op_ClassBodyNil()
           | op_ClassBodyDeclarations( ClassBodyDeclarations )
           ;

ClassBodyDeclarations : op_ClassBodyDeclarationsNil()
           | op_ClBodyDeclPair( ClassBodyDeclaration ClassBodyDeclarations )
           ;

ClassBodyDeclaration  : op_ClassBodyDeclarationNil()
           | op_ClassMemberDeclaration( ClassMemberDeclaration )
           ;

ClassMemberDeclaration : op_ClassMemberDeclarationNil()
           | op_FieldDeclaration( FieldDeclaration )
           | op_MethodDeclaration( MethodDeclaration )
           ;
```

2.3.5.3 Field Declarations

SVENJA field declarations differ from Java field declarations in four aspects:

1. In SVENJA, each field declaration declares exactly one field. Multiple field declarations have to be split into separate declarations.
2. SVENJA does not provide arrays.
3. Static fields are not supported to keep the number of logical rules small.
4. Fields in SVENJA have to be protected (cf. section 2.3.4). Final, transient, and volatile fields are not supported (see section 2.3.5.1).
5. Variable initializers are not included in SVENJA as arrays and static fields are not supported. Instance variables have to be initialized by usual methods (see below).

```
FieldDeclaration     : op_FieldDeclarationNil()
                     | op_FieldDecl( Modifier Type VariableDeclarator )
                     ;

VariableDeclarator   : op_VariableDeclaratorNil()
                     | op_VariableDeclaratorId( VariableDeclaratorId )
                     ;

VariableDeclaratorId : op_VariableDeclaratorIdNil()
                     | op_Identifier2( Identifier )
                     ;
```

2.3.5.4 Method Declarations

Declarations of methods in SVENJA are slightly different from Java: SVENJA does not support arrays and exception handling which makes some productions simpler. As explained above, only the access modifiers `abstract`, `public`, `protected`, `static`, and `native` are allowed. Static Methods are required to create objects of a class as SVENJA does not provide explicit constructors (see section 2.3.5.6).

SVENJA requires each method to return a result. Methods that do not produce any results have to return a dummy value. Thus, the `void` type can be omitted. As in BALI, each method body in SVENJA ends with a return statement (see section 2.3.8.8). The result expression is denoted explicitly in the syntax of method bodies.

```
MethodDeclaration   : op_MethodDeclarationNil()
                    | op_MethodDecl( MethodHeader MethodBody )
                    ;

MethodHeader        : op_MethodHeaderNil()
                    | op_MethodHead( Modifiers Type MethodDeclarator )
                    ;

MethodDeclarator    : op_MethodDeclaratorNil()
                    | op_MethodSig( Identifier FormalParameterList )
                    ;

FormPars            : op_ForParsNil()
                    | op_FormParsPair( FormalParameter, ForPars )
                    ;

FormalParameter     : op_FormalParameterNil()
                    | op_FormalParameter( Type VariableDeclaratorId )
                    ;

MethodBody          : op_MethodBodyNil()
                    | op_EmptyBlock()
                    | op_Block( Block Expression )
                    ;
```

The above production for *MethodHeader* differs from the syntax given in chapter 8.4 of [GJS96]

according to the problems described in chapter 19.1.1 of the Java Language Specification.

2.3.5.5 Static Initializers

Static initializers can be used to initialize static fields of a class. As static fields are not supported by now, static initializers can be omitted.

2.3.5.6 Constructor Declarations

In Java, constructors are used to create and initialize new objects of concrete classes. They are invoked by the `new` expression. To simplify the syntax and logic of SVENJA, it does neither provide constructors nor a `new` expression. Therefore, alternative ways for object creation and initialization have to be used.

Object Creation Instead of constructors, each concrete class `C` has a predefined method `newC` which creates a new `C` object. This method must have the following properties:

1. It has to be native. As SVENJA does not provide a `new` expression, object creation cannot be expressed in SVENJA.
2. It has to be static. Otherwise, it would not be possible to create an initial object of `C`.
3. It cannot perform initialization of fields as it has a fixed body which looks the same for all classes. Thus, initialization has to be done by usual methods (see below).
4. It has to be protected. This is due to the semantics of class invariants (see section 3.2.5) which states that the conjunction of the invariants for all living object of the program has to hold after termination of any public method `m` if it held before execution of `m` (cf. [MPH97b] for more details). In particular, the invariant of the new object would have to hold if the `newC` method was public. As this is in general not true, `newC` must not be declared public. (Object creation from outside the class can be handled by static methods; see below.)

The points above lead to the following definition of the creation method for class `C`: `protected static native C newC();`. Confer appendix B.2 for the specification of `newC` and the transformation into Java.

Object Initialization Initialization of objects can be done in two ways:

1. by protected instance methods. As protected methods are not affected by the semantics of class invariants (see above), it is possible to call protected methods of the uninitialized object. This solution allows to emulate the call of the `super` constructor which is used in Java to initialize inherited fields: One simply has to call the protected initialization method of the superclass.
2. by public static methods. It is necessary to allow creation and initialization of objects of a class `C` from outside `C` and its subclasses. Therefore, a public static method is required. To avoid the problems caused by the semantics of invariants (see above), object creation and initialization have to be performed in one single method. This can be done by calling the `newC` method and initializing the fields of the result as shown in the following example:

```
class C {
  protected FieldType f;
  protected static native C newC();
  public static C createC(FieldType p) {
    C v;
    v = C.newC();
    v.f = p;
    return v;
  }
  ...
}
```

The above solutions allow to create and initialize objects in a way that is as expressive and flexible as in Java. As usage of constructors and the `new` expression is circumvented, SVENJA avoids the treatment of special constructor methods and `new` expressions which vitally simplifies the programming logic.

2.3.6 Interfaces

SVENJA interfaces differ from Java interfaces in two aspects: (1) Constant declarations are not supported in SVENJA. This avoids ambiguities due to multiple inheritance of interfaces. Furthermore, as SVENJA does not provide variable initializers (see section 2.3.5.3), constants could not be initialized. (2) SVENJA does not allow access modifiers for interfaces and abstract method declarations as use of these modifiers is discouraged in Java. Both items are implicitly public and abstract.

```
InterfaceDeclaration : op_InterfaceDeclarationNil()
               | op_InterDecl( Identifier ExtendsInterfaces InterfaceBody )
               ;

ExtendsInterfaces    : op_ExtendsInterfacesNil()
               | op_ExtendsInterfacesPair( InterfaceType ExtendsInterfaces )
               ;

InterfaceBody  : op_InterfaceBodyNil()
               | op_InterfaceMemberDecls( InterMemberDecls )
               ;

InterMemberDecls : op_InterMemberDeclsNil()
               | op_InterMemberDeclsPair(AbstractMethDecl InterMemberDecls )
               ;

AbstractMethDecl : op_AbstractMethDeclNil()
               | op_MethodHeader( MethodHeader )
               ;
```

2.3.7 Arrays

In its current version, SVENJA does not provide arrays for several reasons:

1. Arrays are not a typical feature of object-oriented languages. The only interesting aspect of arrays in the context of verification is the detection of boundary violations. It is obvious how this can be done by certain logical rules for array access.

2. As described above, dealing with arrays requires additional rules in the programming logic.

3. Arrays require a more complex data model of the programming language as additional types, objects, and access functions have to be incorporated.

4. Arrays can be emulated by predefined classes (one would need one predefined class for each primitive types and one for all reference types). These classes would provide methods to create one-dimensional arrays and to set and get the value at a specified index. The transformation of multi-dimensional into one-dimensional arrays is trivial.

Omitting arrays allows to focus on the central aspects of object-orientation.

2.3.8 Blocks and Statements

In this section, we present the syntax of SVENJA statements.

2.3.8.1 Normal and Abrupt Completion of Statements

Java statements either complete normally or abruptly (cf. chapter 14.1 of [GJS96]). Abrupt completion is always initiated by one of the following statements: `break`, `continue`, `return`, and `throw` (including exceptions thrown by the virtual machine). The programming logic gets much simpler if the programming language does not allow abrupt completion. Thus, the above statements are not supported in SVENJA (see section 2.3.8.8 for the treatment of return statements).

Therefore, abrupt completion of statements in SVENJA can only be caused by exceptions thrown by the virtual machine. We discern between memory errors and other exceptions. All situations in which exceptions of the second kind would occur will be detected during verification (cf. [PH97]). Thus, such exceptions will never be thrown in correct SVENJA programs.

Memory errors cannot be detected during verification in our framework as this would require a formalization of the system a program in running on. As SVENJA does not provide any means to catch exceptions (see section 2.3.8.14), exceptions thrown by the virtual machine lead to abnormal program termination. To handle such situations, our programming logic has a refined partial correctness semantics which does not make any statements about programs that abort due to memory errors (cf. [PHM97] for more details). Thus, memory errors do not affect our notion of partial correctness.

2.3.8.2 Blocks

Blocks are used to structure lists of statements. In particular, they are used to define the scope of local variables. Although blocks lead to a more complex binding analysis, SVENJA

provides blocks to stay compatible to Java in a sense that the syntax of SVENJA can be extended towards Java without larger modifications.

```
Block           : op_BlockNil()
                | op_BlockStatements( BlockStatements )
                ;

BlockStatements : op_BlockStatementsNil()
                | op_BlockStatementsPair( BlockStatement BlockStatements )
                ;

BlockStatement  : op_BlockStatementNil()
                | op_LocalVarDeclStmt( LocalVarDeclStmt )
                | op_Statement( Statement )
                ;
```

2.3.8.3 Local Variable Declaration Statements

Local variable declarations in SVENJA have been modified in a similar way like field declarations (see section 2.3.5.3): Only one variable may be declared in one declaration statement and variable initializers are not supported.

In SVENJA, each local variable is assumed to be initialized. Variables of boolean, integer, or reference types are initialized with `false`, `0`, or `null`, respectively. This assumption simplifies the data and state model of SVENJA as it guarantees the absence of references to non-living objects. When SVENJA programs are transformed into Java, this assumption does not cause any problems as Java checks whether a local variable is initialized before its first use. Thus, the assumed initialization of SVENJA is overridden anyway.

```
LocalVarDeclStmt : op_LocalVarDeclStmtNil()
                 | op_LocalVarDecl( LocalVarDecl )
                 ;

LocalVarDecl     : op_LocalVarDeclNil()
                 | op_VariableDeclarator( Type VariableDeclarator )
                 ;
```

2.3.8.4 Statements

SVENJA supports only a small subset of Java statements to reduce the number of logical rules. In particular, all secondary statements (e.g., `for` and `do` statements), statements for abrupt completion (see section 2.3.8.1), and statements for exception handling are omitted.

The syntax of Java statements is quite complex. This is due to the so-called "dangling `else`" problem. As the `else`-branch of `if` statements is optional in Java, an `else`-branch can be bound to several `if` statements in certain situations (cf. section 14.4 of [GJS96] for an example). SVENJA requires each `if` statement to have an `else`-branch which makes statement syntax much easier (`if` statements without an `else`-branch can use the empty statement as default).

SVENJA does not support any expression statements whereas Java provides four kinds of expression statements:

Assignment: SVENJA only supports the simple assignment operator "=" as the other assignment operators are just abbreviations for binary expressions. To keep the rule for assignments simple, multiple assignments are not allowed. If assignments are treated as expressions (like in Java), evaluation of expressions might change the values of local variables and parameters which leads to complex logical rules. Thus, assignments are treated as statements in SVENJA.

Method invocation: As described above, every SVENJA method returns a result (cf. section 2.3.5.4). Therefore it is not necessary to provide an expression statement which allows to drop the result of a method. Furthermore, verification gets easier if only atomic expressions are allowed (see section 2.3.9). Thus, method invocations can not occur as part of compound expressions. As a consequence, SVENJA treats method invocations as statements and requires every method invocation to have the form `v = exp.meth(...);`.

Object creation: As described in section 2.3.5.6, object creation is performed by invocation of a predefined method. Thus, the class instance creation expression is dispensable.

In/decrement expressions: These statements are only syntactical abbreviations and can thus be omitted.

Besides the assignment and method invocation statements, we introduced two further statements for field read and write operations. These operations cannot be treated as method invocations because field access is statically bound in Java. As field access requires other logical rules than assignments, separate statements have to be used.

Casts allow to narrow the static type of an expression. As compound expressions are not supported in SVENJA, casts always have the form `v = (T)exp`. Therefore they are treated as statements. Furthermore, the logical rules for assignment and casts are very similar if both features are treated as statements. Assignments can even be regarded as a special form of cast statements (cf. [PHM97]). The above considerations are reflected by the following productions:

```
Statement : op_StatementNil()
          | op_IfThenElseStatement( IfThenElseStatement )
          | op_WhileStatement( WhileStatement )
          | op_Block2( Block )
          | op_EmptyStatement()
          | op_FieldRead( FieldReadStatement )
          | op_FieldWrite( FieldWriteStatement )
          | op_MethodInvocationStmt( MethodInvocationStmt )
          | op_AssignStatement( AssignStatement )
          | op_CastStatement( CastStatement )
          ;
```

2.3.8.5 Conditional Statements

Java provides two kinds of conditional statements: `if` statements and `switch` statements. As described above, SVENJA requires each `if` statement to have an `else`-branch to avoid the dangling-`else`-problem. This leads to a simple syntax.

```
IfThenElseStatement : op_IfThenElseStatementNil()
                    | op_IfThenElse( Expression Statement Statement)
                    ;
```

`switch` statements are not provided in SVENJA as they can be transformed into semantically equivalent nested `if` statements.

2.3.8.6 Iteration Statements

The syntax of the `while` statement is identical to Java.

```
WhileStatement : op_WhileStatementNil()
               | op_While( Expression Statement )
               ;
```

`do` statements and `for` statements can be transformed into semantically equivalent `while` statements. Therefore, these statements are not provided in SVENJA.

2.3.8.7 Statements for Abrupt Completion

To support abrupt completion of statements (see section 2.3.8.1), Java provides labeled statements, `break` statements, and `continue` statements. They are not supported by SVENJA. The `return` statement is discussed in the next section.

2.3.8.8 The `return` Statement

To avoid the problem of abrupt statement completion, `return` statements are not permitted inside method bodies. To allow methods to return a result, we follow the idea of BALI (cf. [NvO98]). In BALI, each method has exactly one return statement at the end of the method body. This is modeled in the SVENJA syntax by adding a result expression to the method body (see section 2.3.5.4). Thus, an explicit `return` statement is dispensable.

2.3.8.9 Field Read Statements

In Java, a field is either denoted by a primary expression and an identifier or by `super` and an identifier. As field access is statically bound, the latter syntax is an abbreviation for casting `this` to one of its superclasses and accessing the field of the superclass (cf. chapter 15.10 of [GJS96]). Therefore, this syntax is not supported in SVENJA.

```
FieldReadStatement : op_FieldReadStatementNil()
                   | op_FieldRead( LeftHandSide Primary Identifier )
                   ;
```

2.3.8.10 Field Write Statements

The arguments of the above paragraph carry over to writing field access.

```
FieldWriteStatement : op_FieldWriteStatementNil()
                    | op_FieldWrite( Primary Identifier Expression )
                    ;
```

2.3.8.11 Method Invocation Statements

As described in section 2.3.8.4, method invocation statements consist of a method invocation expression and a left-hand-side expression the result is assigned to.

```
MethodInvocationStmt : op_MethodInvocationStmtNil()
                     | op_MethodInvocation( LeftHandSide MethodInvocation )
                     ;

MethodInvocation : op_MethodInvocationNil()
                 | op_StaticInvocation( ClassType Identifier ArgumentList )
                 | op_NormalInvocation( Primary   Identifier ArgumentList )
                 | op_SuperInvocation (           Identifier ArgumentList )
                 ;

ArgumentList     : op_ArgumentListNil()
                 | op_ArgumentListPair( Expression ArgumentList )
                 ;
```

2.3.8.12 Cast Statements

In Java, cast expressions are used for three purposes: (1) to convert a value of a numeric type to a similar value of another numeric type, (2) to confirm, at compile time, that a type of an expression is `boolean`, and (3) to check at run time, that a reference value refers to an object whose class is compatible with a specified reference type (cf. [GJS96], chap. 15.15). The first case is not needed in SVENJA as SVENJA provides only one numeric primitive type and no arrays. Case (2) has not to be supported by SVENJA as the compile time type of expressions can be determined without using casts.

Casts of the third kind are truly needed in SVENJA to narrow the static type of an expression. Consider the following example: We assume a class `List` with method `List insert(...)` and a subclass `SortedList` of `List` which overrides method `insert`. Java (and SVENJA) type rules require the result type of the overridden and the overriding method to be identical (cf. chapters 8.4.2 and 8.4.6 of [GJS96]). Thus, the result type of `insert` of class `SortedList` has to be `List` although the method might only return `SortedList` objects. To overcome this problem, the result of method `insert` has to be casted to `SortedList`.

The above example shows that casts of the third kind (see above) are indispensable in SVENJA. As the SVENJA syntax does not suffer from the ambiguity problems described in chapter 19.1.5 of [GJS96] (because parenthesized expressions are not provided in SVENJA, see section 2.3.9.1), we can simplify the production for cast statements. (See section 2.3.8.4 for the reason why casts are treated as statements.)

```
CastStatement : op_CastStatementNil()
              | op_Cast( LeftHandSide ClassOrInterfaceType Expression )
              ;
```

2.3.8.13 Assignment Statements

For the reasons pointed out in section 2.3.8.4, assignments are treated as statements in SVENJA. As field access is handled by separate statements (see sections 2.3.8.9 and 2.3.8.10), assignments are only used for local variables and parameters. Thus, the left-hand-side of

assignment statements (as well as of field read, method invocation, and cast statements) is a local variable or a parameter.

```
AssignStatement : op_AssignStatementNil()
                | op_Assign( LeftHandSide Expression )
                ;

LeftHandSide    : op_LeftHandSideNil()
                | op_LeftHandVar( Name )
                ;
```

2.3.8.14 Statements for Exception Handling

As SVENJA does not support exception handling, the `throw` statement and the `try` statement are omitted.

2.3.8.15 The `synchronized` Statement

Threads are not provided in SVENJA. Thus, the `synchronized` statement is not supported.

2.3.9 Expressions

SVENJA provides only a very restricted set of expressions. This has three major reasons: (1) SVENJA provides only atomic (or primary) expressions. This eases verification as it simplifies design and application of the logical rules. This restriction may be inconvenient for programmers, but as Java guarantees the evaluation order of expressions (cf. [GJS96], chap. 15.6), every Java expression can automatically be split into atomic expressions (cf. [PH97]). (2) Some Java expressions are treated as statements in SVENJA (e.g., assignment and method invocation; see section 2.3.8.4). (3) Unary and binary operations are mapped to method invocations (see section 2.3.9.3).

2.3.9.1 Primary Expressions

Besides array creation expressions, Java provides seven kinds of primary expressions: literals, the `this` expression, parenthesized expressions, class instance creation, field access, method invocation, and array access. SVENJA supports only two of them: literals and the `this` expression. class instance creation, field access and method invocation are handled by statements (see section 2.3.8.4). Array access is omitted as arrays are not provided in SVENJA. The absence of non-atomic expressions makes parenthesized expressions dispensable.

In Java, *Names* or not primary expressions. This is due to ambiguities between casts to array types and parenthesized expressions. Both cases cannot occur in SVENJA. Thus, SVENJA can use a simplified grammar where *Names* are primary expressions.

```
Primary : op_PrimaryNil()
        | op_Literal( Literal )
        | op_this()
        | op_LocalVar( Name )
        ;
```

As the SVENJA expression syntax does not have to cope with operator precedence and parse ambiguities, it contains only primary expressions. Thus, an expression is simply a primary expression.

```
Expression : op_ExpressionNil()
           | op_Primary( Primary )
           ;
```

2.3.9.2 Expressions Treated as Statements

In SVENJA, class instance creation, field access, method invocation, casts, and assignment is handled by statements, not by expressions (see section 2.3.8.4). Thus, the corresponding expressions are omitted.

2.3.9.3 Unary and Binary Operators

In this section, we describe the handling of the following operators: unary minus/plus, negation, multiplicative operators, additive operators, shift operators, relational operators (except `instanceof`, see below), equality operators, bitwise and logical operators, and conditional and/or operators. All these operators are handled in SVENJA via method invocations. I.e., we assume a class `Operator` which provides static methods that perform the above operations on their parameters. E.g., the expression `v = a + b` would be written in SVENJA as `v = Operator.plus(a, b)`. See appendix B.1 for the definition of class `Operator`.

This technique has two major advantages: (1) The expression syntax is very small. (2) A specification can be assigned to every operator by specifying its corresponding method. This eases verification and allows a simple treatment of errors occurring during expression evaluation (e.g., arithmetic overflow). See [PH97] for more details.

The `instanceof` operator of Java cannot be treated as described above because its second argument has to be a reference type which is an element of the syntax, not a value that could be passed to a method. Many aspects of the behavior of `instanceof` can be modeled by defining a public method for every class that returns a constant representing the class name. Thus, the type of an object can be determined at run time. Except to check whether an object's type is a subtype of a given type, `instanceof` can be replaced by comparing these constants[3]. Consequently, SVENJA does not provide an `instanceof` expression or statement.

2.3.9.4 Remaining Expressions

Array creation and access expressions are not supported as SVENJA does not provide arrays. *Infix and postfix in/decrement expressions* are abbreviations for binary expressions and can be omitted. The *conditional operator* `? :` can be seen as an abbreviation for an `if` statement. Thus, it is dispensable. *Constant expressions* are expressions that can be evaluated at compile time. They are only used in `switch` statements and thus not needed in SVENJA.

This completes the specification of SVENJA. An example program can be found in chapter 4.

[3]This technique does not work for interfaces as interfaces have no concrete methods. In later versions of SVENJA, a public static field could be used instead.

Chapter 3

Interface Specification Language

In this chapter, we present the interface specification language of JIVE, which is called ANJA (**An**notation language for **Ja**va). The next section describes the design decisions made for ANJA. The syntax of ANJA and the embedding of the interface specifications in SVENJA programs is specified in section 3.2.

3.1 Design Decisions

The design of ANJA was influenced by two major aspects: (1) The objective of verifying SVENJA programs and (2) the usage of theorem provers in JIVE. In the following section, we discuss the design decisions made for ANJA in this context.

3.1.1 Specification Technique

We want to use well-known specification techniques wherever it is possible. Thus, we basically adopt the two-tiered Larch technique which was described in chapter 1. To keep things simple, we only use the most important specification primitives, namely pre- and postconditions and class invariants (see section 3.1.2).

Verification requires interface specifications to be declarative. To ease the usage of a Hoare-style programming logic for verification, the mapping of specification primitives to Hoare triples must be clear. Confer [PH97] and [MPH97b] for a detailed description of our specification technique, in particular for the formal meaning of the specification primitives.

3.1.2 Specification Primitives

In JIVE, interface specifications can be formalized by two specification primitives: (1) Pre- and postconditions for methods and (2) class invariants.

Usually, method specifications capture different aspects of method behavior such as functional behavior, side-effects, and sharing properties. To enable structured method specifications, ANJA allows to denote multiple pre-post-pairs for each method. To improve readability, common requirements of all preconditions of one method can be specified in a so-called *requires-clause*. Furthermore, the requires-clause is necessary to formalize the meaning of class invariants. Confer [PH97] for more details on this topic.

In summary, the interface specification of a class (or interface) consists of the class invariant and a specification for each method of the class. Method specifications consists of a requires-clause and a set of pre-post-pairs.

Comparison to Larch/C++ As our interface specification language is similar to the Larch interface specification languages, it might be interesting to compare ANJA to Larch/C++, the Larch language for C++ (cf. [Lea96]).

All of the specification primitives of ANJA are provided by Larch/C++ as well. In addition to that, Larch/C++ contains three features not supported by ANJA (cf. [LB97]):

1. Larch/C++ uses so-called modifies-clauses to express which objects might have changed their values under execution of a method. Similar to that, a trashes-clause lists all objects that might be destroyed by the method. Modifies- and trashes-clauses have two particular drawbacks: (1) They do not fit naturally into the Hoare-logic. (2) They can hardly express sharing properties. Thus, in our framework, invariance properties are expressed by relating the pre- and poststate of a method (cf. [MPH97b] for details).

2. In Larch/C++, history constraints can be used to specify properties that have to hold for any ordered pair of visible states in program execution. E.g., a history constraint could be used to specify that the value of the `age` field of a class `Person` may not be decreased during program execution. Such properties cannot be expressed in ANJA. From a theoretical point of view, history constraints behave very similar to class invariants. Thus, our techniques could be applied to history constraints as well. But we want to focus on the central aspects in the first version of JIVE.

3. Larch/C++ allows to specify redundant method specifications which are used for additional checking and documentation. In ANJA, each method specification has the same meaning: It is transformed into a Hoare triple that has to be proved. Thus, there is no need to discern between different kinds of pre-post-pairs.

3.1.3 Using TPC Formulas vs. Providing Own Syntax

Recall from chapter 1 that the universal (i.e., program-independent) parts of a specification are formalized in the language of the theorem prover component TPC. In JIVE, we use either Isabelle or PVS as TPC. Both systems provide a multi-sorted higher-order specification language which is strongly typed[1] (cf. [Pau94] and [OSR93]).

Basically, there are two alternatives how formulas in ANJA can be denoted:

1. We can adopt the syntax of the TPC language. This allows to pass formulas to the TPC without any syntactical transformations.

2. A special ANJA syntax can be used. This requires each formula or declaration to be transformed into the syntax of the TPC.

As the user has to switch between the PVC and the TPC quite often, both solutions require to use a formula syntax that looks very similar to the TPC syntax. Thus, solution 2 does not

[1] From now on, we follow the convention of Larch and denote types of the programming language by *types* whereas the types of the TPC language are called *sorts*.

allow to exchange the TPC system in an easier way as the syntax would have to be adapted anyway.

The most important difference between the alternatives above is the treatment of sort declarations. Choosing alternative 2 requires to provide sort declarations in the PVC which can be transformed into declarations of the TPC. Alternative 1 allows to move all sort declarations to the TPC. In this case, the PVC would use the sort names without any knowledge of the sorts themselves. In the following, we discuss the pros and cons of these solutions:

1. Declaring sorts in the PVC allows full control of all sort information. In particular, the PVC controls whether first-order or higher-order formulas are used. This decision cannot be made in alternative 1.

2. Declaring sorts in the PVC enables full sort-checking inside the PVC. On the other hand, sort-checking can get quite complex if subsorting is used. Alternative 1 requires each formula to be passed to the TPC to be parsed and sort-checked.

3. Alternative 1 is much easier to implement as neither sort declarations nor sort-checking has to be provided. This argument is in particular true if elaborated higher-order type systems and subsorting shall be supported.

In the long term, full control of the sort system is certainly desirable, but in JIVE, we favorite alternative 1 as it is easier to implement. As a consequence, our prototype might have to deal with higher-order formulas. This does not cause any problems to the programming logic.

3.1.4 Choosing a Theorem Prover

Basically, we are planning to support PVS and Isabelle as theorem prover components. Both systems are very sophisticated and have proved their power in several research projects and practical applications. In the context of our work, both systems have particular advantages: The type system of BALI was proved correct in Isabelle (cf. [NvO98]). We can build on the formalization of BALI for further work, e.g. for a soundness proof of the SVENJA programming logic. The PVS system is used by the PAMELA verification tool (cf. [But97]) for a purpose that is very similar to ours. In particular, PVS was enhanced by functions to prove and type-check single formulas. This feature will be needed in JIVE as well (see section 3.1.3).

As a starting point, JIVE uses the PVS system because the Lopex group has more experience with this system. This decision allows to adopt parts of the PAMELA system to implement the communication between the PVC and TPC. Isabelle will be supported as soon as we have time to implement the interface.

3.2 Specification

This section specifies the syntax of ANJA. The syntax is based on the syntax of the PVS Specification Language (cf. [OSR93] and extension to PVS 2). In its first version, ANJA supports only a very small subset of PVS expressions as it will provide a specification syntax of its own later (see section 3.1.3). We discuss the particular restrictions along with the definition of the supported features.

3.2.0.1 Lexical Structure

All keywords of the PVS language are reserved in ANJA as well. Otherwise, ambiguities would occur when formulas are passed to PVS. To ease lexical analysis, ANJA requires all PVS keywords to be written in upper-case characters. Furthermore, `req`, `pre`, `post`, `inv`, and `decl` are keywords of ANJA (see appendix A for a list of all keywords and special symbols). ANJA identifiers are identical to SVENJA identifiers. To denote the symbols of unary and binary operators, we assume a terminal symbol *PVSOPERATOR*.

To keep the syntax specification modular, we introduce separate productions for all ANJA features. Each phylum which depends on the syntax of the PVS language is prefixed by *PVS*.

```
PVSIdOp       : op_PVSIdOpNil()
              | op_PVSIdentifier( PVSIdentifier )
              | op_PVSOpSym( PVSOpSym )
              ;

PVSIdentifier : op_PVSIdentifierNil()
              | op_PVSIDENTIFIER( IDENTIFIER )
              ;

PVSOpSym      : op_PVSOpSymNil()
              | op_PVSUnaryOp( PVSOPERATOR )
              | op_PVSBinaryOp( PVSOPERATOR )
              ;
```

3.2.1 Names

PVS names are identifiers or operator symbols. To denote names of different theories, a qualified name can be used. The formal parameters of a theory can be instantiated by providing a list of actual parameters in square brackets.

```
PVSName     : op_PVSNameNil()
            | op_PVSSimpleName( PVSIdOp PVSModuleParList )
            | op_PVSQualifiedName( PVSIdOp PVSModuleParList PVSIdOp )
            ;

PVSModuleParList : op_PVSModuleParListNil()
            | op_PVSModuleParListEmpty()
            | op_PVSModuleActuals( PVSModuleActuals )
            ;

PVSModuleActuals : op_PVSModuleActualsNil()
            | op_PVSModuleActualsPair( PVSModuleActual PVSModuleActuals )
            ;

PVSModuleActual  : op_PVSModuleActualNil()
            | op_PVSExpr( PVSExpr )
            | op_PVSSortExpr( PVSSortExpr )
            ;
```

3.2.2 Sorts

PVS specifications are strongly typed. Thus, ANJA must provide means to denote PVS sorts, e.g. in the context of quantified formulas. To keep the syntax simple, only sort names are used. As PVS allows to introduce names for any type expression (e.g., subsorts, function sorts, or enumeration sorts), this does not restrict expressiveness.

```
PVSSortExpr : op_PVSSortExprNil()
            | op_PVSSortName( PVSName )
            ;
```

3.2.3 Sort-Checking

As pointed out in section 3.1.3, sort-checking cannot be performed within the PVC. Thus, all formulas are passed to PVS to be parsed and sort-checked. This is done as follows:

Assume a formula P that occurs as invariant or part of a method specification. P may contain program variables, free logical variables, and names of PVS sorts, functions, variables, etc. As ANJA has no access to PVS theories, we cannot decide within the PVC whether a name denotes a free variable or an item of a PVS theory. Therefore, we pass the formula P to PVS along with all sort information that can be provided. I.e., we transform P into a *closed form* `FORALL` $(v_i : sort_i)$: `P` = `TRUE`. The v_i are those variables for which the PVC has sort information: all program variables (including `this`) of the current scope, all logical variables declared in the decl-clause of the current type (see section 3.2.5), `result` which denotes the result of the current method, and `$` which denotes the current object environment.

The closed form is then sort-checked by PVS. If no errors occur, P is correctly sorted and of sort *bool*. Otherwise, P may be ill-sorted or some free logical variables have not been declared. Errors can be passed back to the PVC and displayed.

3.2.4 Formulas

ANJA supports only a restricted set of PVS expressions. In particular, it does not provide tuple expressions, record expressions, set expressions, coercion expressions, let and where expressions, and cases expressions. Almost every omitted feature can be replace by an application of an appropriate function. ANJA supports numbers, names, binary expressions, unary expressions, if-then-else expressions, function applications, and binding expressions. Furthermore, `result` and `$` are used to denote the result of a method and the current object environment, respectively. `result` may only occur in postconditions of methods (see section 3.2.6). Whereas `result` can be treated as PVS name, `$` has to be a seperate expression as it is not a valid PVS identifier.

```
PVSExpr : op_PVSExprNil()
        | op_PVSNumber( INTLITERAL )
        | op_PVSName( PVSName )
        | op_PVSBinaryExpr( PVSExpr PVSBinaryOp PVSExpr )
        | op_PVSUnaryExpr( PVSUnaryOp PVSExpr )
        | op_PVSIfExpr( PVSExpr PVSExpr PVSExpr )
        | op_PVSApplicationExpr( PVSName PVSArguments )
        | op_PVSBindExpr( PVSBindOp PVSFormals PVSExpr )
        | op_PVSDollar()
        ;
```

Unary and Binary Operators To provide a flexible notation, ANJA supports most of the PVS operator symbols. A list of all operator symbols can be found in appendix A.

```
PVSBinaryOp : op_PVSBinaryOpNil()
            | op_PVSBinary( PVSOPERATOR )
            ;

PVSUnaryOp  : op_PVSUnaryOpNil()
            | op_PVSUnary( PVSOPERATOR )
            ;
```

If Expressions ANJA `if` expressions consist of a boolean expression and two expressions of the same sort, one in the `then` branch and one in the `else` branch. `elsif` is not supported as it is only an abbreviation for nested `if` expressions.

Applications In contrast to PVS, ANJA does not directly support higher-order applications, i.e., the function is determined by a name, not by an expression. Thus, function applications consist of a name and an argument list.

```
PVSArguments : op_PVSArgumentsNil()
             | op_PVSArgumentsPair( PVSExpr PVSArguments )
             ;
```

Binding Expressions Binding expressions are used to denote quantified formulas. ANJA provides universal and existential quantification. In contrast to PVS, lambda abstraction is not supported. As higher-order is not directly supported, ANJA does not allow to quantify over operator symbols.

```
PVSBindOp  : op_PVSBindOpNil()
           | op_PVSExists()
           | op_PVSForAll()
           ;

PVSFormals : op_PVSFormalsNil()
           | op_PVSFormalsPair( PVSFormal PVSFormals )
           ;

PVSFormal  : op_PVSFormalNil()
           | op_PVSFormal( PVSIdentifier PVSSortExpr )
           ;
```

3.2.5 Class and Interface Specifications

Properties of data representations can be specified by invariants. To describe properties that have to hold for a set of classes, invariants may also be used in abstract classes and interfaces. In the following, we use the term *type* to denote classes, abstract classes, and interfaces.

According to [PH97], an invariant is a formula with only one free variable ranging over the type the invariant belongs to[2]. Thus, an invariant is given by an Identifier and a formula.

[2]This can be enforced by omitting the declarations of free logical variables when the invariant is sort-checked in PVS (see section 3.2.3).

All free logical variables (i.e., not program variables) occurring in specifications have to be declared to provide their sorts. To enforce that logical variables with the same name are used for the same purpose within a type specification, variables are declared on type level. Therefore, each type declaration contains a (possibly empty) list of PVS variable declarations.

The productions for class and interface declarations given in sections 2.3.5.2 and 2.3.6, respectively, have to be modified to incorporate the invariant and variable declarations.

```
ANJAInvariant    : op_ANJAInvariantNil
                 | op_ANJAInvDecl( PVSIdentifier PVSExpr )
                 ;

ClassDeclaration : op_ClassDeclarationNil()
                 | op_ClassDecl( Modifiers Identifier Super Interfaces
                                    ANJAInvariant PVSFormals ClassBody )
                 ;

InterfaceDeclaration : op_InterfaceDeclarationNil()
                     | op_InterDecl( Identifier ExtendsInterfaces
                           ANJAInvariant PVSFormals InterfaceBody )
                     ;
```

3.2.6 Method Specifications

Method specifications may occur in all kinds of method declarations, in particular in abstract or native methods. As described in section 3.1.2, method specifications consist of a (possibly empty) set of pre-post pairs and a requires-clause. Requires-clauses, pre-, and postconditions are simply PVS formulas. To provide interface specifications, the syntax of method declarations (see section 2.3.5.4) has to be modified as follows.

```
ANJAPrePostList : op_ANJAPrePostListNil()
                | op_ANJAPrePostListPair( ANJAPrePost ANJAPrePostList )
                ;

ANJAPrePost     : op_ANJAPrePostNil()
                | op_ANJAPrePostPair( PVSExpr PVSExpr )
                ;

MethodHeader    : op_MethodHeaderNil()
                | op_MethodHead( Modifiers Type MethodDeclarator
                                          PVSExpr ANJAPrePostList )
                ;
```

This completes the specification of ANJA. An example for annotated SVENJA programs can be found in the next chapter.

Chapter 4

Example

In this chapter, we present a small example program to demonstrate the usage of SVENJA and ANJA. The program consists of an Interface `IList`, an abstract class `AList`, and a concrete class `List`. `IList` describes the interface of a list data structure. `AList` implements `IList` but does not provide implementations for the abstract methods declared in `IList`. Based on these methods, `AList` defines a method `length` which returns the length of the list. Class `List` inherits from `AList` and implements `IList` by providing implementations for the abstract methods. For brevity, we omitted methods `isempty` and `rest` in all types as they do not reveal any interesting aspects.

Abstract Data Type The types and methods are specified in terms of an abstract data type *ADTList*. *aIList*, *aAList*, and *aList* are abstraction functions that map objects of `IList`, `AList`, and `List` to values of *ADTList*. *aB* and *aI* are used to map boolean and int values of SVENJA to the corresponding PVS values. The PVS definition of *ADTList* looks as follows:

```
ADTList : DATATYPE
BEGIN
      empty : isempty?
      app(l: ADTList, i: int) : isapp?
END ADTList

first :   [ADTList -> int]
rest :    [ADTList -> ADTList]
length :  [ADTList -> nat]
```

In the next paragraphs, we discuss the implementations and specifications of the types mentioned above. This is not the place to describe the Lopex specification technique in detail. Therefore, we assume the reader to be familiar with [PH97] or [MPH97b].

Interface IList To keep things simple, we assume a function *wf* that expresses well-formedness for list representations (e.g., the list has to be acyclic). This keeps the invariant of the discussed types very simple: *wf* has to hold for each list representation. The variable declaration clause of `IList` introduces the variables L and I which are used to denote values of types *ADTList* and *int*, respectively.

Each abstract method is specified by a requires clause and one pre-post-pair. E.g., the requires clause of `first` states that the abstraction of the implicit parameter `this` in the current state (denoted by `$`) is called L and must not be empty. Whenever this condition is met, `first` will return a value the abstraction of which equals the first element of L.

```
interface IList
  inv  X: wf(X, $);
  decl L: ADTList, I: int
{
  IList append(int s)
    req  TRUE;
    pre  aIList(this, $) = L AND aI(s) = I;
    post aIList(result, $) = app(L, I);      ;
  int first()
    req  aIList(this, $) = L AND NOT isempty(L);
    pre  TRUE;
    post aI(result) = first(L);              ;
}
```

Abstract Class AList The abstract methods of `IList` are inherited in `AList` and have thus not to be repeated. The implementation of method `length` shows two interesting aspects of SVENJA code: (1) SVENJA is capable to deal with the subtyping rules of Java which enforce the result types of corresponding methods in the super- and in the subtype to be identical (in other languages, method `rest` would have result type `AList` instead of `IList`). This is achieved by using casts. (2) The absence of complex expressions leads to longer and less readable code. This price has to be payed to ease verification.

```
abstract class AList extends Object implements IList
  inv X: wf(X, $);
  decl L: ADTList
{
  public int length()
    req  TRUE;
    pre  aAList(this, $) = L;
    post aI(result) = length(L); {
      int res;
      boolean b;      b = this.isempty();
      if (b) res = 0;
      else {
        IList ir;     ir = this.rest();
        AList r;      r = (AList)ir;
        int l;        l = r.length();
        res = Operator.plus(1, l);
      }
      return res;
  }
}
```

Class List `List` implements `IList` via singly linked lists. The length of the list is stored explicitly in field `len`. The empty list is represented by a `List` object with length 0. For brevity, we omitted method `first` as it simply returns the content of field `elem`.

The implementation of `List` demonstrates how object creation is handled in SVENJA. Static method `empty` calls the implicit static method `newList`. After that, the `len` field is initialized to 0 which makes the new object represent an empty list.

Using requires-clauses and preconditions makes defensive programming and exception handling dispensable in many cases. E.g., in method `first`, the requires-clause asserts that the list is non-empty. Thus, additional tests and exceptions are not necessary.

Method `length` overrides the inherited method of `AList`. The specification of `length` shows how several pre-post-pairs can be used to express different aspects of method behavior: The first pair specifies the functional behavior (`length` will return the length of the list) whereas the second pair states that execution of `length` does not produce any side-effects. The is expressed by stating that the object environments of the method's pre- and poststate are equal.

```
class List extends AList implements IList
  inv X: wf(X, $);
  decl L: ADTList, I: int, E: ObjEnv
{
  protected int elem;
  protected List next;
  protected int len;

  public static List empty()
    req  TRUE;
    pre  TRUE;
    post aList(result, $) = empty; {
      List l;          l = List.newList();
      l.len = 0;       return l;
  }

  public IList append(int s)
    req  TRUE;
    pre  aList(this, $) = L AND aI(s) = I;
    post aList(result, $) = app(L, I); {
      List res;        res = List.newList();
      res.elem = s;    res.next = this;
      int ll;          ll = this.len;
      int sum;         sum = Operator.plus(1, ll);
      res.len = sum;   return res;
  }

  public int length()
    req  TRUE;
    pre  aList(this, $) = L;
    post aI(result) = length(L);
    pre  $ = E;
    post $ = E; {
      int l;           l = this.len;
      return l;
  }
}
```

Summary The above example demonstrates that SVENJA can be used to develop usual object-oriented programs. SVENJA programs can easily be transformed into Java and tested by using the Java compiler. Inconvenience is caused by the restricted expression syntax. In the next chapter, we describe an idea to overcome this drawback.

As ANJA provides almost full PVS expression syntax, interface specifications can be formalized in a very flexible and convenient way. Sets of pre-post-pairs allow to structure method specifications.

Chapter 5

Conclusion and Further Work

In this report, we defined the programming language and the interface specification language of the JIVE system. We introduced SVENJA, a subset of Java, as programming language. SVENJA provides all typical object-oriented language features such as a class concept, strong typing, encapsulation, subtyping, dynamic binding, and inheritance.

The interface specification language ANJA is based on the specification language of the PVS system. It provides means for convenient and flexible state-of-the-art interface specifications.

We presented a formalization of the abstract syntax of both languages in Sythesizer Generator notation. An example program demonstrated the usage of SVENJA and ANJA. It has shown that the languages allow to implement and specify realistic object-oriented programs.

In progress of the Lopex project, we will refine, extend, and improve the presented in work in three aspects:

1. SVENJA and ANJA will be extended in several aspects:

   (a) Program development in SVENJA will be more convenient if complex expressions and secondary statements (e.g., `for` and `switch` statements) are supported. As a larger abstract syntax makes verification more complex, JIVE should provide two syntactic levels. The user edits programs on a high level (i.e., in a complex syntax). For verification, the high-level program is transformed into a semantically equivalent program using only the features described in this report. Thus, the language can be convenient for programming and easy to handle for verification.

   (b) ANJA will provide means for specification of sorts. This will allow to perform sort-checking within the program verification component. Furthermore, transformations have to be implemented that map ANJA specifications to PVS or Isabelle theories. In the medium-term, both of these theorem provers will be supported.

2. SVENJA and ANJA will be enhanced by means for compositional programming and specification. A semantic-based module concept will be added to SVENJA that will allow syntacical and semantical encapsulation of types (cf. [MPH97a] for a discussion). ANJA will be enriched by features for module specifications such as module invariants. These extensions will allow to compose verified program components and deduce the correctness of the resulting program from the correctness of the components.

3. SVENJA and ANJA have to be evaluated to find out whether certain language features are missing and to detect further possibilities for simplification. The evaluation requires two actions:

   (a) The transformation from SVENJA into semantically equivalent Java programs has to be implemented to enable compiling and running SVENJA programs.

   (b) Larger programs have to be implemented in SVENJA and specified with ANJA. To do that, we have recently begun to translate parts of the LEDA library (cf. [NU95]) into SVENJA and specify their behavior with ANJA. To evaluate the Lopex specification technique outside the world of basic data structures, it will as well be applied to user interface libraries such as the Java AWT.

The presented programming language and interface specification language build a strong basis for the Lopex prototype JIVE. They allow to implement and specify realistic object-oriented programs. The above improvements will make the languages (and the JIVE system) even more convenient.

Bibliography

[AS96] M. Austern and A. Stepanov. *The Java Algorithm Library*. SiliconGraphics, 1996. Available from `http://reality.sgi.com/austern/java`

[But97] B. Buth. An interface between pamela and pvs. Technical report, Universität Bremen, 1997. Available from `http://www.informatik.uni-bremen.de/~bb/bb.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*. Springer-Verlag, 1993.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, Reading, MA, 1996.

[Ham97] G. Hamilton. *JavaBeans*. Sun Microsystems, 1997. Available from `http://java.sun.com/beans/docs/spec.html`

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, 1969.

[LB97] G. T. Leavens and A. L. Baker. Enhancing the pre- and postcondition technique for more expressive specifications. (submitted), 1997.

[Lea96] G. T. Leavens. An overview of Larch/C++: Behavioral specifications for C++ modules. In Hiam Kilov and William Harvey, editors, *Specification of Behavioral Semantics in Object-Oriented Information Modeling*, chapter 8, pages 121–142. Kluwer Academic Publishers, Boston, 1996.

[MPH97a] P. Müller and A. Poetzsch-Heffter. Developing provably correct programs from object-oriented components. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, 1997. Available from `http://www.cs.iastate.edu/~leavens/FoCBS/FoCBS.html`

[MPH97b] P. Müller and A. Poetzsch-Heffter. Formal specification techniques for object-oriented programs. In M. Jarke, K. Pasedach, and K. Pohl, editors, *Informatik 97: Informatik als Innovationsmotor*, Informatik Aktuell. Springer-Verlag, 1997.

[NU95] Stefan Näher and Christian Uhrig. Leda user manual (version r 3.2). Technical Report MPI-I-95-1-002, Max-Planck-Institut für Informatik, June 1995.

[NvO98] T. Nipkow and D. von Oheimb. Java$_{light}$ is type-safe — definitely. In *Proc. 25th ACM Symp. Principles of Programming Languages*. ACM Press, 1998. To appear.

[OSR93] S. Owre, N. Shankar, and J. M. Rushby. The PVS specification language (beta release). Technical report, Computer Science Laboratory SRI International, April 1993.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. Number 828 in LNCS. Springer-Verlag, 1994.

[PH97] A. Poetzsch-Heffter. *Specification and Verification of Object-Oriented Programs*. PhD thesis, Technische Universität München, 1997. (Habilitationsschrift).

[PHM97] A. Poetzsch-Heffter and P. Müller. Logical foundations for typed object-oriented languages. (submitted), 1997.

[RT89] T. W. Reps and T. Teitelbaum. *The Synthesizer Generator*. Springer-Verlag, 1989.

Appendix A

Keywords

This chapter summarizes the reserved keywords and operator symbols of SVENJA and ANJA. All Java and PVS keywords and operator symbols are reserved in JIVE as well for two reasons: (1) The languages may be extended towards full Java and full PVS without changing the list of keywords. (2) Transforming SVENJA programs into Java and passing ANJA formulas to the PVS system require that Java and PVS keywords are not used as identifiers in SVENJA or ANJA.

A.1 SVENJA Keywords:

SVENJA **Keywords:**

abstract	default	if	private	throw
boolean	do	implements	protected	throws
break	double	import	public	transient
byte	else	instanceof	return	try
case	extends	int	short	void
catch	final	interface	static	volatile
char	finally	long	super	while
class	float	native	switch	
const	for	new	synchronized	
continue	goto	package	this	

SVENJA **Seperators:**

```
(  )  {  }  [  ]  ;  ,  .
```

SVENJA **Operators:**

```
=     >     <     !     ~     ?     :
==    <=    >=    !=    &&    ||    ++    --
+     -     *     /     &     |     ^     %     <<     >>     >>>
+=    -=    *=    /=    &=    |=    ^=    %=    <<=    >>=    >>>=
```

A.2 ANJA Keywords:

ANJA keywords consist of PVS keywords and a few reserved words to denote interface specifications.

ANJA Keywords:

decl	inv	post	pre	req

ANJA Special Symbols:

```
$
```

PVS Keywords:

AND	CONTAINING	FALSE	LEMMA	SUBTYPE_OF
ANDTHEN	CONVERSION	FORALL	LET	TABLE
ARRAY	COROLLARY	FORMULA	LIBRARY	THEN
ASSUMING	DATATYPE	FROM	MEASURE	THEOREM
ASSUMPTION	ELSE	FUNCTION	NONEMPTY_TYPE	THEORY
AXIOM	ELSIF	HAS_TYPE	NOT	TRUE
BEGIN	END	IF	O	TYPE
BUT	ENDASSUMING	IFF	OBLIGATION	TYPE+
BY	ENDCASES	IMPLIES	OF	VAR
CASES	ENDCOND	IMPORTING	OR	WHEN
CHALLENGE	ENDIF	IN	ORELSE	WHERE
CLAIM	ENDTABLE	INDUCTIVE	POSTULATE	WITH
CLOSURE	EXISTS	JUDGEMENT	PROPOSITION	XOR
COND	EXPORTING	LAMBDA	RECURSIVE	
CONJECTURE	FACT	LAW	SUBLEMMA	

PVS Special Symbols:

```
$   @   :=   ->   ;   %   .   :   [||]   |->   ||   <-   ,   |
(  )   (#  #)   (|  |)   (:  :)   [  ]   [#  #]   [|  |]   |[  ]|   {  }
```

PVS Infix Operators:

```
|-      |=      IFF     <=>     IMPLIES =>      WHEN
OR      \/      XOR     ORELSE  AND     &       /\      &&        ANDTHEN
=       /=      ==      <       <=      >       >=
<<      >>      <<=     >>=     <|      |>
#       @@      ##      +       -       ++      ~
*       /       **      //      o       ^       ^^
```

PVS Unary Operators:

```
NOT     ~       -       []      <>
```

Appendix B

Definition of Predefined Types and Methods

This chapter presents the definition of the predefined SVENJA types (see section 2.3.5.1) and describes their transformations into Java types. We omitted the interface specifications of predefined types as they might change during progress of our work. Furthermore, we give a definition for the object creation method which is predefined for every concrete class.

B.1 Predefined Types

`Interface:` The interface `Interface` is used as a default superinterface.

```
interface Interface {}
```

Besides the transformation into packages, `Interface` stays unchanged when the SVENJA program is transformed into Java.

`Object:` Class `Object` is the root of the subtype hierarchy. As SVENJA requires each class to have a superclass, we allow `Object` to be its own superclass. This is not allowed for all other classes. In SVENJA, `Object` provides signatures for some of the methods of Java `Object`. As threads, dynamic class loading, and finalizers are not supported in SVENJA, we omitted the corresponding methods.

```
class Object extends Object implements Interface {
    public    native int     hashCode();
    public    native boolean equals(Object obj);
    protected native Object  clone();
    public    native String  toString();
}
```

When SVENJA programs are transformed into Java, `Object` is simply dropped.

`Operator:` The SVENJA class `Operator` provides static methods to perform unary and binary operations (see section 2.3.9.3). To keep things simple, shift operators and bitwise operators are omitted.

```
class Operator extends Object implements Interface {
    public static native int     times    (int a, int b);
    public static native int     div      (int a, int b);
    public static native int     mod      (int a, int b);
    public static native int     plus     (int a, int b);
    public static native int     minus    (int a, int b);
    public static native boolean less     (int a, int b);
    public static native boolean greater  (int a, int b);
    public static native boolean lesseq   (int a, int b);
    public static native boolean greatereq(int a, int b);
    public static native boolean equal    (int a, int b);
    public static native boolean equal    (boolean a, boolean b);
    public static native boolean equal    (Object  a, Object  b);
    public static native boolean notequal (int a, int b);
    public static native boolean notequal (boolean a, boolean b);
    public static native boolean notequal (Object  a, Object  b);
    public static native boolean condand  (boolean a, boolean b);
    public static native boolean condor   (boolean a, boolean b);

    public static native int minus(int a);
    public static native boolean not(boolean a);
}
```

On transformation of SVENJA programs into Java, `Operator` is replaced by the following class declaration which assigns implementations to the native methods of SVENJA.

```
public class Operator implements Interface {
    public static int     times    (int a, int b)         { return a*b;   }
    public static int     div      (int a, int b)         { return a/b;  }
    public static int     mod      (int a, int b)         { return a%b;  }
    public static int     plus     (int a, int b)         { return a+b;  }
    public static int     minus    (int a, int b)         { return a-b;  }
    public static boolean less     (int a, int b)         { return a<b;  }
    public static boolean greater  (int a, int b)         { return a>b;  }
    public static boolean lesseq   (int a, int b)         { return a<=b; }
    public static boolean greatereq(int a, int b)         { return a>=b; }
    public static boolean equal    (int a, int b)         { return a==b; }
    public static boolean equal    (boolean a, boolean b) { return a==b; }
    public static boolean equal    (Object  a, Object  b) { return a==b; }
    public static boolean notequal (int a, int b)         { return a!=b; }
    public static boolean notequal (boolean a, boolean b) { return a!=b; }
    public static boolean notequal (Object  a, Object  b) { return a!=b; }
    public static boolean condand  (boolean a, boolean b) { return a&&b; }
    public static boolean condor   (boolean a, boolean b) { return a||b; }

    public static int     minus    (int a)                { return -a;   }
    public static boolean not      (boolean a)            { return !a;   }
}
```

B.2 Predefined Methods

As described in section 2.3.5.6, each concrete SVENJA class `C` contains a predefined method `newC` which returns a new object of type `C`. This method is defined as follows ($E{+}{+}C$ denotes object environment E after allocating a new object of type `C`; *new(E, C)* yields a new object of type `C` in environment E):

```
protected static native C newC()
  req  TRUE;
  pre  $ = E;
  post $ = E++C AND result = new(E, C);
  ;
```

The translation into Java calls the default constructor and returns its result:

```
protected static C newC() { return new C(); }
```