MAGISTERARBEIT

# Implementation of the VooDo Kern Programming Language

Ausgeführt am

Institut für Computersprachen
der Technischen Universität Wien

unter der Anleitung von

Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Franz Puntigam

durch

Joachim Bickel, Bakk.techn.

Gottschalkgasse 1/23
A-1110 Wien

Wien, August 2005

## Zusammenfassung

VooDo ist eine experimentelle objekt-orientierte Programmiersprache, welche eine Vielzahl an Features aufweist, hauptsächlich das Typsystem betreffend. Beispiele sind Mehrfachvererbung, Module mit Unterstützung für Vererbung, generische Typen und sogenannte *verteilte Optionen*, welche das Hauptmerkmal neu in VooDo im Vergleich zu anderen objekt-orientierten Sprachen darstellen. Optionen ermöglichen es einem Compiler, Synchronisations-Bedingungen statisch zu überprüfen. Es wird angenommen, dass die Implementierung des VooDo Typsystems recht komplex ist und einige potenzielle Probleme enthält.

In dieser Magisterarbeit werden diese Probleme behandelt, indem ein Compiler-Frontend für eine Teilmenge der Sprache, genannt VooDo Kern, implementiert wird, als Proof of Concept und Basis für weitergehende Arbeit. Die Implementierung zeigt, dass die eigentlichen Schwierigkeiten nicht in der Typüberprüfung, sondern in den grundlegenderen Bereichen, insbesondere Namensauflösung und Zyklensuche, zu finden sind.

## Abstract

VooDo is an experimental object-oriented programming language containing a large variety of features, mostly concerning the type system. This includes multiple inheritance, modules that support inheritance, generic types, and so-called *distributed options*, which are the key feature new to VooDo compared to other object-oriented languages. Options allow a compiler to statically check synchronization constraints. It is to be assumed that the implementation of the VooDo type system is quite complex and includes a number of potential problems.

In this thesis we address these problems by implementing a compiler frontend for a subset of the language, called VooDo Kern, as a proof of concept, and to provide a basis for further work. The implementation shows that the actual difficulties are not in type checking itself, but rather in more basic parts, especially name resolution and cycle detection.

# Contents

Contents 3

I. Overview 6

1. Introduction 7

2. VooDo Kern 9
- 2.1. Overview 9
- 2.2. Example 9
- 2.3. Basic Elements 11
  - 2.3.1. Assemblies 11
  - 2.3.2. Units 11
  - 2.3.3. Members 14
  - 2.3.4. Statements 18
- 2.4. Element Sequence 20
- 2.5. Subtyping and Inheritance 21
- 2.6. Synchronization 24
  - 2.6.1. Distributed Options 25
  - 2.6.2. Synchronization Constraints 25
  - 2.6.3. Active Type Modifiers 26
  - 2.6.4. Tokens and Subtyping 26
  - 2.6.5. Conditional Statements 27
- 2.7. Changes 28
  - 2.7.1. Syntax 28
  - 2.7.2. Distributed Options 28

3. Methods and Tools 29
- 3.1. Objective 29
- 3.2. General Approach 29
- 3.3. Implementation Tools 30
  - 3.3.1. Programming Language C# 30
  - 3.3.2. Parser Generator Coco/R 31

4. Examples 32
4.1. Covariant Routine Parameters . . . 32
4.1.1. Problem . . . 32
4.1.2. Solution in VooDo . . . 33
4.1.3. Issues and Implementation . . . 34
4.2. Cycle Detection . . . 35
4.2.1. Approaches . . . 35
4.2.2. Implementation . . . 35
4.3. Lifted Types . . . 36
4.3.1. Approaches . . . 36
4.3.2. Implementation . . . 36

II. Implementation 37

5. Overview 38

6. Syntax Analysis Phase 39
6.1. Syntax Analysis . . . 39
6.1.1. Syntax Grammar . . . 39
6.1.2. Attributed Grammar . . . 40
6.2. Data Structures . . . 40
6.2.1. Units . . . 41
6.2.2. Members . . . 42
6.2.3. Formal Parameter Lists . . . 44
6.2.4. Routine Bodies and Statements . . . 45
6.2.5. Expressions . . . 45

7. Name Resolution 49
7.1. Name Resolution Basics . . . 49
7.1.1. Language Elements . . . 49
7.1.2. Context . . . 52
7.1.3. Member Accesses . . . 53
7.1.4. Creators . . . 56
7.2. Name Resolution/Cycle Detection . . . 56
7.2.1. Considered Approaches . . . 57
7.2.2. Ordered Elements . . . 58
7.2.3. Allowed Forward References . . . 60
7.2.4. Name Resolution in Units . . . 60
7.2.5. Name Resolution in Expressions . . . 61
7.2.6. Resolution of Member Accesses . . . 63
7.2.7. Recursive Routines . . . 63

**8. Inheritance** 65
8.1. Name Resolution . . . 65
8.2. Generic Units . . . 66
8.3. Embedded Private Members . . . 67
8.4. Substituted Members . . . 67
8.5. Multiple Inheritance . . . 68

**9. Semantic Checks** 69
9.1. Basic Type Checking . . . 69
9.1.1. Interfaces . . . 69
9.1.2. Order of Type Comparisons . . . 69
9.1.3. Named Types . . . 71
9.1.4. Modified Types . . . 71
9.2. Substituted Members . . . 72
9.3. Distributed Options . . . 72
9.3.1. Types and Tokens . . . 72
9.3.2. State Information for Objects . . . 73
9.3.3. Routine Bodies . . . 76
9.3.4. Elements . . . 77

**III. Summary** 78

**10. Statistical Information** 79

**11. Future Work** 84
11.1. Changes to the Language . . . 84
11.2. Implementation of Remaining Features . . . 84
11.2.1. Basic Data Types . . . 84
11.2.2. Distributed Options . . . 84
11.2.3. Miscellaneous Features . . . 85
11.3. Code Generator . . . 85

**12. Conclusion** 86

**IV. Appendix** 87

**A. List of Figures, Tables, and Listings** 88

**B. Explanation of Figures** 90

**Bibliography** 93

# Part I.

# Overview

# 1. Introduction

*VooDo* (*Vienna Object-Oriented Language with Distributed Options*) is an experimental object-oriented programming language developed at the Vienna University of Technology, Institute of Computer Languages, Compilers and Languages Group. Conceptually, it is similar to well-established object-oriented languages like Java [9], C# [21], Eiffel [10], or C++ [22], and it includes many features known from other languages, especially concerning the type system. Examples (see also [16]) include multiple inheritance, modules distinct from classes that support inheritance, multi-methods [4], and generic types [19, 2]. What is special about VooDo is the combination of all those features in one language, as well as a new concept called *distributed options*. Based on process types [14], options are an approach to address problems occuring in concurrent programs. They allow the compiler to statically check synchronization constraints and determine the need for semaphore operations.

As VooDo itself is very complex, a scaled-down version called *VooDo Kern* [17] was used as basis for this thesis, which contains only the most important concepts. In the rest of this work, both VooDo and VooDo Kern refer to this smaller version. If the full version is meant, this is denoted specifically.

An implementation of the VooDo language is assumed to be complex and face a number of potential problems, mainly because of the complexity of the type system and related to distributed options.

Since a complete implementation of the whole language is beyond the scope of this work, a compiler frontend for VooDo Kern has been implemented in order to validate or disprove this assumption and to provide a basis for further work. The programming language used in the implementation is C#, with Coco/R as parser generator. The frontend is designed using object-oriented techniques and an emphasis is placed on flexibility.

Part II gives an overview of our implementation. We explain the most important aspects, used approaches, as well as solutions to encountered problems. We also include some statistical information to give an idea about the size and level of complexity of the implementation.

While the compiler frontend is complex, the most problematic part turned out not to be the implementation of the semantic checks, but the implementation of more basic functionality, especially name resolution and cycle detection.

The remainder of Part I contains a description of the VooDo Kern language in Chapter 2, and an overview of the methods and tools used in the implementation in Chapter 3. Furthermore, Chapter 4 offers examples of VooDo Kern features, and a general idea of the implementation of the required concepts.

Part II explains the implementation itself. We provide an outline in Chapter 5. The remaining chapters describe the syntax analysis and internal data structures of the compiler frontend (Chapter 6), how name resolution and cycle detection are implemented and the encountered problems (Chapter 7), the approach used for handling inheritance (Chapter 8), and the semantic checks performed by the compiler frontend (Chapter 9).

In Part III, we summarize the information found during the course of this work. We present statistical information regarding the implementation in Chapter 10. Chapter 11 assesses the additional work required to implement a complete VooDo Kern compiler, and a conclusion can be found in Chapter 12.

The appendices (Part IV) contain a list of all figures, tables and listings (Appendix A), as well as an explanation of the class diagram and example figures (Appendix B).

# 2. VooDo Kern

## 2.1. Overview

VooDo is an experimental object-oriented programming language. It is designed to include a large number of features. Most of those are known from other languages, but new concepts not previously implemented are also included.

Two versions of VooDo have been proposed, a full version including every feature, and a scaled-down version called VooDo Kern, which exists "to make a clear separation between essential language conceptions and non-essential goodies" [17]. This work is based on the latter.

In this chapter, we give an overview of the language. This is meant to provide an introduction and describe the concepts relevant for the implementation of a compiler frontend. Also included are some proposed changes that will, at least in part, be incorporated into the language.

## 2.2. Example

Listing 1 shows a VooDo Kern implementation of a generic stack.

We assume a basic familiarity with other object-oriented languages, for example Java [8], in the following explanations.

The stack is implemented using a singly-linked list. The generic class `Node` represents a node in the list. `Node` has one type parameter, `Element`, which is used to specify the type of the elements in the stack. Each node contains the element itself as well as a reference to the next node, the latter of which may be `Null`, indicated by the `?` before the type. `new` is used to create new instances of the class (see Section 2.3.3).

The `Stack` class uses a `Node` object as its internal representation in the `nodes` variable. Again, a creator named `new` is used to instantiate the class. The methods `push` and `pop` implement the standard stack operations.

An interesting element is the token `unique` (see Section 2.6), which is used for synchronizing access to the stack, as seen in lines 24 and 30. The `when`-statements require `unique`

```
class Node (type Element) is

    elem: Element;
    next: ?Node(Element);

    creator new (e: Element; n: ?Node(Element)): Node(Element) do
        elem = e;
        next = n;
    end creator;

end class;

public class Stack (type Element) is

    token unique = 1;

    var nodes: ?Node(Element);

    public creator new: Stack(Element) do
        nodes = Null;
    end creator;

    public method push (p: Element) do
        when unique do
            nodes = Node(Element).new(p, nodes);
        end when;
    end method;

    public method pop: ?Element do
        when unique, first: Node(Element) = nodes do
            nodes = first.next;
            return first.elem;
        end when;
        return Null;
    end method;

end class;
```

Listing 1: VooDo Kern: Generic Stack

to be at least one, decrement it on entry, and increment it again when the associated block is left. If the token is not at least one, execution is blocked.

Also of note is the condition `first: Nodes(Element) = nodes` in the **when**-statement in Line 30, a dynamic type check. This is needed because the type of `nodes` is a lifted type (see Section 2.3.3), that is, it can be `Null`. The condition checks at runtime if `nodes` contains a reference to an object of the specified type, `Node(Element)`. If yes, the condition is true and the reference is assigned to the named value `first`, which has the specified type. Otherwise the condition is false and the **when**-block is skipped.

## 2.3. Basic Elements

### 2.3.1. Assemblies

The basic unit of compilation in VooDo is called an *assembly*. As multiple assemblies become relevant only once code generation is performed, it is assumed from here on that only a single assembly is used.

### 2.3.2. Units

Like other current object-oriented languages, VooDo uses classes and other class-like concepts as basic building blocks of programs. These so-called *units* are subdivided into *layers*, which are defined implicitly through the contained members:

**Instance Layer:** Instance members are specific to and accessed using an instance of a unit.

**Shared Layer:** The shared layer exists to support a very flexible form of generic types. A unit has a shared layer if and only if it takes parameters (which are not restricted to type parameters like for example in Java [2]). Shared members are the same for all instances of the unit with the same actual parameters. The parameters themselves also belong to the shared layer.

**Static Layer:** Static members are the same for all instances of a unit.

Listing 2 illustrates this by means of a generic class.

```
1 class A (type T) is
2     static var x: String = ...;
3     shared var a: T do ... end;
4     b: T do ... end;
5     ...
6 end class;
```

Listing 2: VooDo Kern: Layers

Class `A` contains a static, a shared, and an instance member. `x` is a static variable, and can be accessed through the class itself (`A.x`), through a parameterized version (`A(String).x`), and through an instance (`v: A(String); t: T; ... t = v.x;`). `a` is a shared variable function. It can use the parameter of the class, and can be accessed through either a parameterized version or an instance. `b` is an instance value function. It can only be accessed through an instance of the class, and can, of course, also use the parameter. For a description of the various kinds of members, see Section 2.3.3.

There are several kinds of units:

**Class:** Classes are similar to those in other languages. A class always has an instance layer and may have a static layer (if static members exist) and a shared layer (if the class takes parameters). Each class must contain at least one creator. Classes are defined using the `class` keyword.

**Class Brand:** Class brands are like classes except that they allow member declarations (as opposed to definitions) and need not necessarily contain a creator. They are comparable to abstract classes in Java. The keyword sequence for defining class brands is `class type`.

**Module:** Modules do not contain an instance layer. Members can only be declared as `static` if there is a shared layer. Those not declared as static are assumed to belong to the shared layer if there is one (that is, the module takes parameters), and to the static layer otherwise. The `module` keyword is used to define modules.

**Module Brand:** Similar to class brands, module brands resemble modules, but allow member declarations. The keyword sequence used is `module type`.

**Brand:** Brands are comparable to interfaces in Java. They do not define different layers and allow only member declarations, no definitions. The layer of the members in a brand depends on what type of unit uses it as a super-unit (see below and Section 2.5 for details). Brands are defined using the keyword `type`.

Listing 3 contains an example for a brand.

```
1  public type StackInterface (type Element) is
2      method push (p: Element);
3      method pop: ?Element;
4  end type;
5
6  public class Stack (type Element): embed StackInterface(Element) is
7      ...
```

Listing 3: VooDo Kern: Brand

Line 6 of this example could be used to replace Line 13 in Listing 1, illustrating the use of a brand with a class. In this case, the declarations in the brand define instance members.

An example for a module can be found in Listing 4.

The module `MyStack` is very similar to the `Stack` class. However, no instances of the module can be created, and all members without layer qualification, that is, in this case, all members, are implicitly declared as belonging to the shared layer. Therefore, `MyStack` is at the same time a type and an instance of the type.

```
public module MyStack (type Element): embed StackInterface(Element) is

    token unique = 1;

    var nodes: ?Node(Element) = Null;

    public method push (p: Element) do
        when unique do
            nodes = Node(Element).new(p, nodes);
        end when;
    end method;

    public method pop: ?Element do
        when unique, first: Node(Element) = nodes do
            nodes = first.next;
            return first.elem;
        end when;
        return Null;
    end method;

end module;
```

Listing 4: VooDo Kern: Module

Note that `MyStack` embeds, that is, inherits from `StackInterface` (see Listing 3). In this case, the members of `StackInterface` are used as shared members.

A summarizing list of the different kinds of units can be found in Table 1.

| Type of Unit | Declarations | Definitions | Creators | Layers |
|---|---|---|---|---|
| Class | - | + | ! | static, instance |
| Class (generic) | - | + | ! | static, shared, instance |
| Class Brand | + | + | + | static, instance |
| Class Brand (generic) | + | + | + | static, shared, instance |
| Module | - | + | - | static |
| Module (generic) | - | + | - | static, shared |
| Module Brand | + | + | - | static |
| Module Brand (generic) | + | + | - | static, shared |
| Brand | + | - | - | none |

Table 1.: Units Summary

Sometimes, it is useful to access only the shared and static layers of a class or class brand, or only the static layer of a class, class brand, module, or module brand. This is possible using the keywords `static`, `shared`, or `module` as suffix of a unit in the same form as a member access. `shared` references the shared and static layer of a unit together (also

called the *module* of a class) and is only available for generic classes and class brands. `static` references just the static layer of a unit, and is available for all units except brands. The semantics of `module` depend upon whether the unit takes parameters or not. If it does, `module` is equivalent to `shared`, otherwise, it is equivalent to `static`.

Classes and modules can be used as objects and as types. Table 2 provides an overview of examples of possible units. Additionally, it shows the use of the units themselves, modules and static layers thereof, and instances of classes, as objects. The "Type" column contains the type of the object in the "Object" column. Note that brands, class brands, module brands, and static and shared layers thereof cannot be used as objects but only as types.

| Unit | Object | Type |
|---|---|---|
| class A | x: A | A |
| | A | A.static |
| | A.static | A.static |
| class B (type T) | x: B(Integer) | B(Integer) |
| | B(Integer) | B(Integer).shared |
| | B(Integer).shared | B(Integer).shared |
| | B(Integer).static | B.static |
| | B.static | B.static |
| module C | C | C = C.static |
| | C.static | C = C.static |
| module D (type T) | D(Integer) | D(Integer) = D(Integer).shared |
| | D(Integer).shared | D(Integer) = D(Integer).shared |
| | D(Integer).static | D.static |
| | D.static | D.static |

Table 2.: Units as Objects

Units can have public or default visibility. Public units are visible everywhere, while units with default visibility are only accessible from within the assembly containing the unit.

### 2.3.3. Members

Unit members in VooDo can be either declared or defined, depending on the unit (see Section 2.3.2). Members can have one of the following visibility states:

**Private:** Members that have private visibility are only visible in the unit in which they are declared or defined and in instances of that unit.

**Public:** Public members are visible wherever the unit or instance is visible.

**Default:** Members not declared as either `public` or `private` have default visibility, that is, they are visible in the assembly containing the unit in which they are declared or defined and in all sub-units of that unit.

There are basically four kinds of unit members, described below.

### Data Members

Data members are *named values* and *variables*. Named values are similar to final variables in Java, as they can only be assigned to once. Variables basically function like those in other programming languages, however, there is an important difference, as write accesses to variables are only possible from inside the object containing the variable.

All data members must be initialized before the first read access. They can either be initialized directly in their definition, in a creator, or in a method. In the last case, it must be guaranteed that no read access is possible before that method is executed. In addition, for named values, it has to be impossible for the initializing method to be executed more than once. These restrictions can be asserted using tokens (see below).

### Routines

There are four kinds of routines:

**Value Functions:** Value functions provide a purely functional subset of the language, in that they are not allowed to have any side effects and may only access named values and other value functions, no variables, variable functions, or methods. Thus, the result of each call of a value function depends only upon the actual parameter values and named values. Value functions, like named values, are declared without a special keyword. For an example see Listing 2, Line 4.

**Variable Functions:** Like value functions, variable functions must not have any side effects. They are, however, allowed to access variables. Thus, the result of a variable function call can change from invocation to invocation, even if the actual parameters stay the same. Variable functions are declared/defined using the keyword `var`, same as variables. An example can be found in Listing 2, Line 3.

**Methods:** Methods are the most general routines. Side effects are allowed to occur, access to all visible members is possible, and methods need not have a return value. The keyword `method` is used to declare or define methods. Listings 1, 3, and 4 contain examples of methods.

**Creators:** In order to create an instance of a class, a creator is used. Creators are basically static or shared methods (depending on whether the class is generic or not) that have access to instance members and in which `this` is available. The return type must be the class itself (optionally modified using type modifiers, see below). In creator declarations/definitions, the `creator` keyword is used. The `Node` and `Stack` classes in Listing 1 each contain a creator.

A special case exists for data members and parameterless functions. For units that allow both declarations and definitions (i.e., class brands and module brands) a parameterless value function declaration is indistinguishable from a named value definition without initial value, and the same is true for variable function declarations and variables. However, this ambiguity only exists on the instance layer, because shared and static data members must be initialized in their definition. The decision between these kinds of members is postponed to an inheriting unit, except if there is an assignment to the member. In this case it can only be a data member.

### Named Types

Named types provide a kind of type constants. Basically, a named type is a different name for an existing type. However, it is also possible to explicitly specify a supertype for a named type, which is used wherever the actual type value is not available. Defined named types must be initialized directly in the definition, while named type declarations can omit the initialization and only specify a supertype. Named types can take parameters.

Listing 5 shows the use of directly initialized named types (with and without parameters) as well as a named type declaration specifying a supertype.

```
1 type StackType (type Element) = Stack(Element);
2 type IntegerStack = Stack(Integer);
3
4 type SomeStack: StackInterface;
```

Listing 5: VooDo Kern: Named Types

A more comprehensive example for the use of named types can be found in Section 4.1.

In addition to unit members, named types can also be formal parameters of generic units or routines. Such type parameters must not take parameters themselves, and can of course only specify a supertype, no initialization. Listings 1, 3, and 4 illustrate the use of named type parameters for generic units.

A special pre-defined named type with one parameter is `?`, which defines a so-called *maybe-type* or *lifted type*. All instances of the type specified as parameter (the *just-type* of the maybe-type) as well as the special module `Null` are valid instances of such a type.

?Null and Null are identical. In contrast to other parameterized named types, lifted types are covariant, that is, a type ?U is a subtype of ?T if U is a subtype of T. This does not pose any problems with the subtyping relation. It is, however, necessary, because type modifiers (see below) are allowed on maybe-types, and apply to the underlying just-type. If lifted types were not covariant, this would break the semantics of type modifiers and thus distributed options (see below and Section 2.6). Examples for lifted types can be found in listings 1, 3, and 4.

### Tokens

Tokens are used for synchronization. They are declared/defined using the keyword token and can contain an initialization. Basically, a token somewhat resembles a variable (in earlier versions of the language, tokens were called *type variables* [16]) in that a token has a numerical value that can change over time. However, tokens cannot be assigned to directly. Token values are of type Natural, which contains all non-negative integers and the special value Infinity.

Dynamically, tokens correspond to semaphores [5], but are implemented in a safer way and provide additional static features, as they can be used to encode state information in the type of an object, by specifying a lower bound for the token values in an object.

Listing 6 shows a class Fork (for the dining philosophers problem [6, 7]) that contains two token definitions without initialization (which indicates an initial value of zero).

```
1 public class Fork is
2     public token up;
3     public token down;
4     ...
5 end class;
```

Listing 6: VooDo Kern: Tokens

If the type of an object is simply Fork, no further information about the values of the tokens is known. To specify a fork that is down on the table, the type Fork[down] is used, that is, an object of type Fork for which the token down has at least the value one. In this context, we also say that the object has one down token. The corresponding type for a fork that is up in the hand of a philosopher is Fork[up]. The token specifications in squared brackets are called *type modifiers*, and the type Fork in this case is known as the *base type* (not to be confused with a supertype, for which this expression is sometimes used).

To specify token values other than one, it is possible to append a numeric value to the token in a type modifier. Additionally, more than one token can be specified. For example, Fork[down:3, up:2] would specify an object of type Fork that has three down tokens and two up tokens. A compiler can check if such a type can actually exist. The

value that can be specified is relative to the token's default value, which is normally zero (not the token's initial value, which is only used inside the unit). Listing 7 shows an example where this is not the case.

```
class A is
    token t;
    type T = A[t];

    method m(p: T) do
        ...
        l: T[t:-1] = p;
        ...
    end method;
    ...
end class;
```

Listing 7: VooDo Kern: Negative Token Value

Here, the base type `T` in Line 7 already contains a type modifier. Thus, the use of a negative token value can make sense. Note that the specified value of a token (in sum over all applied type modifiers) can never be lower than zero.

For further discussions of the use of type modifiers and tokens in general see Section 2.6.

### 2.3.4. Statements

The kinds of statements for use in a routine body in VooDo are described below.

#### Local Data Definitions

A local data definition defines a named value or variable that is only visible within the routine body containing the statement. Syntactically, they resemble data members without visibility or layer specifications. As with data members, local named values and variables must be assigned to before their first use.

#### Assignments

Assignments assign a value to a named value or variable. They are written in the form `x = v`, where `x` is the data object assigned to and `v` the assigned value.

### Invocations

Routine invocations are used for routines that do not return a value. Other routines can only be used in an assignment or as an actual parameter, not as an invocation statement ignoring the result.

### Block Statements

Block statements are conditional statements (`if` and `when`) and `while`-loops. Conditional statements execute at most one of a number of blocks of statements. Each but at most one of the blocks is associated to a list of conditions. The first block for which all conditions are true is selected.

Listing 8 shows an `if`-statement.

```
1 if a.isAvailable() do
2     ...
3 else if b = c, x: T = v do
4     ...
5 else do
6     ...
7 end if;
```

Listing 8: VooDo Kern: `if`-Statement

Normally, conditions are expressions that return a boolean value. However, a special case exists in the form of a dynamic type check, as seen in Line 3 of the example, as well as Listing 1, Line 30. Such a condition includes a local named value definition for the associated block. Here, the value `v` must be of type `T` in order for the condition to be true. In that case, the named value `x` is defined and initialized with `v`.

For a discussion of the use of synchronization constraints in conditional statements and the difference between `if` and `when` see Section 2.6.5.

A `while`-loop consists just of a list of conditions and a block of statements. The statements in the block are repeated as long as the condition is true.

### Return Statements

Return statements consist of the keyword `return` optionally followed by a value to be returned from the routine call. Return statements must contain a return value if and only if the routine has a return type.

## 2.4. Element Sequence

VooDo is designed to prevent cyclic definitions. In some cases this is necessary for a compiler to function (i.e., to prevent an endless recursion) while in others it provides additional safety. Listing 9 illustrates this in the simple case of a named value that is initialized with itself.

```
1 class A is
2     x: Integer = f;
3     f: Integer do
4         return this.x;
5     end;
6     ...
7 end class;
```

Listing 9: VooDo Kern: Invalid Cycle

In contrast to other programming languages, this is not allowed in VooDo.

To allow detection of invalid cycles, VooDo defines a logical sequence on all units, layers of units, and members. Note that this sequence does not have to and in some cases cannot correspond to the order of the elements in the source code. The logical order is defined using references between elements. In the standard case, the referenced element comes before the one referencing it. For the example in Listing 9, the named value `x` references the value function `f`, thus, `f` must come before `x` in the sequence. Similarly, `f` references `x`, so `x` must precede `f`. As both assertions cannot be true at the same time, the example contains an invalid cycle.

The sequence of a number of elements relative to each other is assumed without being explicitly specified in the source code:

- Static members of a unit precede the static layer of the same unit, and shared members precede the shared layer.
- The static layer of a unit precedes all shared members of the unit.
- The static and shared layers of a unit precede the instance layer.
- The instance layer of a unit precedes the instance members of the unit.
- All layers of a unit (but not necessarily all members) precede the unit itself.
- Each layer of a super-unit precedes the same layer of each sub-unit and all members of the sub-unit at this layer.

`this` denotes the instance layer of the object containing the member in which it is used, and thus comes before the member itself in the logical sequence.

In some cases, cycles are necessary, for example when declaring the type of a member, or for recursive routines. Therefore, there are a number of situations where a reference between two elements does not define the relative order of the elements. For the following elements, references are allowed from other elements that precede them in the sequence:

- accesses to data members and routines, provided there is a prefix that does not start with `this`, a unit, or a static layer or module thereof (in other words, accesses to instance members of other objects than the containing one);
- routine invocations in routine bodies (that is, recursive routines); note, however, that both invoking and invoked routines must belong to the same layer, and there most not be a data member in the sequence between the two routines;
- types at the same layer directly following `:` (that is, all except those used as actual parameters, as supertypes in unit definitions, or to initialize named types);
- types at the same layer used as actual type parameters in super-unit specifications (`embed` clauses, see Section 2.5; needed for F-bounded genericity [3], which permits the use of a type as parameter of its own supertype);
- `this` as well as units (and static layers and modules thereof) when used as prefixes.

## 2.5. Subtyping and Inheritance

Subtyping in VooDo, like other object-oriented languages, is based on the *principle of substitutability*, that is, an instance of a subtype can be used wherever an instance of a supertype is expected. The most general type is `{}`, the empty type, which is a supertype of any type.Primarily, subtype relationships between types are explicitly specified by unit inheritance, with the following additions:

- The empty type `{}` is a supertype of all types.
- `Null` is equivalent to itself and to `?Null`.
- A maybe-type `?T` is a supertype of its just-type `T`.
- Lifted types are covariant (see Section 2.3.3).
- A named type is a subtype of its declared supertype, and is equivalent to its type value if available (see Section 9.1).
- Subtype relationships exist between a unit and its embedded units, as well as the static layer or module of a unit and the static layer respectively module of its embedded units.

- A unit is a subtype of its static layer and module (if available).

Additionally, if two types have type modifiers (see Section 2.3.3), the token values of the subtype must imply those of the supertype (see Section 2.6.4).

To simplify the definition of substitutability given below, a subtyping relationship is defined on formal parameter lists. A formal parameter list $s$ is a subtype of a formal parameter list $t$ if

- $s$ and $t$ specify the same number of parameters,
- each parameter (or the corresponding declaration) in $s$ can be used where the parameter at the same position (or the corresponding declaration) in $t$ is expected,
- and if $t$ contains synchronization constraints (see Section 2.6.2), then $s$ also contains synchronization constraints implying those of $t$.

Inheritance is specified using an `embed` clause (for an example refer to Listing 4). Multiple inheritance is supported without additional restrictions. Conceptually, inheritance is defined by copying all members from the inherited unit to the inheriting unit. Private members are renamed so as to prevent conflicts. For non-private members, the following two cases are problematic:

- Two (or more) members have the same name, and none is substitutable for all others (see below).
- Two (or more) members are defined (not just declared) in different embedded units.

The second case is only problematic because it is not possible to decide which definition is to be used, and can thus always be resolved by providing a new definition, that is, overriding the members. For the first case, providing a declaration or definition that can substitute all embedded members of the same name can in some cases solve the problem. However, there are instances where the embedded members are completely incompatible and the respective units cannot be embedded at the same time (for example, if one of the members is a named value and the other a named type).

Brands can only inherit from other brands. Modules and module brands can be sub-units of brands, modules, module brands, and static layers thereof. Classes and class brands can inherit from all kinds of units and static layers and modules thereof.

Members can be overridden by other members if and only if the inherited member is *substitutable* by the overriding member, as defined by the reflexive and transitive closure of the following rules:

- A named value can be used where a named value is expected if the type of the used named value is a subtype of the expected named value's type.
- A named value can be used where a parameter-less value function is expected if the named value's type is a subtype of the value function's result type.
- A parameter-less value function can be used where a named value is expected if the value function's result type is a subtype of the named value's type.
- A value function can be used where another value function is expected if
  - the expected value function's formal parameter list is a subtype of the used value function's formal parameter list,
  - and the used value function's result type is a subtype of the expected value function's result type.
- A variable can be used where a parameterless variable function is expected if the variable's type is a subtype of the variable function's result type.
- A value function or variable function can be used where another variable function is expected if
  - the expected variable function's formal parameter list is a subtype of the used function's formal parameter list,
  - and the used function's result type is a subtype of the expected variable function's result type.
- A function can be used where a method is expected if
  - the method's formal parameter list is a subtype of the function's formal parameter list,
  - and the method has a result type such that the function's result type is a subtype of the method's result type.
- A method can be used where another method is expected if
  - the formal parameter list of the expected method is a subtype of those of the used method,
  - if one of these methods has a result type, then both methods have one, and the used method's result type is a subtype of the expected method's result type.
- A creator can be used where a shared method or a static method (in classes without shared layer) is expected if
  - the method's formal parameter list is a subtype of the creator's formal parameter list,

- the method has a result type,
- the creator's result type (that is, the class containing the creator together with any type modifiers specified in the creator definition) is a subtype of the method's result type.

- A shared method or a static method (in classes without shared layer) can be used where a creator is expected if
  - the creator's formal parameter list is a subtype of the method's formal parameter list,
  - the method has a result type,
  - and the method's result type is a subtype of the creator's result type.
- A creator can be used where another creator is expected if
  - the expected creator's formal parameter list is a subtype of the used creator's formal parameter list,
  - and the used creator's result type is a subtype of the expected creator's result type.
- A named type can be used where another named type is expected if
  - the expected named type's formal parameter list is a subtype of the used named type's formal parameter list,
  - the used type's declared supertype is a subtype of the expected named type's declared supertype,
  - and the used type's type value is a subtype of the expected named type's type value.

Additionally, overriding members can be more visible then overridden members, and can be moved to a lower layer (instance to shared or static, shared to static).

If an inherited generic unit is statically determined (that is, all actual parameters are statically determined), any shared members are inherited as static members.

## 2.6. Synchronization

Synchronization between concurrent threads of execution in VooDo is a focus of the language. It is centered around the concept of tokens (see Section 2.3.3), which are used as *distributed options*. This section describes the synchronization concepts as implemented in this work.

### 2.6.1. Distributed Options

Each token defines an *option* a client of an object can have. Type modifiers specify the currently available options (see Section 2.3.3, especially Listing 6 and the corresponding explanations). For example, if an object is known to be of type `Fork[down]`, we say that one `down` option is available on the object.

Options are a limited resource. When creating an alias to an object, any options required are removed from the original named value or variable. Listing 10 illustrates this, using the `Fork` class defined in Listing 6.

```
1 method m (f: Fork[down]) do
2     ...
3     g: Fork[down] = f;
4     ...
5     h: Fork = g;
6     ...
7 end method;
```

Listing 10: VooDo Kern: Options

In Line 2, the option `down` is available on `f`. However, after Line 3, it is available on `g`, but no longer on `f`, that is, the type of `f` has changed to `Fork`. In contrast, Line 5 does not change any types, as the type of `h` does not specify any required options. Actually, this example is not correct, because `f` is still of type `Fork` at the end of the method, but should be of type `Fork[down]`.

Passing actual parameters is similar to creating aliases through assignments. However, unless specified otherwise (see Section 2.6.3), the options are returned to the object used as actual parameter after the invoked routine returns.

### 2.6.2. Synchronization Constraints

Options become a useful concept only through the use of so-called *synchronization constraints*. These are token values specified by a member. The member can then only be accessed on an object with all required tokens available, that is, the client accessing the member has all required options. Listing 11 provides a more complete version of the `Fork` class.

`Fork` contains two methods specifying synchronization constraints and a creator. The synchronization constraints imply that `take` can only be called on an instance of `Fork[down]`, while for `giveBack` the instance has to be of type `Fork[up]`.

In the example, *active* synchronization constraints, identifiable by `->`, are used. Active synchronization constraints define changing options. After a call to `take`, the object is no longer of type `Fork[down]`, but of type `Fork[up]`, and vice versa for `giveBack`.

```
public class Fork is
    public token up;
    public token down;
    public method take (with down -> up) do ... end method;
    public method giveBack (with up -> down) do ... end method;
    public creator new: Fork[down] do ... end creator;
end class;
```

Listing 11: VooDo Kern: Synchronization Constraints

Only methods and creators may specify active synchronization constraints, all other members, including value and variable functions, are restricted to passive synchronization constraints.

### 2.6.3. Active Type Modifiers

Similar to active synchronization constraints, methods and creators may specify active type modifiers on formal parameters. This is illustrated in Listing 12.

```
method takeFork(f: Fork[down -> up]) do
    f.take;
end method;
```

Listing 12: VooDo Kern: Active Type Modifiers

In this way, it is also possible to consume an option in a method, for example if the object is to be assigned to a member variable that specifies the option, by writing `[token ->]`.

### 2.6.4. Tokens and Subtyping

For subtyping, token values on types have to be taken into account. A type $s$ is a subtype of a type $t$ if, in addition to the base type of $s$ being a subtype of the base type of $t$, the tokens of $s$ imply those of $t$ (see Section 2.5). For synchronization constraints, the equivalent rule is included in the definition of subtyping on formal parameter lists. Note that the formal parameter list of a overriding member must be a supertype, not a subtype, of the formal parameter list of the overridden member because of parameter contravariance.

The tokens of $s$ imply the tokens of $t$ if they can become equal by applying any number of transformations from the following list to the tokens of $s$:

- Tokens to the left of `->` can be removed. For example, `down:2 -> up` implies `down -> up`.

- Tokens to the right of `->` can be added. For example, `down -> up` implies `down -> up:2`.
- Tokens can be removed from the left and right of `->` simultaneously (and they can be removed without restrictions if there is no `->`). For example, `up -> up:2` implies `-> up`, and `up:2` implies `up`.

### 2.6.5. Conditional Statements

Synchronization constraints and type modifiers can be checked statically, eliminating the need for semaphore operations. However, it is also possible and often necessary to dynamically check for token values using conditional statements (`if` and `when`; see Section 2.3.4).

Conditional statements may contain synchronization constraints similar to methods, that is, active or passive. The tokens available are those defined in the current object. In contrast to methods, which can only be called if the required tokens are guaranteed to be available at compilation time, conditional statements perform a dynamic check. An inner block for which the required tokens are available at runtime is executed. The tokens to the right of `->` are added to the containing object upon leaving that inner block.

The difference between `if` and `when` is that the latter blocks execution until the tokens specified in the list of conditions for at least one block are available, while `if` does never block.

Listing 13 illustrates the use of an `if`-statement.

```
1 class Fork2: embed Fork is
2     method takeOrGiveBack do
3         if down -> up do
4             take;
5         else if up -> down do
6             giveBack;
7         end if;
8     end method;
9 end class;
```

Listing 13: VooDo Kern: Synchronization Constraints in Conditional Statement

Through the use of conditional statements, tokens provide the same flexibility as semaphores. However, the use of tokens is safer, as all option requirements to the left of `->` define an atomic action. Therefore, there is a much lower risk of having unbalanced semaphore operations.

## 2.7. Changes

Because of experiences in this implementation, possible changes to VooDo have been suggested, but not yet incorporated into the language description.

### 2.7.1. Syntax

#### Number Literals

The lexical syntax for VooDo Kern only specifies positive integer and floating point numbers. However, because of changes in the use of type modifiers, negative integers are needed (see below). Consequently, all integer constants are of type `Integer`, as opposed to `Natural` as specified in the original description. In the complete language, negative numbers are supported through the use of the `-` operator, which is not available in VooDo Kern.

#### Creators

Creators were originally specified as not having a return type. In order to allow creation of type instances with modified token values, instance-level tokens can be specified to the right of `->` in synchronization constraints. In this implementation, creators must specify a return type, which has to be the containing unit, optionally with type modifiers. The synchronization constraints must not contain instance-level tokens. This is semantically identical to the original form, but easier to implement, as the return type (and its expected options) are explicitly specified in a way syntactically discernible.

### 2.7.2. Distributed Options

These changes will be included in a later version of the language specification:

- According to the original specification, token declarations and definitions can specify a default token value different from zero. This is no longer possible.
- In the original version, type modifiers can only specify positive values. They are always based on the default value of the token in the real base type (the original unit containing the token declaration/definition). In the modified version, type modifiers can specify negative values. They apply to their specified base type (for example, a named value) instead of the real base type (see Section 2.3.3).

# 3. Methods and Tools

## 3.1. Objective

To evaluate the complexity of implementing the VooDo language as well as to provide a basis for further work, we developed a test implementation, which, for reasons of scope, is only a compiler frontend, not including a code generator.

## 3.2. General Approach

Basically, a linear approach was used for the implementation. However, the stages were only planned in a rough way, as it had not been deemed possible to anticipate all potential problems in advance. The distinctions between the stages were therefore somewhat more blurred in the actual implementation process, and in several cases changes as many as three stages back were needed.

Below, the planned stages of the implementation are described in short. These stages correspond approximately to the phases in the compiler described in Chapter 5.

**Syntax Analysis** The syntax analysis provides the basis for all further work. This phase includes any semantic tests needed to differentiate between similar syntactic constructs, but not the attributed grammar used for building the data structures.

**Data Structures** In this stage, we create the classes and interfaces for the internal compiler data structures, as well as the parts of the attributed grammar needed for building the data structures from the syntax tree.

**Data Structure Semantics** For data structure semantics, we add those routines and additional data values that provide the connections between the data structures. This includes the complete name resolution mechanism as well as cycle detection. Another part of this stage is the implementation of unit embedding respectively inheritance.

**Semantic Checks** The semantic checks include the routines and values needed for the complete type checking phase in the compiler. This encompasses any checks that need not be performed during name resolution, but can be done afterwards.

**Further Semantic Checks (optional)** Some of the features listed in Section 11.2 have been seen as optional in case the implementation would be finished sooner than anticipated. As this has not been the case, the stage has been dropped.

Table 3 lists the anticipated proportions for the individual stages of the time needed to develop the complete implementation (compare Table 5 in Chapter 10).

| Stage | Proportion |
| --- | --- |
| Syntax Analysis | 10% |
| Data Structures | 30% |
| Data Structure Semantics | 20% |
| Semantic Checks | 40% |

Table 3.: Anticipated Proportions of Stages

For the implementation itself, we placed an emphasis on flexibility and a clean, object-oriented design to facilitate later extensions.

## 3.3. Implementation Tools

### 3.3.1. Programming Language C#

C# is a modern object-oriented programming language similar to Java [9]. It was originally developed by Microsoft and standardized by the European Computer Manufacturers Association (ECMA) and the International Organization for Standardization (ISO) [21].

At the time of this work, the current version of C# is 1.1, with 2.0 being available as a public beta. For the implementation of the VooDo compiler we chose the latter because of several new features [23], especially support for generic types.

The following reasons were decisive in choosing C# as opposed to other possible languages like C, C++, or Java. The languages in parentheses indicate which other language does not fulfill the respective requirement.

- Object-oriented language. (C)
- Automatic memory management and no need for unmanaged pointers. (C++)
- Availability of an integrated development environment. (Java)
- Familiarity with the language and personal preference of the author. (Java)

An argument for the use of Java that has been considered was portability, as the default C# implementation is only available on Microsoft Windows. However, the availability of the open-source .NET/C# runtime environment *Mono* [18] was deemed sufficient.

### 3.3.2. Parser Generator Coco/R

Coco/R [12] is a parser generator like YACC, but also supports generating a scanner and using an attributed grammar directly. It is available for several programming languages, including C#.

From an input file, Coco/R constructs a recursive descent parser [20]. Thus, in principle, it can only use LL(1) grammars. It is possible, however, to use semantic checks written in the target language to process LL($n$) grammars for an arbitrary $n$ [13].

Coco/R was chosen as parser generator for this work because it was deemed adequate for the task and was the only parser generator for C# with which the author had already been familiar.

# 4. Examples

## 4.1. Covariant Routine Parameters

### 4.1.1. Problem

Because of substitutability, that is, an instance of a subtype can be used where an instance of a supertype is expected, types of formal parameters (that is, types of value parameters and supertypes of type parameters) must be contravariant. Consider the example in Listing 14, which uses covariant parameters where they would be useful to tightly couple two type hierarchies.

```
1  class type Animal is
2      method feed (f: Food);
3      creator new: Cattle do ... end creator;
4  end class type;
5
6  class Cattle: embed Animal is
7      method feed (f: Hay) do ... end method;
8  end class;
9
10 type Food is ... end type;
11
12 class Hay: embed Food is
13     creator new: Hay do ... end creator;
14     ...
15 end class;
16
17 class Meat: embed Food is
18     creator new: Meat do ... end creator;
19     ...
20 end class;
```

Listing 14: Example: Covariant Parameters (Invalid)

Note the method declarations/definitions in lines 5 and 11. However, this is not correct (and not allowed in VooDo), as Listing 15 shows.

Since `a` is declared as `Animal`, the call to `feed` in Line 4, while obviously not correct, is valid and cannot not be rejected by the compiler. For a more elaborate discussion of covariant parameter types (and how they are handled in Eiffel), see Chapter 17 in [11].

```
a: Animal = Cattle.new;
f: Food = Meat.new;
...
a.feed(f);
```

Listing 15: Example: Use of Covariant Parameters (Invalid)

### 4.1.2. Solution in VooDo

Named types in VooDo (see Section 2.3.3) provide a way to partially support covariant parameter types consistent with substitutability. Listing 16 modifies the incorrect example from Listing 14 to be correct VooDo code.

```
class type Animal is
    static type F = Food;
    static method createFood: F do ... end method;
    method feed (f: F);
    creator new: Cattle do ... end creator;
end class type;

class Cattle: embed Animal is
    static type F = Hay;
    static method createFood: F do ... end method;
    method feed (f: F) do ... end method;
end class;

type Food is ... end type;

class Hay: embed Food is
    creator new: Hay do ... end creator;
    ...
end class;
```

Listing 16: Example: Named Types and Covariant Parameters

The example in Listing 17 shows a possible use of these units.

```
a: Animal = Cattle.new;
f: a.F = a.createFood;
...
a.feed(f);
```

Listing 17: Example: Use of Named Types and Covariant Parameters

This solution is somewhat restricted in that it requires the use of the `createFood` method. However, in most cases, this or a similar solution to create the actual argument is necessary anyway. Otherwise the actual type of the object containing the routine to be called must be known, eliminating the need for covariant parameter types.

### 4.1.3. Issues and Implementation

Below is a discussion of the issues this approach has and an overview of how these are handled in the implementation.

#### Type Equivalence and Subtyping

The following problems concerning named types are relevant in the example:

- Type equivalence between a named type and its specified type value can only be asserted in certain circumstances (see Section 9.1.3). It is not possible to guarantee an equivalence between `a.F` and `Food` or `Hay` (which is not necessary in this case).
- The type of `f` must be a subtype of the type of the parameter of `a.feed`. In this case an equivalence can be ascertained even, because `a` is the same object in both cases (see below).
- The type of the parameter of `feed` in `Cattle` (the named type `F`) must be a supertype of the corresponding type in `Animal`. Since a named type can only change covariantly, a special type equivalence is assumed between an overridden and overriding named type, regardless of the specified supertype or type value, but only when used inside the unit (that is, when checking substitutability of members). See Section 9.1.3 for details.

#### Object Identity

The indirect use of covariant parameter types utilizing named types requires knowledge of object identity. The example in Listing 17 is only possible because it can be asserted that `a` is the same object in Line 1 and Line 4. The solution in this case (and in this implementation) is simply to use named values, which can only be assigned to once.

To be able to ascertain object identity, the prefixes of the member accesses must be available. This includes the type `a.F` of `f`, the routine call to `a.feed`, as well as the type of the parameter in `a.feed` (also `a.F`). The implementation handles this using special member access classes that provide information about the object to which the access belongs (see Section 7.1.3). Additionally, it is necessary to analyze member signatures for each member access, so as to for example connect the type of the parameter in the call to `a.feed` with the prefix `a` (also discussed in Section 7.1.3).

## 4.2. Cycle Detection

### 4.2.1. Approaches

Most well-known object-oriented programming languages perform cycle detection only in cases where it is absolutely necessary for a compiler to function, for example in inheritance hierarchies. For other references cycles are not rejected by the compiler and in many cases only found when testing the application.

VooDo uses a different approach in that it tries to prevent invalid cycles wherever possible. Consider the example in Listing 18, which contains a non-trivial invalid cycle (compare Listing 9 in Section 2.4).

```
1  class A is
2      x: Integer = 2 * y;
3      y: Integer = f;
4
5      public var f: Integer do
6          if g > 5 do
7              ...
8          else
9              ...
10         end if;
11     end do;
12
13     public g: Integer do
14         return x;
15     end do;
16     ...
17 end class;
```

Listing 18: Example: Non-Trivial Invalid Cycle

A logical order on all units, layers of units, and members is used to implicitly define invalid cycles (see Section 2.4).

### 4.2.2. Implementation

Cycle detection has turned out to be the most problematic part of the implementation. Basically, references between elements are used to implicitly defined the logical sequence of elements, which corresponds roughly to the order the elements are resolved in. A recursion detection algorithm is used to find invalid cycles, and a number of flags control exemptions to the cycle detection rules. The cycle detection algorithm is explained in detail in Section 7.2.

## 4.3. Lifted Types

### 4.3.1. Approaches

Other object-oriented languages normally allow the use of null references or null pointers as values for most non-trivial types, that is for example classes. Experience has shown, however, that in the majority of cases, no null values should be allowed, for example in actual parameters. This implies a need to perform a large number of dynamic checks if errors resulting from unexpected null values are to be handled gracefully.

In VooDo, the special value `Null` (a module) is only allowed in so-called *lifted types* (also known as *maybe-types*). Listing 19 shows a method with two parameters, the second of which may be `Null`, that is, contain a null value.

```
1 method foo (s: String, i: ?Integer) do
2     ...
3     if x: Integer = i do
4         ...
5     end if;
6     ...
7 end method
```

Listing 19: Example: Lifted Type

To ensure that only lifted types can contain `Null`, all data elements must be initialized before their first use. `Null` is a subtype of all lifted types, but of no non-lifted type.

To ensure static type safety for member accesses, maybe-types do not export the interface of their underlying just-type. Therefore, a dynamic type check (see Section 2.3.4) is needed in order to use

### 4.3.2. Implementation

The implementation of lifted types is straight-forward and has not caused any significant issues.

# Part II.

# Implementation

# 5. Overview

The implementation of the comiler frontend is divided into four projects:

**vkc** The main assembly (in C#, not to be confused with the equivalently termed VooDo concept). Contains the static class `Vkc.Program`, which implements a main routine that checks for a filename as parameter and calls first the parser and then the semantic checking routines.

**Tools** Assorted helper classes and interfaces (namespace `Vkc.Tools`) that do not fit anywhere else. This includes the `Token` class used by Coco/R for source tokens (not to be confused with the VooDo language concept of the same name), as well as error handling classes.

**Parser** The `Scanner.frame` and `Parser.frame` from Coco/R (both slightly modified), and the Coco/R input file `Vkc.Parser.atg`. These are used by Coco/R to generate the `Scanner.cs` and `Parser.cs`, which implement the recursive descent parser used by the compiler. The namespace used is `Vkc.Parser`.

**Data Structures** This is where most of the implementation (except the syntax analysis) resides. It contains the namespace `Vkc.DataStructures` and a number of sub-namespaces thereof.

This assembly is larger than originally anticipated, as no useful factorization could be found, because there must not be any circular dependencies between assemblies, and all classes and interfaces in this one are interdependent in some way.

It should be possible to add a code generator as an additional project, through the use of either sub-classes of the existing data structure classes (modified to utilize abstract factory methods), or the extraction of interfaces for the existing data structures. The latter approach is anticipated to require less changes to the current code.

# 6. Syntax Analysis Phase

The syntax analysis phase in the compiler frontend consists of two parts, the actual syntax analysis using the Coco/R-generated parser, and the construction of internal data structures.

## 6.1. Syntax Analysis

### 6.1.1. Syntax Grammar

The syntax analysis has been implemented in a straight-forward way, by basically translating the VooDo Kern syntax grammar from [17] to an LL(1) grammar for Coco/R. Of course, the VooDo Kern grammar is not fully LL(1), so some changes were needed. Most changes are restructurings or simple resolver routines providing a fixed number of additional look-ahead symbols. These trivial changes are not detailed here.

However, in the following cases, more complex resolver routines are required:

**Data Member definition** Parameterless function declarations and data member definitions without initial value are syntactically identical and need not necessarily be differentiated (see sections 2.3.3 and 6.2.2). However, data member definitions containing an initial value can be discerned from function definitions/declarations by finding an equal sign, outside of parentheses, before either a semicolon, the keywords `do` or `end`, or the end of the file.

**Routine definition** Routine definitions can be told apart from declarations by the keyword `do`, outside of parentheses, before either a semicolon, the keyword `end`, or the end of the file.

**Assignment condition** VooDo contains a special assignment condition used for dynamic type checks (see Section 2.3.4). To differentiate such an assignment condition from a normal condition, a colon must follow the first identifier, and an equal sign must be found at least another token further, outside of parentheses, before a comma or the keyword `do`.

**Assignment** An assignment differs from an invocation in that an equal sign exists, outside of parentheses, before a semicolon, the keywords `end` or `else`, or the end of

the file.

Listing 20 shows the resolver function for the last of these cases as an example.

```
private static Boolean IsAssignment()
{
    Scanner.ResetPeek();
    Token next;
    Int32 level = 0;

    while (true)
    {
        next = Scanner.Peek();
        if (next.kind == _leftPar)
            level++;
        if (next.kind == _rightPar)
            level--;
        if (((level == 0) && ((next.kind == _semicolon) ||
            (next.kind == _end))) || (next.kind == _else) ||
            (next.kind == _EOF))
        {
            return false;
        }
        if ((level == 0) && (next.kind == _equals))
            return true;
    }
}
```

Listing 20: Implementation: Assignment Resolver

### 6.1.2. Attributed Grammar

The attributed grammar is only used for collecting information needed to build the data structure objects (see Section 6.2). All further checks are then performed in these objects.

## 6.2. Data Structures

The compiler frontend uses a number of classes and interfaces to represent the program to be checked internally. In the first phase, the Coco/R-generated attributed grammer (see Section 6.1.2) instantiates these data structures.

### 6.2.1. Units

#### Unit Classes

Figure 1 shows a class diagram for the classes used in representing the units of the assembly to be compiled.

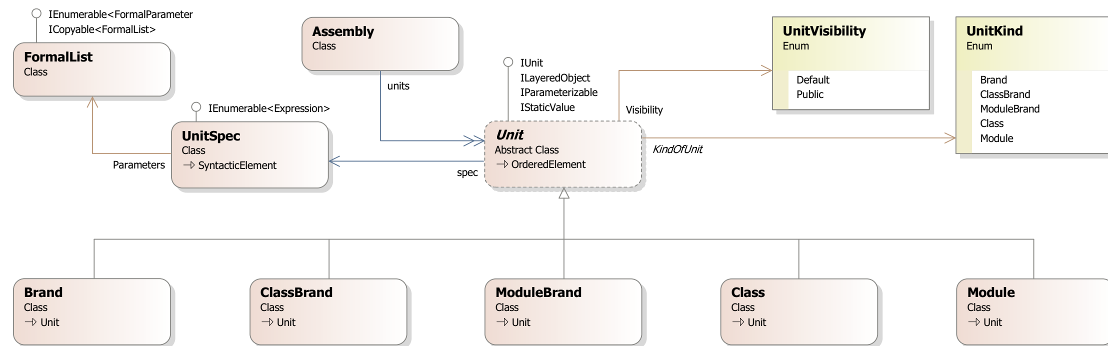



Figure 1.: Class Diagram — Units

The most important object is the `Unit` class. For each unit found in the source code, an instance of a derived class of `Unit` is created by a call to the abstract factory method `Unit.Create`, specifying the kind of unit to create (a `UnitKind` enumeration value), the visibility of the unit (a `UnitVisibility` value) and a `UnitSpec` object, which contains the name of the unit, information about its formal parameters (see Section 6.2.3), as well as the expressions defining any embedded units.

#### Units and Layers

Units are subdivided into layers, which need to be represented. In addition, for generic units, it is necessary to evaluate the concrete units for all combinations of actual parameters used (called a *generic instance* from here on), because of cycle detection (see Section 7.2.6). The nested classes in `Unit` displayed in Figure 2 serve this purpose.

In every unit object, a single `StaticObject` is instantiated that represents the static layer of the unit. For generic units, an instance of `GenericInstance` is created for each combination of actual parameters and used instead of the `Unit` object itself when accessed from outside the unit (the unit cannot be accessed without actual parameters, except for the static layer). For semantic checks of the unit itself, a default generic instance is used, with the formal parameters substituted as actual ones. As with the static layer of the unit itself, a `SharedObject` instance for the shared layer is created in each generic instance, as well as a `GenericStaticObject` instance. The latter is, in fact, mostly a proxy object for the `StaticObject` instance of the unit, which is needed for checking the

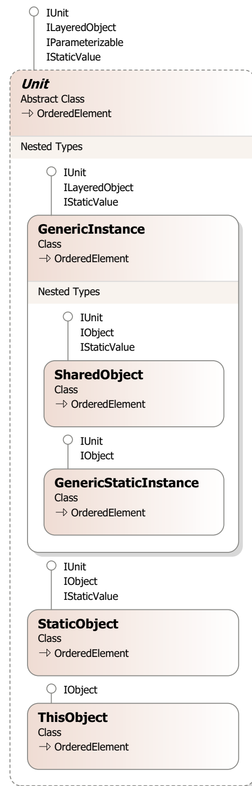



Figure 2.: Class Diagram — Nested Classes of `Unit`

actual parameters used during the semantic checking phase (see Chapter 9).

### Assemblies

Each unit is added to an `Assembly` object, which checks for double unit declarations and is used as the starting point for later phases. Originally, `Assembly` was implemented as a static class. While this has subsequently been changed, only one assembly object exists in `Assembly.CurrentAssembly`. If needed, only small modifications (mainly in the `Unit` class) would be required to support multiple assemblies in a single compilation run.

## 6.2.2. Members

### Member Classes

The classes used for representing members are depicted as a class diagram in Figure 3.

As with units, all member classes are built around a single abstract base class, `Member`.

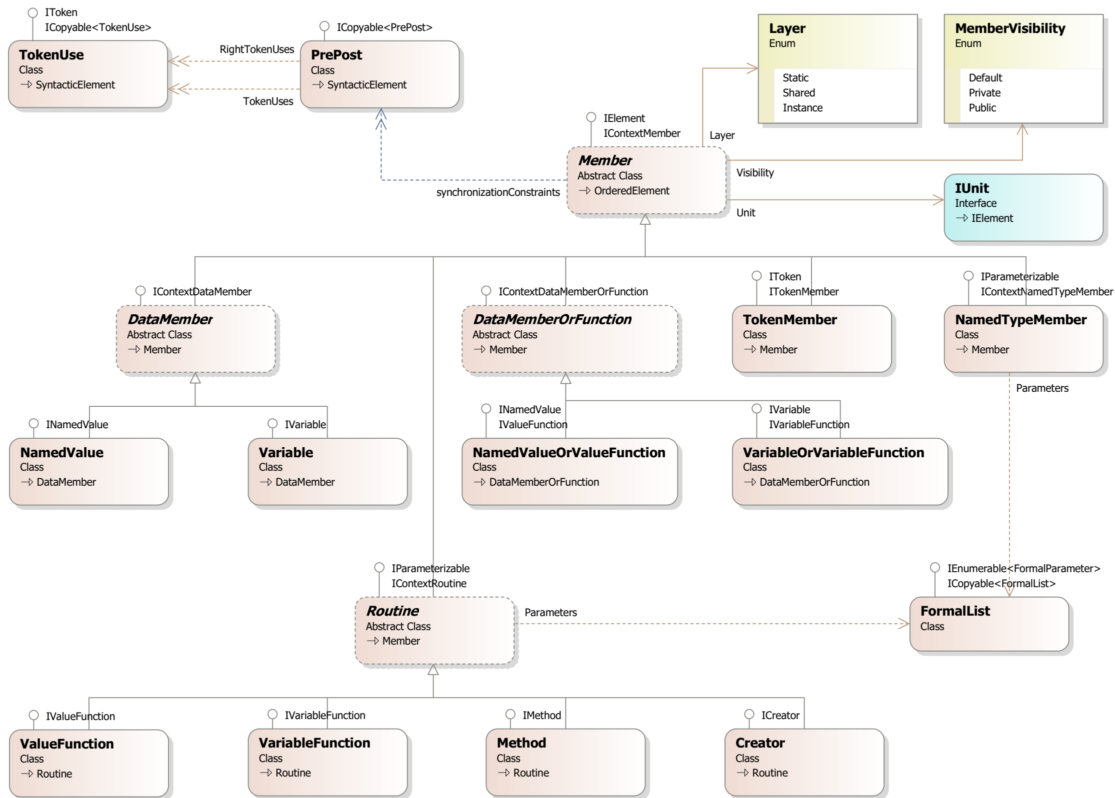



Figure 3.: Class Diagram — Members

However, constructors of the subclasses are used to create the objects instead of an abstract factory method (except for `DataMemberOrFunction`, see below). To these constructors we pass the layer of the member (`Layer` enumeration value), its visibility (`MemberVisibility`), and optional synchronization constraints (a `PrePostList` object, shown as a collection association between `Member` and `PrePost` in Figure 3). Furthermore, depending on the type of member, a list of parameters (see Section 6.2.3), a type expression, an initial value or type expression, and a body may be specified. The last possibility is never actually used, as a member object is created as soon as the declaration is found. If this is later discovered to be a definition, the body is assigned separately.

### Assigning Members to Units

Each member is assigned to the containing unit using the `Unit.AssignMember` method. Its behaviour depends on the kind of the unit and the layer of the member. Static members are always added to the `StaticObject` instance. For non-generic units, instance members are simply added to the member collection in the unit itself, while for generic units, instance and shared members alike are added to a collection of member templates,

and the event MemberTemplateAdded is raised. This event is processed by all existing GenericInstance objects and leads to the creation of a new instance of a subclass of ConcretizedMember, a nested class of Member (each subclass of Member has its own nested subclass of ConcretizedMember). This new member object is then added to either the collection of members in the generic instance or its SharedObject instance, depending on the layer.

It has previously been deemed possible to simply maintain all members in one collection in the unit. However, as has been realized later in the implementation, name resolution of member signatures must be performed separately for each combination of actual parameters, making the current solution outlined above necessary. Additionally, this has turned out to be advantageous for name resolution in general.

#### Data Member Definitions ↔ Function Declarations

A special case exists for data members and parameterless functions, as they are indistinguishable in certain instances (see Section 2.3.3).

The implementation uses DataMemberOrFunction and its subclasses to represent this special case. Both NamedValueOrValueFunction and VariableOrVariableFunction contain a static factory method Create that is used to instantiate each data member found during syntax analysis. These methods check if it is possible to assert that the member is either a data member or a function, in which case the corresponding class is used. Otherwise, a new instance of the containing DataMemberOrFunction subclass is returned.

If, during name resolution, such an ambiguous member is used as a data member (i.e., assigned to), only a flag is set to mark the member as a definition. We had considered early in the implementation to replace the NamedValueOrValueFunction or VariableOrVariableFunction instance with an instance of the correct member class. However, in addition to being unnecessary, such an approach would have required the excessive use of events to replace all existing references, and was thus deemed too complicated.

### 6.2.3. Formal Parameter Lists

Formal parameter lists are used for generic units, routines, and named types. The class diagram in Figure 4 shows the classes used to represent formal parameters.

Formal parameters are a straight-forward representation of the syntactic elements themselves. Note that synchronization constraints are not part of formal parameter lists, but have their own collection class, PrePostList (see Section 6.2.2). This separation was implemented because there are cases where formal parameter lists are needed without synchronization constraints (for example generic units), and vice versa (for example pa-

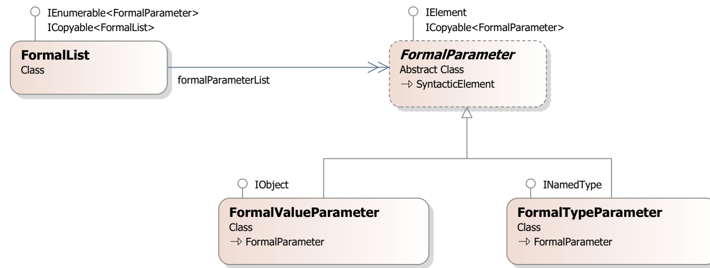



Figure 4.: Class Diagram — Formal Parameter Lists

rameterless routines). Additionally, both element types are resolved separately, giving no advantage to a combined data structure.

### 6.2.4. Routine Bodies and Statements

A class diagram for body and statement classes is shown in Figure 5.

Compared to units and members, routine bodies and statements are trivial. The only interesting aspect are block statements (only `if` and `when` are supported in this implementation, no `while`). A `BlockStatement` object contains a list of `ConditionalBody` instances, each containing a body, and each but at most one also containing a `CondList` object, which represents the conditions for the block. In this way, the handling of a routine body and the body in a block statement is completely transparent (which has to be taken into account during name resolution through the use of different context classes; see Section 7.1.2).

The `Body` class contains a list of statements simply appended in the order found in the source file.

### 6.2.5. Expressions

#### Expression Classes

Expressions are the basis for name resolution. Wherever a reference to another element is needed in the source code, an expression is used. Figure 6 provides an overview of the used classes.

Most concrete expression classes correspond to elements in the syntax grammar and are self-explanatory. However, several special cases are described below.

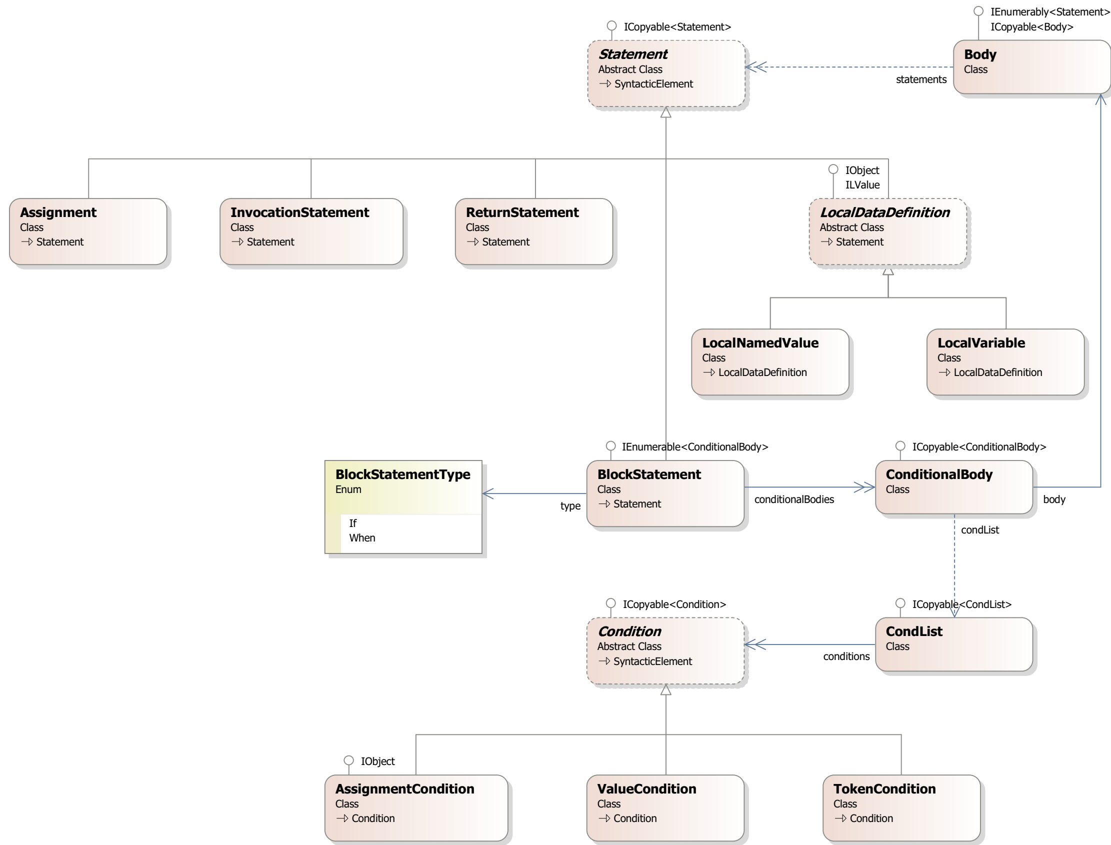



Figure 5.: Class Diagram — Routine Bodies and Statements

### Expressions and Defined Elements

Each expression defines a language element, for example a value or object, a type, or a routine invocation. The `Expression` object for an expression contains the kinds of elements the expression may refer to, which depends on the context in which the expression is used. This is usually set in the constructor, or a `set`-accessor for a property, of the object containing the expression. For instance, a routine that has a result type sets the `Kinds` property of its `resultTypeExpression` object to `ElementKinds.Type`.

This information is checked during name resolution and used to make sure that casts to specific interfaces are safe. See Section 7.1.1 for detailed information.

### Type Expressions

Expressions used as types have the following restrictions:

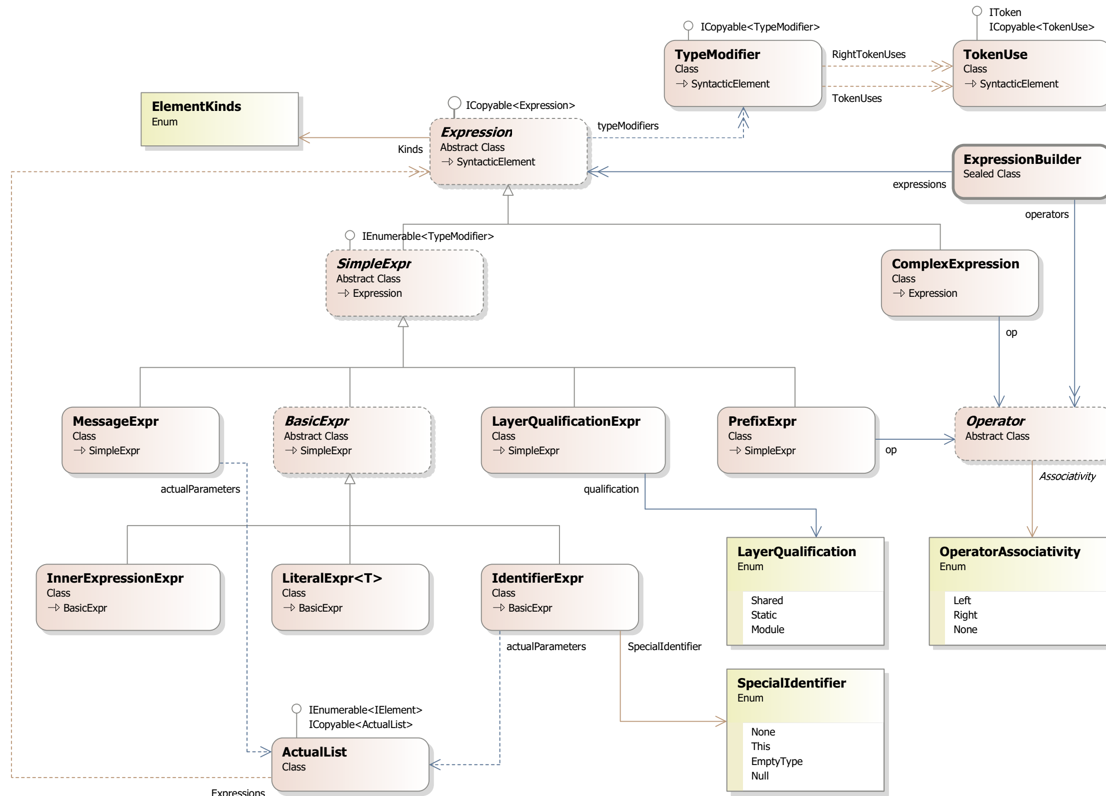



Figure 6.: Class Diagram — Expressions

- Such expressions (which can precede the assignment operator `=` in the lexical ordering) shall not contain the `=` operator except in parentheses.
- Such expressions shall access only units, named types, and literals. They shall not access named values and variables or invoke routines.

The first restriction is handled by syntax analysis and not relevant here. The second one, however, restricts the use of elements in the expression and thus all sub-expressions. Therefore, an additional flag is available in the `Expression` class that indicates an expression being part of a type expression. This flag is automatically set if the kind of elements for the expression is set to include types (but can also be set manually through the `IsTypeExpression` property), and is propagated to all sub-expressions, including actual parameter lists.

For details on the handling of the flag, see Section 7.2.5.

Type expressions may also contain type modifiers (see Section 2.6) always handled by the `Expression` class itself.

### Layer Qualifications

What is called *layer qualifications* here and handled by the `LayerQualificationExpr` class is the possibility of using only the static layer of a unit or only the static and shared layers of a class or class brand. This is similar to a message (and there is no actual distinction in the original VooDo Kern grammar), but sufficiently different in the implementation to warrant using a separate class.

### Operators

Operator support in VooDo Kern is severely limited. Basically, operators are nothing more than a special syntax for routine calls, with two exceptions. These are the `?` operator that denotes maybe-types (see Section 2.3.3), and the `=` operator that is used as assignment as well as comparison operator depending on context. It was thus considered to drop support for operators completely from this implementation.

`PrefixExpression` (for unary operators) and `ComplexExpression` (for binary operators) are the expression classes used. The latter cannot be created directly. In order to support different precedence and associativity, the `ExpressionBuilder` class was implemented, which uses a trivial stack-based algorithm to build a hierarchy of `ComplexExpression` objects.

# 7. Name Resolution

The first section in this chapter discusses the basic approach used for resolving names in the compiler. The more general aspects, including cycle detection, are explained in Section 7.2.

## 7.1. Name Resolution Basics

### 7.1.1. Language Elements

As shown in Section 6.2.5, each instance of `Expression` has assigned the kinds of elements the expression may refer to. Figure 7 contains a class diagram showing the interface hierarchy for the available element kinds and some additional information. As this is a central part of the implementation, the interface members are included.

Note that the `IUnit` and `IMember` interfaces are not shown in this class diagram because they are not directly relevant to name resolution. The additional properties and methods are used internally for the nested classes of `Unit` and member access classes (see Section 7.1.3).

#### `IElement`

The `IElement` interface contains the property `Kinds` (a flag enumeration) used in place of dynamic type checks to find out which of the sub-interfaces the type of an object implements. This is needed because there are cases where a class implements one of the interfaces without it being available in all instances or at all times (i.e., depending on the state of the instance). An example for the first case would be the `Unit` class, which implements `IParameterizable`, but only instances representing generic units have the corresponding flag set in the `Kinds` property. Although such cases could be handled using inheritance (i.e., a `Unit` class that does not implement `IParameterizable` and a `GenericUnit` subclass), the second type (flags changing over time) cannot. Additionally, this solution would increase the complexity of the class hierarchy in the compiler further, which is already quite high.

The `IsAvailableForType` property works in conjunction with the `IsTypeExpression`

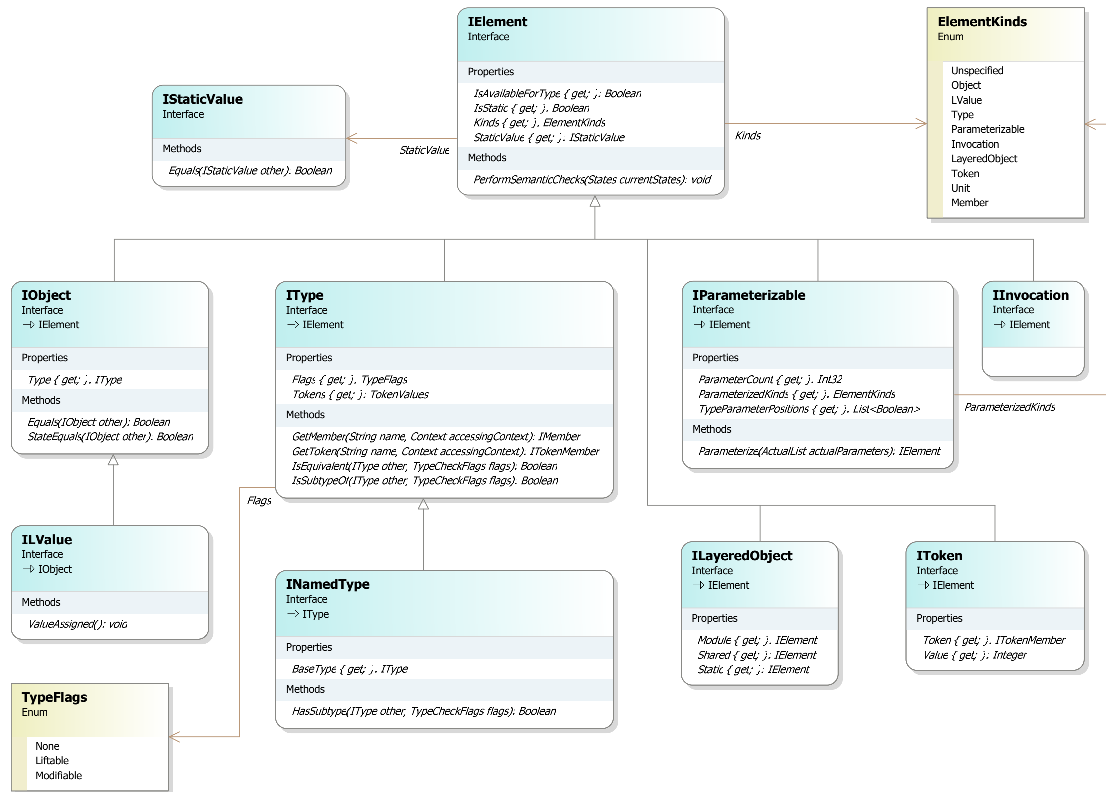



Figure 7.: Class Diagram — Element Interfaces

property of the `Expression` class (see Section 6.2.5). If it is true, the element can be used in type expressions. Otherwise an error is reported when resolving the expression.

`IsStatic` and `StaticValue` are needed to handle statically determined expressions. If `IsStatic` is true, `StaticValue` must return a valid instance of `IStaticValue`. In this implementation, this is only used to assert equality between statically determined expressions since no code generation takes place, and thus no actual values are computed.

The `PerformSemanticChecks` method is used for semantic checks, for details refer to Section 9.3.

### `IObject`

`IObject` is the interface for all elements that can be used as objects. This includes for example class instances, classes and modules, accesses to named value and variable members, local named values and variables, literals, and invocations of routines that have a result type.

Every object in VooDo has a type, contained in the `IType` property. The `Equals` and `StateEquals` methods are used for determining object equality, for the difference between the two methods refer to Section 9.3. `Equals` overloads the default method in the C# `Object` class, and does not override it. These methods must only be called after name resolution is complete, as they depend on the information gathered there in some cases.

#### `ILValue`

This interface is used for all elements usable as l-values, that is, objects that can be assigned to. These are accesses to variable members and named value members without initialization, as well as local variables and uninitialized named values. The `ValueAssigned` method is called when an assignment is resolved and used by the `DataMemberOrFunction` class (see Section 6.2.2) to determine that it is to be used as a data member definition.

#### `IType/INamedType`

Every element usable as type implements the `IType` interface. At this point, only the `GetMember` and `GetToken` methods are of interest, all other properties and methods are used during semantic checking (see Section 9.1).

The `GetMember` and `GetToken` methods are used for accesses to the members of an instance of a type. This is normally only relevant when accessing an `IType` instance through the `Type` property of an `IObject` instance (see above). The need for two different methods results from the fact that token members are handled differently in a number of ways. Especially, they cannot be accessed like other members.

The `INamedType` interface is used for named type member accesses and type parameters. It only adds properties and methods relevant for semantic checks.

#### `IParameterizable`

Elements that take parameters do not directly implement the interface for the category into which they fall. Instead, `IParameterizable` is used. This interface has three properties, two of which, `ParameterCount` and `TypeParameterPositions`, provide information about the parameters the element takes. The last, `ParameterizedKinds`, corresponds to the `Kinds` value of the element returned by the `Parameterize` method. This method takes an `ActualList` as parameter and is used to get the parameterized version of the element.

`IInvocation`

`IInvocation` is implemented by elements that can be invoked without returning a value (i.e., methods without a result type). It is not used for routine calls that return a result, which implement `IObject` instead.

As no further information is needed for an invocation, this interface is only used as a placeholder and does not contain any members. Note that an `IInvocation` instance already contains all actual parameters needed, the unparameterized element implements `IParameterizable` instead (see above).

`ILayeredObject`

The `ILayeredObject` interface is used in the `LayerQualificationExpr` class (see Section 6.2.5). It defines objects that have layers, that is, the `Unit` and `GenericInstance` classes, not the other layer classes. For this reason, `ILayeredObject` is separate from the `IUnit` interface.

`IToken`

The `IToken` interface is straight-forward. It provides properties containing the token member used, as well as the minimal value of the token.

### 7.1.2. Context

Resolving names is largely dependent on the context the name is used in. That is, the same identifier used in one unit may point to a different element than in another unit. This implementation uses the class `Context` and its subclasses for this task. A class diagram is depicted in Figure 8.

Instances of these classes are created as needed in a hierarchical way. Each instance of `Context` may contain a reference to a containing context. If an identifier is to be resolved, the `ResolveName` or `ResolveSpecialIdentifier` methods are used. If the requested identifier cannot be found in the current context, the request is propagated to the containing context.

Every context object contains a reference to a unit and a member, which may be null. The context first used for name resolution is called the *primary context*, and it is propagated through the context hierarchy during a resolution request. This makes it possible to find out in which unit and member (if any) the access takes place at any point in the hierarchy. An example where this knowledge is required would be checking the visibility

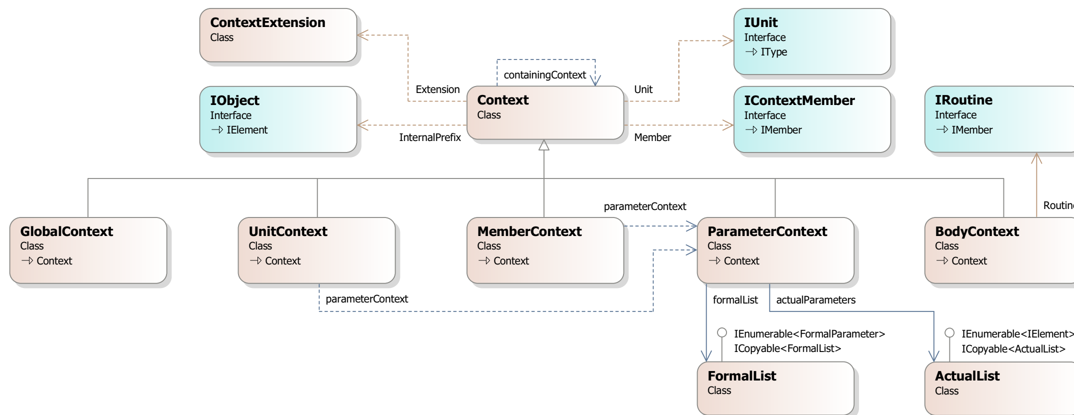



Figure 8.: Class Diagram — Context

of a unit member access.

For members that take parameters, the `MemberContext` instance for the member is the containing context of the `ParameterContext` instance for the formal paramter list. However, for generic units, the parameter context uses the same containing context as the unit context. The unit context has an additional reference to a `ParameterContext` used in addition to the containing context when an identifier cannot be resolved directly. The `MemberContext` class also contains such a reference used for creators (see Section 7.1.4).

Note that each `ParameterContext` instance contains a reference to a list of actual parameters. If not available, the formal parameters are substituted for the actual ones instead (see Section 6.2.1).

Figure 9 shows the hierarchy of `Context` classes for the body of a method (with parameters) in a generic unit.

`ContextExtension` is needed for inheritance, for details refer to Section 8.2.

A special case exists for creators, see Section 7.1.4.

### 7.1.3. Member Accesses

This section deals with resolving the names of unit members. As with other types, the `GetMember` and `GetToken` members of the `IType` interface are implemented for this purpose. These methods are called either directly from a `MessageExpr` instance (see Section 6.2.5) or a from a `UnitContext` object.

One aspect of member resolution is the fact that members are distributed over several layers. Each layer class (see Section 6.2.1) implements `IUnit`, which is a sub-interface of

Figure 9.: Example — Context Hierarchy

`IType`, and thus implements its own versions of `GetMember` and `GetToken`. These methods look in all layer classes accessible to themselves. For example, the default generic instance in a generic unit tries to find a member first in its own member collection, and then in the ones of its shared layer and static layer.

It was considered early in the implementation that classes representing unit members (i.e., subclasses of `Member` and `Member.ConcretizedMember`) directly implement the necessary `IElement` sub-interfaces (in addition to `IMember`). There are, however, the following problems with this approach:

- VooDo does not allow write access to member variables except from within the object containing the variable, that is, when it is possible to access the variable using `this` (the expression *internal* member access is used from here on). This implies that an access to a member variable is different depending on whether this condition holds or not. Concretely, the element representing the variable must have the value `ElementKinds.LValue` set only if accessed from within its own object.
- Routines and named types can be parameterizable. As type parameters are available, the complete member signature must be resolved separately for each combi-

nation of actual parameters. Objects representing the combination of member and actual parameters are thus needed.

- To use distributed options, it is necessary to know the object to which a member belongs, that is, the element used as prefix for the member access.

Therefore, we chose a different concept, using the abstract base class `MemberAccess` (a nested class of `Member`) and a hierarchy of subclasses. For each access to a member in the source code (be it an explicit access using `MessageExpr` or an implicit one using `IdentifierExpr`, `PrefixExpr`, or `ComplexExpr`), a new access object is retrieved by using the `GetAccess` method (of the `IMember` interface), containing information about the accessing unit and member (if available), the element used as prefix to access the member (or null if the access was implicit), a boolean value indicating if the access is internal, and a list of actual parameters if required.

The reason for the need of a boolean value to indicate an internal access in addition to the prefix arises from the handling of member signatures. For each member access object created, the signature of the corresponding member is resolved anew. Any internal member accesses inside the signature actually refer to members in the prefix of the original member access. This is modelled by using a new `Context` object for resolving the signature, which has the `InternalPrefix` property set to that prefix. Whenever this property is set for a context in the hierarchy (it is propagated downwards if not set), any internal member access objects are passed the prefix it holds. Listing 21 illustrates this.

```
1  class A is
2      static type T = {};
3
4      method m: T do ... end method;
5      ...
6  end class;
7
8  class B is
9      var x: A = ...;
10
11     method n do
12         ...
13         var y: A.T;
14         ...
15         y = x.m;
16         ...
17     end method;
18     ...
19 end class;
```

Listing 21: Example: Member Access

The member access to `T` (Line 4) for `x.m` in `B.n` (Line 15) is internal, but uses the `IObject` instance for `x` as prefix.

An exception to the above are tokens, which are used directly as members, because the

need for a distinct member access class is not existent in that case.

### 7.1.4. Creators

Name resolution in creators has to be handled in a particular way, as creators are static or shared routines, but have access to instance members, and an explicit `this` is available.

The context used for resolving names in a creator body is a normal `BodyContext` object. However, its containing context is not the `MemberContext` instance of the creator, but a special one. This, like a `UnitContext` object for a generic unit, contains an additional reference to the `ParameterContext` instance of the creator, and its containing context is the `UnitContext` instance for the class representing the instance layer of the unit. Figure 10 shows the context hierarchy for a creator that takes parameters and is a member of a generic unit.

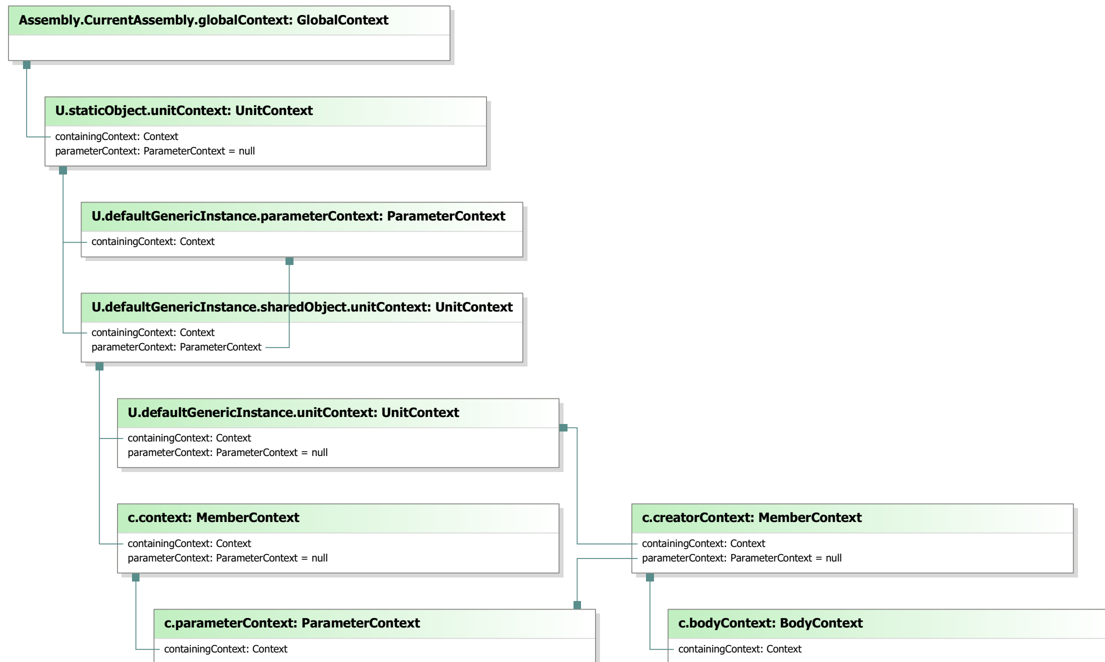



Figure 10.: Example — Creator Context Hierarchy

## 7.2. Name Resolution/Cycle Detection

Despite being anticipated as a small part of the implementation, cycle detection has turned out to be the most complicated problem. A number of approaches were considered and implemented in part until the current solution was found.

### 7.2.1. Considered Approaches

#### Cycle Detection Separate from Name Resolution

The first basic approach was to implement name resolution by traversing the implicit syntax tree with no consideration for cycles. This is obviously not correct and would lead to an endless recursion of the compiler. However, it has been convenient for implementing the basics of name resolution (see Section 7.1).

#### Name Resolution/Cycle Detection per Unit

Based on a previous version of [17], in this approach, each unit had to be handled separately. Apart from that, the approach was basically similar to the following one, including the two-phase concept, while being quite different in the actual implementation.

#### Global Two-Phase Name Resolution/Cycle Detection

After [17] had been changed to incorporate the current description of element order (see Section 2.4), it became necessary to reimplement cycle detection. A two-phase approach was considered, with the first phase containing the resolution of types (except in routine bodies) and the second phase all the rest, with all types being available at that point. This would have simplified the name resolution/cycle detection phase considerably.

However, it would have implied that no type expression accesses any object in a circular way, especially any unit used as object. Without this restriction, there are no advantages in using two phases.

Apart from the use of two different phases, the actual implementation of this approach was very similar to the current one, as the name resolution methods were the same for both phases, with only a parameter determining the current phase.

#### Global One-Phase Name Resolution/Cycle Detection

The current solution corresponds to a large extend to the description of the member sequence, as presented in Section 2.4. There are, however, still a number of special cases that require consideration.

### 7.2.2. Ordered Elements

Cycle detection is based on the concept of ordered elements. These are elements (see Section 7.1.1) upon which the logical order is defined. Thus, ordered elements are basically units, layers of units, and members.

The abstract `OrderedElement` class is used as base class of ordered elements, and provides the basic name resolution/cycle detection algorithm. It is shown in listings 22 and 23.

```
public abstract class OrderedElement
{
    protected static readonly
        Dictionary<OrderedElement, OrderedElement> incompleteElements =
            new Dictionary<OrderedElement, OrderedElement>();

    internal static void ResolveIncompleteElements()
    {
        while (OrderedElement.incompleteElements.Count > 0)
        {
            foreach (OrderedElement element in
                OrderedElement.incompleteElements.Values)
            {
                element.ResolveNames(true);
                break;
            }
        }
    }

    protected NameResolutionState nameResolutionState;
    protected Boolean mustComplete;

    protected abstract Boolean ResolveNames();

    protected abstract void HandleCyclicReference();

    ...
```

Listing 22: Implementation: `OrderedElement` Class (1)

The most important part is the non-abstract `ResolveNames` method. Its parameter, `mustComplete`, is true if the element must be finished before name resolution on the calling one can continue, which implies that the element represented by the current instance of `OrderedElement` comes before the one calling this method in the logical order. Three cases can be distinguished for the state of the current element:

- Name resolution for the element has not been started yet. It is started using the overridden `ResolveNames` method.
- Name resolution for the element is in progress. This implies a recursive call of this method, meaning that a cycle has been detected. There are several sub-cases:
    - `mustComplete` is true in both the previous and current call. This would mean

```
    ...

    public Boolean ResolveNames(Boolean mustComplete)
    {
        if (!OrderedElement.incompleteElements.ContainsKey(this))
            OrderedElement.incompleteElements.Add(this, this);

        Boolean completed = true;

        switch (this.nameResolutionState)
        {
            case NameResolutionState.NotStarted:
                this.mustComplete = mustComplete;
                this.nameResolutionState = NameResolutionState.InProgress;
                completed = this.ResolveNames();
                break;

            case NameResolutionState.InProgress:
                if (this.mustComplete && mustComplete)
                    this.HandleCyclicReference();
                return false;

            case NameResolutionState.Complete:
                return true;
        }

        if (completed)
        {
            this.nameResolutionState = NameResolutionState.Complete;
            OrderedElement.incompleteElements.Remove(this);
            Assembly.CurrentAssembly.Ordering.Add(this);
        }
        else
            this.nameResolutionState = NameResolutionState.NotStarted;

        return completed;
    }
}
```

Listing 23: Implementation: `OrderedElement` Class (2)

that the element must be finished before itself, that is, an illegal cycle has been detected.

  - In the previous call, `mustComplete` is false, but it is true now. This must not happen, as it would imply the need to finish an element even though the element depending on it need not be finished.
  - `mustComplete` is false in this call. Independent of whether it was true before or not, this means the current call is unnecessary and could only cause problems because of recursion. It is thus aborted.

- Name resolution has been completed before. There is nothing left to do, so the method is exited.

It is possible that `ResolveNames` exits even though name resolution of the element is not

complete, and that no later call exists because no other element depends on this one. For this reason, a collection of unfinished elements is maintained, which can later be used to make sure this phase has been completed for all elements.

There are two restrictions on methods that override the parameterless `ResolveNames` method in `OrderedElement`:

- It must be possible for the method to be started again if it is aborted, either by using some sort of checkpoint concept or by being reentrant.
- If `mustComplete` is false, it must not become true for any depending element.

### 7.2.3. Allowed Forward References

The term *forward reference* applies when an element references another element that does not come before it in the logical sequence. Basically, forward references are handled using the `mustComplete` parameter of the general `ResolveNames` method. If it is false, the reference from the calling to the called element can be a forward reference.

Not all name resolution happens directly in the elements themselves, however. Thus, other objects need to be informed about which references may be forward references. This is done using the `NameResolutionFlags` enumeration, which includes the following values:

`TopLevelTypeIncomplete` Is only significant if the expression it is passed to is a type expression. Indicates that the reference to the type defined by the expression may be a forward reference.

`TypeAsParameterIncomplete` This is passed to expressions defining embedded units and modified to `TopLevelTypeIncomplete` for the actual parameter list, allowing the use of recursive type parameters.

`RoutineIncomplete` Is used whenever an expression may reference a routine, if a recursive call is valid at that point.

`UnitIncomplete` Indicates that a forward reference to a unit is valid (see Section 8.1).

`ActiveTypeModifiers` Is used to specify that active type modifiers are allowed at that point. Does not have any influence on forward references.

### 7.2.4. Name Resolution in Units

For name resolution, the unit itself as well as its layers are considered ordered elements. Name resolution normally starts with the unit itself (which is the instance layer if one

exists), except if invoked through a dependence. Table 4 shows how units and layers depend on each other and on the respective members. A dependence indicates that the `ResolveNames` method of the depending element calls the one of the element it depends upon, with the `mustComplete` parameter set to true if its own was set to true also.

| Element | Depends on |
|---|---|
| Unit | Static Layer |
| | Embedded Instance Members |
| Unit (generic) | Default Generic Instance |
| Generic Instance | Shared Layer |
| | Embedded Instance Members |
| Shared Layer | Static Layer |
| | Embedded Shared Members |
| | Formal Parameters |
| | Shared Members |
| Static Layer | Embedded Static Members |
| | Static members |

Table 4.: Unit and Layer Dependencies

Note that the unit does not depend on any instance members. This is because instance members come after the unit in the logical order. Nevertheless, the `ResolveNames` method of the unit calls the one of each instance member, but with `mustComplete` set to false regardless of what it is set for itself. This ensures that every member is in fact resolved eventually.

Shared members depend on the static layer, and instance members on the containing unit, which is the instance layer in this case.

### 7.2.5. Name Resolution in Expressions

#### Parameters and Return Values

Expressions are resolved by calling the `ResolveNames` method of the `Expression` class. This method takes three parameters:

`context` A `Context` object used for name resolution.

`flags` A `NameResolutionFlags` value (see above).

`mustComplete` A boolean value.

The `flags` and `mustComplete` parameters determine how cyclic references are handled. If `mustComplete` is true, only cyclic references specified in `flags` are allowed to occur.

Otherwise no further checks are needed, and `mustComplete` is set to false for all calls to the `ResolveNames` method of other elements.

The `ResolveNames` method of the `Expression` class returns a boolean value indicating if name resolution has been completed successfully. The resolved element defined by the expression can be found in the `DefinedElement` property.

There are three cases to be distinguished after the method returns:

- The `DefinedElement` property is set to null. In this case, the expression could not be resolved. This can happen for a variety of reasons, for example an unknown identifier has been found, or an object that is used as a prefix could not be resolved completely.
- `DefinedElement` contains an element, but the return value of `ResolveNames` is false. This means that name resolution is basically complete for the expression. However, it could not be completed for the defined object. Depending on the value of `mustComplete` and `flags`, this can mean that an error message has already been printed.
- The return value is true. This implies that `DefinedElement` contains an element and the `ResolveNames` method (if any) of that element has completed successfully.

How these cases are handled depends on the element referencing the expression.

### Prefixes

Forward references are allowed for reading accesses to data members as well as routine invocations if a *safe prefix* is used. Safe prefixes do not contain `this` as well as units (and static/shared layers thereof). This is also handled by the `Expression` class and its subclasses.

### Type Expressions

As explained in Section 6.2.5, expressions defining types are subject to restrictions. Particularily, only certain other elements may be used in such expressions. In the `ResolveNames` method, the `IsTypeExpression` flag described in the section mentioned above is used. If true, as soon as the element is identified, its `IsAvailableForType` property is checked. If false, an error is reported.

### 7.2.6. Resolution of Member Accesses

For resolving member accesses, several steps are needed:

1. The `IObject` element containing the member (respectively its type) is resolved.
2. The `IMember` instance for the member to be accessed is retrieved.
3. A member access object (see Section 7.1.3) is created.
4. The member access is resolved, which includes resolving the signature of the member itself.

The last step is required, because the member may take type parameters on which other parameters or the return value depend, so these have to be resolved for each member access. For this reason, a new instance of `ParameterContext` is created, which allows a list of actual parameters to be assigned. Whenever the name of a parameter is to be resolved, the element of the corresponding actual parameter is returned instead.

For generic units, a similar concept is used, which implies that each distinct generic instance, that is, each combination of actual parameters, is resolved separately, including all unit signatures.

### 7.2.7. Recursive Routines

One special case for name resolution are recursive routines, which are specified in a way to make sure no data member initialization depends recursively on itself by disallowing forward references among routines if any data member is between the routines in the logical sequence. The concepts outlined above cannot enforce this constraint, so a dedicated algorithm was needed.

Each routine has two lists of other routines. The first (the *direct list*) contains routines that may not be referenced, either directly or indirectly through another routine. The second list (the *indirect list*) contains those routines that may not be referenced indirectly via at least one data member. In addition, each data member has a list of routines not to be referenced (directly or indirectly).

Following is an overview of the algorithm. $a$ and $b$ are routines, $x$ is a data member. $a \rightarrow b$ denotes that $a$ references $b$.

**$a \rightarrow b$**
- If $b$ is in the direct list of $a$: Invalid recursion, output error and exit.
- Add $a$ and all routines on its indirect list to the indirect list of $b$.

- Add all routines on the direct list of $a$ to the direct list of $b$.

**$a \rightarrow x$**
- Add $a$ and all routines on its direct and indirect lists to the list of $x$.

**$x \rightarrow a$**
- If $a$ is in the list of $x$: Invalid recursion, output error and exit.
- Add all routines on the list of $x$ to the direct list of $a$.

The operations are called in the `ResolveNames` method of the `Expression` class if the defined and calling elements are routines or data members.

# 8. Inheritance

Inheritance or *embedding* in VooDo Kern is defined by copying all members of the embedded class into the embedding class (see Section 2.5). This is to be done at the source level, and not as binary code.

For the implementation, two approaches were considered:

- Members are not copied, but only references to the (already resolved) embedded members are used.
- Actual copies of the embedded members are created (albeit not directly at source level, but using the data structures created during syntax analysis).

The first approach seems more efficient at first glance. However, as the language is specified under the assumption that the second variant is used, the actual implementation is complex and has many issues. Especially, the fact that name resolution has to be performed anew in the inheriting class negates any efficiency advantage. Thus, the second approach has been used in the implementation.

## 8.1. Name Resolution

Embedded units and members are only resolved when needed, that is, either before the first non-embedded member on the corresponding layer in the embedding unit is resolved (for static and shared layers) or before that layer itself. The following steps are performed:

1. The expressions defining the embedded units are resolved. This is done only once, usually before resolving the first static member. Note that the embedded units themselves need not be resolved completely at this point (this is what `NameResolutionFlags.UnitIncomplete` is used for, see Section 7.2.3).
2. For each embedded unit:
    - The corresponding layer of the embedded unit is resolved.
    - The members to be embedded are collected.

3. Embedded members are added to the unit as if found during syntax analysis, basically. See Section 8.4 for how substituted members are handled.

The `IUnit` interface defines a `GetMembersToEmbed` method that is used by the embedding unit to retrieve a list of `IMember` objects for a specified layer. These are already copies of the original members, with any modifications that are needed made.

## 8.2. Generic Units

Generic units have to be handled specially for embedding, as parameters of the embedded unit need to be translated. In many cases, this is a simple name replacement, but sometimes, more complex expressions are used. Additionally, the name used for a parameter in an embedded class may have a different meaning in the embedding class.

For this reason, the concept of *context extensions* is available. A context extension is a translation table from names to expressions that can be assigned to a `Context` object (see Section 7.1.2). It is possible to create chains of context extensions needed for multiple levels of embedding.

If a context has an extension assigned, it first tries to find the name to be resolved in it. If successfull, the attached expression is resolved. The context used for that is either the one used for the containing (i.e., embedding) unit's `embed` clause if the context extension is the last in the chain, or an instance of `Context` that uses the former (directly or indirectly) as containing context.

When creating a copy of a `Member` instance that is to be used as embedded member, a `Dictionary<String, Expression>` object can be specified, which is used to construct a context extension. This is later passed to the context for the embedded member on creation.

Another special case exists if the actual parameters for a generic unit are statically determined. In this case, all shared members are considered to be static members instead. For simple accesses to unit members, this does not make much difference. When embedding such a statically determined generic unit, however, all shared members actually become static members. To implement this, the `GetMembersToEmbed` method of `GenericInstance` and `SharedObject` (see Section 6.2.1) check if their instance is statically determined whenever the members of the static or shared layer are requested, and returns the correct and appropriately modified members.

## 8.3. Embedded Private Members

Even though not directly visible in the embedding class, private members have to be embedded, because other non-private embedded members might depend on them. To make sure there are no conflicts, private members are to be renamed on embedding.

Each `Unit` object has a unique identifier that is created when the object is instantiated during syntax analysis. This identifier is appended to the name of any inherited private member. For example, the private named value `x` in the unit `A`, which has the unique identifier 7, would be named `x_7` when embedded in another unit. This works recursively, so if an already embedded private member is embedded again, another unit identifier is appended.

Every embedded member contains a reference to its original, which allows retrieval of the identifier of its unit. Therefore, the recursive method performing name translation is defined by the `IMember` interface. Name resolution of members takes place in the `UnitContext` class, which first checks if the primary context (see Section 7.1.2) has an embedded member assigned. In this case, it uses the `TranslatePrivateName` method of that member to get the name an embedded private member would have. If a member with that name exists, it is returned. Otherwise, name resolution continues normally.

## 8.4. Substituted Members

Each `Unit` instance has, in addition to the collection holding instance members, if available, another collection of `IMember` objects containing all members in the unit, including those on the static and shared layer. For each member that is embedded, this collection is checked for a member with the same name. If such a member exists, its `AddSubstitutedMember` method is used with the newly embedded member. Otherwise the new member is itself added to this collection as well as the correct collection for its layer.

For member templates (used with generic units, see Section 6.2.2), it is important that all members of generic instances are informed when a substituted member is added (because it is needed during name resolution). Therefore, the `AddSubstitutedMember` method raises the event `SubstitutedMemberAdded`, which is handled by those member copies.

Because of multiple inheritance, it is possible that a new member already has a substituted member, or that an embedded member with the same name already exists in the collections. The `AddSubstitutedMember` method handles this by creating a new `EmbeddedMemberAccumulation` instance (see below) and raising another event, `Replaced`, which is handled by the unit as well as the layer object to which the member belongs. These objects then remove the previous member from all relevant collections and add

the new member accumulation instead.

The check of whether a member can substitute another one or not is part of the semantic checking phase, see Section 9.2.

## 8.5. Multiple Inheritance

VooDo fully supports multiple inheritance, that is, a unit can embed two or more other units. As embedding of members is based on the member names, this can lead to problems if members from different embedded units have the same name (see Section 2.5).

Multiple embedded members are handled by the `EmbeddedMemberAccumulation` class. An instance is created as soon as a second member of the same name as an already embedded member is embedded. Each object of this class contains two collections, one for all embedded members with the same name, and another one for *active members*, that is, members that could potentially substitute all others.

When a new member is added to an accumulation that already contains at least one embedded member, it is checked for basic compatibility and whether it is to be active or not. If the new member has a higher visibility, all previous active members are removed. The same is true if all previous members are declarations and the new one is a definition.

A special case exists for `DataMemberOrFunction` members (see Section 6.2.2), for which it is not known if they are declarations or definitions. Therefore, such members are kept in the active member collection until the semantic checking phase, were this ambiguity does not exist any longer.

# 9. Semantic Checks

What is called *semantic checks* here includes all those checks that need not (or in some cases cannot) be performed during name resolution. Thus, this implementation uses a third and last phase for these checks. The following assumptions can be made during this phase:

- All expressions are resolved, all references are available.
- For every member it is known whether it is a declaration or definition, including members represented by `DataMemberOrFunction` instances (see Section 6.2.2). This is the case because all such members that have not been assigned to at some point (which makes them data member definitions) can be asserted to be function declarations.

Because of the first assumption, semantic checks can be implemented fairly easily by traversing the syntax tree, which in this implementation does only exist implicitly. Thus, all unit, member, and statement objects contain `PerformSemanticChecks` methods.

## 9.1. Basic Type Checking

### 9.1.1. Interfaces

Type checking is the central task of the semantic checking phase. It is necessary to be able to determine if two types are equivalent, or if one is a subtype of the other. For this reason, the `IType` interface (see section 7.1.1) defines two methods, `IsEquivalent` and `IsSubtypeOf`. The `INamedType` class defines another method, `HasSubtype`. All these methods take two parameters, the first being another `IType` instance, and the second a `TypeCheckFlags` enumeration value (see below).

### 9.1.2. Order of Type Comparisons

Beside the default subtyping relationship that exists between a unit and its embedded units, several special cases are defined (see Section 2.5). The type checking methods

have to take these cases into account by performing type comparisons in the correct order to prevent endless recursion situations. Listing 24 shows the `IsSubtypeOf` and `IsEquivalent` methods of the `Unit` class. Note that this is only used for non-generic units, as for generic units, the unit itself is no type, and a generic instance is used instead.

```
public Boolean IsEquivalent(IType other, TypeCheckFlags flags)
{
    if (other == this)
        return true;

    if ((other is INamedType) &&
        ((INamedType) other).IsEquivalent(this, flags))
    {
        return true;
    }

    return false;
}

public Boolean IsSubtypeOf(IType other, TypeCheckFlags flags)
{
    if (this.IsEquivalent(other, flags))
        return true;

    if (other is EmptyType)
        return true;

    if ((other is INamedType) &&
        ((INamedType) other).HasSubtype(this, flags))
    {
        return true;
    }

    if ((other is LiftedType) &&
        ((LiftedType) other).HasSubtype(this, flags))
    {
        return true;
    }

    if ((other is ModifiedType) &&
        ((ModifiedType) other).HasSubtype(this, flags))
    {
        return true;
    }

    if (other.IsEquivalent(this.staticObject, flags))
        return true;
    if (other is IUnit)
    {
        foreach (IUnit embeddedUnit in this.embeddedUnits)
            if (embeddedUnit.IsSubtypeOf(other, flags))
                return true;
    }

    return false;
}
```

Listing 24: Implementation: Type Checking Methods in `Unit`

As can be seen, in addition to `INamedType`, `LiftedType` and `ModifiedType` also contain a `HasSubtype` method, which is needed, as shown in this example, for the type checking to be invertible for these cases.

For an explanation of the `TypeCheckingFlags` value, see *Modified Types* below and Section 9.3.1.

### 9.1.3. Named Types

Named types are a special case in that an equivalence can be defined, but is not always visible. More specifically, named type member definitions contain a type value in addition to the supertype that is also defined for declarations and type parameters. However, because the value can change to a subtype through inheritance, this type value is only available in the following cases:

- The access to the named type is internal, that is, from a member in the same object, using either `this` as prefix or no prefix at all.
- The named type is statically determined, that is, is either at the static layer, or at the shared layer in a generic instance with statically determined actual parameters. Additionally, the prefix used for accessing the named type must be statically determined.

In these cases, a type equivalence between the named type and its type value can be asserted. Otherwise, only a subtype relation between the named type and any supertype of its (implicitly or explicitly) specified supertype is available. The implementation of this difference is trivial, as the member access class used, `NamedTypeMemberAccess` in this case, knows the prefix and whether the access is internal, static, or neither of these cases.

Another special case occurs when embedding named types, in that an overriding named type member is seen as equivalent to the overridden member, but only when both are used without prefix. The only case where this equivalence is used is the substitutability check for members. This is required to allow the use named types in member signatures, which would otherwise become invalid on embedding if the used named type member is substituted.

### 9.1.4. Modified Types

Type checking for modified types is done separately for the base type and any type modifiers. One problem, however, is the fact that the real base type need not necessarily

be known. Listing 25 shows an example where the ModifiedType instance has an already modified type as its base type, and a subtyping relationship must be determined.

```
1  class A is
2      token t;
3      type T = A[t];
4
5      method m(p: A[t:2]) do
6          ...
7          l: T[t] = p;
8          ...
9      end method;
10     ...
11 end class;
```

Listing 25: Example: Base Type of Modified Type Unknown

Rather than requiring to be able to access the type value of the named type and using a loop to get to the real base type, the BaseTypeOnly flag of the TypeCheckingFlags enumeration is used. As an example, this is illustrated by means of the IsEquivalent method of the ModifiedType class in Listing 26. The IsSubtypeOf and HasSubtype methods are similar. For information about the token equality check see Section 9.3.1.

## 9.2. Substituted Members

For checking if a member may override another one, the IsSubstitutableBy method defined in the IMember interface is used. Its implementation for the different member classes is straight-forward as the example in Listing 27 shows for the Member and NamedValue classes (the former is needed for the latter).

## 9.3. Distributed Options

Distributed options are a main feature of VooDo. The implementation of the functionality subset (see Section 11.2.2) contained in this work has proven to be less complicated than originally anticipated.

### 9.3.1. Types and Tokens

The IType interface defines a property Tokens, which is an instance of the TokenValues class. This class is used to manage the tokens a type has. Beside constructors for creating instances using the tokens available in a unit, a list of type modifiers, or a list of synchronization constraints, it contains methods for combining instances (for example to apply type modifiers) and for checking if type modifiers are available.

```
public Boolean IsEquivalent(IType other, TypeCheckFlags flags)
{
    if (other == this)
        return true;

    if ((other is INamedType) &&
        ((INamedType) other).IsEquivalent(this, flags))
    {
        return true;
    }

    if (other is ModifiedType)
    {
        if (this.baseType.IsEquivalent(((ModifiedType) other).baseType,
            TypeCheckFlags.BaseTypeOnly) &&
            (T.Flags.IsSet(flags, TypeCheckFlags.BaseTypeOnly) ||
            this.Tokens.Equals(other.Tokens,
            T.Flags.IsSet(flags, TypeCheckFlags.CompareExpectedTokens))))
        {
            return true;
        }
    }
    else if (this.baseType.IsEquivalent(other, TypeCheckFlags.BaseTypeOnly)
        && (T.Flags.IsSet(flags, TypeCheckFlags.BaseTypeOnly) ||
        this.Tokens.Equals(other.Tokens,
        T.Flags.IsSet(flags, TypeCheckFlags.CompareExpectedTokens))))
    {
        return true;
    }

    return false;
}
```

Listing 26: Implementation: `IsEquivalent` Method in `ModifiedType`

`TokenValues` supports what shall be called *expected tokens*, that is, token values to the right of `->` in type modifiers and synchronization constraints. The methods used for checking availability take a boolean parameter indicating whether these expected values should also be checked or not. This is what the `CheckExpectedTokens` value of the `TypeCheckFlags` enumeration is for in the type checking methods (see Section 9.1.4).

The public interface of the class is shown in Figure 11.

### 9.3.2. State Information for Objects

Normally, the type of an object is fixed throughout the program. Because of distributed options, however, this is only true globally, not locally within an expression or routine body. Therefore, the current state (that is, type) of each object for which it has changed must be kept during type checking.

The `States` class is used for this function. It holds the current type of any object required and allows its modification through a number of methods. Additionally, checks for type

```
class Member : OrderedElement, IElement, IContextMember
{
    ...
    public virtual Boolean IsSubstitutableBy(IMember other)
    {
        if ((this.Visibility == MemberVisibility.Public) &&
            (other.Visibility != MemberVisibility.Public))
        {
            return false;
        }

        if ((this.layer == Layer.Static) && (other.layer != Layer.Static))
            return false;
        if ((this.layer == Layer.Shared) && (other.layer == Layer.Instance))
            return false;
        if (!this.Tokens.Implies(other.Tokens, true))
            return false;

        return true;
    }
    ...
}

class NamedValue : DataMember, INamedValue
{
    ...
    public override Boolean IsSubstitutableBy(IMember other)
    {
        if (!base.IsSubstitutableBy(other))
            return false;

        if ((other is INamedValue) &&
            ((INamedValue) other).Type.IsSubtypeOf(this.type,
            TypeCheckFlags.None))
        {
            return true;
        }

        if ((other is IValueFunction) &&
            !((IValueFunction) other).HasParameters &&
            ((IValueFunction) other).ResultType.IsSubtypeOf(this.type,
            TypeCheckFlags.None))
        {
            return true;
        }

        return false;
    }

    ...
}
```

Listing 27: Implementation: `IsSubstitutableBy` Method in `Member` and `NamedValue`

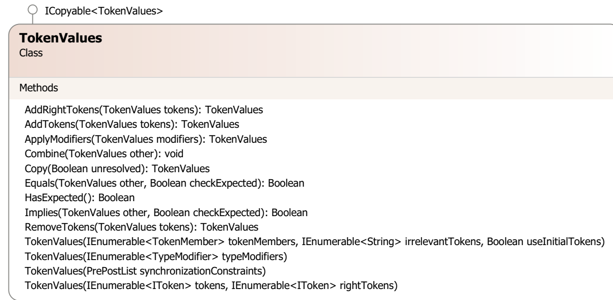



Figure 11.: Public Interface — `TokenValues`

equality and subtyping relations can be performed directly. There are also methods for creating copies of a `States` object and combining multiple instances into one, which is needed for block statements (see below). In addition to the types of objects, the class can also hold a `TokenValues` object for `this`.

Figure 12 contains the public interface for `States`.

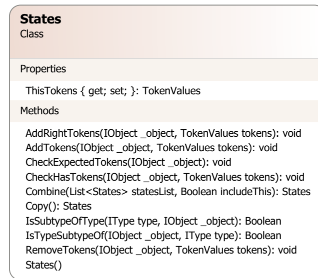



Figure 12.: Public Interface — `States`

It is possible for two or more `IObject` instances to represent the same object (for example, two member accesses to the same named value member). These instances should then be treated the same in some cases, but different in others. For example, when using a variable member as an actual parameter in two different places, the two actual parameter lists cannot be considered the same, because the variable value could have changed in the meantime. However, any state information about the variable would be known to have changed and thus can be kept. The inverse example would be two calls to a value function (with no parameters or the same actual parameters), which can be assumed to return the same object, but no state information needs to be saved, as it cannot have changed.

For this reason, the `IObject` interface defines two methods, `Equals` and `StateEquals` (see Section 7.1.1). The first one is used when object identity must be guaranteed, while the second one is used when it needs be asserted that the objects are the same regarding state information.

### 9.3.3. Routine Bodies

In the `PerformSemanticChecks` method of a routine, a new `States` object is created and a `TokenValues` instance for the synchronization constraints of the routine (if any) is assigned to its `ThisTokens` property. The states object is then passed to the body.

The `PerformSemanticChecks` method of the body calls the ones of the contained statements in order. Following is a list of cases where actions regarding distributed options are taken and what these actions are:

- In local data definitions and assignments, it is checked if the assigned value is a subtype of the type of the named value or variable assigned to. If yes, the tokens of that type are removed from the assigned value. Otherwise, an error is printed.
- Whenever a member is accessed, the following steps take place:
  - The current tokens of the prefix used (or `this` if none) are checked to conform to the synchronization constraints of the member.
  - If the member is a routine that takes parameters, each actual parameter is checked for the tokens required by the formal parameters, and those tokens are removed.
  - The tokens required by the synchronization constraints are removed from the prefix (or `this` if none).
  - If the member is a routine that takes parameters, the token values to the right of `->` in the type modifiers of each formal parameter are added to the corresponding actual parameter. If a type modifier is not active, the tokens specified are assumed to be present at the left and right of `->`.
  - The right tokens in the synchronization constraints are added to the prefix (or `this`). As above, the tokens specified in the synchronization constraints are used if there is no `->`.
- For block statements, the following actions are taken for each block:
  - A copy of the current (outer) `States` instance is created.
  - Any tokens in synchronization constraints for the block are removed from `this`. However, if a token is not available, no error is reported, and the token value is set to zero. This is because block statements dynamically check for the availability of tokens.
  - The `PerformSemanticChecks` method for the inner block is invoked.
  - Tokens to the right of `->` (or tokens if there is no `->`) in the synchronization constraints are added to `this`.

After checking all blocks this way, the newly created `States` objects are combined by taking the minimum value of each token for each object known. If an unconditional block (that is, an `else`-block without a condition) is part of the block statement, the previously current `States` object is not included; otherwise it is.

- Return statements that contain a value are handled just like assignments, with the type against which the assigned value is checked being the routine result type.
- Any return statement implies that synchronization constraints for all enclosing blocks must be checked. Therefore, whenever a return statement is encountered, an event is raised that contains a copy of the current `States` object. This event is handled by block statements and the routine itself, making sure all synchronization constraints are satisfied.

### 9.3.4. Elements

When preforming semantic checks in routine bodies and other parts of members, it is necessary to propagate the checks to contained elements. For example, an `IObject` instance can be a function call containing actual parameters, and these parameters have to be checked using the current `States` object. Therefore, the `IElement` interface (see Section 7.1.1) defines a `PerformSemanticChecks` method that takes a `States` instance as parameter.

# Part III.

# Summary

# 10. Statistical Information

Table 5 contains information about the types, lines of code, and hours of implementation time, organized roughly by the implementation phases described in the Part II (compare Table 3 in Section 3.2). *Types* includes classes, interfaces, structures, enumerations, and delegates, also counting nested types. The number of types for the individual phases sees each type as belonging to only one phase. However, most types are relevant in more than one phase, and many even in the complete compiler frontend, except for syntax analysis.

Note that all numbers except the overall number of types and lines of code are rough estimates.

| Part | Types | Lines of Code | Hours |
|---|---|---|---|
| Syntax Analysis | 5 | 1100 | 20 |
| Data Structures | 45 | 6000 | 40 |
| Name Resolution/Cycle Detection | 60 | 3500 | 200 |
| Inheritance | 25 | 1500 | 30 |
| Semantic Checks | 45 | 1000 | 80 |
| Framework | 2 | 4500 | 30 |
| Overall | 182 | 17600 | 400 |

Table 5.: Statistical Information

Name resolution/cycle detection has proven to be the most complex phase, as can be seen especially from the implementation time needed. The other phases correspond approximately to the original anticipations, with the exception of the semantic checks perhaps, which has proven less time consuming than estimated, as the features that have been implemented were decidedly less problematic than expected.

Overall, we can note that the compiler frontend is quite complex. Especially the more than 180 types, most of which are interdependent in some way, underline that fact.

Tables 6 through 10 give an overview of all types in the implementation and provide some additional information. Not included are the types generated by Coco/R.

| Name | Abstract | Base Class | Interfaces | Nested Types | Instance Members: Fields | Properties | Methods | Constructors | Events | Overall | Static Members: Fields | Properties | Methods | Constructor | Events | Overall |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| AccessOperator | | Operator | 0 | | | 2 | 2 | 1 | | 5 | | | | | | |
| ActualList | | | 3 | | 2 | 4 | 12 | 3 | | 21 | | | | | | |
| AdditionOperator | | Operator | 0 | | | 2 | 2 | 1 | | 5 | | | | | | |
| Assembly | | | 0 | | 5 | 3 | 9 | 1 | | 18 | 1 | | | 1 | | 2 |
| Assignment | | Statement | 1 | | 4 | | 3 | 2 | | 9 | | | | | | |
| AssignmentCondition | | Condition | 3 | | 5 | 5 | 11 | 2 | | 23 | | | | | | |
| BasicExpr | + | SimpleExpr | 3 | | | | | 2 | | 2 | | | | | | |
| BlockStatement | | Statement | 3 | | 3 | 1 | 8 | 2 | | 14 | | | | | | |
| Body | | | 3 | | 3 | 3 | 12 | 2 | 1 | 21 | | | | | | |
| BodyContext | | Context | 0 | | 2 | 1 | 4 | 1 | | 8 | | | | | | |
| Brand | | Unit | 6 | | | 2 | 5 | 1 | | 8 | | | | | | |
| Class | | Unit | 6 | | | 1 | 2 | 1 | | 4 | | | | | | |
| ClassBrand | | Unit | 6 | | | 2 | 3 | 1 | | 6 | | | | | | |
| ComparisonOperator | | Operator | 0 | | | 3 | 3 | 1 | | 7 | | | | | | |
| ComplexExpression | | Expression | 1 | | 3 | 2 | 4 | 2 | | 11 | | | | | | |
| Condition | + | SyntacticElement | 1 | | 1 | | 3 | 2 | | 6 | | | | | | |
| ConditionalBody | | | 1 | | 4 | 1 | 7 | 2 | 1 | 15 | | | | | | |
| CondList | | | 1 | | 4 | 1 | 7 | 2 | | 14 | | | | | | |
| Context | | | 0 | | 5 | 5 | 18 | 3 | | 31 | | | | | | |
| ContextExtension | | | 0 | | 2 | 1 | 3 | 2 | | 8 | | | | | | |
| Creator | | Routine | 7 | 1 | 1 | 4 | 9 | 2 | | 16 | | | | | | |
| →ConcretizedCreator | | ConcretizedRoutine | 7 | | 2 | 2 | 5 | 1 | | 10 | | | | | | |
| DataMember | + | Member | 5 | 2 | 5 | 11 | 19 | 2 | | 37 | | | | | | |
| →ConcretizedDataMember | | ConcretizedMember | 5 | | 5 | 5 | 12 | 1 | | 23 | | | | | | |
| →DataMemberAccess | | MemberAccess | 4 | | 3 | 2 | 6 | 1 | | 12 | | | | | | |
| DataMemberOrFunction | + | Member | 7 | 2 | 3 | 14 | 23 | 2 | | 42 | | | | | | |
| →ConcretizedDataMemberOrFunction | | ConcretizedMember | 7 | | 3 | 10 | 18 | 1 | | 32 | | | | | | |
| →DataMemberOrFunctionAccess | | MemberAccess | 4 | | 3 | 2 | 6 | 1 | | 12 | | | | | | |
| EmbeddedMemberAccumulation | | | 9 | 1 | 14 | 35 | 63 | 1 | 2 | 115 | | | | | | |
| →EmbeddedMemberAccumulationAccess | | | 7 | | 5 | 13 | 25 | 1 | | 44 | | | | | | |
| EmptyType | | | 3 | | | 6 | 12 | 1 | | 19 | 1 | | | 1 | | 2 |
| Errors | + | | 0 | | | | | | | | 3 | 1 | 7 | | | 11 |
| ExponentiationOperator | | Operator | 0 | | | 2 | 2 | 1 | | 5 | | | | | | |
| Expression | + | SyntacticElement | 1 | | 7 | 6 | 16 | 2 | | 31 | | | | | | |
| ExpressionBuilder | | | 0 | | 2 | | 3 | 1 | | 6 | | | | | | |
| Flags | + | | 0 | | | | | | | | | | 1 | | | 1 |
| FormalList | | | 3 | | 3 | 5 | 17 | 2 | | 27 | | | | | | |
| FormalParameter | + | SyntacticElement | 2 | | 3 | 6 | 14 | 2 | | 25 | | | 1 | | | 1 |
| FormalTypeParameter | | FormalParameter | 4 | | | 6 | 14 | 2 | | 22 | | | | | | |
| FormalValueParameter | | FormalParameter | 3 | | | 4 | 11 | 2 | | 17 | | | | | | |
| GlobalContext | | Context | 0 | | | | 4 | 1 | | 5 | | | | | | |
| IdentifierExpr | | BasicExpr | 3 | | 3 | 2 | 4 | 3 | | 12 | | | | | | |
| InnerExpressionExpr | | BasicExpr | 3 | | 1 | 2 | 5 | 2 | | 10 | | | | | | |
| Int32EventArgs | | EventArgs | 0 | | 1 | 1 | 1 | 1 | | 4 | | | | | | |
| InvocationStatement | | Statement | 1 | | 2 | | 3 | 2 | | 7 | | | | | | |
| LayerQualificationExpr | | SimpleExpr | 3 | | 2 | 1 | 3 | 2 | | 8 | | | | | | |
| LiftedType | | | 3 | | 1 | 6 | 15 | 1 | | 23 | | | | | | |
| Literal‘1 | | | 4 | | 1 | 8 | 16 | 1 | | 26 | 1 | | 1 | 1 | | 3 |
| LiteralExpr‘1 | | BasicExpr | 3 | | | 1 | 3 | 2 | | 6 | | | | | | |
| LocalDataDefinition | + | Statement | 4 | | 5 | 5 | 11 | 2 | | 23 | | | | | | |
| LocalNamedValue | | LocalDataDefinition | 4 | | | 1 | 2 | 2 | | 5 | | | | | | |
| LocalVariable | | LocalDataDefinition | 4 | | | 1 | 3 | 2 | | 6 | | | | | | |

Table 6.: Type Information — Classes (1)

| Name | Abstract | Base Class | Interfaces | Nested Types | Instance Members | | | | | | Static Members | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | Fields | Properties | Methods | Constructors | Events | Overall | Fields | Properties | Methods | Constructor | Events | Overall |
| LogicOperator | | Operator | 0 | | | 2 | 2 | 1 | | 5 | | | | | | |
| Member | + | OrderedElement | 3 | 2 | 14 | 17 | 41 | 2 | 2 | 76 | | | | | | |
| →ConcretizedMember | + | OrderedElement | 3 | | 9 | 17 | 35 | 1 | 1 | 63 | | | | | | |
| →MemberAccess | + | OrderedElement | 2 | | 6 | 7 | 11 | 1 | | 25 | | | | | | |
| MemberContext | | Context | 0 | | 1 | 1 | 3 | 2 | | 7 | | | | | | |
| MemberEventArgs | | EventArgs | 0 | | 1 | 1 | 1 | 1 | | 4 | | | | | | |
| MessageExpr | | SimpleExpr | 3 | | 3 | 1 | 3 | 2 | | 9 | | | | | | |
| Method | | Routine | 7 | 1 | | 2 | 5 | 2 | | 9 | | | | | | |
| →ConcretizedMethod | | ConcretizedRoutine | 7 | | 1 | | 1 | 1 | | 3 | | | | | | |
| ModifiedType | | | 3 | | 4 | 7 | 14 | 2 | | 27 | | | 2 | | | 2 |
| Module | | Unit | 6 | | | 1 | 3 | 1 | | 5 | | | | | | |
| ModuleBrand | | Unit | 6 | | | 2 | 4 | 1 | | 7 | | | | | | |
| MultiplicationOperator | | Operator | 0 | | | 2 | 2 | 1 | | 5 | | | | | | |
| NamedTypeMember | | Member | 6 | 2 | 6 | 16 | 24 | 2 | | 48 | | | | | | |
| →ConcretizedNamedTypeMember | | ConcretizedMember | 6 | | 6 | 10 | 15 | 1 | | 32 | | | | | | |
| →NamedTypeMemberAccess | | MemberAccess | 4 | | 6 | 6 | 12 | 1 | | 25 | | | | | | |
| NamedValue | | DataMember | 6 | 1 | | 2 | 5 | 2 | | 9 | | | | | | |
| →ConcretizedNamedValue | | ConcretizedDataMember | 6 | | 1 | | 1 | 1 | | 3 | | | | | | |
| NamedValueOrValueFunction | | DataMemberOrFunction | 9 | 1 | | 1 | 5 | 2 | | 8 | | | 1 | | | 1 |
| →ConcretizedNamedValueOrValueFunction | | ConcretizedDataMemberOrFunction | 9 | | 1 | | 1 | 1 | | 3 | | | | | | |
| Null | | | 4 | | | 7 | 15 | 1 | | 23 | 1 | | | 1 | | 2 |
| Operator | + | | 0 | | 2 | 4 | 4 | 1 | | 11 | | | 1 | | | 1 |
| OrderedElement | + | | 0 | | 2 | | 3 | 1 | | 6 | 1 | | 1 | 1 | | 3 |
| ParameterContext | | Context | 0 | | 3 | 1 | 3 | 2 | | 9 | | | | | | |
| PrefixExpr | | SimpleExpr | 3 | | 2 | 1 | 3 | 2 | | 8 | | | | | | |
| PrePost | | SyntacticElement | 1 | | 3 | 3 | 7 | 2 | | 15 | | | | | | |
| PrePostList | | List‘1 | 7 | | | 1 | 3 | 2 | | 6 | | | | | | |
| Program | + | | 0 | | | | | | | | | | 2 | | | 2 |
| Resources | | | 0 | | | | | 1 | | 1 | 2 | 2 | 3 | | | 7 |
| Resources | | | 0 | | | | | 1 | | 1 | 2 | 2 | 3 | | | 7 |
| Resources | | | 0 | | | | | 1 | | 1 | 2 | 2 | 3 | | | 7 |
| ReturnStatement | | Statement | 1 | | 2 | | 3 | 2 | | 7 | | | | | | |
| Routine | + | Member | 6 | 2 | 9 | 17 | 32 | 2 | | 60 | | | | | | |
| →ConcretizedRoutine | | ConcretizedMember | 6 | | 6 | 11 | 22 | 1 | | 40 | | | | | | |
| →RoutineInvocation | | MemberAccess | 4 | | 4 | 3 | 6 | 1 | | 14 | | | | | | |
| Settings | | ApplicationSettingsBase | 1 | | | | | 1 | | 1 | 2 | 1 | 1 | 1 | | 5 |
| Settings | | ApplicationSettingsBase | 1 | | | | | 1 | | 1 | 2 | 1 | 1 | 1 | | 5 |
| Settings | | ApplicationSettingsBase | 1 | | | | | 1 | | 1 | 2 | 1 | 1 | 1 | | 5 |
| SimpleExpr | + | Expression | 3 | | | 3 | 6 | 2 | | 11 | | | | | | |
| Statement | + | SyntacticElement | 1 | | 3 | 1 | 7 | 2 | 1 | 14 | | | | | | |
| States | | | 0 | 1 | 2 | 1 | 11 | 2 | | 16 | | | | | | |
| →IObjectEqualityComparer | | | 1 | | | | 2 | 1 | | 3 | | | | | | |
| StatesEventArgs | | EventArgs | 0 | | 1 | 1 | 1 | 1 | | 4 | | | | | | |
| SyntacticElement | + | | 0 | | 1 | | | 1 | | 2 | | | | | | |
| Token | | | 0 | | 6 | | | 1 | | 7 | | | | | | |
| TokenCondition | | Condition | 1 | | 1 | 1 | 3 | 2 | | 7 | | | | | | |
| TokenList | | List‘1 | 6 | | | | | 1 | | 1 | | | | | | |
| TokenMember | | Member | 5 | 1 | 5 | 12 | 18 | 2 | | 37 | | | | | | |
| →ConcretizedTokenMember | | ConcretizedMember | 5 | | 1 | 6 | 9 | 1 | | 17 | | | | | | |
| TokenUse | | SyntacticElement | 3 | | 5 | 6 | 9 | 2 | | 22 | | | | | | |
| TokenValues | | | 1 | | 4 | | 15 | 5 | | 24 | | | | | | |
| TypeModifier | | SyntacticElement | 1 | | 4 | 3 | 7 | 2 | | 16 | | | | | | |

Table 7.: Type Information — Classes (2)

| Name | Abstract | Base Class | Interfaces | Nested Types | Instance Members | | | | | | Static Members | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | Fields | Properties | Methods | Constructors | Events | Overall | Fields | Properties | Methods | Constructor | Events | Overall |
| Unit | + | OrderedElement | 6 | 3 | 19 | 33 | 59 | 1 | 1 | 113 | | | 2 | | | 2 |
| →GenericInstance | | OrderedElement | 5 | 2 | 17 | 24 | 41 | 1 | | 83 | | | | | | |
| →GenericInstance+GenericStaticObject | | OrderedElement | 4 | | 1 | 20 | 33 | 1 | | 55 | | | | | | |
| →GenericInstance+SharedObject | | OrderedElement | 5 | | 7 | 20 | 38 | 1 | | 66 | | | | | | |
| →StaticObject | | OrderedElement | 5 | | 7 | 20 | 38 | 1 | | 66 | | | | | | |
| →ThisObject | | | 2 | | 1 | 5 | 8 | 1 | | 15 | | | | | | |
| UnitContext | | Context | 0 | | 1 | | 4 | 2 | | 7 | | | | | | |
| UnitSpec | | SyntacticElement | 2 | | 3 | 5 | 8 | 1 | | 17 | | | | | | |
| ValueCondition | | Condition | 1 | | 2 | 1 | 4 | 2 | | 9 | | | | | | |
| ValueFunction | | Routine | 7 | 1 | | 2 | 6 | 2 | | 10 | | | | | | |
| →ConcretizedValueFunction | | ConcretizedRoutine | 7 | | 1 | | 1 | 1 | | 3 | | | | | | |
| Variable | | DataMember | 6 | 1 | | 3 | 7 | 2 | | 12 | | | | | | |
| →ConcretizedVariable | | ConcretizedDataMember | 6 | | | 1 | 2 | 1 | | 4 | | | | | | |
| VariableFunction | | Routine | 7 | 1 | | 2 | 5 | 2 | | 9 | | | | | | |
| →ConcretizedVariableFunction | | ConcretizedRoutine | 7 | | 1 | | 1 | 1 | | 3 | | | | | | |
| VariableOrVariableFunction | | DataMemberOrFunction | 9 | 1 | | 1 | 5 | 2 | | 8 | | | 1 | | | 1 |
| →ConcretizedVariableOrVariableFunction | | ConcretizedDataMemberOrFunction | 9 | | 1 | | 1 | 1 | | 3 | | | | | | |

Table 8.: Type Information — Classes (3)

| Name | Base Interfaces | Declarations | | | |
|---|---|---|---|---|---|
| | | Properties | Methods | Events | Overall |
| IContextDataMember | 4 | | | | |
| IContextDataMemberOrFunction | 6 | | | | |
| IContextMember | 2 | 3 | 3 | | 6 |
| IContextNamedTypeMember | 4 | | | | |
| IContextRoutine | 4 | | | | |
| IContextTokenMember | 4 | | | | |
| ICopyable‘1 | 0 | | 1 | | 1 |
| ICreator | 3 | | | | |
| IDataMember | 2 | 4 | 8 | | 12 |
| IDataMemberOrFunction | 4 | | | | |
| IElement | 0 | 4 | 5 | | 9 |
| IInvocation | 1 | | | | |
| ILayeredObject | 1 | 3 | 3 | | 6 |
| ILValue | 2 | | 1 | | 1 |
| IMember | 1 | 10 | 22 | 1 | 33 |
| IMemberAccess | 1 | 2 | 3 | | 5 |
| IMethod | 3 | | | | |
| INamedType | 2 | 1 | 2 | | 3 |
| INamedTypeMember | 2 | 6 | 6 | | 12 |
| INamedValue | 3 | | | | |
| IObject | 1 | 1 | 3 | | 4 |
| IParameterizable | 1 | 3 | 4 | | 7 |
| IRoutine | 2 | 6 | 8 | | 14 |
| IStaticValue | 0 | | 1 | | 1 |
| IToken | 1 | 2 | 2 | | 4 |
| ITokenMember | 2 | 3 | 4 | | 7 |
| IType | 1 | 2 | 6 | | 8 |
| IUnit | 2 | 13 | 18 | | 31 |
| IValueFunction | 3 | | | | |
| IVariable | 3 | | | | |
| IVariableFunction | 3 | | | | |

Table 9.: Type Information — Interfaces

| Name | Values |
|---|---|
| BlockStatementType | 2 |
| DataAccessKind | 4 |
| ElementKinds | 10 |
| Layer | 3 |
| LayerQualification | 3 |
| MemberKind | 4 |
| MemberType | 5 |
| MemberVisibility | 3 |
| NameResolutionFlags | 6 |
| NameResolutionState | 3 |
| OperatorAssociativity | 3 |
| ReturnCode | 4 |
| SpecialIdentifier | 4 |
| SynchronizationConstraints | 3 |
| TokenValuesSource | 3 |
| TypeCheckFlags | 3 |
| TypeFlags | 3 |
| TypeModifierKind | 3 |
| UnitKind | 5 |
| UnitVisibility | 2 |

Table 10.: Type Information — Enumerations

# 11. Future Work

This chapter describes the necessary work to get a full compiler for VooDo Kern.

## 11.1. Changes to the Language

The VooDo language is not completely finalized, so there may be changes to the language specification that would have to be incorporated into the compiler. On the other hand, these modifications are expected to be in the details and should not require excessive revisions of the existing codebase.

## 11.2. Implementation of Remaining Features

This section lists the features not included in the current implementation.

### 11.2.1. Basic Data Types

To fully support the various literals, base types, that is, `Integer`, `Float`, and `String`, have to be implemented. The same is true for a boolean data type to check conditions in block statements for having the correct type.

### 11.2.2. Distributed Options

The following functionality is needed for complete support of distributed options:

- Dependent options and alternative type modifiers.
- Alternative synchronization constraints.
- Special handling of equality comparisons in block statements.

All of these features are anticipated to be implementable in a straight-forward way, without the need to revise large amounts of the existing work.

### 11.2.3. Miscellaneous Features

#### While-Loops

There is currently no support for loops of any kind. `while`-loops as specified in the language would thus have to be implemented.

#### Semantic Checks

A number of non-critical semantic checks are expected to be easy to implement:

- Checks on whether data members are correctly initialized or not.
- The uniqueness-constraint on write accesses and state-dependent read accesses, which would provide a way of ensuring atomic accesses to data members.
- Member access and side effect safety of functions (see section 2.3.3).
- Checks for *level-safe variable functions* in the conditions of `when`-blocks.

## 11.3. Code Generator

The most important, but also most extensive part left to be implemented for a complete compiler is a code generation module, which could either directly output binary code, or code in a different programming language that can then be translated using the appropriate compiler.

Additional features in the frontend depending upon and supporting code generation:

- The `import` statement and the repository management needed.
- Cross-assembly visibility checks.

# 12. Conclusion

VooDo is an experimental object-oriented programming language that includes a large number of features, especially concerning the type system. While most of those are known from other languages, VooDo is special in that it combines all those features in one language. Additionally, there are new concepts, most prominently *distributed options*, which are used for statically checking synchronization. As the complete language is fairly complex, a scaled-down version called VooDo Kern was used as basis for this work.

It is to be anticipated that a compiler for the VooDo language is fairly complex, most importantly as a result of the needed type checking mechanisms.

The focus of this work has been the implementation of a compiler frontend for VooDo Kern. Additionally, several small changes to the language have been made. The frontend has been implemented using the C# programming language and Coco/R parser generator.

As has been shown in Part II, developing a compiler for VooDo Kern is a complex task, as anticipated. The main problems, however, are not to be found in the implementation of the semantic checks. Rather, name resolution and cycle detection have turned out to be the most difficult part. Together, name resolution, cycle detection, and inheritance, while anticipated to make up around twenty percent of the implementation work, have accounted for more than half of it. Semantic checks are an important part of the implementation and should not be underestimated, especially if all features are to be supported. Still, the actual complexity of this part is significantly less than expected.

As the current implementation has been developed with an emphasis on flexibility and the use of clean object-oriented methods, it should not be too difficult to add the additional functionality described in Chapter 11. Hopefully, the compiler frontend developed in the course of this work can provide a basis for a complete VooDo compiler, or at least help in its realization.

# Part IV.

# Appendix

# A. List of Figures, Tables, and Listings

## Figures


1. Class Diagram — Units . . . 41
2. Class Diagram — Nested Classes of `Unit` . . . 42
3. Class Diagram — Members . . . 43
4. Class Diagram — Formal Parameter Lists . . . 45
5. Class Diagram — Routine Bodies and Statements . . . 46
6. Class Diagram — Expressions . . . 47
7. Class Diagram — Element Interfaces . . . 50
8. Class Diagram — Context . . . 53
9. Example — Context Hierarchy . . . 54
10. Example — Creator Context Hierarchy . . . 56
11. Public Interface — `TokenValues` . . . 75
12. Public Interface — `States` . . . 75


## Tables


1. Units Summary . . . 13
2. Units as Objects . . . 14
3. Anticipated Proportions of Stages . . . 30
4. Unit and Layer Dependencies . . . 61
5. Statistical Information . . . 79
6. Type Information — Classes (1) . . . 80
7. Type Information — Classes (2) . . . 81
8. Type Information — Classes (3) . . . 82
9. Type Information — Interfaces . . . 83
10. Type Information — Enumerations . . . 83

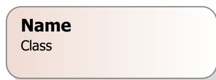

## Listings


1. VooDo Kern: Generic Stack . . . 10
2. VooDo Kern: Layers . . . 11
3. VooDo Kern: Brand . . . 12

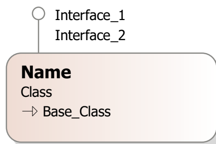
4. VooDo Kern: Module . . . 13
5. VooDo Kern: Named Types . . . 16
6. VooDo Kern: Tokens . . . 17
7. VooDo Kern: Negative Token Value . . . 18
8. VooDo Kern: `if`-Statement . . . 19
9. VooDo Kern: Invalid Cycle . . . 20
10. VooDo Kern: Options . . . 25
11. VooDo Kern: Synchronization Constraints . . . 26
12. VooDo Kern: Active Type Modifiers . . . 26
13. VooDo Kern: Synchronization Constraints in Conditional Statement . . . 27
14. Example: Covariant Parameters (Invalid) . . . 32
15. Example: Use of Covariant Parameters (Invalid) . . . 33
16. Example: Named Types and Covariant Parameters . . . 33
17. Example: Use of Named Types and Covariant Parameters . . . 33
18. Example: Non-Trivial Invalid Cycle . . . 35
19. Example: Lifted Type . . . 36
20. Implementation: Assignment Resolver . . . 40
21. Example: Member Access . . . 55
22. Implementation: `OrderedElement` Class (1) . . . 58
23. Implementation: `OrderedElement` Class (2) . . . 59
24. Implementation: Type Checking Methods in `Unit` . . . 70
25. Example: Base Type of Modified Type Unknown . . . 72
26. Implementation: `IsEquivalent` Method in `ModifiedType` . . . 73
27. Implementation: `IsSubstitutableBy` Method in `Member` and `NamedValue` 74

# B. Explanation of Figures

The following elements are used in the class diagrams contained in this work:

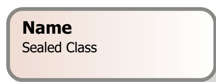



A class.

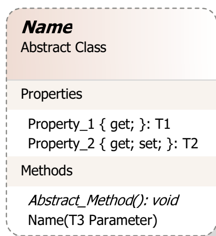



A class that is a sub-class of another and implements two interfaces.

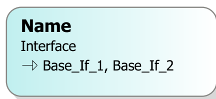
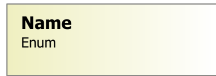
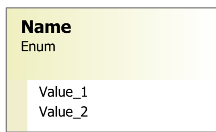
An abstract class.




A sealed class.




An abstract class containing a read-only property, a read-write property, an abstract method without parameters and no return type, and a constructor with one parameter.

| | |
|---|---|
|  | An interface with two base-interfaces. |
|  | An enumeration. |
|  | An enumeration with values shown. |
| | An association for a field that always contains exactly one instance of the type pointed at. |
| | An association for a property. |
| | An association for an abstract property. |
| | An association for a field that may be empty/null. |
| | A collection association for a field that always contains at least one element of the type pointed at. |
| | A collection association for a property that may be empty, that is, be null or contain no elements. |
| | A subtyping relationship. The class or interface pointed at is a base-class or base-interface of the one at the beginning. |

For object hierarchies, these elements are used:

| | |
|---|---|
| **Name: Type**<br>Member_1<br>Member_2 | An object. |
| | A reference to an object. The member at the beginning (line) contains a reference to the object at the end (block). |

# Bibliography


[1] Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman: *Compilers.* Addison Wesley, January 1985, ISBN 0-20110-194-7.

[2] Bracha, Gilad, Martin Odersky, David Stoutamire, and Philip Wadler: *Making the future safe for the past: Adding genericity to the java programming language.* In *13th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '98)*, Vancouver, BC, Canada, October 1998. `http://www.cis.unisa.edu.au/~pizza/gj/Documents/#gj-oopsla`.

[3] Canning, Peter, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell: *F-bounded polymorphism for object-oriented programming.* In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 273–280, New York, NY, USA, 1989. ACM Press, ISBN 0-89791-328-0. `http://webcourse.cs.technion.ac.il/236801/Winter2004-2005/ho/WCFiles/p273-canning.pdf`.

[4] Chambers, Craig: *Object-oriented multi-methods in cecil.* In *ECOOP '92 European Conference on Object-Oriented Programming, Proceedings*, number 615 in *LNCS*, Utrecht, The Netherlands, June/July 1992. Springer-Verlag. `http://www-plan.cs.colorado.edu/diwan/class-papers/cecil-oo-mm.pdf`.

[5] Dijkstra, Edsger W.: *The structure of the "THE"-multiprogramming system.* In *SOSP '67: Proceedings of the first ACM symposium on Operating System Principles*, pages 10.1–10.6, New York, NY, USA, 1967. ACM Press.

[6] Dijkstra, Edsger W.: *Cooperating sequential processes.* In *The Origin of Concurrent Programming: From Semaphores to Remote Procedure Calls*, pages 65–138, New York, NY, USA, 2002. Springer-Verlag New York, Inc., ISBN 0-387-95401-5.

[7] Dijkstra, Edsger W.: *Hierarchical ordering of sequential processes.* In *The Origin of Concurrent Programming: From Semaphores to Remote Procedure Calls*, pages 198–227, New York, NY, USA, 2002. Springer-Verlag New York, Inc., ISBN 0-387-95401-5.

[8] Flanagan, David: *Java in a Nutshell.* O'Reilly & Associates, March 2005, ISBN 0-59600-773-6.

[9] Gosling, James, Bill Joy, Guy Steele, and Gilad Bracha: *The Java Language Specification, Second Edition.* Addision-Wesley, June 2000, ISBN 0-201-31008-2. `http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html`.

[10] Meyer, Bertrand: *Eiffel: The Language.* Prentice Hall, 2nd edition, 1992.

[11] Meyer, Bertrand: *Object-Oriented Software Construction.* Sams, 2nd edition, May 1997, ISBN 0-136-29155-4.

[12] Mössenböck, Hanspeter: *Coco/R for various languages - Online Documentation.* `http://www.ssw.uni-linz.ac.at/Research/Projects/Coco/`.

[13] Mössenböck, Hanspeter: *A generator for production quality compilers.* In *CC '90: Proceedings of the third international workshop on Compiler compilers*, pages 42–55, New York, NY, USA, 1991. Springer-Verlag New York, Inc., ISBN 0-387-53669-8.

[14] Puntigam, Franz: *Concurrent Object-Oriented Programming with Process Types.* Der Andere Verlag, Osnabrück, Germany, 2000.

[15] Puntigam, Franz: *Typsysteme.* Skriptum zur gleichnamigen Vorlesung, Technische Universität Wien, Institut für Computersprachen, Arbeitsgruppe Programmiersprachen und Übersetzerbau, Argentinierstraße 8, A-1040 Wien, Österreich, 2001. `http://www.complang.tuwien.ac.at/franz/typsysteme.html`.

[16] Puntigam, Franz: *Synchronization with type variables.* In *Workshop on Object-oriented Language Engineering for the Post-Java Era*, Darmstadt, Germany, July 2003. `http://www.complang.tuwien.ac.at/franz/papers/Punt03b.ps.gz`.

[17] Puntigam, Franz: *VOO-DO Language Specification (Kern–Draft).* Technical report, Vienna University of Technology, Institute of Computer Languages, Compilers and Languages Group, Argentinierstraße 8, A-1040 Vienna, Austria, 2005.

[18] Quintana, Julio David: *Getting to know Mono.* Linux Journal, 2003(111):4, July 2003, ISSN 1075-3583.

[19] Reynolds, John C.: *Types, abstraction, and parametric polymorphism.* In *Information Processing 83*, pages 513–523, Amsterdam, The Netherlands, 1983. Elsevier Science Publishers B.V. (North-Holland).

[20] Wirth, Niklaus E.: *Compiler Construction.* Addison Wesley, July 1996, ISBN 0-20140-353-6.

[21] ECMA International, Rue du Rhóne 114, CH-1204 Geneva, Switzerland: *C# Language Specification*, 2nd edition, December 2002. `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf`, Standard ECMA-334 (≡ ISO/IEC 23270:2003).

[22] International Organization for Standardization, 1, Rue de Varembé, Case postale 56, CH-1211 Geneva 20, Switzerland: *ISO/IEC 14882:2003: Programming languages — C++*, October 2003.

[23] Microsoft Corporation, One Microsoft Way, Redmond, WA 98052-6399, USA: *C# Version 2.0 Specification*, May 2004. `http://download.microsoft.com/download/8/1/6/81682478-4018-48fe-9e5e-f87a44af3db9/SpecificationVer2.doc`.