Using Oberon’s Active Objects for Language Interoperability and Compilation

Patrik Reali

Diss. ETH No. 15022

Using Oberon's Active Objects for Language Interoperability and Compilation

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH
(ETH ZÜRICH)

for the degree of
Doctor of Technical Sciences

presented by
Patrik René Celeste Reali
Dipl. Informatik-Ing. ETH
born July 11, 1971
citizen of Isorno, Switzerland

accepted on the recommendation of
Prof. Dr. Jürg Gutknecht, examiner
Prof. Dr. Bertrand Meyer, co-examiner

2003

Programming in different languages is like composing pieces in different keys, particularly if you work at the keyboard. If you have learned or written pieces in many keys, each key will have its own special emotional aura. Also, certain kinds of figurations "lie in the hand" in one key but are awkward in another. So you are channeled by your choice of key. In some ways, even enharmonic keys, such as C-sharp and D-flat, are quite distinct in feeling. This shows how a notational system can play a significant role in shaping the final product.

— D. Hofstadter; G ̈odel, Escher, Bach: an eternal golden braid

Acknowledgments

A thesis is supposed to be the work of one person. Nevertheless, this work would have not be possible without the collaboration, support, feedback, and encouragement from many persons.

First of all, I would like to thank Prof. J. Gutknecht for giving me the chance to work in his group, and for his liberal supervision of the thesis. Prof. B. Meyer kindly accepted to be co-examiner and provided very competent feedbacks.

I wish thank all the past and present members of the language and run-time research group, which created such an interesting and challenging environment, in particular Pieter Muller, who shared the same office with me for more than four years, and for introducing me to the secrets of system construction and of south african culture. Many thanks go also to A. Disteli, B. Egger, A. Fischer, T. Frey, P. Kramer, E. Oswald, K. Skoupy, E. Zeller, and E. Zueff.

The Institute for Computer Systems at ETH provided a very lively frame to any kind of discussions and activities, ranging from technical topics to wine tasting. Life at ETH would have been dull without H. Domjan, R. Karrer, C. Kurmann, C. von Praun, E. Ruiz, H. Sommer, M. Taufer, and all the past and present members of the institute. Never miss a coffee break or an Assistentenabend for any reason!

The student assignments of G. Banfi, R. Laich, and R. Morelli has been relevant for this thesis, and greatly helped me.

Some external collaborations need also to be mentionned: P. Januschke for the interesting and productive discussions about the design and implementation of OberonX; M. van Hacken for fruitful collaboration in project Hostess; A. Freed and C. Langreiter for the constant encouragement.

I'm deeply indebted to S. Brawer and A. Fischer for reading early drafts of this thesis and pointing out many errors or inconsistencies.

Last but not least, I would like to thank my parents Annie and Carlo, and my wife Andrea, for their support and the great display of patience (or forced unto them).

Table of Contents

Abstract

The modern run-time environments provide a high-level object-oriented programming model, reflection capabilities, and a large amount of libraries. These platforms are particularly suitable to host programs written in many different languages. To let all languages reuse the common libraries, the platform has a reflection mechanisms to access the metadata information, which can be used by the compilers to provide language interoperability in a seamless way.

Writing a compiler for such platforms becomes an exercise in programming model mapping. Most of the languages rely on closely related flavors of the object-oriented model, but the mappings from one model to another are not always trivial.

This thesis investigates the implementation of a language interoperability platform based on an active object-based object-oriented model.

First, it analyzes the relations between the type systems of the platform and its subset used for interoperability, and the languages for the platform. Each language type system must be mappable to the platform type system, and the interoperability type system must be bidirectionally mappable to every language type system. Many languages implement a different flavor of the object-oriented model. A few simple mapping rules to transform each flavor onto another one are presented.

Second, the Aos kernel implementing Active Oberon's active object model is introduced as language interoperability platform. The language Active Oberon is presented as natural notation for the active object model. The Paco scope-parallel Active Oberon compiler is a case study that shows a non-trivial application relying on active objects. It parses each scope concurrently and solves forward references in the code with parser synchronization, instead of requiring a second parsing pass.

Third, the Jaos Java Virtual Machine for Aos is presented. The jit-compiler maps the Java model to the active object model, and compiles the java byte-code using the data-layout and facilities provided by the kernel; the jit-compiler and the Java reflection libraries use the same metadata repository as the Active Oberon compiler: this allows both languages to interoperate seamlessly. Plugins for the metadata repository and for the kernel's dynamic loader allow to use multiple metadata persistency formats and multiple object-file formats at the same time.

Kurzfassung

Die modernen Laufzeitumgebungen definieren ein objektorientiertes Programmiermodel, Reflektionsfunktionalität und umfangreiche Funktionsbibliotheken. Diese Plattformen eignen sich besonders gut um Programme zu unterstützen, die in verschiedenen Programmiersprachen entwickelt worden sind. Um den Zugriff auf die gemeinsamen Funktionsbibliotheken zu ermöglichen, bietet die Plattform durch Reflektion Zugriff auf Metadaten an, damit die Compiler eine automatisierte und transparente Sprachinteroperabilität gewährleisten können.

Ein Compiler für solche Platformen wird zum Programmiermodelübersetzer. Obwohl die meisten Programmiersprachen auf Varianten des objektorientieren Modells basieren, sind die Abbildungen von Modelvariante zu Modelvariante sind nicht immer trivial, und oft mit semantischen Verlusten behaftet.

Diese These untersucht die Implementation einer Sprachinteroperabilitätsplattform basierend auf aktiven Objekten.

Zuerst werden die Beziehungen zwischen dem Programmiersprachtypsystems, dem Plattformstypsystemen, und dem gemeinsamen Interoperabilitätstypsystem analysiert. Das Programmiersprachtypsystem soll vollständig auf das Plattformtypsystem abbildbar sein. Das Interoperabilitätstypsystem soll dazu bidirektional auf eine Untermenge des Programmiersprachtypsystem abbildbar sein. Es wird vorgestellt, wie die verschiedenen Varianten des objektorientierten Models aufeinander abgebildet werden können.

Zweitens wird der Aos-Kernel als Sprachinteroperabilitätsplattform vorgestellt. Das auf aktiven Objekten basierte Modell und die Active Oberon Programmiersprache werden eingeführt. Der parallele Paco Compiler für Active Oberon wird als Fallstudie einer nicht trivialen Anwendung aktiver Objekte gezeigt. Paco übersetzt jeden Sichtbarkeitbereich im Quellecode nebenläufig, und behandelt Symbolreferenzen, die vor der Symboldefinition auftreten, als Synchronisationsproblem, statt einen zweiten Durchlauf über den Quellcode zu verlangen.

Drittens wird die Jaos Java Virtual Machine für Aos vorgestellt. Der just-in-time Compiler bildet das Java-Model auf Oberons aktive Objekte ab. Die Abbildung verwendet so weit wie möglich die Datenauslegung und Funktionalität des Kernels; der jit-Compiler und die Java Reflektionsklassen bauen auf den selben Metadaten wie der Paco Compiler. Damit können beide Sprachen transparent interoperieren. Die Plugin-basierte Architektur des Metadaten-Systems und des dynamischen Laders erlauben die gleichzeitige Verwendung von unterschiedlichen Metadaten- und Objektfileformate.

1 Introduction

Eine Verbesserung erfindet nur der,
welcher zu fühlen weiss:
dies ist nicht gut

— Friedrich Nietzsche

Language interoperability has been in use since the second programming language appeared, but in such a small extent, that awareness about it came only at a later stage. Every time two software components exchange information, they have to agree on a common protocol to be able to share the information: this is interoperability at work.

The Internet is probably the best-known example of interoperability among systems. It provides the infrastructure for the exchange of information across heterogeneous machines connected through a network. A database system is also an example of interoperability: the queries are formulated in an application and submitted to the database though a well-defined software interface, and both systems can be implemented using different programming languages. More recently, whole platforms dedicated to language interoperability have appeared: Microsoft's .NET is the most refined example up-to-now, and it is designed to make programs written in more than two dozen imperative and functional languages interoperate in a completely automated way. In practice, interoperability requires clear and accepted standards and protocols for information interchange.

Software components are the main reason for the emergence of interoperability, and language interoperability in particular. Components solve a specific programming problem and make the results available to the other components through an interface. Components are usually implemented using the language best suited to solve the problem[1], and implicitly pose the problem of how to make software written in different languages communicate.

[1]or the only one known to the software developer

1.1 Motivation

Interoperability is the ability of two or more software components to cooperate despite differences in language, interface, and execution platform [Weg96]. Language interoperability focuses how to cope with the differences in the programming languages across software components.

Language interoperability has many obvious advantages, which reflects its application fields. First, it allows component developers to choose the most appropriate language to solve the problem, often dramatically reducing software complexity. Second, it allows and simplifies the reuse of existing software written in a different language. Third, language interoperability is also beneficial for system and compiler constructors, because it requires and offers a common infrastructure which has not to be replicated in every language's compiler: for this it offers an higher run-time abstraction, reducing the gap between language and platform to be filled by the compiler.

1.1.1 The Problem Perspective

From the problem perspective, the choice of an appropriate language can dramatically reduce a software component's complexity, which is often the limiting factor in software development. A programming language is a tool for building software: it provides primitive concepts to reason about problems. The closer the language or library components are to the problem, the simpler the program gets. Complexity is then handled and hidden by the compiler converting the program to the underlying run-time environment. The complexity of a program is approximately proportional to its size, as it roughly corresponds to the number of state changing operations used in the program[2]; thus the perceived program complexity is the difference between the problem complexity and the language abstraction level. It can also happen that the language model is too abstract for a problem, in which case the language is not suitable to implement the problem. This may be restricted to a limited part of the language. In other cases, the language abstraction may be too low, in which case the implementation can become very complex, because most of the implementation details are exposed to the program. The first forty years of computing saw the creation of hundreds of programming languages [Wex81, BG96], specialized towards restricted application domains. These languages simplify and speed up the software development, by reducing the gap between the problem to be solved and the language. On the other hand, many generic purpose languages like Algol 68, Ada, PL1, Clipper, and C++ were also created, but more often than not, these languages tremendously suffered under their own complexity, as they tried to address every problem with an appropriate abstraction, or offered only very simple primitives that were not expressive enough for complex problems[3].

[2]there are more refi ned software complexity measures like Halstead's Elements of Software Science [Hal77] or McCabe's cyclomatic number [McC76]; this thesis follows Halstead's considerations and spirit, but for this discussion the approximation presented above should be enough

[3]Let's make an example: counting the number of unique entries in a list. In UNIX this is a single shell line: `cat entries | sort | uniq | wc -l`. The same program in Perl would take a dozen lines, in any assembly a few pages. Note that the UNIX solution is truly a manual composition of existing software components

1.1.2 The Component Perspective

The investment in developing a software component can be very high and the risk involved cannot be ignored[4]. Reuse of software components bears many advantages, in particular the availability of software solutions for a well-defined problem that cost only a small fraction of the price to develop them, because their price is shared among all software projects (re-)using them. A fundamental question with components is how to communicate with the rest of the world, by relying on a standard for interoperability. In fact, most of the component standards have to define how to represent the data they use to communicate and of to define their interfaces in a language independent way. Component standards like COM, CORBA, JNI and RMI define a programming model, the semantic thereof, and a language to define their interfaces. The main achievement of these standards was to open the borders across different software systems, and allow interoperability among them. However, the border remains, and crossing it is more like crossing a state border than a city border. JNI, as an example, requires the developer to convert the interface by using some tools, to access all fields using a method, and to explicitly take care of the object pinning to avoid their collection. This solution lacks the automatic propagation of the component's interface. CORBA also suffers from the same problem: in the GNOME project, where components can be accessed through using various programming languages, the time gap between a component release and the release of the wrappers for accessing it from other languages can take up to eight months [dI02]. Microsoft's .NET platform improves the situation by adding the interface definitions[5] to the components and forcing all the language to define their interfaces using the common model. In this way, every component can be understood by all languages that understand the common model and it becomes readily available without need to create interface conversions. The availability of well-integrated simple to use solutions for creating and using software components will play an important role in enabling a real component-oriented development and market therefore.

1.1.3 The System and Compiler Perspective

Providing support for components and interoperability requires creating software systems to provide this support. In particular, these systems must allow be created around the component model which becomes the vehicle of all information in the system. These systems require reflection capabilities to be able to inspect but also design the components. Java institutionalized the use of reflection for inspection purposes, in .NET the reflection API supports metadata and code emission, Aos allows inspection and is intermediate-language agnostic by allowing multiple loaders (possibly including a jitter). The common platform is a huge gain for language and compiler developers, because it provides much functionality, which they would otherwise have to design themselves. This is quite obvious when comparing the Sable and the Jaos JVMs: in Sable most of the effort was spent creating a complete system, like the data-layout design, whereas Jaos main efforts where in the mapping of Java's object model to Active Oberon's object model and the implementation

[4]many software projects fail or are delivered over budget. See [Gla99] for an instructive anecdote collection

[5]further called the metadata

of the just-in-time compiler. In the last decades, language developers often faced the dilemma of either implementing a whole compiler and system or to provide a language to C translator (choice that most of them took, for obvious reasons).

The transition from to a system providing support for language interoperability is less difficult than it seems. In practice, this requires to move some parts of the compiler, in particular the symbol table and possibly the back-end, to the system to be available for all compilers.

Thus, the availability of software platform exposing a higher-level abstraction and metadata services is a tremendous simplification for language and compiler designers, which are freed from many details and become model mappers instead of bit fiddlers.

1.2 This Thesis

This thesis relates my investigation on language interoperability, documents the Active Oberon's language and its Active Objects-based model, and present the Active Oberon Parallel Compiler ”Paco”and the Java on Aos ”Jaos”JVM case studies.

This thesis investigates language interoperability among software components having an object oriented programming model. We restrict the languages considered to imperative programming languages implementing an object-oriented model, because are based on the same concepts and can thus be compared. The Active Object model of Active Oberon is used as a reference.

The thesis shows that the various flavours of the object-oriented models are in fact equivalent. It is possible to translate one model into another one and still convey the same information. The common ideas of object orientation, type compatibility across subtypes, and the object facettes (inheritance of specifications) are present in all flavours.

The thesis also presents the Active Object Model used in the Active Oberon Language, and shows its soundness and usefulness in the construction of non-trivial software like a whole kernel, an operating system and compiler.

The idea of realizing interoperability through the sharing of metadata information and run-time structures, and the analysis of the conversions among the various object model flavours are quite simple and intuitive, whereas the realization of those ideas requires a fairly large amount of software to concretely show an example[6].

This thesis makes a few contributions:

- the various flavours of the object-oriented model used in a few well-known languages are classified and the conversions among them are detailed, including the semantic loss incurred during the transformations

- the Active Object Model used in Active Oberon is presented; the language report for the language itself is included, with many examples of concurrent structures implemented in

[6]fairly large for one person, but not for today's standards, where the ”lines of code”(LOC) metric for software products is being replaced by the number of CD-ROMs required to install it

Active Oberon; a non-trivial example, the Active Oberon compiler itself implemented in Active Oberon is also presented

- Active Oberon's Parallel Compiler "Paco"is presented. Paco is a scope-concurrent compiler. In particular, concurrency is used to parse forward references in a single pass. A testing framework for programming languages named "Hostess"is also presented
- the Java on Aos "Jaos"JVM is presented. Jaos is a JVM for the Aos system. In Jaos, the reflection API is implemented to access the common metadata repository and thus provide interoperability with Active Oberon. The incompatibilities between Active Oberon and Java are presented
- the Aos Kernel is shown to be complete enough for supporting systems other than Oberon. The plugin mechanisms of Aos can be easily used to allow multiple loaders and thus multiple intermediate languages on the same platform

1.3 Overview

The thesis consists of two parts. The first part of the thesis investigates language interoperability. The second part documents the case study used to support our assertions and collect experience.

First Part

Chapter 2 introduces a conceptual framework based on Tanenbaum's layered system and Szyperski's component description, and defines the terminology used throughout the rest of the thesis.

Chapter 3 presents language interoperability as a type mapping problem; the constraints of such mapping are listed. The object-oriented models of a few languages are classified and mappings among models, including the semantic losses incurred, are shown.

Second Part

Chapter 6 describes the Aos platform under the interoperability perspective. This chapter presents the platform's model, and the conventions used, in particular the application binary interface (ABI), which are used by the Active Oberon and Java compilers to implement the language features following a common scheme.

Chapter 4 presents the requirements on a object oriented language providing concurrency, shows the Active Object model and includes the Active Oberon Language report, a natural and high level notation for describing and implementing active object on the Aos system.

Chapter 5 describes the Paco compiler used on Aos; the compiler's symbol table is used a common metadata repository to support language interoperability; furthermore, Paco's scope-concurrent parser is documented, and used as an example of a possible and elegant use of active objects. A framework for language regression testing called "Hostess"is also presented.

Chapter 7 describes the Jaos JVM and the details of its implementation; the chapter includes a list of the factors limiting language interoperability between Oberon and Java.

Chapter 8 draws the conclusions of this thesis, summarizes the achievements reached during this work, and proposes a few future directions of research.

Appendices

Appendix A.1 presents some examples of the Active Oberon Language showing the use of concurrency in the implementation of a fe well-known algorithms and structures.

Appendix B makes a list of points that could be improved in the Oberon and Active Oberon languages from the compiler constructor perspective.

2 Conceptual Framework

This chapter introduces the conceptual framework used throughout the thesis. In Section 2.1, an extension of Tanenbaum's layered system model is presented; Section 2.2 explains interoperability problems bound with components; SectionAos Approach introduces the Aos system as language interoperability platform; Section 2.4 classifies the existing strategies and platforms for language interoperability using the model and terminology used in the chapter.

2.1 Layered system

In this section, a model to describe and classify computer and software systems is introduced; this model structures a system into a stack of layers with growing abstraction level. Each layer consists of machines with an own language and programming model.

The idea of a layered system was introduced by Tanenbaum [Tan98]; we extend his classification by adding a few more notions to the conceptual framework, which are then helpful for a better understanding of systems for language interoperability. Figure 2.1 shows a layered system.

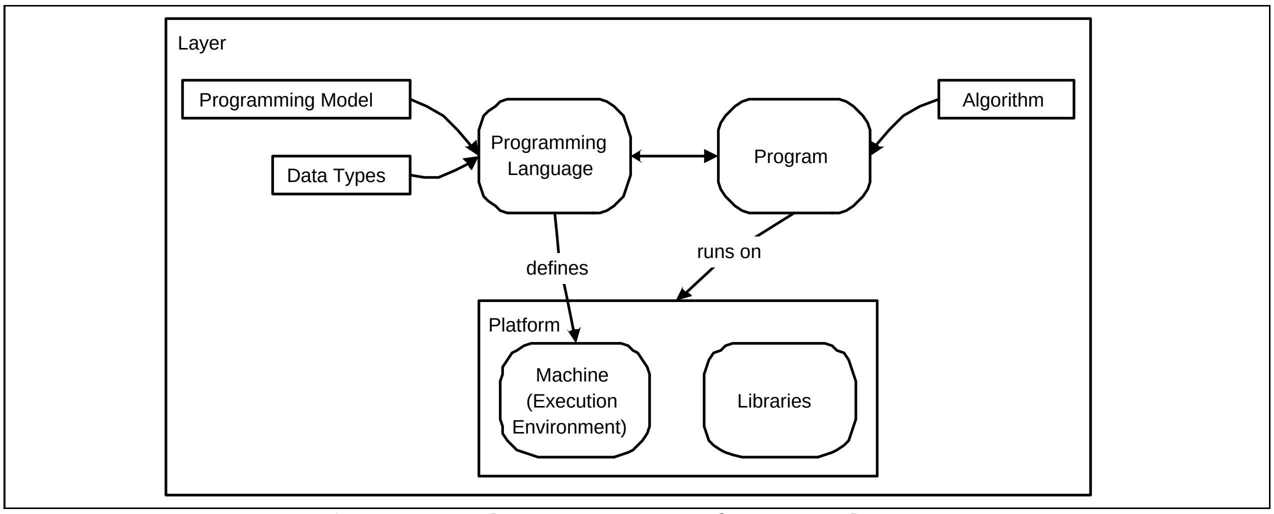


Figure 2.1: The Components of a Layered System

2.1.1 Program and Programming Language

A *program* is the formal description of how a task has to be performed. It defines a set of *data* which have to be manipulated by a sequence of *instructions*.

A *programming language* is a formal notation for expressing programs. It consists of a model for expressing information by using data primitives with well-defined semantics, and a set of instructions to manipulate the data.

Programming languages can range from machine language—extremely flexible but not very expressive—to modern programming languages defining highly expressive abstractions. The most important characteristics of a language are the programming model it uses and the data types it handles.

The *programming model* defines the semantic and organization of the program and data, and how the computation is performed. Possible models are imperative, object-oriented, functional, and logical.

The *data type*[1] provides a set of primitives type that can be used in a program, define the meaning of the data, its location, and compatibility to other data; these characteristics differ in the level of abstraction and complexity. Some type systems employ composition to create new user defined types (like arrays and structures).

The programming model and type system reflect the application domain of the language.

Machine language has an imperative programming model and a type system defining registers[2] and memory locations that allow any kind of data.

High-level languages have a different and more abstract programming model; location of the data can be procedure local or system-wide (the exact address is hidden by the abstraction), each data has a precisely defined type which limits the operations on it. Some languages provide a wide range of data kinds, while other are specialized in particular operations (e.g. strings, or numerical values).

2.1.2 Machines

Machines execute programs; they consist of a programming language and storage.

Machines may exist physically like a microprocessor, or be purely theoretical definitions, like virtual machines that are available only in software. From the conceptual point of view, this distinction is irrelevant, as software and hardware are logically equivalent. In practice, it may be advantageous to have a machine implemented by software, as changes are easier to make in software; this is common during the design phase of a machine.

A machine is not required to be implemented: machines with useful mathematical properties can be designed, to help proving the properties of other machines built on top of them.

[1]languages often refer to it as type system

[2]Registers can be general purpose or limited special purpose (e.g. fbating points, data, addresses, condition codes, or counters)

2.1.3 Libraries

Libraries are collections of programs that fulfill a purpose. The user of a library does not need to care about the implementation.

Libraries can be considered as additional instructions provided by the language, but for practical reasons are treated separately, because these instructions are not essential for the language.

As an example, many mathematical functions present in a language as primitives are implemented with library functions in the run-time, because the hardware does not have them. In some extreme cases like the ARM processor, every floating-point operation is implemented in a library, because there is no FPU unit.

2.1.4 Platforms

Platforms provide an execution environment for a program; they consist of a machine and a set of libraries, and implicitly rely on an application binary interface (ABI), which includes a data layout and a calling convention used to access the libraries. Platforms are the cornerstones on which systems are built.

Following this definition, the Java Virtual Machine is in fact a platform. Language and standard classes are tightly bound: classes `Object`, `String`, and `Throwable` are implicitly used in the language although they are part of the standard API [Szy98, p. 221]. This is not the case for the Java byte-code, which knows only about references, but has no knowledge of any particular class.

2.1.5 Layers

Layers group machines that have similar abstraction levels. Layers are hierarchically ordered, with the hardware on the lowest level and with growing abstraction.

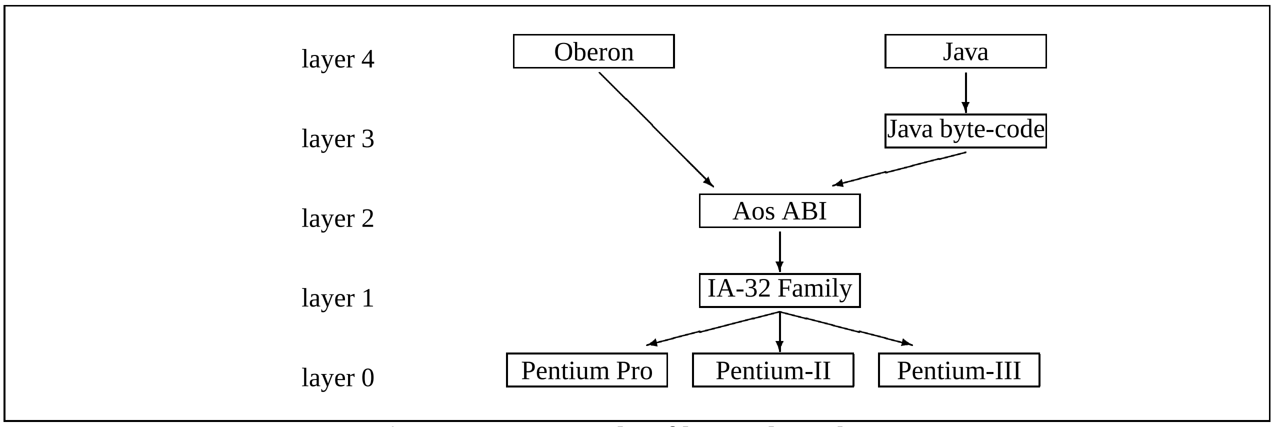


Figure 2.2: Example of layered machines

Figure 2.2 shows an example of 5 layers of different abstraction level and a few machines that are part of them.

2.1.6 Across Layers: Abstraction and Mapping

Abstraction is the step from a lower layer to a higher one. It consists of hiding, composition, and restriction of the machine features (data and the instruction set) to create a new machine.

These operations can be used together. Composition and removal are by far the most used abstraction steps: the primitives of a layer are not available in an upper layer, but more expressive primitives are offered, which are built by grouping primitives of the lower layer.

In particular, removal can be used to create a generalization of a group of machines, by hiding some primitives (or only part of their semantic) and keeping the part common to all machines. As an example, removing the timing information and the microprocessor specific instructions from the specification of a microprocessor allows to create an abstraction that embraces all microprocessors in a family. Another example is abstracting the memory location of the data: data becomes either stack or heap allocated, the data address cannot be defined or queried in the language and remains implementation dependent.

Mapping is the step from a higher layer to a lower one, the opposite of abstraction: the primitives of the layer are translated to groups of primitives in the lower layer with an equivalent semantic. Special cases of mapping have a well-known name: compilation is the translation of the programs to a language of a lower layer, whereas interpretation is the interactive execution of programs through a program of the lower layer.

Creating new machines or languages as abstractions of existing ones is uncommon. New machines can be designed from scratch independently from the machines they will be built on. Nevertheless, a mapping must exist (and symmetrically an abstraction step), otherwise they remain a purely theoretical construct. Often, the semantic gap between two layers can be big and the mapping not obvious; the introduction of intermediate layers can reduce the gap. Software engineers are by far more interested in mapping than abstracting, because it corresponds to the algorithms needed to implement a machine on a platform.

To be precise, a higher-level machine is usually mapped to a lower level platform. The mapping of some operations (like the `new` Java byte-code memory allocation) can be very complex, and often translated to a library call for simplicity. Libraries can be considered here as collections of common instruction patterns, factorized together to reduce the code size.

2.1.7 Machine Description

A machine is described by its mapping to another well-defined machine in a lower layer. The target machine can be either an existing machine or a hypothetical one.

The mapping to a real machine has the advantage of being readily available for use, but it binds the machine to a single implementation. The semantic gap between the two machines may be big, and cause the description to be correspondingly large. A compiler can be used as language definition, but the amount of information in the compiler is exceedingly high.

Hypothetical machines are designed to have useful mathematical properties, which are helpful in proving the properties of the mapped machine. Their abstraction level is usually closer to the machine to be described, and hold smaller descriptions. [KP97] for Oberon and [SSB01] for Java and the JVM are examples of formal descriptions of a language.

Describing a machine in a human language (usually English) is also possible to some limited extent. It can be useful to start with, but is not formal enough for a precise description.

2.1.8 Type Enforcement

One relevant property of a machine is the *enforcement* of the type information. Enforcement is performed either *by the machine* itself, *by the application* mapping programs to the current machine, or *by the developer* writing the programs to be mapped.

Enforcement by the machine requires the machine instructions to be aware of the type system and to perform only the appropriate operations on it. Many run-time environments define the a type system without enforcing it: on the run-time layer an ABI is defined, but the assembly language provides the instruction which can perform manipulations breaking the type invariants; these machines rely on the application to generate programs that comply with the type invariants. If the application is not aware of the type system, then the developer must write the program according to the invariant.

As an example, the run-time of a system may have a structure describing garbage collection relevant information. This structure may be only a convention that the compiler has to comply with, but nothing prevents a compiler from generating an incorrect structure. On the other hand, a machine may enforce a type system with abstract pointers by only supplying instructions that manipulate pointers through symbol names.

2.2 Components and Interoperability

This section introduces language interoperability using software components.

By definition, components address a single problem, and are cooperating parts of a whole system: these are exactly the boundaries where using different programming languages is sensible.

2.2.1 Definition

Szyperski [Szy98] defines components as follows: *software components are binary units of independent production, acquisition, and deployment that interact to form a functioning system.*

This definition captures most of important aspects of a software component. Components are functional units, and are part of a bigger system. This implies that a component system must provide a component model and a component binding (or composition) mechanism to support interaction among components. The definition fails to mention that components also define a programming interface through which they are accessed. A component system must define a way to describe the interfaces of the components in a formal way, like the various interface definition languages (IDL) do.

Components provide a highly specialized solution to a single problem. It is thus obvious to use the techniques and tools that help to build a better component. The choice of an appropriate implementation language belongs to these tools and can vary depending on the application domain of the component.

Intercommunication allows interaction and information exchange among the components of a system. It requires a common component model—known and enforced by each component— and a per-component interface built using this model and accessible to the other components.

Component composition defines the way and time when two or more components are bound together to create a system.

The rest of the section discusses these aspects of components under the perspective of language interoperability.

2.2.2 Component Model

The component model defines the architecture of a component, in particular which programming model it is based on, and how it can be accessed by other components.

Programming Model

Components define the programming interface of an object. The interface consists of the visible state of the component and a set of operations to access and control the component.

The way components are realized in the various programming languages depends on the language itself and its modeling features. Many component systems design components to be objects, classes or modules. This gives components a clear structure and model, but also creates some confusion about what is a component[3]. In fact, every programming model that allows defining some kind of interface can be used as a base for a component model.

The component model defines how to construct a component interface.

In its simplest form, the interface consists of a simple shared data structure, like a file or a pipe, which is used to pass information from components. These components do not have an own interface, but use a system library to access the common structure. This primitive but very effective form of intercommunication among components was in use even before the term of software component emerged. These interfaces are simple to provide because they never change, and are the same for the whole system; the metadata information is usually part of the data. In such a system, the component itself is usually never referenced.

Unix's pipes are a simple way to let two components communicate together; using shared files is another way; in particular, files even allow to decouple the components and let then execute at different times. In a text editing environment, the text editor and the printer—here considered as two components of the system—communicate through a postscript file: the editor creates it, the printer consumes it; they do not have to be executed at the same time or on the same machine.

[3]Sometimes the terms components and objects are used as synonyms

Using an object oriented programming model (or simply *object model*) allows components to be directly referenced and associated with one or more interfaces. This adds more flexibility, but also complexity to the design: many interfaces are available, each component may provide one or more, and they may change during one component lifetime.

Furthermore, some component models allow the component interface to change at run-time (e.g. COM) and to be component-specific.

Overall system complexity is increased because new component interfaces can be designed at any time, and can be bound to components at any time.

In a language interoperability context, two problems arise: first, the need for a semantic-preserving translation of the component interfaces in each language; second, how to provide the same component model in each language[4].

Component Description

The component model must not only define a model, but also provide a way to describe the interfaces of each component. The description is made using an interface description language (IDL), which lists the features of a component, usually its visible state and operations.

The component description can take many forms. It can be serialized into a file accompanying the component, or be loaded in memory in the reflection support to allow run-time inspection of the component itself.

Various forms of IDL exist. The Modula tradition uses symbol-files (i.e. file describing the symbols used in the compilation unit); Java includes this information in the classfile generated by the compiler; /DotNET extended the Portable Executable (PE) file format to embed this information; CORBA defines an IDL for describing the object's interfaces.

Component Model Support

To implement a component in a given language requires enforcing the component model in the program. If the programming language's model includes the component model's semantic, this can be automatically ensured by the compiler, otherwise the same semantic has to be provided by the programmer in the code.

Programming COM components in the C language is an example of a component model that is not part of the language; some operations must be provided in the program itself, like querying interfaces or releasing the component to the garbage collector. On the other hand, the Component Pascal language model includes the COM model and thus frees the programmer from these kinds of details, because the compiler takes care of them.

A similar example is provided by the JNI API, which allows C and C++ to interoperate with Java (this implicitly defines components to be equivalent to Java objects). The C language uses a model differing from Java one, thus all objects used shall be explicitly registered with the garbage

[4]this can already be a problem when using a single language

collector, and all field access shall be performs through a function call. All these operations are part of the Java language and are automatically provided in Java programs.

Interface Translation

A component is accessed through its interface. If the component is programmed in a different language than the one currently used, the interface must be translated. The translation shall preserve the interface semantic as expressed in the original language. This can be difficult or even impossible, if the interface relies on semantics that are not available or expressible in the target language.

Translations of an interface in source form are called *stubs*.

There are many approaches to interface translations:

- *hardwired:* when communicating through the operating system (e.g. pipes, files), the interface is provided by an unchangeable system library
- *by hand:* the programmer can write by hand the interface in the target language
- *by tool:* a tool to generate an interface in another language is used. `javah` is an example of a tool translating Java class definitions to C header files
- *integrated:* the translation can be performed transparently by the compiler on-demand

These techniques allow the compilation of components using different languages. If the code is compiled using different conventions (memory layout and calling convention), the linker will have to provide glue code or wrappers to perform data conversions when inter-component calls are performed.

As a side remark, integrated translation is possible only if the model used for the interfaces is a subset of both languages' models. This topic is covered in detail in the next chapter.

2.2.3 Component Composition

There are many ways and times to compose components. A component can be aware of other components, or even require a particular one, by requiring a particular interface to communicate.

We distinguish among these *binding types*:

- *none:* components are completely decoupled, unaware of each other. Example: components communicating through a file
- *static:* components are statically bound together before loading and deploying. Example: statically linked applications made from components
- *dynamic on-load:* components are linked together when loaded. Example: the Oberon system's module loading scheme, and components communicating through a pipe

- *dynamic on-demand:* components are linked on their first use; the component is linked when the first access to it performed. Example: DLLs and Java classes

In practice, the binding type also imposes the binding time.

A system may provide tools (usually called builders) to compose components together. As an example, many commercial tools exist to compose Java Beans together.

A component can also provide a *binding restriction*, to specify which other components are acceptable as service providers:

- *no restriction:* any component
- *interface restriction:* any component providing a given interface
- *version restriction:* a particular component version (or version range)

2.2.4 Inter-Component Consistency

The independence of production and provenance of the single components creates a consistency problem: the system must be able to check if a component provides the features or interfaces the system is expecting from it.

Consistency check can be performed at different life-stages of a component:

- at *implementation* time the interfaces are defined
- at *linking* time the component versions are checked
- at *run* time the dynamic types of the components and interfaces are checked
- at system *death* time (or crash) the user realizes that some assumptions were wrong

Early tests allow to produce more efficient code, e.g. in a statically typed language, but don't necessarily remove later tests. A test at link-time should always be performed to ensure that the components did not change in-between.

2.2.5 Language Interoperability

Components are an enabling technology for language interoperability. The use of different languages for each component and the related consistency problems must all be addressed during the conception of a component system.

The only additional problem created by the use of different languages is interface translation. All other problems are already present and addressed in component system based on a single language.

2.3 The Aos Approach

2.3.1 Overview

The Aos object model consists of active objects, definitions, and modules. This model is suitable for many application domains, in particular for imperative, object oriented, parallel, component, and system programming.

The Aos kernel provides a complete and efficient implementation of the Aos object model for single-processor and multi-processor machines; active object-based systems can be implemented on top of the kernel. It has a flexible design and can be used as a low-overhead operating system on hardware ranging from small devices to powerful server machines. It is currently available for the Intel IA-32 single-processor and multi-processor machines and Intel StrongARM-based machines.

The kernel provides dynamic loading and linking of modules, and support for the allocation and automatic reclamation (garbage collection) of class instances. It provides activity synchronization and object protection against concurrent access for active objects. Other services provided by the kernel include exception handling, and a plug-in architecture for dynamically installing the hardware drivers whenever needed.

In spite of the many requirements to the kernel, its implementation is lean and efficient; the core of the kernel fits into less than 50 KB, while the main hardware drivers (disk, keyboard, display, and network) need additional 140KB.

Loaders and linkers for object files can be dynamically installed into the kernel. This makes the system independent from a particular intermediate representation and persistence format. Including a just-in-time compiler in the loader easily supports other intermediate languages.

A metadata repository for the Aos object model is available. It can be used to support compilation and reflection in the system. Multiple drivers for loading metadata in the repository can be dynamically installed; this allows the repository to be independent from any persistence format, and permits language interoperability at the source level.

2.3.2 Aos Object Model

Active objects, definitions, and modules are the main parts of the Aos object model.

Active Objects

Active objects define resources and activities at the same time. A resource consists of a resource state and functions to change or query its state. An activity defines the intrinsic behavior of an active object. Objects are part of a single inheritance hierarchy.

Active objects execute their own activity independently of other activities, in a completely autonomous way. In fact, only the object itself can decide to preempt or terminate its activity. Other active objects can only indirectly affect an activity, e.g. by allocating a shared resource.

The object-centric behavior is reflected in the condition-based synchronization primitive: an activity is preempted by specifying a condition to wait for; the activity is automatically resumed once the condition is (re-)established. Conditions are boolean functions on an object state.

Resources can define their functions (or part of them) to act like monitors to protect themselves against concurrent access, providing exclusive access to the resource to an activity.

Active objects are a successful example of generalization. Instead of separating activities and objects, like many operating systems do, or—even worse—creating hybrids like in Java and C# , Active Oberon makes every object a potential active object. Ordinary passive objects (i.e. the resources) as simply active objects with empty intrinsic behavior, just a special case.

The use of conditions instead of signals or semaphores is decisive, because they change the perspective under which waiting is done from the resource to the activity. Conditions expose explicitly the resource state an activity is waiting for, as opposed to semaphores and signal which set the focus on the waiting but don't consider the reason for the preemption. By making the condition explicit and delegating the condition evaluation to the system, this mechanism becomes structured and safe at the same time. It is better structured because it can be used as a dynamic precondition, and it comes as a single instruction. It is safe because it is not possible to forget to resume an activity, whenever the condition is fulfilled. Further advantages are the explicit expression of the condition to be waited for, and the ability to have many one conditions per resource.

Definitions

Definitions are contracts that define a set of functions an object shall provide. They represent a facet of an object, and are orthogonal to the object hierarchy. As you would expect, an object can have many facets.

Definitions permit the construction of software components, because they allow to use possibly type incompatible objects for the same purpose.

Modules

A module is a static container for states, functions, and type declarations (including active objects). It defines a compilation and deployment-unit, and provides an aspect-oriented grouping of features. Modules are organized in a strict hierarchy where reciprocal references are not allowed.

Modules and classes share some commonalities: they both define resources with optional exclusive access, granting ordered access to the resource state though their functions. On the other hand, modules express a static view of the system they construct, and are thus restricted to singleton instances. Because of this, they are allowed to define constants and types.

Many language designers and object orientation advocates have removed modules from the language, because they assert that classes are a generalization of modules. This is in part correct, as a module roughly correspond to a singleton class. More often than not, the unification of classes and modules makes code hard to read, because static and non-static members, which should belong to different scopes[5] are mixed. This unification also forces the use of inheritance instead of import,

[5]static is used for singleton, but this is an attribute to apply to a whole class, not to single members of a class

which clearly changes the semantic of the program [Szy92]. Modules provide a clear separation of concerns, by containing only static members.

The clear structure provided by modules is helpful in understanding and structuring a complex software system. In particular, dependencies between modules must be solved during the design phase of the modules. This is a welcome restriction when it comes to bootstrapping a software system, because modules do prevent cyclic dependencies (and thus data races or even deadlocks among modules). A system like the Java Standard Classes can enormously suffer from cyclic dependencies during initialization: the transitive hull of `java.lang.Object` contains about 300 classes[6], whereas `System.Object` indirectly references more than 450 other classes[7]. Static initialization of a class is performed on-demand, and can begin with any class. A cyclic static dependency arises whenever the initialization of a class directly or indirectly expects the same class to be initialized[8].

Cyclic static dependencies among compilation units prevent separate compilation, because they require compiling all compilation units belonging to the transitive hull of the target unit at once.

2.3.3 Component Interoperability

The Aos kernel supports data and binary code interoperability through a predefined object-file format. The shared symbol tables provide source level interoperability and persistence format independence.

The kernel implicitly defines many API and conventions. The calling convention is the most relevant one, as it allows binary compatibility among compilation-units.

Loaders and linkers can be dynamically installed. This mechanism makes the kernel independent from the persistence format used for the object-files. It possible to have a JIT-compiler as part of the loader, and thus support different intermediate representations.

A shared metadata repository providing reflection services for the Aos object model is available on top of the kernel. The repository allows to share metadata information among compilers for different languages. Furthermore, using dynamically installed metadata loaders makes the repository independent from any predefined persistence format.

2.3.4 Multilayer Interoperability Platform

Aos is not just an interoperability platform built around its object model. The flexibility and extensibility of the kernel allows constructing other platforms on top of it; these platforms can provide an object model different from Aos' one. By using the common repository, these higher-level models can still interoperate with components written for the lower layer.

[6]measured on Classpath 0.3

[7]measured on the .NET Framework 1.0

[8]I uncovered one such cyclic static dependency in the Classpath code while testing Jaos: the function `String.toLowerCase` calls `Locale.getDefault()`, and the static initialization of `Locale` uses `String.toLowerCase`. When `Locale` is initialized before `String`, this causes an exception because `Locale` is accessed by `String` before being initialized

The ability to have different object models on the same platform is important. It releases the languages from the conformance to a simple—usually rather restrictive—object model, though still keeping interoperability with the lower levels, at least for the features present in both models.

The Jaos JVM is a case study in this direction. Jaos maps the Java object model to the Aos object model, enabling Java components to interoperate with Oberon ones. Furthermore, component interoperability using the Java model is available too.

This approach proves that it is possible to construct a platform that supports many programming models at the same time, and allows interoperability among their components (obviously with some limits).

2.4 Related Work

Language interoperability existed to some extent since the second programming language appeared. Most systems allow it only in a primitive way, to support interaction with the operating system. Some recently released systems consider interoperability unavoidable or even as their goal, and offer more powerful and better automated support for language interoperability.

2.4.1 Interoperability through shared files

Using a file to pass information among components—usually one producer and one consumer—is probably the most common way to let components interact.

The common model consists of a byte stream. The components can implicitly define a more constraining model, e.g. by defining a protocol or a grammar for the file contents, but they have to perform the consistency checks themselves. Examples of such protocols range from XML to the Postscript language.

The component's interface is provided by the operating system's file access API. This API is hardwired in the system and the same for each component.

A peculiarity of this interoperability scheme is that components are completely decoupled: they are not required to run at the same time (in fact weeks or even years can pass), and they can run on different machines and environments.

This kind of interoperability is possible on every system and programming language providing files.

Model	raw data and an hardwired system API. Additional metadata may be available as part of the data
Interface translation	the system API for accessing fi les is hardwired as part of the system and cannot be changed. Languages may provide a slightly different API to access fi les
Binding type	API bound statically fi le bound dynamic on-demand
Binding restriction	fi le name match
Consistency Check	implementation-time (compiler checks API access) link-time (library must be available) run-time (fi le must be available)

2.4.2 Interoperability through pipes or channels

Pipes and channels are communication facilities provided by the operating system to let two components communicate together. Like files, their model is a raw byte stream[9] and the interface is provided by a system library.

Furthermore, channels can provide synchronization of the communicating components. The most important difference with the file-based interoperability is the tighter coupling of the components, which are executed at the same time.

Examples are Unix sockets, which include pipes, TCP connections [Pos81], and UDP datagrams [Pos80]. Some languages also provide such facilities mostly based on the CSP [Hoa78, Hoa85] model: Ada rendez-vous, and Occam channels are examples thereof.

Model	raw data (byte stream); the interface is hardwired and provided by the system
Interface translation	the system API for accessing fi les is hardwired as part of the system and cannot be changed. Languages may provide a slightly different API
Binding type	API bound statically channel bound dynamically on-load
Binding restriction	none
Consistency Check	implementation-time (compiler checks API access) link-time (library must be available)

2.4.3 Application Binary Interface as common layer

Many operating systems define an application binary interface (ABI) to allow the use and sharing of the system libraries. The ABI includes the data layout and the calling convention. All programs wanting to use the libraries or interact together must comply with the ABI.

Although not part of the ABI, a further protocol to follow is the object file format, because this format contains the information necessary for binding the software pieces together.

The compiler provides the compliance with the ABI, by generating the code and the data structures according to the ABI.

[9]unlike fi les, pipes are always accessed sequentially

As an example, the whole Macintosh operating system libraries [App] use the Pascal ABI. The library interfaces are available in both Pascal and C. Development on this system could be done in both languages, as long as the library calls are performed using the Pascal ABI; the compiler may choose to use different conventions for its own code, and apply the run-time conventions only to the library or external calls.

This strategy is a generalization of the previous two; the difference here is, the components are linked together instead of being linked to the system.

Model	defi ned by the ABI interfaces are defi ned composing the ABI primitives
Interface translation	by hand or by tool
Binding type	static or on-load
Binding restriction	interface compatibility (often only the symbol names are checked)
Consistency Check	implementation: interface type checks link-time: symbol existence check

2.4.4 Microsoft COM

Microsoft's Component Object Model (COM) [Rog96, Szy98] is a binary standard defining objects implementing one or more interfaces. This component system is built on top of the Windows platform.

The COM model provides a higher abstraction than the model provided by the platform. The model is also higher than the C and Visual Basic languages' models; operations like querying an interface or registering the object to the garbage collector must be implemented explicitly. A few languages exist that support COM's model directly, e.g. Component Pascal.

Model	objects one or more per-instance interfaces; interfaces defi ne a set of method signatures; support for garbage collection through reference counting
Interface translation	by hand; by tool
Binding type	static; on-demand (DLLs)
Binding restriction	version number
Consistency Check	implementation-time; link-time; run-time

2.4.5 CORBA

Common Object Request Broker Architecture (CORBA) [COR02] is a standard to provide access and communication among distributed objects defined by the Object Management Group (OMG). CORBA defines an object-oriented model. Every computer participating in a distributed application can host objects and make them available to other machines through an Object Request Broker (ORB), which takes care of the communication. Internet Inter-ORB Protocol (IIOP) is the standard that defines the communication protocol between object brokers.

CORBA provides an interface description language (IDL) to describe the object's public interface. The IDL programming language agnostic; OMG has defined standard mappings between the IDL and the programming languages C, C++ , Java, Ada, COBOL, Smalltalk, Lisp, and Python.

CORBA type system is very flexible: base types include three different precisions of integers and floating-point numbers plus fixed-point; standard and wide characters and strings; boolean; valuetype; and octet for binary values. Constructed types include the structures (”`struct`”), `union`, and `enum`. Fixed and variable length structs, arrays, strings, and wstrings can be declared. An `any` type can assume any legal IDL type at runtime. The CORBA object reference is also an IDL type.

Model	objects implementing multiple interfaces; interfaces defi ne a set of method signatures
Interface translation	by hand; by tool
Binding type	on-demand
Binding restriction	interface check
Consistency Check	implementation-time; link-time; run-time

2.4.6 Java

The Java Virtual Machine [LY99] (JVM) defines an abstract platform for the execution of Java programs [GJS96]; the platform includes an object-oriented model, an intermediate representation, and an object-file format.

The Java language is not suited for low-level implementation, thus another language is needed for implementing low-level parts of the JVM; time-critical functions can also be implemented in another language. Interoperability is explicitly supported through native methods and the Java Native Interface (JNI) API [Lia99], mainly with C and C++ . The JNI was added to the language at a later stage to solve the previously mentioned problems; interoperability was never meant to be a main goal for the JVM, although its ubiquity make it an appreciate target.

The Remote Method Invocation (RMI) interface allows to interoperate with objects located on remote machines. The Java type system can be mapped to the CORBA IDL, whereas only a subset of the IDL can be mapped to java. When communicating with the IIOP protocol and using only the Java compatible subset of the IDL, allows Java to access remote objects in environments other than Java.

The JVM has also been used as a platform for languages other than Java [Tol], although the Java-centricity of its model and the consequent limitations have quite reduced the expectations and the practicability of this approach [GC00].

Model	Object instances as components; component interfaces defi ned by class and interface signatures; single class inheritance and multiple interface inheritance; C programs must explicitly program access to fi elds and methods and to the garbage collector
Interface translation	by tool to C or C++ (javah)
Binding type	on-demand (fi rst use)
Binding restriction	limited interface check[10]
Consistency Check	implementation-time; link-time; run-time

2.4.7 .NET

Microsoft’s .NET [ECM01] is a language interoperability platform. It defines a virtual machine with an object-oriented model, an own intermediate language (MS-IL), and a common metadata repository.

The .NET object model is an extension of Java's one including delegates, events, and properties; more built-in types are also available. A fine but important difference is the ability to explicitly bind a method implementation to an interface's method: this allows different implementations of methods having the same name, but defined in different interfaces. This allows to map languages with multiple class inheritance to /DotNET.

In .NET , the concepts of name-space, class, and compilation-unit are completely orthogonal, allowing a greater modeling flexibility.

.NET enforces its object model by allowing access to the data only through operations of the intermediate language that hide the details away, and refer to the data through its symbol name. Applications written for the .NET platform are type safe.

Like Java, .NET also support interaction with *unmanaged* programs, i.e. programs that do not provide strict type checking (like C). These programs run in a separated part of the platform provided for unmanaged code.

Language interoperability is an explicit goal of the .NET platform. Microsoft has designed a whole platform providing higher abstractions, and about two dozens of imperative, object oriented, and functional languages[11] are available for it. The platform specifies a metadata format and the compilers are in charge of the translation: a few languages had to be adapted to fit into the different model, or to allow access to types that do not exist in the language. .NET supports automated language interoperability. Some languages (C and C++) had to be severely restricted to be able to run on the .NET platform or their programs must run as unmanaged code.

Model	Object instances as components; component interfaces defi ned by class and interface signatures; single class inheritance and multiple interface inheritance
Interface translation	integrated (a common metadata repository is part of the platform)
Binding type	on-load; on-demand (fi rst use)
Binding restriction	interface and version
Consistency Check	implementation-time; link-time; run-time

2.4.8 Web Services (SOAP, WSDL, UDDI)

Web Services are components that expose their functionalities through a web-based (HTTP) protocol. They are emerging technique used for distributed component interoperability. The protocols for providing web services are still being defined, but three of them are widely gaining recognition and support: SOAP for the service access, WSDL for the service description, and UDDI for service location.

Simple Object Access Protocol (SOAP) [SOA00] is a lightweight protocol for exchange of information in a decentralized, distributed environment. It is an XML based protocol that consists of three parts: an envelope that defines a framework for describing what is in a message and how to process it, a set of encoding rules for expressing instances of application-defined datatypes, and

[11] C# , JScript, Managed C++ , VB.NET, APL, COBOL, Component Pascal, Forth, Eiffel, Fortran, Haskell, Mercury, Mondrian, Oberon, Perl, Phyton, RPG, Scheme, Smalltalk, Standard ML, TMT Pascal

a convention for representing remote procedure calls and responses. SOAP can potentially be used in combination with a variety of other protocols; however, it is usually used with HTTP[12].

Web Services Description Language (WSDL) [WSD01] is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information. The operations and messages are described abstractly, and then bound to a concrete network protocol and message format to define an endpoint. Related concrete endpoints are combined into abstract endpoints (services). WSDL is extensible to allow description of endpoints and their messages regardless of what message formats or network protocols are used to communicate, however, the only bindings described in this document describe how to use WSDL in conjunction with SOAP 1.1, HTTP GET/POST, and MIME.

The focus of Universal Description Discovery and Integration (UDDI) [UDD02] is the definition of a set of services supporting the description and discovery of (1) businesses, organizations, and other Web services providers, (2) the Web services they make available, and (3) the technical interfaces which may be used to access those services. Based on a common set of industry standards, including HTTP, XML, XML Schema, and SOAP, UDDI provides an interoperable, foundational infrastructure for a Web services-based software environment for both publicly available services and services only exposed internally within an organization.

[12].NET 's remoting allows to freely mix the formatters (SOAP, binary, or user defi ned) and the communication channels (TCP, HTTP, or user defi ned)

3
Interoperability

I'd rather speak clearly,
and run the risk of banality.

— Alain de Botton

3.1 Introduction

3.1.1 Interoperability Issues

Of the many benefits of language interoperability, *component reuse* across language boundaries is the most remarkable one. Interoperability allows you to use the language you prefer or find the most appropriate for solving a problem, without having to renounce to the existing components nor to the potential consumers thereof that use a different language. Ideally, interoperability allows to write even more powerful components, because they can be implemented with an appropriate language, and then used in any program, not only those written in the same language.

Mixing languages inside one project is not likely to happen, first of all because it would require more knowledge (and thus cost more) than using a single language. Nevertheless, it will be possible to access components independently of the language they are written in.

A programming platform should provide a programming model that is powerful enough to support as many languages as possible. This would allow the to have a reasonable choice of languages on a platform, having different programming models and addressing different problems, in a way that allows the developer to choose the language closest to the problem. The choice of the platform primitives is purely a technical choice; the limiting factors are usually the system's efficiency and complexity, as abstract or generic features are usually more costly and slower than specialized ones. The rest of the chapter will show that it is also possible to provide simple features and to map the more generic ones to them; the drawback for this flexibility is the lack of enforcement for the semantic of the non-primitive features (i.e. those mapped). Some choices may restrict the set of languages that can be mapped to the platform, if some language relies on a model whose abstraction level is lower than the platform one: for example, if the platform provides a model with only abstract structures and classes (but no direct access raw memory), then languages that allow pointer arithmetic will not be mappable to the platform[1].

Such a platform should also define one o more subsets of the programming model, which must be understood by all languages wishing to interoperate using the model provided by the subset. The software components have to comply with this model subset, and expose only features that can be defined using this model. This choice is more delicate, because it cannot be ignored by the languages: if a language has to be mapped to the interoperability platform, it must be able not only to map its own model to the platform model, but also to map the common subset to its own model. This seems scary at first, but is not very restrictive: a component's granularity is usually quite coarse, and the information passed across the component's boundaries doesn't usually need to rely on sophisticated data types. Consider a web server as example: the HTML pages are provided by a dozen of cooperating components implemented in different languages: it is possible to use any language (which internally could use the most exotic or pathologically eclectic programming model), as long as it complies to the common data model which is simply raw text.

It is also conceivable to define more than one interoperability model on the platform. Components would then have to specify to which interoperability model they comply. For web applications, a text stream is quite enough, for supporting software frameworks an object-model based subset would be more appropriate. These two models can coexist on the same platform.

3.1.2 Scope and Overview

This chapter summarizes and tries to answer most of the questions about language interoperability among languages having an object oriented programming model. We consider interoperability among components and compilation units. This simplifies the discussion by restricting interoperability to sharing data and execution on method-granularity, and thus to the programming model and type system of the languages.

The previous chapter showed that interface mapping is the problem to be solved in language interoperability among software components. In the programming languages, this question can be formulated in terms of programming models and type systems used to create the component's interfaces. For a mode detailed definition of software components, please refer to the previous chapter.

We informally define a type as a classification. A given value can either meet the criteria set by the classification of not. This idea is nicely depicted by Cardelli and Wegner in [CW85].

We assume the existence of a platform type system modeling the platform[2] and a subset thereof called interoperability type system.

[1]This is not completely correct, as one may allocate a big object and emulate an unstructured heap inside it, where a language could allocate and dispose its own structures. But this is obviously only an ugly workaround

[2]This choice does not restrict the generality of the discussion: it is always possible to defi ne the type system of the source language as platform type system

Language interoperability raises a few interesting questions about the relations among the various type systems and models in a system providing it:

- What are the requirements for language interoperability?
- What properties shall the platform type system have?
- What properties shall the interoperability type system have?
- What properties shall a language type system have?
- Which platform type system is the better one?
- Which interoperability type system is the better one?
- Are there equivalent models?

The platform type system is the model provided by the run-time on which all languages will execute; the interoperability type system is the model used for sharing information and thus for interoperability.

Section 3.2 will analyze the requirements for language interoperability, Section 3.3 will give a fine-grained description of the various object model flavors, Section 3.4 will show that the object-oriented models can be transformed into each other with some limitations and semantic loss, Section 3.5 will show some examples of mappings, Section 3.6 recapitulates the chapter, draws the conclusions, and answers the questions set upon entering the chapter, and Section 3.7 introduces our own approach to language interoperability.

3.2 Type Mapping

In the previous section, a type was informally defined as a classification criterion. We can make this definition slightly more format, but still intuitive:

Definition 1 *A* type t *is a set of values.*

Definition 2 *A* type system $\mathcal{T}$ *is a set of types* t*.*

This definitions allow to use a more convenient mathematical notation. For the rest of the section, the notation t is used to denote types, $\mathcal{L}$ denotes the source type system (L as in language) and $\mathcal{P}$ denotes the destination type system (P as in platform).

Type theory has already been investigated in depth, and it is outside the scope of this thesis to build a new type theory[3].

When mapping types from one type system to another one, a few operations are possible:

[3]the works of Cardelli extensively this fi eld and defi nes a complete formalism to deal with it

Definition 3 Equivalence, *when a type $t_L \in \mathcal{L}$ is mapped to the type $t_P \in \mathcal{P}$ such that $t_L \equiv t_P$.*

Equivalence is the simplest case. Both the source and the destination type systems have the same type.

Example 1 *When mapping the Oberon type system to the Java type system (or the equivalent JVM's one), the Oberon type `LONGINT` is mapped to the Java type `int`. Those two types have the same semantic and exactly the same domain.*

Definition 4 Generalization, *when the destination type includes the source type. $t_L \in \mathcal{L}$, $t_P \in \mathcal{P}$, and $t_L \subset t_P$.*

Generalization means that the destination type contains all the values from the source type (and possibly more), and can thus be used to represent those values.

Example 2 *The Oberon type `INTEGER` (16-bit signed integer) can be mapped to the C# type `int` (32-bit signed integer). The destination type includes all the values of the source type.*

The wider domain of the destination type should not confuse the reader. Although the set may contain more values, those values will never be used. The set of valid values and the result of all operations on them are defined in the source type; all operations are applied according to the semantic of the source type.

Example 3 *The Oberon type `ARRAY 32 OF REAL` can be mapped to the C# type `float[]`. The fixed-size array is mapped to the dynamically-sized array defining all the arrays of 32-bit floating-point values. It would be also possible to map to `double[]`, thus having an even wider destination domain.*

Although the statically sized array (with size 32) is mapped to a dynamically sized array, arrays of size other than 32 will never be created nor used, because the source type system defines no operation that changes the size of a statically sized array.

Definition 5 Composition, *when the source type is mapped to a composition of destination types. $t_L := t_{P_1} \times \cdots \times t_{P_n}$.*

Composition is used to create types that don't exist in the destination system, or when the destination system only provides simple primitive types to be composed together.

Example 4 *The Active Oberon type `HUGEINT` (64-bit signer integer) is mapped to a tuple (`LONGINT x LONGINT`) of Oberon types. The first longint is interpreted as the most significant word, the second as the least significant word.*

One important point appears here: the values in the source and destination sets are not required to be the same. The values in the destination type system can be interpreted to provide the appropriate meaning in the source type system. This is only useful if there is no type similar (or "close enough") to a particular type in the source system.

Independently of the mapping, the most important characteristic of the destination system is the ability to represent all the elements of the source system; in practice, this means that the destination set must have at least as many elements as the source set.

When mapping a type to another one, we implicitly assumed the existence of a value mapping function $m_{t_i \to t_j}$ such that

$$m_{t_x \to t_y}(x) = y, x \in t_x y \in t_y$$

We denote the mapping function from type system $\mathcal{X}$ to type system $\mathcal{Y}$ as

$$M_{\mathcal{X}\to\mathcal{Y}}(t) = \begin{cases} \dots \\ \langle t_y, m_{t_x \to t_y} \rangle & t = t_x \in \mathcal{X} \\ \dots \end{cases}$$

where t_y is a type or a composition of types from $\mathcal{Y}$. As a shortcut, we won't write the type-mapping function $m_{t_x \to t_y}$ when it is an identity function.

Rule 1 (Complete Type Conversion) *Each function $m_{t_X \to t_Y}$ must be total and injective.*

When mapping a type t_X to a type t_Y with a mapping function $m_{t_X \to t_Y}$, all values of the type t_X must be mappable to the destination type t_Y. The mapping function must satisfy that

$$x_1, x_2 \in t_i, x_1 \neq x_2 \Rightarrow m_{t_i \to t_j}(x_1) \neq m_{t_i \to t_j}(x_2)$$

In other words, all values are mapped to distinct values in the codomain, and the codomain must be as big as the domain to be able to fit all those values. As a corollary

$$m_{t_X \to t_Y} \Rightarrow \|t_X\| \leq \|t_Y\|$$

Rule 2 (Complete Type System Conversion) *The function $m_{\mathcal{L}\to\mathcal{P}}$ mapping the types of $\mathcal{L}$ to the types of $\mathcal{P}$ must be a total function.*

When mapping a type system to another one, all types of the source must be mappable to the destination system. $\forall t \in \mathcal{L}, m_{\mathcal{L}\to\mathcal{P}}(t) \in \mathcal{P}$.

In practice, it doesn't matter if a type is mapped using equivalence, generalization, or composition. Note that although the each $m_{t_X \to t_Y}$ must be injective, $M_{\mathcal{X}\to\mathcal{Y}}$ must not be injective.

Example 5 *When mapping Oberon to the JVM type system, $M_{\mathcal{O}\to\mathcal{J}}$ is a legal mapping.*

$$M_{\mathcal{O}\to\mathcal{J}}(t) := \begin{cases} \dots \\ \texttt{int} & t = \texttt{LONGINT}, \texttt{INTEGER}, t = \texttt{SHORTINT} \\ \dots \end{cases}$$

These considerations hold only if the source program is executing in complete isolation on the destination platform, hence no foreign code is used. This situation is quite common: all translation tools that convert the code from a language to execute in a particular run-time environment belong to this group.

Example 6 *The Paco compiler for Active Oberon maps the Active Oberon type system to the type system provided by the Intel processors*[4]*. The type BOOLEAN is converted to an 8-bit integral, but only the values 0 and 1 are used. The compiler is responsible to enforce this limitation* [5]

$$M_{OBERON \rightarrow x86}(\texttt{BOOLEAN}) = \langle \texttt{byte}, m_{\texttt{BOOLEAN} \rightarrow byte} \rangle$$

$$m_{\texttt{BOOLEAN} \rightarrow byte}(x) := \begin{cases} 0 & \textit{if } x = \texttt{FALSE} \\ 1 & \textit{otherwise} \end{cases}$$

To let two or more languages interoperate, an interoperability type system $\mathcal{I}$ is defined.

Rule 3 $\mathcal{I} \subseteq \mathcal{P}$. *The interoperability type system is a subset of the platform type system.*

$\mathcal{I}$ is a subset of $\mathcal{P}$, because $\mathcal{I}$ is expressed in terms of $\mathcal{P}$. It is not needed to have the whole $\mathcal{P}$ as interoperability system, because this would impose to many restrictions on the interoperability platform.

Interoperability causes a mapping in the opposite direction, because the $\mathcal{I}$ type system must be mapped to the $\mathcal{L}$ type system, i.e. the language must understand the interoperability model. It obvious that this mapping must at least statisfy the rules (1) and (2). This is not enough, because this works only for the receiving (or reading) values, but not for passing them. Let's make an example:

Counterexample 7 *Source language is Oberon, destination is C# , interoperability subset is* `int`*. Following the rules (1) and (2),* $M_{C\#\ \rightarrow Oberon}(\texttt{int}) = \texttt{HUGEINT}$ *is legal. This is fine for out-parameters, return values, and for reading fields, but not for passing parameters, because it would allow Oberon to pass a value that is outside of the domain expected by C# .*

From the previous counterexample is obvious that the functions $m_{t_X \rightarrow t_Y}$ for mapping $\mathcal{I}$ must also be surjective.

Rule 4 *All fuctions* $m_{t_X \rightarrow t_Y}$ *of* $M_{\mathcal{I} \rightarrow \mathcal{L}}$ *must be bijective.*

In practice, this further restriction means that the domain and the codomain of the function must contain the same number of elements. Obviously, the following rule must hold too:

[4]Processors provide a very simple type system consisting of 8, 16, and 32-bit integrals and 32, 64, and 80-bit fbats, and the combinations thereof

[5]In an internal seminar, N. Wirth once asked for which values of `b:BOOLEAN` the expression `(b & !b)` holds `TRUE`. The answer is, when `b := SYSTEM.VAL(BOOLEAN, 5);`.

Rule 5 (Identical Mapping)

$$\forall t \in \mathcal{I}, M_{\mathcal{L}\to\mathcal{I}}(M_{\mathcal{I}\to\mathcal{L}}(t)) = t$$

This rule ensures that types mapped from $\mathcal{I}$ to $\mathcal{L}$ are mapped back to the same type of $\mathcal{I}$.

To understand the consequences of the previous rules, let's make an examples.

Example 8 (Oberon and the .NET CLS) *Can the (original) Oberon language's type system be used for accessing the types of the .NET framework? In practice, the question is whether the CLS can be mapped to Oberon or not. The mapping function for the built-in types is as follows:*

$$M_{CLS\to Oberon}(t) \begin{cases} \texttt{BOOLEAN} & t = \texttt{bool} \\ ? & t = \texttt{char} \\ \texttt{SYSTEM.PTR} & t = \texttt{object} \\ ? & t = \texttt{string} \\ \texttt{REAL} & t = \texttt{float32} \\ \texttt{LONGREAL} & t = \texttt{float64} \\ \texttt{SHORTINT} & t = \texttt{int8} \\ \texttt{INTEGER} & t = \texttt{int16} \\ \texttt{LONGINT} & t = \texttt{int32} \\ \texttt{LONGINT} \times \texttt{LONGINT} & t = \texttt{int64} \\ \texttt{LONGINT} & t = \texttt{native int} \\ \texttt{SYSTEM.BYTE} & t = \texttt{unsigned int8} \end{cases}$$

Most of the built-in types of the CLS can be mapped to Oberon. The first problem is the type `char`: the corresponding Oberon type (`CHAR`) is only 8-bit wide. The compiler designer faces here two choices:

1. *redefine CHAR to 16-bit to fit the Unicode character set. This will also require changing the definition of the character constants and of the conversion functions (`CHR` and `ORD`) to take, respectively return, a `LONGINT` instead of an `INTEGER`.*
2. *map `char` to the `INTEGER` type and let the user work with the raw numeric constants (or provide some functions to do the conversion)*

Although more devastating, the first choice is the better one, because `CHAR` was defined in function of the platform on which Oberon was designed (and happened to be ASCII); Oberon basic types are meant to map directly the types provided by the underlying platform; thus changing the type definition follows the spirit of the language.

Strings are also a problem, because they are objects, and the original Oberon has no support for objects.

The Oberon built-in types are mapped as follows:

$$M_{Oberon\to CLS}(t) = \begin{cases} \texttt{unsigned int32} & t = \texttt{SET} \\ M^{-1}_{CLS\to Oberon}(t) & \textit{otherwise (identical mapping rule)} \end{cases}$$

There has been a lot of discussion on the object model used by the .NET platform. Many have meant that only languages with exactly the same model can run on the platform. The large number of languages with different programming models (e.g. Eiffel, ML, ...) proves the contrary. Even the modest previous example shows that Oberon's `SET` type can be mapped to the platform without need to be compliant to the CLS[6]. The question often raised is, if languages with a different object oriented model can coexist with .NET . The answer is already stated in the rules (1) and (2): yes, if a mapping exists.

It is important to understand that the type system used to describe the classes specifies only their interfaces. In other words, a class constists of fields, methods, and is assignment compatible with some other types (at most one class and an unlimited number of interfaces). For a client, it doesn't matter if multiple class inheritance was used or only interface inheritance. A client needs only a flat view of a class specifying its members.

In fact, changing the CLS to support multiple class inheritance would not change the framework, nor break any compiler: the view a client has of a class would remain the same: a collection of fields and methods.

For a language providing CLS-compliant classes (called an extender in the ECMA-335 jargon) this wouldn't change much either: the language would not have to provide support for multiple inheritance (which would be provided by the platform), but only to define a class, and provide the implementation for its methods like it already does: the language would simply not use it.

The next two sections will make this point clear, by showing that the various object model flavors can be mapped onto each other, and by showing where some semantic loss occurs.

3.3 Object Model Flavors

3.3.1 Definitions

Many programming languages use an object-oriented programming model, which we further refer to as *object model*. Object models build on objects, inheritance, and polymorphism. Multiple inheritance of classes is the most generic model, and all other object models are just special cases of it. Whether multiple inheritance simplifies or complicates the object model remains a matter of debate. Many language designers opted for a weaker model by providing a limited form of classes and inheritance, mostly to simplify the implementation of the model, and to avoid problems inherent to the multiple inheritance[7].

A class consists of fields (representing the class state) and methods (operations on the class state). The class members have a few orthogonal properties.

Members can be either *concrete* or *deferred*. Deferred member are placeholders without implementation; the implementation is deferred to the subclasses of the current class[8]. Another property of members is *static* (or *singleton*) or instance binding.

[6]obviously, it won't be possible to interoperate using this type

[7]name and implementation clashes

[8]The term *abstract* is also commonly used to denote deferred members

Constructors are methods specially called at class instantiation to initialize and parametrize the class instances.

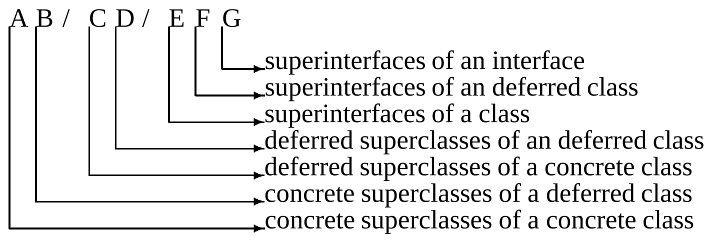
Different flavors of the object oriented model are created by a selective combination of these features. Flavors differ on the restrictions on the classes and class' members, and on the kind and amount of superclasses they can have. Three degrees of inheritance are possible: none, single, and multiple. Class kinds differentiate themselves on the kind of members they have: *concrete* classes have only concrete members, *deferred* classes have both concrete and deferred members, *interfaces* have only deferred methods, *modules* have only static members. Furthermore, a class may be *final* and can thus not be further specialized by other classes.

Classes can be composed into a hierarchy using no inheritance, single inheritance, or multiple inheritance. Class kinds and inheritance can be combined in a completely orthogonal way. Because only concrete classes can be instantiated, the leaf elements of a class hierarchy are always concrete classes; for the same reason, only concrete classes can be final.

During the rest of the chapter, we will avoid the term class without qualification to avoid confusion.

We introduce a compact notation to express an object model's inheritance pattern in a compact form. Figure 3.1 shows the detailed notation, while Table 3.1 gives an overview of the object models of a few common programming languages described with this annotation. Please note that the compact form doesn't show the differences in each language's class definition.

inherits from	concrete	deferred	interface
concrete	A	C	F
deferred	B	D	G
interface	0[a]	0[b]	H



A B / C D / E F G
superinterfaces of an interface
superinterfaces of an deferred class
superinterfaces of a class
deferred superclasses of an deferred class
deferred superclasses of a concrete class
concrete superclasses of a deferred class
concrete superclasses of a concrete class

0: no inheritance
1: single inheritance
N: multiple inheritance

[a] an interface cannot inherit from a concrete class

[b] an interface can only inherit from a deferred class consisting only of deferred methods, which is just an interface

Figure 3.1: The Object Model Notation

When designing a language or a virtual machine, the question of the object model to use is of capital importance, and in particular, whether one model is better than the others. From the previous section we know that the language model and the platform model are not required to be the same, as long as a mapping exists. Thus, the obvious question is, whether a mapping from one object model to another one exists.

Active Oberon[a]	interface	single interface inheritance
	class	single class implementation inheritance and multiple interface implementation
	notation	10/00/N01
Oberon.Net	abstract class	single abstract class inheritance
	class	multiple abstract class inheritance
	notation	00/N1/N11
Java, C#	interface	multiple interface inheritance
	class	single class implementation inheritance and multiple interface implementation
	notation	11/11/NNN
Eiffel[b]	class	multiple class inheritance
	notation	NN/NN/NNN
COM	interface	no inheritance
	class	multiple interface inheritance
	notation	00/00/N00

***Table 3.1**: Some Programming Languages and their Object Model*

[a] doesn't have abstract methods and classes
[b] interfaces are classes containing only deferred methods

Documenting all the possible mappings between any two variants doesn't make sense, as the number of mappings grow exponentially with the number of object model flavors. Our approach is to factorize the problem into small primitive steps. These steps can then be sequentially combined to provide the mapping from model to model.

This notation simple but has a few limitations. Oberon.NET definitions are classified as deferred classes, but in the language spirit they are interfaces with default implementations. Java and C# have the same classification, but the semantic of a class inheriting from an interface is different: in Java, the class must contain a method with the same name, in C# the class must provide an implementation for the interface, which can be bound to the interface (and not to the class).

3.3.2 Examples

Active Oberon The Active Oberon language (Chapter 4) consists of concrete classes and definitions; the first are called classes or objects, the latter pure definitions[9]. Classes are build a single inheritance hierarchy and can implement multiple interfaces. Interfaces are composed using single inheritance. Methods are implicitly deferred when declared in an interface. Table 3.2 resumes Active Oberon's model.

Oberon.NET The Oberon.NET language [Gut01a] has a novel approach. Deferred classes define the members that concrete classes have to provide. There's no inheritance among concrete

[9] Active Oberon uses the same naming conventions as Oberon.NET

class kind	*fields*	*methods*	*inherits from*	
module	static concrete	static concrete		
object	concrete	concrete	single	object
			multiple	definition
definition		deferred	single	definition

***Table 3.2**: Overview: Active Oberon's model*

classes (concrete classes are implicitly always final). Table 3.3 resumes Oberon.NET characteristics.

class kind	*fields*	*methods*	*inherits from*	
module	static concrete	static concrete		
object	concrete	concrete	single	object
			multiple	definition
definition	concrete	concrete	single	definition
	deferred	deferred		
pure definition[a]		deferred		

***Table 3.3**: Overview: Oberon.NET's model*

[a]pure defintions are implicit

Although classified as deferred classes, Oberon.NET definitions are ideally much closer to interfaces, because they define a view or aspect of an object (like interfaces do), and may provide a default implementation for their methods.

Microsoft's .NET CLR The Object Model in Microsoft's .NET CLR has concrete classes, deferred classes, and interfaces. This model supports multiple inheritance only among interfaces, otherwise only single inheritance; final[10] classes are supported. Methods can be declared as deferred. Table 3.4

class kind	*fields*	*methods*	*inherits from*	
interface		deferred	multiple	interface
deferred[a]	[static] concrete	[static] concrete	multiple	interface
		deferred	single	deferred
			single	concrete
concrete	[static] concrete	[static] concrete	multiple	interface
			single	deferred
definition		deferred	single	concrete

***Table 3.4**: Overview: .NET CLR's model*

[a]called abstract

[10]called *sealed*

Although classified the same way as Java, the CLR model provides a fine but importand difference. A class can explicitly implement the interface's methods. This allows having different implementations for homonymous methods specified in different interfaces. A class is not forced to define in its scope the methods defined in its interfaces, but must provide an implementation to all of them.

3.4 Mappings and Conversions

3.4.1 Notation

The algorithms in this section follow a common notation.

First, a *description* gives a quick informal description of the algorithm and its goal. The *Explanation* makes clear how the algorithm works. The *preconditions* give the preconditions required by the algorithm. The *sematic loss* is the list of semantic meanings or restrictions that are lost due to the weaker or different destination model. If needed, a *notes* paragraph will carry some additional information and comments.

3.4.2 Concrete to Deferred Class

Description This conversion removes multiple concrete class inheritance. It does so by splitting concrete classes in a deferred class and a concrete class and substituting the concrete class hierarchy with a deferred class hierarchy. Using the shorthand notation, this corresponds to the transition from `AB/../...` to `00/AB/...`.

Explanation Given a concrete class `C`. `C` is declared as deferred; a concrete final class `C′` subclassing deferred class `C` is created. Instantiation of `C` is replaced with instantiation of `C′`. Table 3.5 gives an example of the mapping.

	Before	After
Declarations	`class C {` `...` `}`	`abstract class C {` `...` `}` `final class C′:C {}`
Allocation	`new C();`	`new C′();`

***Table 3.5**: Example: Concrete to Deferred Class*

Preconditions

1. `C` is concrete.
2. `C` is not final.

Semantic Loss The destination model cannot enforce that:

1. `C` and `C′` represent the same class.
2. `C′` is to be used only for instantiation.

Notes Class `C′` is declared as final to prevent other classes from subclassing it. `C′` is an helper class that exists only to allow instances of `C` to be allocated. Only `C` is to be subclassed. Both preconditions are no limitations to the algorithm: if a class is not concrete, than this conversion is not needed, and final classes cannot be subclassed (i.e. they don't cause any inheritance of concrete classes) and thus don't need to be split.

3.4.3 Field Deferral

Description Field deferral moves the concrete field declarations from the deferred class `C` to the classes `C′` that concretize `C`.

Explanation Given a concrete field `f` in deferred class `C`. All concretizing subclasses `C′` of `C` must provide a concrete implementation of `f`. If `C` is public and `f` is private, then `f` visibility must be changed to at least protected. Table 3.6 shows an example of this conversion.

	Before	After
Declarations		
	`deferred class C {`	`deferred class C {`
	`  T f;`	`  deferred T f;`
	`}`	`}`
	`class C′:C {`	`class C′: C {`
		`  T f as C.f;`
	`}`	`}`

***Table 3.6**: Example: Field Deferral*

Preconditions

1. class `C` is deferred

Semantic Loss

1. if `C` is public and `f` is private, f visibility scope has to be broadened.

Notes The field visibility has to be changed from private to protected, if the class `C` is public. The reason for this change in visibility is, all direct and indirect subclasses of `C` can concretize `f`, thus whenever `C` is visible, so must be the declaration of `C.f` in `C′`. The concretizing class `C′` must also be able to cope with name clashes (fields with the same name may be declared in more than one superclass).

3.4.4 Method Deferral

Description Defer a method `m` in a deferred class `C` by moving the implementation code to all classes `C′` that concretize `C`.

Explanation Given a concrete method `m` with implementation `i` in deferred class `C`. Declare `m` as deferred. In every class `C′` that concretizes `C`, a method `C.m` with implementation `i` is declared. If `C` is public and `m` private, `m` visibility must be changed to at least protected. All members accessed in `i` must also be (or be made) accessible in `C′`. Table 3.7 shows an example of this conversion.

	Before	After
Declarations		
	`deferred class C {`	`deferred class C {`
	`  T m(...) {i}`	`  deferred T m(...);`
	`}`	`}`
	`class C′: C {`	`class C′ : C {`
		`  T C.m {i}`
	`}`	`}`

***Table 3.7**: Example: Method Deferral*

As a variant to avoid code duplication, all implementations `i` can be wrapped into a class `CS`. Classes implementing `C.m` must forward the call to `CS.m`. The members accessed inside `i` must be visible only to `CS`. Table 3.8 shows an example of the alternative implementation.

Preconditions

1. class `C` is deferred
2. explicit method binding or renaming must be supported to avoid name clashes in `C′` and subclasses thereof

	Before	After
Declarations		
	`deferred class C {`	`deferred class C {`
	`  T m(...) {i}`	`  deferred T m(...);`
	`}`	`}`
		`final class CS : C {`
		`  static T m(self: C; ...) {i}`
		`}`
	`class C′ : C {`	`class C′ : C {`
		`  T C.m(...) {CS.m(this, ...);}`
	`}`	`}`

***Table 3.8**: Example: Method Deferral, Alternative Implementation*

Semantic Loss The mapping incurs in the following sematic losses:

1. method `m` may have its visibility broadened
2. all members accessed in `i` may have their visibility broadened to be visible in `C′`
3. the model cannot enforce that concretizations of `C` must use the implementation `i`

The alternate implementation incurs the following semantic loss:

1. deferred method `m` may have its visibility broadened
2. all members accessed in `i` may have their visibility broadened to be visible in `CS`
3. the model cannot enforce that concretizations of `C.m` must forward the call to `CS.m`
4. the model cannot enforce that `CS` is never used for other purposes

The variant is slightly better than the first mapping, because the visibility of the members accessed by `m` has must be less broadened (the other losses are the same).

3.4.5 Deferred Field Remotion

Description Deferred fields can be removed by replacing them with method calls. This is useful in models that don't provide deferred fields.

Explanation Deferred fields `f` are replaced by a deferred method `set.f` and a deferred method `get.f`. Classes concretizing the field `f` must also concretize both methods, which will set and get the value of the field `f`. Table 3.9 shows an example of this conversion.

	Before	After
Declarations	`class C {` `deferred T f;` `}` `class C′ : C {` `T f as C.f;` `}`	`class C {` `deferred T get.f();` `deferred void set.f(T value);` `}` `class C′ : C {` `T f as C.f;` `T C.get.f()` `{ return f; }` `void C.set.f(T value)` `{ f = value; }` `}`
Field Access	`... = f;` `f = ...;`	`... = C.get.f();` `C.set.f(...);`

***Table 3.9**: Example: Deferred Field Removal*

Preconditions

1. field f is deferred

Semantic Loss

1. the model cannot enforce that `set` and `get` access the field `f` instead of another one

Notes This substitution is useful for models that do not support deferred fields. Some models may have equivalent features to implement deferred fields, like properties in C# .

Altough field deferral and field removal are often used together, we have splitted them to make clear which step causes the semantic losses.

3.4.6 Interface Hierarchy Removal

Description Interface Hierarchy Removal removes the inheritance hierarchy among interfaces by flattening it into the concretizing class.

Explanation Given an interface `I` with superinterfaces `SI` and concretizing class `C`. Copy all members of `SI` into `I`, remove superinterface `SI` from `S`, and make it superinterface of `C`. Bind the concretizations of all methods `m` in `C` to both `SI` and `S`. Table 3.10 shows an example of this conversion.

	Before	After

Declarations	`interface SI {`
	`  T m(...);`
	`}`
	
	`interface S: SI {`
	
	`}`
	
	`class C: S {`
	`  T S.m(...) {...}`
	`}`
		`interface SI {`
		`  T m(...);`
		`}`
		
		`interface S {`
		`  T m(...);`
		`}`
		
		`class C: S, SI {`
		`  T S.m, SI.m (...) {...}`
		`}`

Conversions	`SI si;`
	`S  s;`
	
	`si = s;`
		`SI si;`
		`S  s;`
		
		`si = (SI)((AnyType)s);`

***Table 3.10**: Example: Interface Hierarchy Removal*

Preconditions

1. `S` and `SI` are interfaces

Semantic Loss

1. the inheritance relationship between `SI` and `S` is lost (conversion is still possible, as both are always concretized in a common class)
2. cannot enforce that class `C` inheriting from `S` also inherits from `SI`.
3. cannot enforce that class `C` uses the same implementation `i` for `S.m` and `SI.m`.

Notes This models still provides polymorphism, but has a extremly limited inheritance. It is arguable, if such a model can still be considered object oriented.

3.4.7 Deferred Field Concretization

Description This transformation makes deferred fields concrete.

Explanation Given deferred field `f` in class `C`. Change definition of `f` to concrete field; remove all concrete declarations of `f` in all concretizing subclasses of `C`. Table 3.11 shows an example of this conversion.

	Before	After
Declarations		
	`class C {`	`class C {`
	`  deferred T f;`	`  T f;`
	`}`	`}`
	`class C′ : C {`	`class C′ : C {`
	`  T f as C.f;`	
	`}`	`}`

***Table 3.11**: Example: Deferred Field Concretization*

Preconditions

1. Field `f` is deferred

Semantic Loss None

3.4.8 Deferred Method Concretization

Description Deferred Method Concretization concretizes a deferred methods in a class.

Explanation Given a deferred method `m` in class `C`, 0 or more concretizations `m′` in `C′` with the same implementation `i`. Change definition of `m` to concrete with implementation `i`; remove concretizations `m′` in all classes `C′`. Table 3.12 shows an example of this conversion.

	Before	After
Declarations		
	`class C {`	`class C {`
	`  deferred T m(...)`	`  T m(...) {i}`
	`}`	`}`
	`class C′ : C {`	`class C′ : C {`
	`  T m′ as C.m {i}`	`}`
	`}`	

***Table 3.12**: Example: Deferred Method Concretization*

Precondition

1. all concretizations `m'` must have the same implementation `i`
2. `i` must refer only to members visible to `C`

Semantic Loss

1. the ability to provide different implementations `i'` to `m` is lost.

3.4.9 Single to Multiple Inheritance

Description This promotion maps a model with single inheritance to one with multiple inheritance. This corresponds to the transition from `../../111` to `../../NNN`, from `../11/...` to `../NN/...`, and from `11/../...` to `NN/../...`.

Explanation No changes to perform, as single inheritance is a special case of multiple inheritance.

Semantic Loss None

3.4.10 Interface to Deferred Class

Description This promotion converts an interface into a deferred class. This corresponds to the transition from `../../ABC` to `../AD/000`, where `D` = `max(B,C)`.

Explanation No changes to perform, because interfaces are special cases of deferred classes.

Semantic Loss None

3.4.11 Deferred Class to Concrete Class

Description This transformation transforms a deferred class into a concrete class.

Explanation Deferred classes with no deferred members are special cases of concrete classes. To convert remove the deferred qualifier. Table 3.13 shows an example of this conversion.

Preconditions

1. `C` neither contains nor inherits any deferred members

	Before	After

Declarations	`deferred class C {`
	`  ...`
	`}`
		`class C {`
		`  ...`
		`}`

Table 3.13: *Example: Deferred Class to Concrete Class*

Semantic Loss

1. The deferred property of `C` is lost

3.4.12 Signals to Conditions

Description This section shows how to implement signals using conditions.

Explanation Signals are simulated with a ticket algorithm, which assigns to every waiting thread a ticket; signal notification corresponds to calling the next ticket. Table 3.14 shows an example of this conversion, as implemented in Jaos.

```
java/lang/Object = OBJECT
  VAR  in, out: LONGINT;

  PROCEDURE wait*();
    VAR goal: LONGINT;
  BEGIN {EXCLUSIVE}
    goal := in; INC(in);
    AWAIT(goal < out);
  END Wait;

  PROCEDURE notify*();
  BEGIN {EXCLUSIVE} INC(out) END notify;

  PROCEDURE notifyAll*();
  BEGIN {EXCLUSIVE} out := in END notify;
END java/lang/Object;
```

Table 3.14: *Example: Implementing Signals With Conditions*

Preconditions None

Semantic Loss None

3.4.13 Conditions to Signals

Description This conversion replaces Conditions with Signals.

Explanation Every AWAIT is mapped to a Wait, protected with a while-loop; leaving a critical section causes the conditions to be reevaluated. Table 3.15 shows an example of this conversion.

	Before	After

Protection	`BEGIN {EXCLUSIVE}`
	`  ...`
	`END`
		`Monitor.Enter();`
		`  ...`
		`  Monitor.PulseAll(this);`
		`Monitor.Exit();`

Synchronization	`AWAIT(cond);`	`while (!cond) {`
		`  Monitor.Wait(this);`
		`}`

***Table 3.15**: Example: Replacing Conditions with Signals*

3.5 Examples

This section shows a few examples how to use the conversions shown in section 3.4.

The conversions consist of applying the various steps. Each step also contains a list of preconditions and losses that the conversion has. The conversion is possible if all preconditions in the conversion are satisfied; semantic losses remain and must be explicitly dealt by the compiler (but are exposed to the other languages).

3.5.1 Active Oberon to Oberon.NET

The first example we propose is the conversion from Active Oberon to Oberon.NET.

Active Oberon has interfaces (called pure definitions) and concrete classes. Neither deferred members nor deferred classes are supported. An interface can inherit from a single interface (this inheritance is called refinement), whereas concrete classes inherit from a single concrete class and multiple interfaces. The model is `1/00/N01`. Active Oberon's concurrency model is based on active objects.

Oberon.NET has deferred classes (called definitions) and concrete classes; interfaces are special cases of deferred classes (pure definitions). Definitions contain fields and both concrete and deferred methods, and can inherit (refine) from single definition; concrete classes inherit (implement) from multiple definitions; there is no class inheritance. The language model is `0/N1/N11`.Oberon.NET's concurrency model is based on active objects.

Mapping Active Oberon to Oberon.NET requires removing the single inheritance of concrete classes, which is not supported in Oberon.NET. The concurrency model is the same. The mapping consists of the following step:

1. Concrete To Deferred Class

Preconditions The only precondition for this conversion is that the classes mapped shall not be final, but as Active Oberon doesn't have this kind of classes, this is no problem.

Semantic Loss The conversion splits all classes into a definition and an instantiation class. Oberon.NET cannot enforce that both classes represent in fact the same class.

3.5.2 Oberon.NET to .NET CLR

The mapping from Oberon.NET to .NET is not obvious: the multiple inheritance of deferred classes is very close to the generic multiple inheritance, whereas .NET supports only single inheritance of implementations. In practice, a `0/N1/N11` is to be converted to a `1/11/NNN` model, and Active Oberon's Active Objects mapped to .NET's signals. The conversion steps required are:

1. Field Deferral
2. Deferred Field Remotion
3. Method Deferral
4. Conditions to Signals

Preconditions Name clashes among fields and methods are addressed by Oberon.NET, which requires explicit implementation of the interface methods; the same is provided by the CLR. Interface implementation binding is supported by both Oberon.NET and the CLR, thus method name clashes are no problem.

Semantic Loss Field and Method Deferral may cause the visibility of fields and methods to be broadened: Oberon.NET is immune to this, because all members in a deferred class are always public. Methods can access private static fields in the enclosing module; if such a method is deferred, those fields must be made public breaking information hiding. The CLR cannot enforce all concretizations of a method to use the same implementation.

3.5.3 Eiffel to .NET CLR

Eiffel has the most generic model considered here: multiple inheritance of classes. All other models are in fact special cases of this model. Mapping Eiffel to the CLR requires converting multiple inheritance of concrete classes into multiple inheritance of interfaces. This requires the most conversion steps.

1. Concrete to Deferred Class
2. Field Deferral
3. Deferred Field Remotion
4. Method Deferral

Semantic Loss Two problems affect Eiffel programs mapped to the CLR. First, an Eiffel class is mapped to an interface and a class, and the CLR cannot enforce the fact that these two entities are in fact two projections of the same one. Second, information hiding may take a toll in the mapping when fields and methods implementations are deferred, because some fields may have to be made more visible than they are meant to.

These losses don't affect Eiffel, because it can be compensated by some additional checks in the compiler. The problem is for third party languages that import components written in Eiffel, and could perform illegal changes or operations (e.g. accessing a field which is invisible for Eiffel programs, but made public for allowing the conversion of the model).

3.6 Wrap-up

3.6.1 Discussion

Through a combination of the mapping algorithms, it is possible to map most of the object model flavors into each other with some limited semantic loss. Only *multiplicity and subtyping cannot be removed*: all models can be translated to the most restrictive model, multiple implementation of interfaces with no inheritance, but only models with single inheritance can be transformed into a model with single implementation of interfaces.

Semantic loss mostly affects member's visibility and the enforcement of some properties. In a single language environment, the semantic gap between language and platform model is bridged by the compiler. The same is done in a multilanguage environment, but the compiler has authority only on the own language, other languages are constrained only by the common model.

We recognize two extremes in the set of object models: (`NN/NN/NNN` multiple class inheritance, as used in Eiffel) and no interface inheritance but with multiple interface implementation (`00/00/N00`, the COM model). It is possible to convert one into the other one, but we can hardy

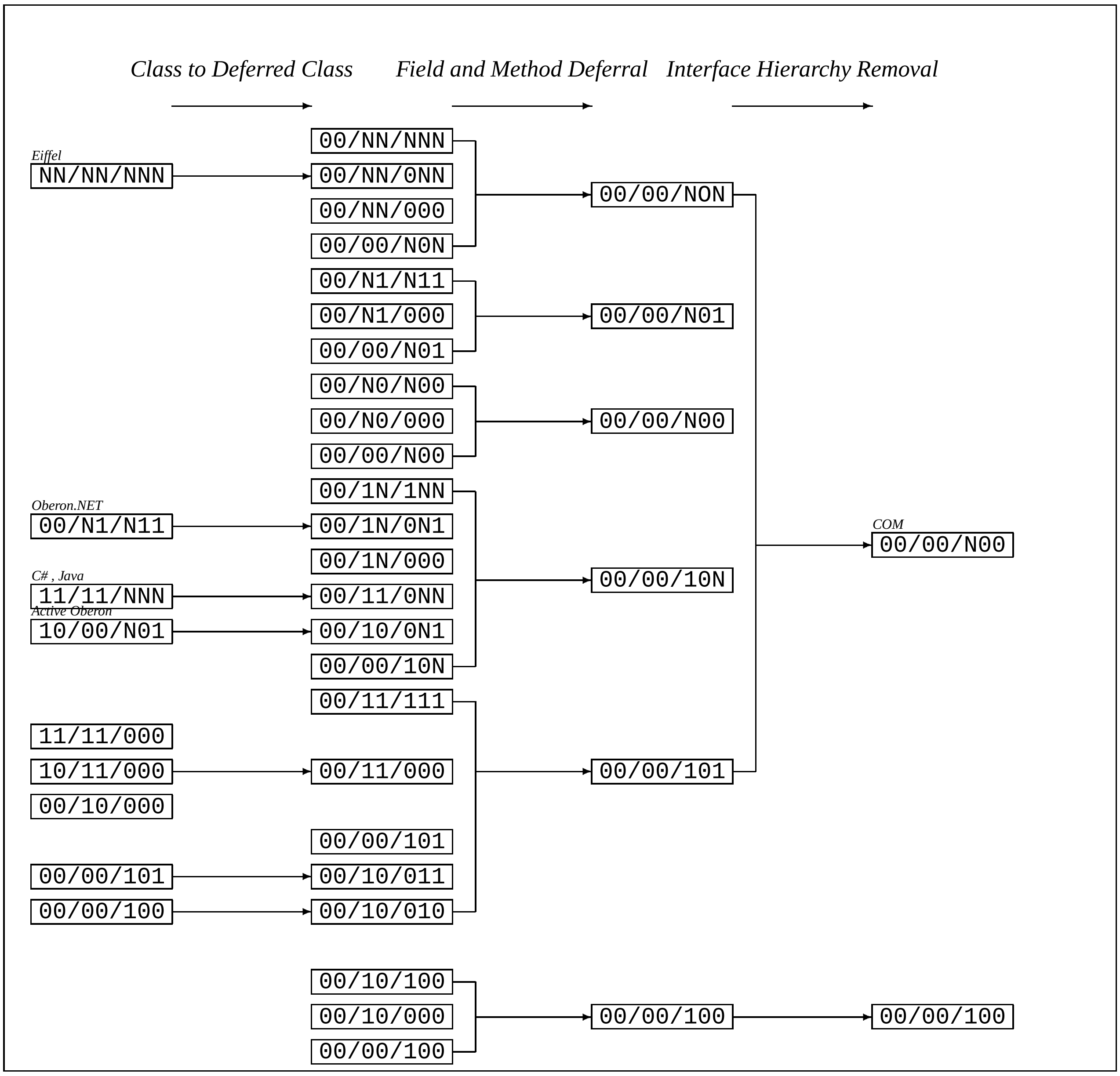


Figure 3.2: Object Model Conversions

consider them equivalent. In fact, they are the object models with the highest and the lowest abstraction level: the first one is the most expressive object model, the latter one the most flexible. In terms of our layered system, they are on different layers.

Thus the question about which object model flavor is the best one, or the most generic one, is in fact a wrong or underspecified question. Both models can be offered by an execution environment to support an object-oriented language. The minimalistic object oriented environment must support at least classes implementing multiple interfaces. In a single-language environment there is no semantic loss, the only difference is that the enforcement of the consistency is moved from the platform model to the compiler.

In a platform designed to support interoperability between languages, the problem is different, as the platform object model defines the primitives that are used to interoperate, and thus every language complying with the platform must understand and support them. This doesn't mean that the platform object model must be a subset of the model with lowest abstraction level among the supported languages, only that every compiler must be able to map the common types to the language types. Languages with a higher object model can still be mapped to the common model to implement their model, interoperability is only possible through the lower model. Thus, the interface of a component written to interoperate is restricted to the common object model.

A solution to this limitation is the use of *multiple interoperability platforms having different abstraction levels*. The platforms map their model to the lower one, but at the same time enforce the consistency of their own model. As an example, we can imagine a multi-platform with three layers: `NN/NN/NNN`, `11/11/NNN`, and `00/00/N00`; as the object models are mapped to the lower layers, every language will be able to use the data defined even in the highest layer, and the higher layers will have access to the data defined in the lower layers. This will also allow languages with a higher object model to interoperate together without semantic loss.

3.6.2 Recapitulation

After the detailed explanations of the previous sections, we are now ready to answer the questions set upon entering the chapter.

What are the requirements for a language interoperability platform?

- support for shared common metadata
- a programming model reflecting the application domain of the system
- every language type should be mappable to the platform type system[11]
- a type system for describing the interfaces for interoperating among the various languages

[11]strict equivalence is not required

What properties shall the interoperability type system have?

- must be mappable (with a bijective mapping) to a subset each language's type system.
- the common type system must be a subset of the platform type system

What properties shall a language type system have?

- the language's type system must be mappable to the platform type system
- a subset of the language type system should be mappable (with a bijective function) to the interoperability type system[12]

Which common programming model and type system are the best one?

- the one that matches the application domain

Are there equivalent models?

- it is possible to convert most of the object models into each other (but this may introduce inefficiencies)
- the inheritance multiplicity in an object model cannot be reduced
- restricting an object model moves the model enforcement from the model to the compiler

3.7 The Jaos Approach

The Jaos project's first goal was to have a JVM on top of the Aos-Kernel, to prove that the kernel is powerful enough to support languages and systems other than Oberon. This was to be achieved by reusing all the conventions, functionalities and API provided by the kernel, in particular the support for object-oriented languages, and the memory management with garbage collection.

While implementing the mechanism for linking the Java native methods, it became clear that language interoperability could be another application of Jaos: Oberon and Java are based on similar programming models, the compiled code follows the same conventions; the only missing element was the automated sharing of the metadata information[13]. We thus decided to let the Oberon compiler and Jaos use the same symbol table, augmented by a few functionalities, to make automated metadata sharing possible.

Jaos falls under the interoperability systems built on top of a common platform shared among many languages. Complexity and implementation is moved from the compiler to the platform offering the common services. Focus was set on the maximal reuse of available components.

The key points in Jaos' design are:

[12] interoperability is not mandatory

[13] like other systems, sharing was done by the means of a stub generator

- built on top of a high-level run-time environment with support for object-oriented features, memory management with garbage collection, and thread support
- common metadata repository to share information; each compiler or virtual machine can provide a metadata loader plug-in to load metadata into the repository on request
- installable loaders; for each object-file format an installable loader and linker plug-in can be installed. The loaded components have to carry some additional metadata to make dynamic linking possible

This framework is very flexible; it allows sharing information without forcing every compiler to change its symbol-file and object-file format, allowing loading and use of compilation-units compiled for other systems.

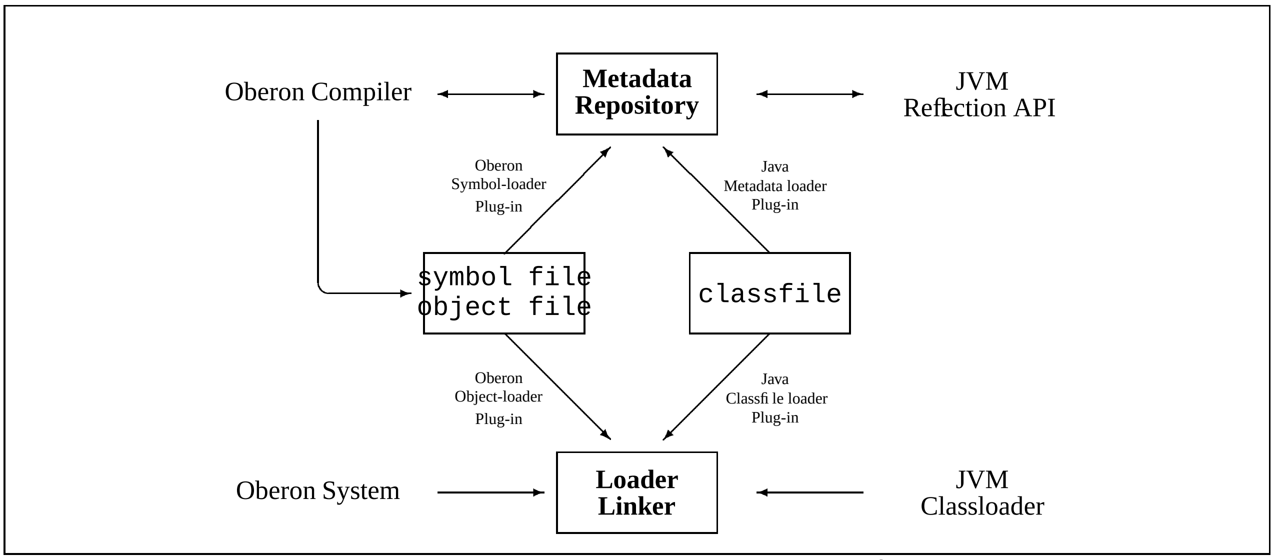


Figure 3.3: The Jaos Interoperability Platform

3.7.1 Run-time environment

The run-time environment provided by the Aos kernel is targeted at strongly typed, garbage collected, object-oriented and imperative languages.

The type descriptor structure associated with every dynamically allocated object provides a single virtual method table and information to provide run-time type checks; more method tables can be bound to a class through the interface mechanism; array structures always carry the array length for boundary checking.

The memory management allocates objects on the heap and a garbage collector performs automatic storage reclamation with object finalization.

Tasks are bound to object instances; the kernel provides efficient object protection against concurrent access and thread synchronization by the means of conditions.

3.7.2 Metadata Repository

Section 5.2 presents a more detailled description of the metadata repository; here only a short overview is given.

The metadata repository consists of the Oberon compiler's symbol table improved for this purpose with method overloading and support for recursive import needed by Java; the repository is made compiler independent, and serves as symbol table for the Oberon compiler and the JVM reflection mechanism used by the Java JIT-Compiler, class loader, and class linker.

Through an installable plug-in mechanism, metadata loaders can be provided. This makes the repository independent from the metadata storage and allows to access information stored in different formats (currently Oberon symbol-files and Java class-files) without prejudice of format.

The type system provided by the metadata repository is the Active Oberon model. Oberon has a richer, although slightly less abstract type system[14]. A few conflicts are caused by the different and non-orthogonal handling of the compilation-unit and namespace concepts in both languages: in Oberon, a module may contain many classes, whereas in Java there's only one class per compilation-unit. On the other hand, defining modules to be equivalent to packages would have caused even more problems. The detailed discussion about the mapping is in Section 7.2.1.

The type system used for interoperability is the intersection of the Oberon and the Java type systems. It includes all types that are common to both languages: integers, floating points, classes, and interfaces. A problem was posed by the different character set.

3.7.3 Loader and Linker

The dynamic loader and linker upon request loads object-files into the memory and resolves the external references to other compilation-units.

Loaders can be installed at run-time through a plug-in mechanism. This allows using more than one object-file format at the same time.

The kernel has a structure for loaded compilation-units, which contains not only data and code, but also linking information in the form of entries. Loaders must fill the entry list, while linkers search it to find the actual address of an entry. The entries consist of a fingerprint describing the entry (produced by the metadata repository) and the actual address of the entry.

[14]e.g., there are four different kinds of arrays in Oberon, which differ in allocation and detail, whereas Java as only one array type which is the generalization of the concept

4

Active Objects in Practice: the Active Oberon Language

This chapter describes the Active Oberon extension to the Oberon language; this extension is currently implemented and used in the Aos system [Mul02]. The extension introduces concurrency in the language and an improved component modeling by the means of its object model; it differs from the further language extension introduced in the Oberon.Net project by Gutknecht [Gut01a, Gut01b].

4.1 Introduction

4.1.1 History and Related Work

Programming language development at ETH Zurich has a long reaching tradition. The Oberon language is the latest descendant of the Algol, Pascal, and Modula family. Pascal [JW74] was conceived as a language to express small programs; its simplicity and leanness made it particularly well-suited for teaching programming. Modula [Wir77] evolved from Pascal as a language for system programming, and benefited from the practical experience gained during the development of the Lilith workstation [Ohr84] and of the Medos operating system [Knu83]. The need to support the programming-in-the-large paradigm was the motivation for Oberon [Wir88]. The Ceres [Ebe87] and Chameleon [HP92] platforms, and the Oberon operating system [GW92] projects were developed in parallel with the language, and allowed to test and evaluate the language improvements.

Many experimental language extensions have been proposed for Oberon at, and outside of the ETH. Object Oberon [MTG89], Oberon-2 [MW91], and Froderon [Fr¨o97] explored adding further object-oriented features to the language; Oberon-V [Gri93] proposed additions for parallel operations on vector computers; Oberon-XSC [Jan98] added mathematical features to support scientific computation; module embedding [Rad98] was also proposed. Concurrency was first added to the operating system through a specialized system API in Concurrent Oberon [SL94] and XOberon [Bre95]; attempts to model concurrency in the language itself were also done by Radenski [Rad95].

Active Oberon is the first exponent of a new generation of languages in this family. Our motivation is to support concurrency and component modeling in the language in a clean, seamless way.

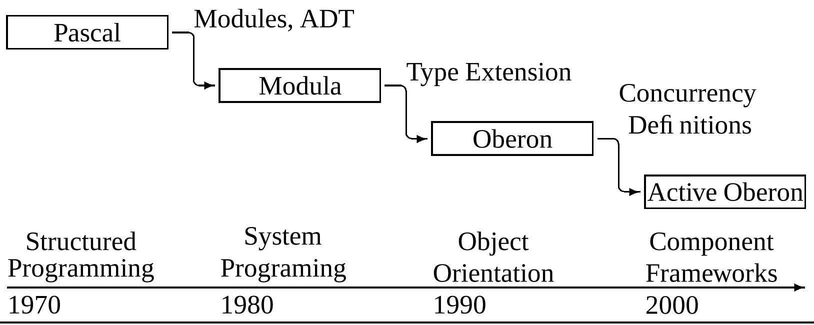


Figure 4.1: The Pascal Language Family Evolution

4.1.2 The Active Object Model

Active Oberon concurrency's model is based on active objects. Active Objects are objects that include an own thread of control. In other words, each object carries an own independent activity, which is executed concurrently to the other objects activities. The new concepts are integrated into the language by generalizing the class construct to include concurrency. Passive classes, as used in Oberon, are a special case of Active Oberon classes.

The support for concurrency requires the language to address atomicity, synchronization and activity. Atomicity ensures that several operations are executed without interruption, as if they were only a single instruction. Synchronization forces the activity to wait for a condition to be established. Activity is the code concurrently executed by a thread.

The question a language designer usually faces is, shall concurrency be added through a library (thus providing an optional implementation) or embedded into the language? A library is a collection of useful abstractions built using the language itself, and is at first sight a comfortable place to add things that don't belong to the language, but can be constructed using it. Many language designers made this choice, including [SL94] for Oberon. This can be an approach for experimenting with concurrency, but on the long run is a dead end: concurrency is a metaphor that cannot be mapped well to an existing non-concurrent model, but rather requires a new model that can be provided only changing the language model.

Providing atomicity through a library (usually with a lock and unlock call) is not satisfactory: atomicity is to be used in a structured way, by wrapping a block of instructions, as for every lock there must be an unlock; a library call will never be able to enforce this property. The library approach provides an imperative view, possibly wrapped in object-oriented cloths, but with insufficient abstraction. This is obviously not desirable.

As an example, Java tries to partially solve this problem by embedding atomicity in the language with synchronized methods and synchronized statement blocks (here synchronized in fact means a critical section, i.e. atomicity), but leaving synchronization and activity in the libraries. Libraries are supposed to be a collection of useful additions to the language, but the language should be independent of the libraries, and even able to exist with a different set of libraries. This choice makes matters even worse: why should the language provide atomiticy, if it doesn't define (know about) concurrency? For the sake of consistency, concurrency should be addressed either in the language or in the libraries, but not in both. C# instead is completely concurrency agnostic; atomicity, synchronization, and activities are provided by the libraries.

Believing now that the language model must support concurrency, we think that the active object model is the most suitable for our language.

The model must have the following properties:

1. simple to understand (concurrency is already hard enough)
2. atomicity must be a structured construct.
3. objects define the atomicity's scope, because objects are the unit of reasoning.
4. synchronization must be structured.
5. the condition for synchronization must be explicit.
6. activities must be bound to objects.

Atomicity

A statement block can require atomic execution, thus atomicity must encompass a statement block. This enforces the structured property of atomicity. The granularity can range from one single statement to a whole procedure.

Because every atomic region in an object will modify the state of the object it is in, it makes sense to group all the atomic regions in the same instance in monitor-like way.

Synchronization

Objects are our first-class citizens, and since the whole model deals with their state, it makes sense to model synchronization on the object state itself. In this way, both above properties are fulfilled: the program specifies on which object state it is waiting for; using a single statement (instead of a pair like wait and notify), the construct is automatically structured, and the condition explicit.

Activity

Every object class can define its own activity in the class body. When an object is instantiated, the body is automatically executed in a concurrent thread.

4.1.3 Language Design

The design of Active Oberon was influenced by the experiences made with Object Oberon and Oberon-2. We follow the Object Oberon notation for declaring classes and methods, because we think it is more expressive than the one in Oberon-2: methods belong to the class scope and therefore they must be declared there; this way, other methods and fields belonging to the record scope can be accessed without explicit qualifier. Protection against concurrent access through the EXCLUSIVE modifier is easier to read if the methods are declared in the same scope. Active Oberon departs from the Object Oberon design, in that records are generalized to be both classes and records, instead of letting classes and records co-exist in the same system. Another important difference is the decision to let the compiler handle forward references. The syntax of Object Oberon and Oberon-2 was designed to simplify the compiler construction, whereas we chose to simplify the programmer's task by avoiding the needless redundancy of forward declarations and declarations, leaving it to the compiler to handle them.

Java [GJS96] and C# [Cor01] share some similarities with Active Oberon. They are both object-oriented languages stemming from the imperative language world, and the concurrency protection mechanism with object instance bound monitors is the same. On the other hand, they both put the accent on the object-orientation in such an extreme manner, that class methods and class fields seem just special cases of the instance fields and methods, because they belong to a class namespace. Furthermore, Java has no support at all for statically allocated structures: everything is dynamic, even user-defined constant arrays; for this reason, to perform at an acceptable speed, Java programs must rely on complicated and expensive compiler optimizations. All languages in the Oberon family treat modules and classes as orthogonal concepts, each has its own scope; the module semantics is different from a class' semantics as shown in [Szy92] [1]: modules group static components and related implementations, and provide a deployment and structuring primitive. In fact, Java and C# had to introduce concepts like packages, namespaces and assemblies, that de facto reintroduce modules with just another name. We think that static structures and modules still have a very valid reason to be part of a programming language.

The AWAIT statement was proposed and investigated in [BH72] who showed its conceptual simplicity and elegance, but also thought it would be impossible to implement it efficiently. We repropose it in Active Oberon, with the conviction that it is a real improvement compared with signals and semaphores, because of the unification and clarity it brings; it becomes particularly obvious in an object-oriented programming style, where signals and semaphores are completely inappropriate because of their unstructured use, as they can be inserted arbitrarily in a program. Pieter Muller's thesis [Mul02] proves, that with the appropriate restrictions, AWAIT can be implemented efficiently. The language Ada 95 [Int95] has also a construct called *barriers*, which is semantically very similar to Active Oberon's AWAIT, although with a coarser procedure-width granularity.

Concurrent Oberon was a first attempt to provide concurrency in an Oberon system. It was

[1]as a comparison, B. Meyer advocates exactly the opposite [Mey97]

done through a specialized API defining a Thread type and functions to create, stop and resume execution. Protection was achieved through a single global system lock. There was no provision for a synchronization primitive. This model is too weak to support Active Oberon, because locks and synchronization are tightly bound (when synchronization is performed, locks are released), and the locking mechanism is too coarse; a single lock would make a multiprocessor-based system—where many threads run at the same time—useless.

4.1.4 General Remarks

Components are the latest technique for coping with programming-in-the-large problems that reach across subsystems boundaries. Programming languages have undergone many changes to address the evolution in the scope of the problem to be solved. Structured programming tried to make the implementation of algorithms easier; Abstract Data Structures addressed the combination of data structures in a first step toward programming-in-the-large; then came object orientation, which improved the way to model systems; now we have component frameworks to combine software systems together.

This shows that, where components are used, the implementation of the internals can be done using object orientation, and the implementation of the single objects and deployment units is done using the concepts of structured programming and typing. Since each paradigm comes with additional costs for the augmented flexibility, the choice of the implementation technique is driven by the trade-off between efficiency and generality. In theory, nothing speaks against the use of components everywhere and this would make sense because it is the most generic approach and can be considered a generalization of the other paradigms; only the practical impact on program efficiency can be used as an argument for using the other techniques.

For this reason, we think that all these paradigms have their place and should co-exist in a multi-purpose language, as the language may be used to address problems of different size and scope.

The step taken in Oberon.Net [Gut01a] to completely discard sub-classing and to offer only definitions is very courageous because it only relies on the most general concept in the language, and all others are shown to derive from it (although by removing them, a comparison is not possible anymore). The cost of using only interface calls instead of method calls should be investigated; we think this will not limit the applications written with Oberon.Net in any way, as the language will not be used for low-level time-critical system implementation (the .Net platform is already implemented). On the other hand, Active Oberon has to address this problem, as the whole Aos kernel is implemented using it, and efficiency is an issue there.

The design objectives of the Active Oberon's extension were to generalize the language in a way that a) the new concepts would fit into the language, and b) the old concepts would become special cases of the new ones. Generalization is a very useful tool, because the language is enriched with more expressive tools, whereas the complexity of the language does not grow much: an old concept is substituted by a new one that is richer and more powerful. The same process is well known in other sciences like physics, where sometimes a new law unifies a couple of old ones. The new, unique law precisely describes the physical world and removes the old ones, now superseded. Even if the new law is more complex than the old laws, the simplification due to the use of only one law instead of many has a beneficial effect. Computer science often is confronted with the desire to add as many features as possible without trying to generalize the currently available ones, which raises the complexity of many systems to an intolerable level. Generalization permits to add features without increasing complexity and thus allows to create bigger and more powerful systems.

4.2 Object Oriented Extensions

4.2.1 Pointer to Anonymous Record Types

```
TYPE
  (* examples of pointer to record types *)

  (* pointer to named record type *)
  Tree = POINTER TO TreeDesc;
  TreeDesc = RECORD  key: INTEGER;  l, r: Node  END;

  (* pointer to anonymous record types *)
  Node = POINTER TO RECORD  key: INTEGER;  next: Node  END;
  DataNode = POINTER TO RECORD  (Node)  data: Data  END;
  DataTree = POINTER TO RECORD (Tree)  data: Data  END;
```

The types `Node` and `DataNode` are *pointers to anonymous record* types; `Tree` is a *pointer to named record* type.

These types can only be dynamically instantiated (with NEW), no static instance is possible; this is enforced by the fact that the record type is anonymous and it is not possible to declare a variable having that type.

Both `RECORD` and `POINTER TO RECORD` types are allowed as base types of pointer to anonymous record type; a record type cannot extend a pointer to an anonymous record, thus preserving the property of allowing only dynamic instances.

4.2.2 Object Types

```
TYPE
  (* object types *)
  DataObj = OBJECT  VAR data: Data;  l, r: DataObj  END DataObj;
```

The type `DataObj` is an *object type*.

The type `OBJECT` has a syntax different from the `POINTER TO RECORD` type; it must match the `[ DeclSeq ] Body` production instead of the `FieldList` production. This implies that procedures can be declared inside an object type. We call them methods or type-bound procedures.

Only an object type can extend another object type. Object types must be dynamically instantiated with NEW, subject to the rules imposed by initializers (Section 4.2.4).

4.2.3 Type-bound Procedures

```
TYPE
  Coordinate = RECORD x, y: LONGINT END;
  VisualObject = OBJECT
    VAR  next: VisualObject;

    PROCEDURE Draw*;    (*draw this object*)
    BEGIN HALT(99); (*force extensions to override this method*)
    END Draw;
  END VisualObject;

  Point = OBJECT (VisualObject)
    VAR  pos: Coordinate;

    PROCEDURE Draw*;    (*override Draw method declared in VisualObject*)
    BEGIN  MyGraph.Dot(pos.x, pos.y)
    END Draw;
  END Point;

  Line = OBJECT (VisualObject)
    VAR pos1, pos2: Coordinate;

    PROCEDURE Draw*;
    BEGIN  MyGraph.Line(pos1.x, pos1.y, pos2.x, pos2.y)
    END Draw;
  END Line;

VAR
  objectRoot: VisualObject;

PROCEDURE DrawObjects*;
VAR  p: GraphObject;
BEGIN
  (* draw all the objects in the list *)
  p := objectRoot;
  WHILE p # NIL DO  p.Draw;  p := p.next  END;
END DrawObjects;
```

Procedures declared inside an object are called *type-bound procedures* or *methods*. Methods are associated with an instance of the type and operate on it; inside a method implementation, if another method is visible using the Oberon scope rules, it can be accessed without qualification.

A method can overwrite another method of the same name inherited from the base type of the record, but it must have the same signature. The visibility flag is part of the signature.

Given an object instance o of type T with type-bound procedures P and Q, $o.P$ is the call to the method P in the context of o. Inside any method of T, another method can be called with Q (no specification of the self object is required). A method P can call the method it overrides in its superclass with the notation $P\uparrow$. Supercalls are legal only inside a method.

4.2.4 Initializers

A method tagged with & is an *object initializer*. This method is automatically called when an instance of the object is created. An object type may have at most one initializer. If present, it is always public, and can be called explicitly, like a method; if absent, the initializer of the base type is inherited. An initializer can have a signature differing from the inherited initializer from the base object type, in which case it must have a different name too.

If an object type *T* has or inherits an initializer *P* with signature (p0: T0; .... pn: Tn), then the instantiation of a variable *o:T* by `NEW` requires the parameters needed by the initializer: `NEW(o, p0, ..., pn)`. The initializer is executed atomically with NEW.

```
TYPE
  Point = OBJECT (VisualObject)
    VAR  pos: Coordinate;

    PROCEDURE & InitPoint(x, y: LONGINT);
    BEGIN  pos.x := x;  pos.y := y
    END InitPoint;
  END Point;

  PROCEDURE NewPoint(): Point;
  VAR  p: Point;
  BEGIN  NEW(p, x, y);  (*calls NEW(p) and p.InitPoint(x, y) *)
    RETURN p
  END NewPoint;
```

4.2.5 SELF

The keyword `SELF` can be used in any method or any procedure local to a method of an object. It has the object type and the value of the current object instance the method is bound to. It is used to access the object whenever a reference to it is needed or to access a record field or method when shadowed by other symbols, i.e. fields that are hidden by a local variable with the same name.

```
TYPE
  ListNode = OBJECT
    VAR data: Data;  next: ListNode;

    PROCEDURE & InitNode (data: Data);
    BEGIN
      SELF.data := data;    (* initialize object data *)
      next := root; root := SELF   (* prepend node to list *)
    END InitNode;
  END ListNode;

VAR
  root: ListNode;
```

4.2.6 Delegate Procedure Types

```
TYPE
  MediaPlayer = OBJECT
    PROCEDURE Play; .... play a movie .... END Play;
    PROCEDURE Stop; .... stop movie .... END Stop;
  END MediaPlayer;

  ClickProc = PROCEDURE {DELEGATE};
  Button = OBJECT
    VAR
      onClick: ClickProc;
      caption: ARRAY 32 OF CHAR;
    PROCEDURE OnClick;
    BEGIN  onClick  END OnClick;

    PROCEDURE & Init(caption: ARRAY OF CHAR;  onClick: ClickProc);
    BEGIN  SELF.onClick := onClick; COPY(caption, SELF.caption)
    END Init;
  END Button;

PROCEDURE Init(p: MediaPlayer);
VAR  b0, b1, b2: Button;
BEGIN
  (* Reboot -> call system reboot function *)
  NEW(b0, "Reboot", System.Reboot);

  (* MediaPlayer UI: bind buttons with player instance *)
  NEW(b1, "Play", p.Play);
  NEW(b2, "Stop", p.Stop);
END Init;
```

Delegate types are similar to procedure types; they are compatible to both methods and procedures, while procedure types are only compatible with procedures.

Delegate procedure types are annotated with the `DELEGATE` modifier. Both methods and procedures can be assigned to a delegate. Given a variable *d* with delegate type *t* , *o* an object instance, and *M* a method bound to *o*, it is allowed to assign *o.M* to *d* if the method *M* and *t* have the same signature. The object self-reference is omitted from the procedure type signature. Whenever *d* is called, the assigned object *o* is implicitly passed as self-reference. Assignment and call of procedures remains compatible with the Oberon definition.

4.2.7 Definitions

A *Definition* is a syntactic contract[2] defining a set of method signatures. A definition *D0* can be *refined* by a new definition *D1*, which will inherit all methods declared in *D0*. Definitions and their methods are globally visible. An object can implement one or more definitions, in which case it commits itself to give an implementation to all the methods declared in the definitions.

```
DEFINITION Runnable;
  PROCEDURE Start;
  PROCEDURE Stop;
END Runnable;

DEFINITION Preemptable REFINES Runnable;
  PROCEDURE Resume;
  PROCEDURE Suspend;
END Preemptable;

TYPE
  MyThread = OBJECT IMPLEMENTS Runnable;
    PROCEDURE Start;
    BEGIN .... END Start;
```

[2][BJPW99] describes four levels of contracts: 1. syntactic contracts (type systems), 2. behavioral contracts (invariants, pre- and post-conditions), 3. synchronization contracts, 4. quality of service contracts

```
    PROCEDURE Stop;
    BEGIN .... END Stop;
  END MyThread;
```

The keyword IMPLEMENTS is used to specify the definitions implemented by an object type. An object type can implement multiple definitions.

Definitions can be thought to be additional properties that a class must have, but that are orthogonal to the object type hierarchy. A object's method can be invoked through the definition, in which case the run-time checks if the object instance implements the definition and then invokes the method; if a definition is not implemented by the object instance, a run-time exception occurs.

```
  PROCEDURE Execute(o: OBJECT; timeout: LONGINT);
  BEGIN
    Runnable(o).Start;
    Delay(timeout);
    Runnable(o).Stop;
  END Execute;
```

4.3 Concurrency Support

4.3.1 Active Objects

The declaration of an object type may include a `StatBlock`, called the *object body*. The body is the object's activity, to be executed whenever an object instance is allocated after the initializer (if any) completed execution; the object body is annotated with the `ACTIVE` modifier. At allocation, a new process is allocated to execute the body concurrently; the object is called an *active object*.

If the `ACTIVE` modifier is not present, the body is executed synchronously; NEW returns only after the body has terminated execution.

The system holds an implicit reference to an active object as long as the activity has not terminated to prevent garbage collection of the object. The object can live longer than its activity.

```
TYPE
  (*define the object and its intended behavior*)
  Object = OBJECT

  BEGIN {ACTIVE}  (*object body*)
    ... do something ...
  END Object;

PROCEDURE CreateAndStartObject;
VAR  o: Object;
BEGIN
  ... NEW(o); ...
END CreateAndStartObject;
```

The active object activity terminates whenever the body execution terminates. As long as the body executes, the object is kept alive (i.e. cannot be garbage collected). After that, the object becomes a passive one, and will be collected according to the usual rules.

4.3.2 Protection

A *Statement Block* is a sequence of statements delimited by `BEGIN` and `END`. It can be used anywhere like a simple statement. It is most useful when used with the `EXCLUSIVE` modifier to create a critical region to protect the statements against concurrent execution.

```
PROCEDURE P;
VAR x, y, z: LONGINT;
BEGIN
  x := 0;
  BEGIN
    y := 1
  END;
  z := 3
END P;
```

An object can be viewed as a resource and various activities may potentially compete for using the object or for exclusive access to the facilities it provides; in such cases some kind of access protection is essential. Our protection model is an instance-based monitor.

```
(* Procedures Set and Reset are mutually exclusive*)
TYPE
  MyContainer = OBJECT
    VAR  x, y: LONGINT;    (* Invariant:  y = f(x) *)

    PROCEDURE Set(x: LONGINT);
    BEGIN {EXCLUSIVE}    (* changes to both x and y are atomic *)
      SELF.x := x;  y := f(x)
    END Set;

    PROCEDURE Reset;
    BEGIN
      ...
      BEGIN {EXCLUSIVE}    (* changes to both x and y are atomic *)
        x := x0;  y := y0;
      END;
      ....
    END Reset;
  END MyContainer;
```

Every object instance is protected and the protection granularity is any statement block inside the object's method, ranging from a single statement to a whole method. A statement block can be protected against concurrent access by annotating it with the modifier `EXCLUSIVE`. Upon entering an exclusive block, an activity is preempted as long as another activity stands in an exclusive block of the same object instance.

An activity cannot take an object's lock more than once, re-entrancy is not allowed.

Every module is a special *singleton instance* object, thus its procedures can also be protected. The scope of protection is the whole module, like in a monitor [Hoa74].

4.3.3 Synchronization

```
TYPE
```

```
Synchronizer = OBJECT
  awake: BOOLEAN

  PROCEDURE Wait;
  BEGIN {EXCLUSIVE} AWAIT(awake); awake := FALSE
  END Wait;

  PROCEDURE WakeUp;
  BEGIN {EXCLUSIVE} awake := TRUE
  END WakeUp;
END Synchronizer;
```

The built-in procedure `AWAIT` is used to synchronize an activity with a state of the system. `AWAIT` can take any boolean *condition*; the activity is allowed to continue execution only when *condition* is true. While the condition is not established, the activity remains *suspended*; if inside a protected block, the lock on the protected object is released, as long as the activity remains suspended (to allow other activities to change the state of the object and thus establish the condition); the activity is restarted only if the lock can be taken.

The system is responsible for evaluating the conditions and for resuming suspended activities. The conditions inside an object instance are re-evaluated whenever some activity leaves a protected block inside the same object instance. This implies that changing the state of an object outside a protected block won't have the conditions re-evaluated.

When several activities compete for the same object lock, the activities whose conditions are true are scheduled before those that only want to enter a protected region.

Appendix A.1.6 shows the synchronization inside a shared buffer.

4.4 Other Language Extensions

This section describes a few minor changes made to better integrate the extensions into the language.

4.4.1 Declaration sequence and forward references

In Active Oberon, the definition scope of a symbol ranges over the whole block containing it. This implies that a symbol can be used before being declared, and that names are unique inside a scope.

4.4.2 HUGEINT

The 64-bit signed integer type HUGEINT has been added to the language. It fits into the numeric type hierarchy as follows:

$$\mathrm{LONGREAL} \supseteq \mathrm{REAL} \supseteq \mathrm{HUGEINT} \supseteq \mathrm{LONGINT} \supseteq \mathrm{INTEGER} \supseteq \mathrm{SHORTINT}$$

Table 4.1 shows the new conversion functions added for HUGEINT.

No new constant definition is needed; constants are typed according to their value.

Name	*Argument Type*	*Result Type*	*Function*
SHORT(*x*)	HUGEINT	LONGINT	identity (truncation possible)
LONG(*x*)	LONGINT	HUGEINT	identity
ENTIERH(*x*)	real type	HUGEINT	largest integer not greater than x

***Table 4.1**: New Type Conversion Procedures*

4.4.3 Untraced Pointers

Untraced pointers are pointers that are not traversed by the garbage collector. A structure or object referenced only through an untraced pointer may be collected at any time.

Untraced pointers are defined using the UNTRACED modifier.

```
TYPE  Untraced = POINTER {UNTRACED} TO T;
```

4.4.4 IA32 Specific Additions

The functions in Table 4.2 have been added to the Intel IA32 version of the compiler.

The `PUTx` and `GETx` have been added for security sake, to cope with untyped constants.

Name	*Function*

PUT8(adr: LONGINT; x: SHORTINT)
PUT16(adr: LONGINT; x: INTEGER)
PUT32(adr: LONGINT; x: LONGINT)
PUT64(adr: LONGINT; x: HUGEINT)
	Mem[adr] := x

GET8(adr: LONGINT): SHORTINT GET16(adr: LONGINT): INTEGER GET32(adr: LONGINT): LONGINT GET64(adr: LONGINT): HUGEINT	RETURN Mem[adr]

PORTIN(port: LONGINT; x: AnyType)
PORTOUT(port: LONGINT; x: AnyType)
	x := IOPort(port)
	IOPort(port) := x

CLI STI	disable interrupts enable interrupts
 EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP AX, BX, CX, DX, SI, DI AL, AH, BL, BH, CL, CH, DL, DH	PUTREG/GETREG constants 32-bit register 16-bit register 8-bit register

***Table 4.2**: IA32 Version Additions to SYSTEM*

4.4.5 Miscellaneous

A few Oberon-2 extensions have been adopted in Active Oberon:

- ASSERT
- FOR

- read-only export
- dynamic arrays

Pointer variables are automatically initialized to NIL.

4.5 Interoperability Additions

Supporting interoperability with other languages requires the ability to access and use their features in Oberon. These extensions are not defined in the language, but are implemented in the compiler ad-hoc. We use the modifier syntax to enable these features in the compiler whenever needed.

The `OVERLOADING` modifier enables overloading in a module. Procedures with the same name can be declared in a block, as long as their respective signatures differs. Procedure calls are resolved according to the parameters[3].

[3]This concession does not imply that we agree with overloading. We consider overloading a bad feature to avoid, and are considering an alternative implementation avoiding overloading along the lines of [Mey99]

5 Case Study: The Active Oberon Compiler

Smart data structures and dumb code
works a lot better than the other way around

— E. Raymond

5.1 The Compiler Framework

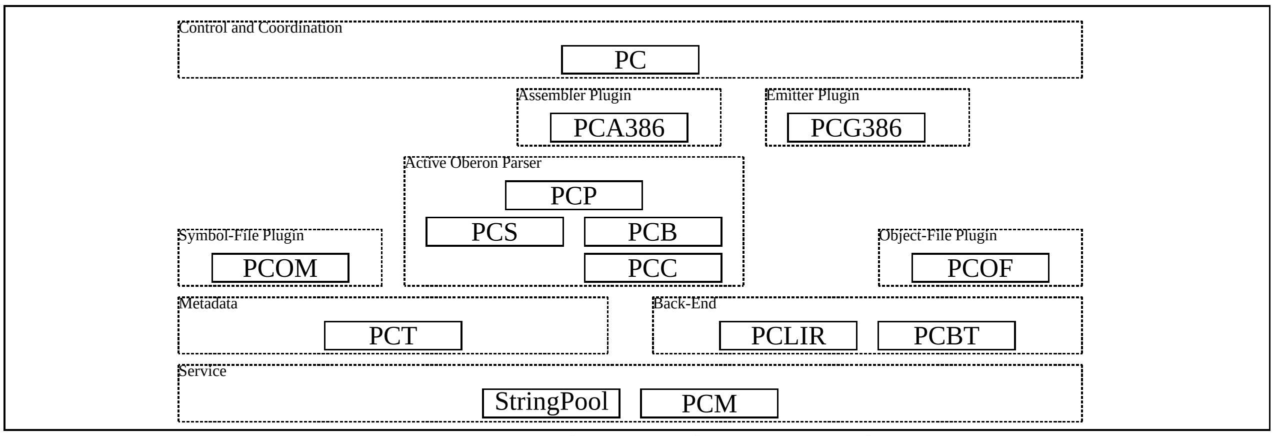


Figure 5.1: The Compiler Framework

The Active Oberon Compiler framework is composed of core components and installable components (plug-ins). The core components implement the architecture independent features of the compiler, whereas the plug-ins can be dynamically installed into the framework to re-target the compiler for other architectures, systems, or standards.

The *Service* component contains functionalities used by all components in the framework. `StringPool` stores and handles strings of any length; `PCM` provides an abstraction layer for the I/O operations.

The *Metadata* component is an extended symbol-table and repository for symbol-table information. A detailed description can be found in Section 5.2.

The *Back-End* component defines an assembly-like low-level intermediate code representation (LIR) and structures for data allocation, fixup lists (for the object file), and system calls. A detailed description can be found in Section 5.4.

The *Parser* component performs lexical and semantic analysis, and translates the parsed code into the LIR defined by the Back-End Component. The scopes are parsed concurrently. A detailed description can be found in Section 5.3.

The *Code Generator* plug-in component translates the LIR code into machine code. A description of the Plug-ins can be found in Section 5.5.1 (Intel IA32) and 5.5.2 (Strong-ARM).

The *Assembler* plug-in component can be installed into the Parser to add assembler language parsing support for the target architecture.

The *Symbol-File* plug-in component handles the serialization of the Symbol Database entries, implementing the persistence of the data; it is plugged into the Symbol Database. Symbol-files store the metadata information of a compilation unit in a compact form for separate compilation[1].

The *Object-File* plug-in component is installed in the Back-End component; it stores in an object-file the generated code and the fixup lists for each external symbol for the linking.

5.2 Metadata Repository

5.2.1 Introduction

Software is used to manipulate data; the way this data is organized is also a kind of data, but on a higher abstraction level: it is called *metadata*. Obviously, one program's data can be another one's metadata: a compiler is an example of a program whose data—variables, methods, and types—corresponds to the generated application's metadata. The reflection mechanism provided in some modern runtime environments is another example of a program dealing with metadata; the reflection provides information about which classes are available, what fields and methods do they contain, and so on.

The information provided by the metadata often depends on the use and context in which the metadata iself is used, and on the kind of data used. In a runtime environemnt, the metadata consists of the information used by the compiler's symbol tables and by the reflection API. In this case, the metadata includes some obvious attributes like the name, size, data type, and location; additional information may include the context where the data is found. The type system used to categorize the data is an important part of the metadata, because it defines the values and meanings of the data models available to the user. In most cases, the type system is customizable by the user, which can use some templates provided by the system (e.g. arrays or records) to create new types appropriated to his own needs.

[1]some compilers simply use the source code for this purpose, but this takes more space and the code has to be checked for correctness

The *metadata repository* is the collection of metadata, and of operations on the metadata collection. Operations include metadata maipulation, creation of new metadata, export and import of metadata (which implies a serialization format), and some basic consistency checks.

A metadata repository can have many uses, in particular as symbol-table for a compiler, or for performing reflection on a system and for metaprogramming.

The rest of this section presents the programmer's interface of the metadata repository implemented by the `PCT` module.

5.2.2 Type System

The metadata repository provides the same type system as the Active Oberon's language. Information is structured and stored using three object classes (and their extensions): structures, symbols, and scopes.

Structures represent a type; thus, they define the semantic of a value, in particular the encoding and the legal operations on it. The class `Struct` is the root class for all structures.

The primitive built-in structures are defined in the `Basic` class; predefined instance of this class correspond to the primitive types of the type system and are `Byte`, `Boolean`, `Char8`, `Char16`, `Char32`, `Int8`, `Int16`, `Int32`, `Int64`, `Float32`, `Float64`, `Set`, `Ptr`. These types are known throughout the whole compiler.

Used-defined types are created using the `Array`, `Record`, `Pointer`, and `Delegate` classes. *Arrays* aggregate elements of the same type, and can be statically or dynamically sized. *Records* aggregate states of any type, and can further be an extension of another record, inheriting type, states and functions (subclassing). *Delegates* define the interface of a routine. *Pointers* are dynamically heap-allocated structures[2].

Records, classes, and interfaces are special cases of the record structure. The symbol table automatically enforces the restrictions associated with each special case. In particular, classes and interfaces are implemented as pointer to records, interface's methods have no implementation, and subclasses of classes must retain the pointer-based semantic.

Listing 5.1 shows the definitions of the various structures.

Symbols define a named entity: either a state, a routine, a named type, or a predefined value; in practice, symobls are used to represent variables, constants, methods, and named types. States contain information about the state of the program, and are associated with a structure defining the information's semantic; routines are user-defined operations on the program's state; types are used to anchor a structure in a scope; values are special cases of states which have a predefined value and cannot be changed by the program (compile-time constants).

Scopes are namespaces and containers for symbols. Table 5.1 shows the available visibility rules. Scopes come in three flavors: record scope, procedure scope, and module scope. These correspond to different uses and implementations of the symbols they contain. Table 5.2 gives the terminology used in the rest of the chapter.

[2]structures are otherwise statically allocated

```
TYPE
  Struct* = POINTER TO RECORD
    owner-: Type;       (* canonical name of structure, if any *)
    size*:  Attribute;  (* back-end: size information *)
    sym*:   Attribute;  (* fingerprinting information *)
  END;

  Basic* = POINTER TO RECORD (Struct)
  END;

  Array* = POINTER TO RECORD (Struct)
    base-:  Struct;     (* element type *)
    len-:   LONGINT;    (* array size or open) *)
    mode-:  SHORTINT;   (* array size: static or open *)
    base-:  Struct;     (* element type *)
    len-:   LONGINT;    (* array size (iff mode = static) *)
  END;

  Record* = POINTER TO RECORD (Struct)
    scope-: RecScope;   (* record contents *)
    brec-:  Record;     (* base record*)
    btyp-:  Struct;     (* base type, for dynamic records = Pointer to brec*)
    ptr-:   Pointer;    (* pointer to self, (POINTER TO RECORD) *)
    intf-:  POINTER TO Interfaces;
  END;

  Pointer* = POINTER TO RECORD (Struct)
    base-: Struct;      (* pointed type *)
  END;

  Delegate* = POINTER TO RECORD (Struct)
    scope-: ProcScope;  (* parameter list *)
    return-: Struct;    (* return type, or NoType *)
  END;
```

Listing 5.1: Structure declaration

access	*scope of visibility*	*applicable to*
private	defi nition scope	states, routines
internal	whole module	states, routines
protected	whole type and all its extensions	methods, fi elds
public	everywhere	states, routines

Table 5.1: *Paco Visibility Rules*

	ModuleScope	ProcedureScope	RecordScope
State	global variable	local variable parameter	fi eld
Routine	procedure	local procedure	method

Table 5.2: *Terminology*

Modules are namespaces, deployment, and compilation units at the same time.

5.2.3 Basic Use

Metadata is created once and never modified. This restriction simplifies the consistency check of the metadata and its use. All fields have read-only access, structures can be initialized only once. Each structure invariant is established upon creation and holds for the whole object lifetime without need of re-checking it.

Creating Structures

Basic types are predefined and cannot be changed.

The repository provides functionalities to create user-defined structures by composing existing structures into arrays, records, and pointers. Initialization can be performed only once. Listing 5.2 shows the functions to initialize the composed structures.

```
PROCEDURE NewOpenArray(base: Struct; VAR res: LONGINT): Array;
PROCEDURE NewStaticArray(len: LONGINT; base: Struct; VAR res: LONGINT): Array;
PROCEDURE NewRecord(base: Struct; scope: RecScope; VAR res: LONGINT): Record;
PROCEDURE NewClass(base: Struct; implements: Interfaces;
                   scope: RecScope; VAR res: LONGINT): Pointer;
PROCEDURE NewInterface(implements: Interfaces; scope: RecScope; VAR res: LONGINT): Pointer;
PROCEDURE NewPointer(base: Struct; VAR res: LONGINT): Pointer;
PROCEDURE NewDelegate(return: Struct; scope: ProcScope; VAR res: LONGINT): Delegate;
```

Listing 5.2: Structure Creation Functions

Scopes and Symbols

Each symbol belongs to a scope. The scopes provide functions to create new symbols. Depending on the kind of scope, the appropriate symbol is create, like described in Table 5.2.

```
Scope = OBJECT
VAR
 state-:  SHORTINT;
 module-: Module;
 parent-: Scope;

 PROCEDURE Await(state: SHORTINT);
 PROCEDURE CreateValue(name: Name; vis: SET; c: Const; VAR res: LONGINT);
 PROCEDURE CreateType(name: Name; vis: SET; type: Struct; VAR res: LONGINT);
 PROCEDURE CreateVar(name: Name; vis: SET; type: Struct; VAR res: LONGINT);
 PROCEDURE CreateProc(name: Name; vis: SET; scope: Scope; return: Struct; VAR res: LONGINT)
END;
```

Listing 5.3: The Scope Class and the Symbol Creation

The repository provides functions to lookup symbols in the scopes; the whole scope hierarchy is traversed until a matching symbol is found. In modules using procedure overloading, the lookup function searches for the procedure with the smallest signature distance. Listing 5.4 shows the lookup functions.

```
  PROCEDURE Find(current, search: Scope; name: Name; mode: SHORTINT): Object;
  PROCEDURE FindProcedure(current, search: Scope; name: Name; parCount: LONGINT;
                          pars: ARRAY OF Struct; identicSignature: BOOLEAN): Proc;
  PROCEDURE FindSameSignature(search: Scope; name: Name; par: Parameter;
                          identic: BOOLEAN): Proc;
```

Listing 5.4: Lookup Functions

The lookup functions only match symbols visible to the current scope according their visibility flags. The `identicSignature` flag limits the search to procedure having exactly the same signature.

The signature distance is the number of type conversion operations needed to convert the actual parameter types to the signature formal types.

5.2.4 Advanced Use

Scopes have a state, which reflects the information they contain; as compilation or loading progresses, the scope's state is incremented.

```
CONST
    (*Scope.state*)
  structdeclared = 1;   (* all structures declared *)
  structallocated = 2;  (* all structures allocated *)
  procdeclared = 3;     (* all procedures declared *)
  modeavailable = 4;    (* body mode available *)
  complete = 5;         (* code available *)

  PROCEDURE ChangeState(scope: Scope; state: SHORTINT);
```

Listing 5.5: Scope States

The `Await` method of the scope class synchronizes the execution with the scope's state. It is used when another activity is owns a scope to make sure that the metadata is available.

The procedure `ChangeState` increments the state a scope is in; this can trigger some additional operations: changing a scope's state to `StructAllocated` computes the size of the structures defined in the scope and the allocation of the states. `ProcDeclared` causes the methods of a `RecScope` to be allocated.

Metadata from other modules can be automatically imported into the repository. To remain independent from the various persistence formats, metadata loaders can be installed at run-time to handle each format.

```
TYPE
  ImporterPlugin* = PROCEDURE (self: Module;  VAR new: Module;  name: Name);

  PROCEDURE Import (self: Module; VAR new: Module; name: Name);
  PROCEDURE AddImporter (p: ImporterPlugin);
  PROCEDURE RemoveImporter (p: ImporterPlugin);
```

Listing 5.6: Loading External Metadata Information

5.3 Scope Concurrent Parser

5.3.1 Overview

The Paco compiler implements a recursive descent and scope concurrent parser for the language Active Oberon.

The simple design of the language and the unambiguous syntax are easily parsed with a recursive descent parser; techniques that are more complicated bring no advantage.

Forward references are uses of a symbol before the symbol definition. Because Active Oberon's objects define their methods in the object scope itself, forward references among methods and objects cannot be avoided. Forward references are a well-known problem in other languages; their handling relies on a technique requiring two passes on the source code: the first to parse the declarations, and the second to use the declared symbols in the implementation.

The original Oberon language was designed to avoid forward references[3], such to make compiler construction simpler; because most of the modules in use are still written in the original language, we try to avoid two passes whenever possible.

Syntax trees allow a first optimization of 2-pass compilation: the first pass performs syntax checking and generates the tree; the second pass is traverses the tree instead of the source, and performs semantic checking and analysis, and code generation.

The Oberon compiler OP2 [Cre90] generates a parse tree containing the whole parsed program. Syntactic and semantic analysis are performed in a single pass during the construction of the tree, which serves as separation between front-end and back-end. To parse Active Oberon, we modified the parser into a 1-1/2 passes parser: if a symbol is not know during the first pass, the compiler saves the actual scanner position, scope, and the position in the parse tree; such tuples are processed at the end of the first pass, when all symbols are declared. This has the advantage of executing the second pass only whenever needed. Furthermore, reprocessed are only the program parts that could not be understood during the first pass. The disadvantage of this technique lies in its complex implementation: for every symbol in a production, the parser must decide whether to insert it into the parse tree or to delay it; the implementation of the parser is doubled in size and complexity.

The Paco compiler takes a different approach: *dependencies in a scope require multi-pass compilations, dependencies between scopes require synchronization*, in other words parsing is divided into concurrent tasks each handling a single scope, in a way to make the dependencies external. Whenever Paco's single-pass recursive descent parser encounters a scope declaration

[3]there is one exception: pointer types are allowed to refer to a type defi ned farther in the same scope

(a procedure, or record/class declaration), it launches a new parser for it. Accessing symbols declared in another scope becomes a synchronization problem, removing the problem inherent to the sequentiality of the parsing. As a consequence of this parsing strategy, forward references are not allowed inside a scope, but only outside.

Scope concurrent parsing has been extensively investigated for the implementation of a Modula-3 concurrent compiler by Wortman and his research group [SSW87, Wor90, SW91, WJ92]; this project shows possibilities and limitations of the technique in conjunction with multiprocessor machines. Focusing the investigation on compiler construction, they however did not realize that this technique could also be used to handle forward references. In fact, allowing symbols to be declared farther in the text removes a restriction in the parser: as a scope concurrent parser has no notion of textually before or later, the declaration position had to be tested against the actual position of the parser. In our parser, this restriction can be removed.

5.3.2 Implementation of Concurrency in the Parser

The Paco compiler recursive descent parser launches a new parser when a nested scope is encountered. Listing 5.7 shows an example.

```
PROCEDURE ProcDecl;
  (* ProcDecl = PROCEDURE IdentDef [FormalPars] ";" Body ident. *)
  VAR parser: ProcedureParser; scope: ProcedureScope;
      res: LONGINT; i: IdentDef; returnType: Struct;
BEGIN
  Check(procedure);
  IdentDef(i);
  NEW(procscope); PCT.InitScope(scope, currentScope);
  FormalPars(procscope, returnType);
  Check(semicolon);
  currentScope.CreateProc(i.name, i.vis, scope, returnType, res);
  IF res # Ok THEN Error(res) END;
  NEW(parser, scope, scanner); (* launch parser *)
  SkipScope;
  IF scanner.name # i.name THEN Error(...) END;
  Check(ident)
END ProcDecl;
```

Listing 5.7: While parsing a procedure, a new parser is launched

Each scope consists of three separated parts: data declarations, procedure declarations, and the scope body. This division reflects the information dependencies of the compilation.

The scope's state variable tells the current compilation stage reached by the parser:

1. Local Data Declared
2. Local Data Allocated
3. Local Procedures Declared
4. Code Available

Symbol lookup across scope boundaries requires synchronizing the current thread with the looked-up scope state. It is unnecessary to wait for the completion of a scope, before performing a lookup; this rule can be weakened as follows:

- during data declarations parsing, lookup in other scopes is delayed until the scopes are allocated
- during implementation parsing, lookup in other scopes is delayed until the scope procedures are declared

These optimizations reflect the fact, that data declaration requires only information about other declarations of structures, whereas during the code's parsing both a data and procedure declarations are required (but the code associated with a procedure can be ignored). The scope lookup function (Listing 5.8) is modified to handle this synchronization; note that the first lookup is in the own scope and thus requires no synchronization.

```
PROCEDURE Find(scope: Scope; name: Name; requiredState: State): Object;
  VAR p: Object;
BEGIN
  p := scope.lookup(name);
  scope := scope.next;
  WHILE (scope # NIL) & (p = NIL) DO
    scope.Await(requiredState);
    p := scope.lookup(name);
    scope := scope.next
  END;
  RETURN p
END Lookup;
```

Listing 5.8: Scope Lookup with State Synchronization

The semantic of the language requires two exceptions to the lookup optimization.

First, the built-in function `new` must know whether the object to be allocated has an initializer and is active; thus `new` can only be parser when the scope of the class to be allocated has reached the `CodeAvailable` state.

Second, the lookup rule assumes implicitly that only the current and the parent scopes are searched. Procedure parameters and record and class contents require the parent scope to search through them; this potentially creates a circular dependency that can potentially lead to a deadlock situation.

Procedure signatures are accessed both from the scope calling the procedure and from the scope using the signature as parameter. To remove this cyclic dependency, we consider the procedure signature part of the parent scope. The parameters are parsed by the parent before launching the child parser for the procedure (Listing 5.7).

Records and classes are more complex to handle. Allocating a scope that contains a record requires the record size to be known, whereas a record can use types declared in a parent scope and requires these types to be allocated to be able to allocate its own fields and calculate the structure size. Classes suffer from the same problems.

Considering records as part of the parent scope would solve the problem, but restrict the handling of the forward references too: all types used in a record were required to be textually declared before the record declaration or outside the scope declaring the record. This may be tolerable for records but not for classes. The solution is in the weakening of the lookup condition: definition lookups outside record scopes are required to be at least `Data Declared` (instead of `Data Allocated`); allocation of the structures will be performed independently of the scope owning them (either the record or the parent scope).

5.3.3 Other Implementation Details

Parse Tree

The Paco compiler builds a parse tree only for the expressions, to reduce the pressure on the memory management[4].

Dynamically-sized local arrays (called semi-dynamic arrays in [Mor97, Jan98]), method calls, and procedure overloading cannot be compiled directly; the expression's parse tree allows to delay the code emission till the appropriate code can be emitted, or to allow the emission of better code.

Procedure overloading is the most problematic case: the parameters to be parsed are needed to determine which procedure is to be selected. Furthermore, if the signature is not identical with the procedure found, the parameter values may have to be casted; only then is possible to pass the parameters to the callee[5]

The method calling convention requires a partial out-of-order passing of the parameters, because the object self-reference is passed as last hidden parameter, whereas it comes textually before all other parameters. A naive implementation would load self into a register before all parameters are pushed, but leads to possibly very inefficient code[6]; furthermore, this implementation leads to a wrong implementation because of side effects: a change of the object self-reference due to a side effect would go unnoticed, because the value has been pre-loaded.

5.4 Back-End Component

The Back-End of the Paco compiler is a framework providing support for code generation. It defines a low-level intermediate code representation, and structures to keep track of references to external entities (fixup chains). Code generators and object-file generators are dynamically installed at run-time using a plug-in mechanism.

[4]memory allocation is the second cost factor in the compiler, after disk access

[5]Oberon.Net [Gut01a] avoids this problem by allowing to specify the procedure signature for every procedure call, thus easing the selection of the appropriate procedure

[6]fi rst, a register less is available for the computation and passing of the parameters and could cause register spilling; second, if a parameter requires a function call, the register has to be saved on the stack

5.4.1 Intermediate Code

An assembler-like low-level intermediate code representation (LIR) [Muc97] is used as interface between the compiler's front-end and back-end. The LIR defines an abstract machine with the following properties [Muc97, HP96]:

- load-store architecture with infinite registers
- instructions in SSA form; registers are assigned only once
- compare and branch instructions, no condition codes
- typed registers (signed integers, unsigned integers, floating points, addresses)
- flat memory model
- stack registers (SP, FP)
- parameters and local variables on the stack

The LIR has infinite *Registers*. Every register has a type that describes the size and encoding of its contents. Defined types are: *Int8*, *Int16*, *Int32*, *Int64*, *Float32*, *Float64*, *Address*; Int types are either signer or unsigned. Address is an alias to unsigned Int32 or Int64, depending on the hardware address size. The special registers *SP* (top of stack) and *FP* (procedure activation frame) are predefined and have type Address.

The instruction set consists of a few dozens instructions divided into three main groups: data transfer instruction, arithmetical and logical instructions, and control flow instructions.

Form	dst	s1	s2	s3	val	addr	explanation
00							op
0C					val	NIL	op(val)
					val	addr	op(val+addr)
01		s1					op(s1)
02		s1	s2				op(s1, s2)
02C		s1	s2		val		op(s1, s2, val)
03		s1	s2	s3			op(s1, s2, s3)
10	d						d := op()
11	d	s1					d := op(s1)
12	d	s1	s2				d := op(s1, s2)
1M	d	s1			val	NIL	d := op(MEM[val+s1])
	d	ABS			val	addr	d := op(MEM[val+addr])
M1		s1	s2		val	NIL	MEM[val+s1] := op(s2)
		ABS	s2		val	addr	MEM[val+addr] := op(s2)
XX							special format

***Table 5.3**: LIR Instruction Forms*

Each instruction has a form describing the source and destination of the instruction. The form has a source and a destination magnitude, each described by a digit or a letter. A digit corresponds to the number of registers, letter C to a constant value, and letter M to an external reference. As an example, instructions of form 1M take an external reference as source and use one register as result. Table 5.3 shows the defined instruction forms.

Data transfer		
loadc	load value	0C
loadsp	load stack pointer	01
load	load from memory	1M
store	store to memory	M1
push	push onto stack	01
pop	pop from stack	10
move	move memory block	03
ret	pass value to caller	01
result	receive value from callee	10
saveregs	save registers in use	00
loadregs	restore registers in use	00

Table 5.4: *LIR Data Transfer Instructions*

The data transfer instructions group all instructions that move data from and to memory or stack. A special case are the `ret` and textttresult instruction, which return and retrieve a value from caller to callee, and the `saveregs` and `loadregs` which save and restore the register contents. Table 5.4 shows the data transfer instructions.

The arithmetic and logic instructions group all instructions that transform values. These instructions consist of bit-pattern operations, arithmetical operations, and conversions.

All instructions of form 12 require both source registers to have the same type. The destination type is same as the source type(s); only exceptions are the conversion instructions and `setcc`.

Conversions are signed (sign extensions), unsigned (zero extension) or untyped (bit-patters is copied).

The control flow instructions change the flow of the program. They allow jumps, conditional jumps, subroutine calls, and throwing exceptions. Jumps and conditional jumps are always local, whereas subroutines can be external to the current code. Table 5.6 shows the control instructions.

Conditional jumps and conditional set take two registers and perform their operation only if the expected relation between the two register's values is true. Table 5.7 shows the available conditions; these suffixes can be used with the `jcc` and `setcc` instructions. Only `tae` and `tne` are defined.

The `Instruction` (Listing 5.9) and `Code` structures (Listing 5.10) contain the intermediate code. *Instruction* defines a single LIR Instruction; the destination register, if needed, is implicitly defined by the instruction itself. *Code* contains a list of instructions (usually the code for one routine) organized in pieces of instructions; this is done in order to reduce the pressure on memory

Arithmetic and Logical			
	not	signed negation	11
	neg	bitwise negation	
	abs	remove sign	
	convs	signed conversion	
	convu	unsigned conversion	
	copy	untyped move	
	add	addition	12
	sub	subtraction	
	mul	multiplication	
	div	division	
	mod	modulo	
	and	logical and	
	or	logical or	
	xor	logical xor	
	bts	set bit	
	btc	clear bit	
	ash	arithmetic shift	
	bsh	logical shift	
	rot	logical rotation	
	setcc	conditional set	12

Table 5.5: *LIR Arithmetic Instructions*

Control			
	jcc	compare and branch on condition	02C
	jmp	jump	0C
	call	call subroutine (pc := Dest)	
	syscall	call system-call	
	callreg	call subroutine (pc := R_{s_1})	01
	trap	throw exception	0C
	tcc	throw exception on condition	02C
	case	case	XX
	casel	case line	XX
	casee	case else	XX

Table 5.6: *Table: LIR Control-Flow Instructions*

	CC(R1, R2)
e	R1 equal to R2
ne	R1 not equal to R2
lt	R1 less than R2 (signed)
le	R1 less or equal than R2 (signed)
gt	R1 greater than R2 (signed)
ge	R1 greater equal R2 (signed)
a	R1 above R2 (unsigned)
ae	R1 above or equal R2 (unsigned)
f	R2-th bit in R1 set
nf	R2-th bit in R1 clear

***Table 5.7**: LIR conditions*

```
TYPE
  Instruction = RECORD
    op: Opcode;

      (* source *)
    src1, src2, src3: Register;
    val: LONGINT;
    adr: PCM.Attribute;  (*external reference*)

      (* destination *)
    dstSize: Size;
    dstCount: SHORTINT;  (*use count*)

    suppress: BOOLEAN;
    info: OBJECT;  (*emitter info*)
  END;
```

Listing 5.9: The Instruction Structure

```
  Piece = OBJECT
    VAR instr: ARRAY PieceSize OF Instruction;
    PROCEDURE & Init;
  END Piece;

  Code = OBJECT
    VAR
      pc: LONGINT;
      info: OBJECT;

    PROCEDURE GetPiece(VAR src: LONGINT;  VAR p: Piece);
    PROCEDURE Traverse(p: TraverseProc; reversed: BOOLEAN; context: PTR);
    PROCEDURE & Init;
  END Code;
```

Listing 5.10: The Code Structure

allocation and garbage collection by reducing memory fragmentation [7]. Therefore, Instruction is defined as flat structure instead of a hierarchy of structures with different type for every instruction form.

Code Example

```
PROCEDURE GCD(a, b: LONGINT): LONGINT;
BEGIN
  WHILE (a > 0) DO
    IF b > a THEN Swap(a, b) END;
    a := a - b;
  END;
  RETURN b
END GCD;

PROCEDURE GCD
    1  load    SD3, 12[FP]
    2  loadc   SD4, 0
    3  jle     SD3, SD4, 24
    4  load    SD8, 8[FP]
    5  load    SD9, 12[FP]
    6  jle     SD8, SD9, 16
    7  savereg
    8  loadc   D14, 12
    9  add     D15, FP, D14
   10  push    D15
   11  loadc   D17, 8
   12  add     D18, FP, D17
   13  push    D18
   14  call    Swap
   15  loadreg
   16  load    SD25, 12[FP]
   17  load    SD26, 8[FP]
   18  sub     SD27, SD25, SD26
   19  loadc   D28, 12
   20  add     D29, FP, D28
   21  store   0[D29], SD27
   22  label   185
   23  jmp     1
   24  load    SD35, 8[FP]
   25  ret     SD35
   26  exit
```

Listing 5.11: Translation Example for a Simple Loop

Listing 5.11 shows an example of the intermediate code generated for simple Active Oberon procedure. The procedure contains a WHILE-loop, an IF-statement and a procedure call.

The complete `Swap` procedure call takes place from line 7 to line 15. The parameters for `Swap` are pushed from line 8 to line 13 and the call itself is at line 14.

[7]This has a big impact on the speed of the compiler: memory management is, after file IO, the second most time-consuming operation in the compiler

Statistics

Table 5.8 shows the LIR instruction usage after compilation of the whole Aos system with Paco.

The most used instructions are `load` and `loadc`. The `label` instruction is used only internally as destination for the jump statements and to store the source code position for debugging purposes and produces no code.

Some instructions are never used (`setb`, `setbe`, `seta`, `setae`, `btc`), because the code patterns that generate these instructions are never used in the whole system. In particular, the setX instructions are generated when a expression holding a boolean result is directly assigned to a variable.

Equality tests are used more often than relational tests.

There are about three times more `push` instructions than `call` and `callreg`, thus procedures have on average three parameters.

5.4.2 Back-End Data

The PCBT module defines and handles the information needed and created by the back-end, including the list of procedures and variables accesses in a module, and the fixup-lists thereof.

```
TYPE Fixup = POINTER TO RECORD
    offset-: LONGINT;
    next-: Fixup
  END;
```

Listing 5.12: Fixup Type

A fixup list 5.12 contains all code offsets where an entity is referenced, and is used for each entity whose location is not known at compilation time; because the generated code is position-independent, roughly all entities accessed through an absolute address fall in this category.

The `GlobalVariable` and `Procedure` types (Listing 5.13) contain the information about variables and procedures that the back-end needs for code generation. In particular, they each have a reference to the module defining the entity and a list of fixups.

The `Module` type (Listing 5.14) defines all the information about a compilation unit for the back-end. It has functionalities to register the uses of variables and procedures inside the module and create the fixup lists for them.

5.4.3 Code Emitter Plug-in

Paco can be retargeted at run-time to generate code for different processors. This is done with a plug-in mechanism; each code generator can install itself on-demand into the back-end.

The structure `CodeGenerator` (Listing 5.15 of the back-end defines the interface for code generators. A code generator must provide an implementation for each of the required functionalities; it can be installed at any time by simply registering its own implementation of those features to the back-end.

load	621094	21.39%	setae	0	0.00%
loadc	479755	16.52%	setf	203	0.01%
store	115251	3.97%	setnf	5	0.00%
in	145	0.00%	result	17304	0.60%
out	256	0.01%	result2	2	0.00%
savereg	97624	3.36%	pop	450	0.02%
loadreg	97624	3.36%	ret	5274	0.18%
label	471135	16.23%	ret2	4	0.00%
je	26289	0.91%	push	299217	10.31%
jne	30789	1.06%	callreg	14058	0.48%
jlt	2600	0.09%	kill	17044	0.59%
jle	4498	0.15%	loadsp	5466	0.19%
jgt	5083	0.18%	convs	27528	0.95%
jge	5388	0.19%	convu	2469	0.09%
jb	408	0.01%	copy	72	0.00%
jbe	285	0.01%	not	406	0.01%
ja	606	0.02%	neg	1511	0.05%
jae	93	0.00%	abs	717	0.02%
jf	681	0.02%	mul	9030	0.31%
jnf	2390	0.08%	div	2287	0.08%
jmp	43898	1.51%	mod	635	0.02%
call	83105	2.86%	sub	20338	0.70%
syscall	5404	0.19%	add	251788	8.67%
enter	17472	0.60%	and	19723	0.68%
exit	20734	0.71%	or	1054	0.04%
trap	7761	0.27%	xor	84	0.00%
tae	20246	0.70%	bts	131	0.00%
tne	3948	0.14%	btc	0	0.00%
sete	504	0.02%	ash	13784	0.47%
setne	207	0.01%	bsh	191	0.01%
setlt	55	0.00%	rot	29	0.00%
setle	21	0.00%	phi	8522	0.29%
setgt	46	0.00%	move	8136	0.28%
setge	39	0.00%	inline	456	0.02%
setb	0	0.00%	case	737	0.03%
setbe	0	0.00%	casel	8721	0.30%
seta	0	0.00%	casee	737	0.03%

***Table 5.8**: Instruction Usage Statistics after compilation of the whole Aos system*

```
GlobalVariable = OBJECT ( Variable )
  VAR
    owner-: Module;
    entryNo: INTEGER; (* object-file information set by PCOF *)
    link-: Fixup; (* occurrences of var in code *)
    next-: GlobalVariable; (* next GVar in the list *)

    PROCEDURE & Init (owner: Module);
END GlobalVariable;

Procedure = OBJECT ( PCM.Attribute )
  VAR
    owner-: Module;
    public-: BOOLEAN;
    locsize: LONGINT;
    parsize: LONGINT;
    codeoffset-: LONGINT;
    entryNr, fixlist: LONGINT;
    next-: Procedure;
    link-: Fixup;

    PROCEDURE & Init (owner: Module; public: BOOLEAN);
END Procedure;
```

Listing 5.13: PCBT Entities

The code generator must register with `SetMethod` an emission procedure for each opcode in the intermediate representation.

The `Prepass` method used to perform a first pass on the code, for collecting information of preparing the data structures.

5.5 Plug-Ins

This section presents two code generator plug-ins for the Paco compiler: the Intel code generator and the ARM code generator. The ability of targeting two completely different microprocessor, a CISC and a RISC, confirms the soundness of the back-end design.

The Intel Code Generator has the complex task of translating the SSA notation of the LIR for a processor with only four registers and complex addressing modes; a peephole optimization for this purpose is presented.

The ARM Code Generator translates the LIR code into StrongARM 7 code.

5.5.1 Intel Code Generator Plug-in

Introduction

The Intel Code Generator translates the LIR code into native Intel IA-32 code [Int00].

The limited and non-orthogonal register set of the Intel IA-32 processors is the major problem faced by the code generator. Only six registers are available; furthermore, a few instructions

```
Module = OBJECT ( PCM.Attribute )
VAR
locsize: LONGINT; (* data section size*)
constsize: INTEGER; (* const section size*)
nr: INTEGER;
const: ConstArray;
OwnProcs-: Procedure; (* this module’s procedures; terminated by psentinel *)
ExtProcs-: Procedure; (* external procedures used; terminated by psentinel *)
OwnVars-: GlobalVariable; (* this module’s used variables; terminated by sentinel *)
ExtVars-: GlobalVariable; (* external used variables; terminated by sentinel *)
syscalls-: POINTER TO ARRAY OF Fixup; (* syscalls’ fixup lists *)

PROCEDURE & Init;

(* ResetLists - remove entries from OwnVars, ExtVars, OwnProcs, ExtProcs *)
PROCEDURE ResetLists;

(* NewConst - Create a new constant *)
PROCEDURE NewConst (VAR a: ARRAY OF SYSTEM.BYTE; len: LONGINT): LONGINT;

(* UseVar - insert a fixup, add to ExtVars or OwnVars *)
PROCEDURE UseVariable (v: GlobalVariable; offset: LONGINT);

(* AddOwnProc - Insert local procedure into OwnProcs list *)
PROCEDURE AddOwnProc (p: Procedure; codeOffset: LONGINT);

(* UseProcedure - insert a fixup entry, add to ExtProcs list (if external) *)
PROCEDURE UseProcedure (p: Procedure; offset: LONGINT);

(* UseSyscall - Add a syscall fixup *)
PROCEDURE UseSyscall (syscall, offset: LONGINT);
END Module;
```

Listing 5.14: PCBT Module

```
  CodeGenerator = RECORD
    Init: PROCEDURE (): BOOLEAN;
    Prepass: PROCEDURE (code: Code);
    GetCode: PROCEDURE (VAR code: CodeArray; VAR codelength: LONGINT);
    Done: PROCEDURE (VAR res: LONGINT);
  END;

  EmitProc = PROCEDURE (code: Code; VAR instr: Instruction; pc: LONGINT);

  PROCEDURE SetMethod (op: Opcode; p: EmitProc);
```

Listing 5.15: Code Emitter Interface

require a specific register to work with. On the other hand, the processor implements a register-memory model with complex addressing modes, which allows to some explicit computations into the address computation.

This small register set problem is further stressed by the SSA-like notation of the LIR code, which tends to put a lot of pressure on the register set by using a new register every time a computation is done. It this thus crucial, that the code generator performs an aggressive optimization to reduce the register usage by using the complex addressing modes of the processor.

This section introduces a peephole optimization to construct complex addressing modes from a register-based notation. We use for this purpose a finite state machine, which models the addressing modes as states and the instructions as transitions. This clear and well-known abstraction is easily translated in a program.

Complex Addressing Modes

IA-32 processors have three operand types: registers, constant values, and memory values. Almost each instruction can use any of the operand types as source or destination. A memory operand has the form

$$disp[R_b][R_i : scale]$$

which corresponds to the memory address

$$disp + R_b + R_i * scale$$

where *disp* is a value called displacement, R_b and R_i registers called base and index, and scale a value (1, 2, 4, or 8). Base, index, and scale are optional.

Mode	Notation	Value in
Register	R_x	R_x
Immediate	`val`	val
Absolute	`@Addr`	MEM[Addr]
Relative	d[R_b]	MEM[d+R_b]
Indexed	d[R_b][R_i]	MEM[d+R_b+R_i]
Scaled	d[R_b][R_i :f]	MEM[d+R_b+f*R_i] $f \in \{1, 2, 4, 8\}$

Table 5.9: *Complex Addressing Modes*

For clarity, we divide our memory address into the *absolute*, *relative*, *indexed*, and *scaled* addressing modes depending on which components of the memory address are used. Table 5.9 shows the details of the various addressing modes.

Addressing Modes Transitions

A complex addressing mode can be translated into a few instructions using only registers. Symmetrically, it is possible to replace some instructions with a complex addressing mode.

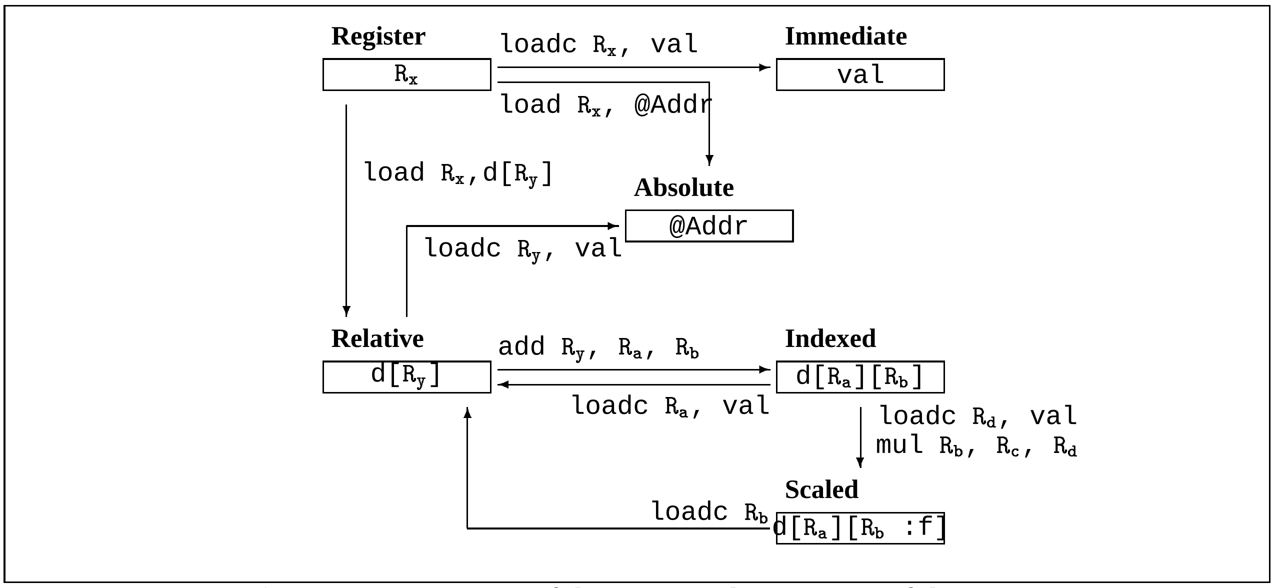


Figure 5.2: Overview of the states and transitions of the FSM

These transformations can be elegantly displayed using the well-known finite state machine (FSM) model, where states represent addressing modes, and transitions are triggered by instructions. Figure 5.2 shows such a FSM. The *Register* is the starting point of the FSM. Depending on how the registers in the addressing mode are assigned, transitions may follow. The instructions triggering a transition can be removed from the code, as they operation is then implicitly executed as part of the address computation. The transitions are repeated as long as matching code pattern for a register in the address exists.

Table 5.10 shows the details of the transitions.

This optimization can be applied only inside basic blocks.

Algorithm

The optimization can be performed in one pass together with code emission, using a *lazy code selection* technique like in [M¨os00, ATCL+98].

We choose to apply this optimization in a separate backward traversal of the code. The reasons for this are as follows:

1. makes the FSM simple, because a forward traversal requires the introduction of additional transitional states, which do not correspond to addressing modes

2. allows better dead code removal, because starting point of the optimization are always uses of registers. Encountering an instruction assigning a register that has not been traversed, means that the register is not used. A forward traversal may emit code whenever a mode cannot be further optimized and its value is needed for the next instruction, independently if the value will ever be used

From State	Transition	To State
Register R_x	load R_x, @Addr	**Absolute** @Addr
	loadc R_x, val	**Immediate** val
	load R_x, d[R_y]	**Relative** d[R_y]
Relative d[R_x]	loadc R_x, val	**Absolute** val + d
	add R_x, R_a, R_b	**Indexed** d[R_a][R_b]
Indexed d[R_a][R_b]	loadc R_a, val	**Relative** d+val [R_b]
	loadc R_b, val	**Relative** d+val [R_a]
	loadc R_d, val add R_a, R_c, R_d	**Indexed** d+val [R_c][R_b]
	loadc R_d, val add R_b, R_c, R_d	**Indexed** d+val [R_a][R_c]
	loadc R_d, val mul R_b, R_c, R_d	**Scaled** d[R_a][R_c :val]
	loadc R_d, val mul R_a, R_c, R_d	**Scaled** d[R_b][R_c :val]
Scaled d[R_a][R_b :f]	loadc R_b, val	**Relative** d+f*val [R_a]
	loadc R_d, val add R_a, R_c, R_d	**Indexed** d+val [R_c][R_b]
	loadc R_d, val add R_b, R_c, R_d	**Indexed** d+val*f [R_a][R_c :f]

***Table 5.10**: FSM Transition Table*

3. backward traversal simplifies the register allocation for Intel processors, because it allows propagating information about the hardware register in which the virtual register shall be mapped. This is useful when generating instructions that use only a specific register. We can thus use a very simple round-robin register allocation algorithm without having to introduce an extra pass to perform register coloring.

4. backward traversal is better adapted to Intel's non-orthogonal instructions, which allows only one operand to be a memory mode, while the second operand is restricted to register or immediate mode[8].

The algorithm for performing the addressing mode optimization can be easily derived from the FSM. Listing 5.16 shows an excerpt of the optimization code.

Example

We show an example of peephole optimization using the algorithm presented in this section.
We take the following Oberon code:

```
PROCEDURE P(i: LONGINT);
VAR  a: ARRAY 4 OF LONGINT;
```

[8]for some instruction only register mode

```
PROCEDURE FSM(VAR addr: Address);
BEGIN
 CASE addr.mode OF
 ...
 | ScaledMode:
   src := FetchInstruction(addr.index);
   IF src.op = loadc THEN
     (* Indexed -> Relative *)
     addr := Relative(addr.disp+addr.scale*src.val, addr.base);
     src.delete := TRUE;
     FSM(addr)
   ELSIF src.op = add THEN
     ...
   END
 ...
 END
END FSM;

PROCEDURE AddressMode(pc: LONGINT);
  VAR addr: Address; instr: Instruction;
BEGIN
  instr := FetchInstruction(pc);
  CASE instr.form OF
  ...
  | Form01, Form11:
    addr := Register(instr.src1);
    FSM(addr)
  ...
  END (* CASE *)
END AddressMode;
```

Listing 5.16: Addressing Mode Selection

```
BEGIN
  a[i] := i
END P;
```

Its intermediate representation is:

```
; i @ 8[FP]
; a @ -16[FP]
0  load  R0, -16
1  add   R1, R0, FP
2  load  R2, 8[FP]
3  loadc R3,  4
4  mul   R4, R2, R3
5  add   R5, R4, R1
6  store 0[R5], R2
```

The FSM processes the code starting from instruction 6. It first tries to optimize the destination 0[R5]:

Mode	Address	Transition
Relative	0[R5]	`5 add R5, R4, R1`

Indexed	0[R4][R1]	`3 loadc R3, 4`
		`4 mul R4, R2, R3`

Scaled	0[R1][R2: 4]	`0 load R0, -16`
		`1 add R1, R0, FP`

Scaled	-16[FP][R2: 4]	

Register R2 is not optimized, because it is used twice in the code, this loads its value only once from memory. The third optimization step (from Scaled to Scaled) is not shown in the FSM table for space reasons, but was added to the table because it is a very common pattern in code.

The optimized code is:

```
2  load  R2, 8[FP]
6  store -16[FP][R2:4], R2
```

The resulting code uses only one register instead of five; it is also much shorter than the original one.

Conclusions

Table 5.11 shows the total number of instructions and the delete instruction after the addressing mode optimization.

The table shows that almost all constants are inlined using a complex addressing mode. Also interesting is that about half of all `add` and `ash` instructions are used in address computations and can be removed by using a complex addressing mode.

	Generated	Suppressed	Percentage
Total	2903547	873032	30.07%
`load`	621094	251528	40.50%
`loadc`	479755	463144	96.54%
`add`	251788	150800	59.89%
`ash`	13784	7541	54.71%

***Table 5.11**: Instruction removal when using complex addressing modes*

Note that `ash` (arithmetic shift) is automatically used instead of `mul` by the compiler for multiplications by a power of 2[9].

5.5.2 ARM Code Generator Plug-in

The ARM code generator produces code for the Intel StrongARM SA-110 processor[Int][10], which is ARM v4 compatible [Sea00]. The code generator was written by B. Egger as part of his diploma thesis [Egg01].

The ARM processors have been designed to allow very small, yet high-performance implementations with very low power consumption. ARM processors are thus ideal for embedded systems, and are currently found inside many mobile phones and palmtops.

The SA-110 processor is a 32-bit RISC processor with a load-store architecture, 32 generic purpose registers, no FPU, and with hardware MMU support.

ARM processors have some peculiarities that relevant for the code generation.

First, for each instruction carries a condition mask; the instruction is executed only if the current condition code matches the mask. This feature allows working with conditions, without inserting jumps in the code, which would stall the processor's pipeline.

Second, every instruction including operands is exactly 32 bits wide; this makes the loading of 32 bit values, in particular absolute addresses, complicated. Addressing is as far as possible performed using PC-relative displacement, but this is limited. When data lies far apart, like the constant table in Oberon, then this data must be embedded in the code in a such that the displacement is small enough.

Third, FPU operations are implemented in a software library[11]

The implementation details of the code generator can be found in [Egg01].

5.6 Interoperability with Java

This section describes the problems limiting transparent language interoperability between Oberon and Java. The focus is set on the compiler and the Oberon language. Section 7.4 describes the interoperability limits found in Jaos.

[9]this optimization is called strength reduction

[10]and its successors SA-1100 and SA-1110

[11]it is also possible to use an external coprocessor

The shared metadata repository provides automated language interoperability for in the Oberon language. Importing compilation-units definitions is performed by Oberon's `IMPORT` command, which makes the definitions available through a new namespace.

5.6.1 Overview

The shared metadata repository provides a common uniform access to the metadata. This allows for a transparent access of the information in a language independent form. This permits to use metadata created by all language processors running on the platform, independently of the language the data is defined in.

The implementation under Oberon is relatively simple: the compiler uses the repository as symbol table; furthermore, a metadata loader for Oberon symbol files (Oberon persistent metadata) is installed at run-time in the repository.

The metadata repository generates a fingerprint for each symbol in the repository. The Oberon object-file format uses this fingerprint to label the entities to be linked, and every loaded module must carry a list of the fingerprints and entry points it offers to allow dynamic linking at run-time. If a loaded compilation-unit provides this linking information, then it can be linked from other compilation-units independently of the language it was written in.

Only the `ImportList` production of the Active Oberon language has been changed to allow interoperability. This to allow more flexibility in the identification of imported compilation-units.

5.6.2 Problems

The common type system is the intersection of Oberon's and Java's type system. Interoperability is thus limited or impossible for all features whose semantic differs or that are not present in the other language.

Problems are caused by the different character sets, strings, arrays, monitors protecting against concurrent execution, and the compilation-unit initialization strategy.

Characters

Oberon and Java use two different character sets. Oberon characters are in an own proprietary 8-bit encoding, close to ISO-LATIN; Java 16 bit characters follow the Unicode 3.0 specification. Both character sets include the ASCII characters, but the different encoding sizes make interoperability impossible.

The Active Oberon type CHAR16 has been defined to allow interoperability with Java;' `char`. This was unavoidable, because Oberon is used to implement the native methods of the Java VM; using INTEGER instead is possible, but semantically not clean.

Identifiers

Oberon and Java use different rules for their identifiers (entity names). They differ in the character set used, the allowed length, and the characters allowed in an identifier.

Identifiers are part of the metadata and are defined in the repository. As the whole Aos system is based on the Oberon character set, the identifiers are encoded using Oberon's character set. The repository itself doesn't limit the identifier length or contents in any way, but for interoperability reasons the Java identifiers are subjected to the same restrictions as Oberon's ones, in particular the may contain only letters and digits.

Although the conversion of the Java identifiers causes a loss of information, problems where never encountered. Our guess is that most of the identifiers are English nouns since programmers usually write code using English; non-English text is usually limited to strings or comments. It is also uncommon (and to be considered bad taste, although common in the C world) to have identifiers differ only on an underscore.

Strings

Oberon and Java have two different approaches to strings. In Oberon are they ad-hoc constants in array of char format, whereas Java defines a built-in type `java.lang.String`.

Under Oberon, a few conversion routines are available to convert Java strings, but they must be called explicitly.

Arrays

Array semantic differs a lot between Java and Oberon. Oberon takes a more low-level approach, whereas Java defines arrays in a generic and highly abstract way.

The main discrepancy is that Java arrays are defined as classes. To model this semantic, the dynamically allocated arrays are embedded into a class, which is not compatible with any of the array types defined in Oberon. Oberon developers can explicitly access the mapped primitive (i.e. the dynamic array inside the class), but makes the implementation dependent on the mapping function; this coding style could barely qualify as clean (although it may be required in the JVM itself for bootstrapping reasons).

Protection

Code protection against concurrent access is modeled with very close semantics in both languages. Each object can acts as a monitor; code portions can be declared as synchronized or exclusive relative to the monitor.

The difference is in reentrancy. The Aos kernel offers non-reentrant locks, while Java defines locks to be reentrant. Because of this difference, `java.lang.Object` defines its own locking primitives; Java locks are based on Aos ones to achieve locking, but with the additional features to handle reentrancy.

Active Oberon modules extending Java classes shall avoid `EXCLUSIVE` and explicitly call `jjlObject.Lock` and `jjlObject.Unlock`.

Loading and Initialization

Oberon and Java differ in the way compilation-units are loaded and initialized. Oberon compilation-units are clearly organized in a hierarchy; Java classes are also hierarchically organized, but references from a class are not limited to the classes defined in the hierarchy, but can refer to any class in the universe.

Consequence of this design weakness and to avoid needlessly loading hundreds of classes, a class is loaded and initialized on-demand when first accessed.

To fulfill Oberon's semantic which requires imported modules to be initialized before their clients can execute, whenever Oberon imports a java class, this class must be initialized first, regardless of its use by Oberon. Only classes directly imported have to be initialized in such a way; all other classes are initialized using the Java mechanism.

Other problems

The introduction of overloading in Oberon causes some problems; in particular, procedure types and overloading do not mix well. In Oberon, procedure types may be declared, and variables of such type assigned with procedures. The value of the expression to be assigned (r-value) does not depend on the designator to be assigned. This means that assigning an overloaded procedure is impossible, because the compiler has no means to determine which of the many procedures is meant.

An elegant workaround to this problem is present in Oberon.Net, where a notation to select procedures by their signature is introduced.

5.6.3 Two Examples

Listing 5.17 shows an example of language interoperability with Oberon importing two Java classes.

The example creates two Java strings using the overloaded `valueOf` functions. The strings are then printed in the Oberon Log (using `Out`) and on the Java console (using `System.out`). The use of `SHORT` for forcing the cast from the `UNICODE` to the `CHAR` type reveals the incompatibility between the type character types.

The module interface browser is another application using the common metadata repository for accessing interface data. Without change, this application is able to print the any compilation-unit interface using Oberon notation. This implicitly proves that the information is not converted, but accessible in a common format.

Listing 5.18 shows the interface of class `java/lang/Classloader` in Oberon notation.

```
MODULE Java;  (** PRK  **)

IMPORT
    Out,
    String := "java/lang/String", System := "java/lang/System";

PROCEDURE PrintString(s: String.String);
  VAR i, l: LONGINT; ch: CHAR;
BEGIN
  i := 0; l := s.length();
  WHILE i < l DO
    ch := SHORT(s.charAt(i));
    Out.Char(ch);
    INC(i)
  END
END PrintString;

PROCEDURE Test*;
  VAR s0, s1: String.String;
BEGIN
  s0 := String.valueOf(12345678H);   (* valueof(int) *)
  s1 := String.valueOf(100000000H);  (* valueOf(long) *)

  PrintString(s0); Out.Ln;
  PrintString(s1); Out.Ln;

  System.out.println(s0);
  System.out.println(s1);
END Test;

END Java.
```

Listing 5.17: Example of Oberon importing Java classes

```
MODULE java/lang/ClassLoader;

IMPORT
  Object := "java/lang/Object", String := "java/lang/String",
  Class := "java/lang/Class", InputStream := "java/io/InputStream",
  Enumeration := "java/util/Enumeration", URL := "java/net/URL";

TYPE
  ClassLoader = OBJECT (Object.Object)
      PROCEDURE loadClass (p0: String.String): Class.Class;
      PROCEDURE getResource (p0: String.String): URL.URL;
      PROCEDURE getResourceAsStream (p0: String.String): InputStream;
      PROCEDURE getParent (): ClassLoader;
      PROCEDURE getResources (p0: String.String): Enumeration.Enumeration;
  END ClassLoader;

  PROCEDURE getSystemResource (p0: String.String): URL.URL;
  PROCEDURE getSystemResourceAsStream (p0: String.String): java/io/InputStream.InputStream;
  PROCEDURE getSystemClassLoade* (): ClassLoader;
  PROCEDURE getSystemResources (p0: String.String): Enumeration.Enumeration;

END java/lang/ClassLoader.
```

Listing 5.18: Java Classloader Class in Oberon notation

5.7 Compiler Testing and Maintenance

In many academic research projects, the software often plays the role of ”proof of concept”; it shows that the idea investigated can be implemented and allows to make measurements to prove or discard a theory; the software is mostly a prototype that never reaches the product ripeness. Since the launch of Project Oberon [WR92], our goal was to produce software that can be used for the daily work. This creates additional constraints on the software, which must be correct, usable, and fully functional; for this reason maintenance and testing play a role in the implementation of the software presented in this thesis.

Our *testing strategy* is based on regression testing and user feedback (beta testing). The whole system and a dedicated test-suite called Hostess (Section 5.7.2) are used as pre-release compiler tests. The Aos system has about five hundred modules, and the compiler must be able to compile them all correctly. Although this is a good parser and compiler stress test, it nevertheless fails to identify the errors in the generated code until a user executes a faulty piece of code. A test-suite on the other hand, can selectively test each feature, and —in case of error— identify the fault automatically.

Beta testing the compiler by using the language in the undergraduate programming courses is also an important mean of detecting errors. Many language features and weak-spots were tested for curiosity or lack of documentation by our students, often resulting in unexpected language constructs.

Another software-relevant aspect is *maintenance*. Software maintenance is necessary to correct errors, to implements specification changes, and to enhance performance. As a research project, our software is bound to be changed many times, when different features and implementations are tested and evaluated. It is thus very important that the compiler implementation is robust and easily maintainable; the code must be written in a self-documentary fashion, because documentation will be written only when the project is frozen at a later stage.

5.7.1 Designed for ease of maintenance

In the second part of [Rea00], the design underlying the compiler implementation is explained focusing on the software maintenance aspect; it is stressed that the compiler implementation relies on strong typing to check invariants automatically and intercept errors as soon as they happen. We designed our compiler data structures using type extension in the fashion proposed by Griesemer [Gri93], Fr¨ohlich [Fr¨o97], and Mikheev [Mik00] as opposed to a single structure designed to fit all needs[12] as documented by Wirth [GW92, Wir96] and Crelier [Cre90, Cre91], which is more appropriate for programming-in-the-small. Using different types is in our opinion the best way to deal with heterogeneous data structures like a symbol table or a parse tree[13]. We can only agree

[12]A record with many different meanings selected by a mode fi eld, like in a variant record, but where every fi eld is visible and accessible

[13]An overview of the advantages and disadvantages of object oriented design can be found in [M¨os98, Chap. 13]

with E. Raymond's quote [Ray99] (paraphrasing F. Brooks [Bro95]): *"Smart data structures and dumb code works a lot better than the other way around"*.

Using a type for each concept modeled in the software has the advantage to make many *invariants* obvious and automatically checked by the compiler (either statically or at run-time). Code becomes more robust as inconsistent structures can be detected at the instantiation time, which has an extremely positive effect on debugging too. Maintenance is also simplified, because the invariants are enforced by the language and not by the programmer. As a side effect, a few explicit run-time consistency checks can be removed from the program, making the code leaner and simpler.

Having a separate type for every object kind makes the source code longer, because more types must be declared. It nevertheless improves the *readability* of the software, as only the interface of a compilation unit must be understood: when the structures are not expressive enough, the reader has to browse through the implementation to understand the structure and find the hidden dependencies between the fields defined in a data structure; this is of course more complex and time consuming. It's noteworthy that the data structures do not affect the size of the compiled code and that memory usage is reduced because the structures better fits the size of the data.

For efficiency reasons, we used information hiding sparingly. To execute efficiently, a compiler needs all the information available: enforcing information hiding would only make the compiler slower, because the same information would have to be recomputed many times or accessed through procedure calls. We also designed the compiler to add information, but never change it; thus the invariants must be only checked when data is created. Read-only export of the data in this case is enough to ensure that the invariant, once established, will never be broken, while giving a fast and efficient access to the data.

We recognize the following design choices as having a beneficial effect on the software:

- using separate types for distinct concepts to have invariants checked automatically
- precondition testing in procedures
- error detection as soon as possible
- immutable data (create, never change) in conjunction with read-only visibility
- avoiding side-effects and global variables

Most of the features used in the implementation rely on the underlying language that must provide them. Languages without strict type checking make such implementations more difficult. Support for assertions and contracts allow to go further in this direction and improve software reliability a lot.

5.7.2 Regression Testing with Hostess

Regression testing is the process of testing changes to programs to make sure that the older programming still works with the new changes. Before a new version of a software product is released, the old test cases are run against the new version to make sure that all the old capabilities still work. The reason they might not work is because changing or adding new code to a program can easily introduce errors into code that is not intended to be changed [Wha]. Of course, regression testing doesn't prove or ensure program correctness, but it allows avoiding past and obvious errors. As the test suite grows, the number of possible errors covered by the test suite increases. For each language feature introduced, new tests must be written; error fixing is done by first writing a new test case for the test suite.

Modern software engineering techniques as extreme programming [Bec99] also stress the use and design of test cases while implementing the application, and the importance of using a test-suite to perform regression testing.

We tested our compiler using *Hostess* [RvA], a freely available test-suite for Oberon Language Processors. Hostess has allows to test single language features separately, simplifying the compiler bootstrapping and maintenance. Hostess is designed to perform many functions:

- check parser acceptance of correct code
- check parser rejection of erroneous code
- check execution of generated code
- support for language dialects

The test suite contains only the system independent files (test cases and schedule). For every system, an application to execute the test cases must be written.

Hostess concepts

Let P be a *language processor*: an application that takes source code as input; P is the application to be tested against the test-suite. Let be $o := P(i)$ the result of applying input i to P with output $o \in \mathcal{O}$ where

$$\mathcal{O} = \{skip, crash, reject, accept, error, run\}$$

$P(i)$	**Description**
skip	i was ignored
crash	i caused an exception in P
reject	P rejected i (it doesn't belong to the language)
accept	P accepted i (it does belong to the language)
error	execution of i causes an exception
run	execution of i terminates without exception

***Table 5.12:** Test results*

The outputs are described in Table 5.12. A *test-case* c is a tuple (i, o) which describes the expected output o for input i. We define $test$ as follows:

$$test(c, P) : \begin{cases} pass & o = P(i) \\ fail & o \neq P(i) \end{cases}$$

Usually $skip$ and $crash$ will never be expected results, and they should cause the test to fail. A *test-suite* $\mathcal{C}$ is a set of test cases $\{c_0, ..., c_n\}$. A language processor P is *compliant* relative to test suite $\mathcal{C}$ iff

$$\forall c_k \in \mathcal{C} \rightarrow test(c_k, P) = pass$$

which means that P returns the expected result for all submitted tests.

Support for Language Dialects

The testing methodology explained above works well when there is only one clearly defined standard to be tested. In our case, the language has evolved into many, mostly backward compatible, language dialects and extensions. To cope with this situation, we extend our definition of test case c to (i, o, L), where L is the set of languages addressed by the test. Obviously, if P is a processor for language l, it is expected to pass all tests c_k where $l \in L_k$ and to reject the others. The tests cases for the dialects other than l are used as negative tests. Thus

$$test'(c, P) : \begin{cases} pass & (l \in L) \wedge (P(i) = o) \\ pass & (l \notin L) \wedge (P(i) = reject) \\ fail & otherwise \end{cases}$$

P complies to language l iff

$$\forall c_k \in \mathcal{C} \rightarrow test'(c_k, P) = pass$$

This requirement is sometimes too restrictive. If language l' is an extension of language l, then a language processor P for l' will result not to be compliant to l; this although P understands l to some extent. To cope with this often occurring case, we introduce a less stringent requirement: P *accept*s language l iff

$$\forall (c_k \in \mathcal{C}) \wedge (l \in L_k) \rightarrow test(c_k, P) = pass$$

The weakened constraint *accepts* is useful, because it express that the compiler understands to some extent language l; it just doesn't require to fail the test cases for other language dialects. In a real case, we would expect an Active Oberon compiler to comply to Active Oberon and to accept Oberon-1.

Hostess Implementation

The Hostess [RvA] test suite contains about 150 test cases[14] for the languages Oberon, Oberon-2, and Active Oberon. Hostess was launched as a joint effort of the author (maintainer of the ETH-OP2 and Paco compilers) and of M. van Hacken (maintainer of the ooc compiler [Ooc]). Hostess'

[14]this number can be misleading; positive tests can be grouped in a single test case, only negative tests must be separated into distinct test cases to be sure that every test case is rejected or causes an exception

goal is *”to provide a set of conformance tests for Oberon language consumers (e.g. compilers and language processors)*”; this implies that Hostess must be platform independent and dialect independent. The test schedule is formulated in XML format [BPSMM00], to simplify its parsing and readability. Listing 5.19 shows an extract of the test schedule.

```
<testcases
    profile="BUILT-IN TYPES"
    default-lang="O1 O2 AO">

    <test id="int0" type="run" file= "HOTInt0.Mod">
        test integer constants, conversions, operators
    </test>

    <test id="interr0" type="reject" file="HOTIntErr0.Mod">
        error LONGINT -> INTEGER
    </test>
</testcases>
```

Listing 5.19: A snapshot of Hostess’ test schedule

Language Grey Zones

During the preparation of the Hostess’ test cases we run into the *”testing the undefined”* problem. Wirth’s Oberon Language Report [Wir88] is very short; the main reason for this is that many details are left out[15], often relying on the reader’s common sense. This approach makes the report simple to understand and the language easy to learn, but creates *language grey zones*. Therefore, compiler implementations can differ on important features, though being completely compatible with the language definition; this has a negative effect on the portability of programs to different Oberon compilers.

The undefined language parts where also a problem when implementing the test cases. The most problems where caused by the untyped constants and by exception and error handling. One of the most common problem was the use of constants in built-in polymorph functions and procedures in conjunction with constant folding, one of the simplest thus most often implemented compiler optimization. More details on these problems are discussed in Section B.

[15]in particular, the report specifi es which programs are legal, but doesn’t make the distinction between errors and exceptions

6
The Aos Kernel as Language Interoperability Platform

6.1 Introduction

Aos [Mul00, Mul02] is a compact multiprocessor kernel for active object-based systems. It has a flexible design and can be used as a low-overhead operating system on hardware ranging from small devices to powerful server machines. It is currently available for the Intel IA-32 single-processor and multi-processor machines and Intel StrongARM-based machines.

The Aos kernel programming model consists of classes composed in a single hierarchy, and implementing zero or more interfaces; modules are containers for procedures, types, and variables, and are used as compilation unit, unit of deployment, and namespace. Other types include statically and dynamically-sized arrays and structures, available as stack-allocated value-types and heap-allocated reference-types.

The kernel provides dynamic loading and linking of modules; and support for the allocation and automatic reclamation (garbage collection) of class instances.

The kernel provides complete support for active objects, by providing object-bound activities, synchronization and atomicity.

Other services provided by the kernel include exception handling, and a plug-in architecture for dynamically installing hardware drivers for disks, network-cards, and graphic adapters.

In spite of the many requirements to the kernel, its implementation is lean and efficient; the core of the kernel fits into less than 50 KB, while the main hardware drivers (disk, keyboard, display, and network) need additional 140KB.

6.2 Aos as Interoperability Platform

The object oriented programming model implemented by the kernel, and the exposure of all the services required to support a mordern programming language, make Aos a very interesting run-time platform. The whole system can also easily be extended by simply adding new modules to it, to provide features that are missing.

The pervasive plugin architecture allows to integrate other systems on Aos, even in parallel to

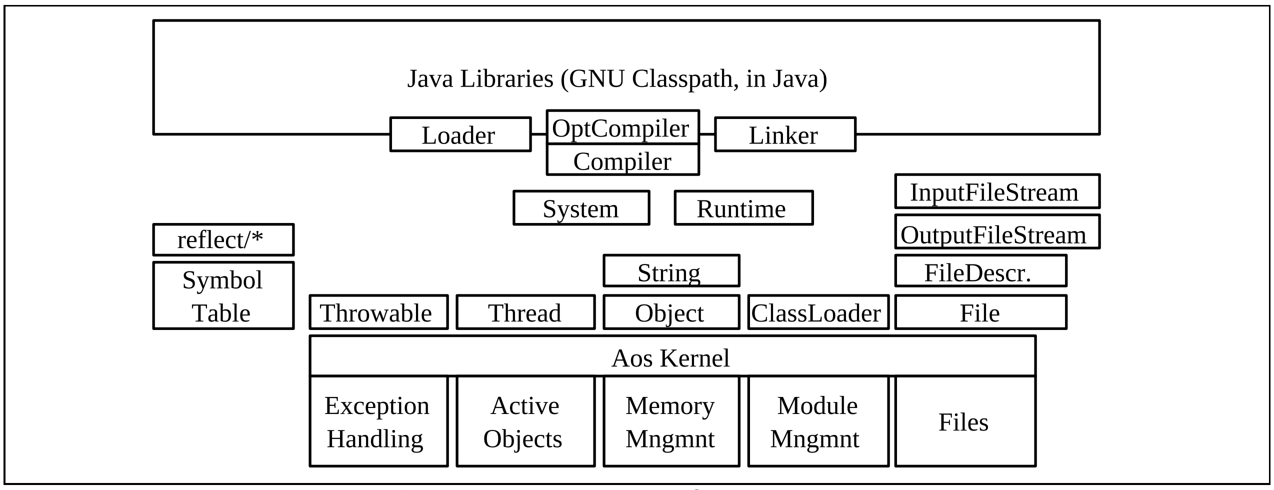


Figure 6.1: Overview of a JVM on Aos

the existing ones. New module loaders can be added allow loading other object-file formats on-demand. This makes Aos independent of the object-file format and even independent of the intermediate code used. Code like the intel assembler, the Java byte-code or CLR-IL can be supported by the providing a just-in-time compiler for them. Other higher-level models like OMI [Fra94] or even source code [Mar96, Zel] can also be supported. Figure 6.1 gives an example of a the Jaos JVM on top of the Aos kernel.

The higher-level programming model offers by Jaos can dramatically reduce the offert of porting another language or run-time on top of it. Comparing the SableVM JVM built on top of Linux and the Jaos JVM on top of Aos is instructive: SableVM was a medium-size project with a dozen of graduate students working on it, whereas Jaos was a one-man project.

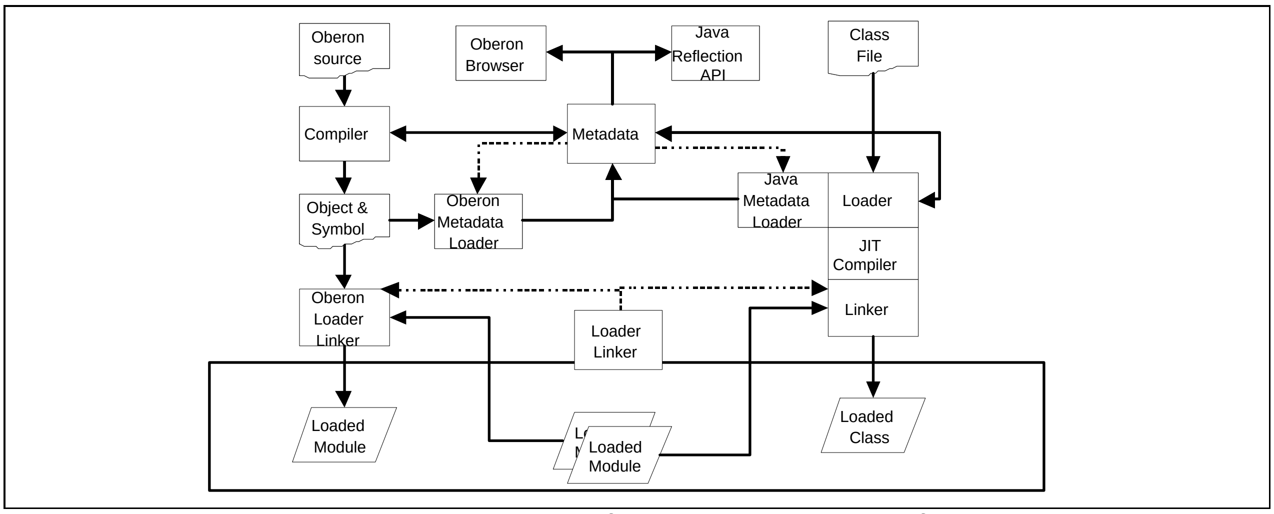


Figure 6.2: Overview of an Interoperability Platform

By also providing a common shared metadata service, Aos can be turned into a language interoperability platform. The metadata repository provides reflection capabilities. The information is structured using the Active Oberon model. A key feature of the metadata is its plugin mechanism, which allows loading metadata available in different format. Figure 6.2 gives an example with Oberon and Java.

6.3 Run-time Layer: API and Conventions

This section describes the API and conventions defined by the Aos run-time layer. This layer is used as interoperability layer.

Complying to the conventions defined here ensures that all components rely on the same model, thus allowing a seamless sharing of structures and functionalities among components.

The part of the layer defined by the API is enforced by the underlying code, the conventions must be enforced by the compilers producing the code. The compliance of the dynamically allocated objects to the type system is partially enforced by the garbage collector, which requires the data to be conform to the defined protocol; because of the unpredictable execution of the garbage collector, it is not wise to rely on it for this purpose[1].

The description of the interoperability layer is divided in four parts: type system description, concurrency support API, module description, and calling convention description.

6.3.1 Type System

This section describes the type system provided by the run-time layer of the Aos kernel.

Aos type system consists of basic types, compositions of types in arrays and objects. Object types build a single inheritance hierarchy; furthermore each object type defines fields and methods associated with it, and can be allocated on the heap or on the stack. Object types are used to implement both structures and classes. Modules act as deployment units and namespaces containing types, variables, procedures, and constants.

Basic Types

Table 6.1 shows the core basic types used in the run-time layer; more types can be defined. Each type defined a meaning, size, alignment, and encoding of the values having that type; it furthermore restrict the legal operations applicable on the values. All types are stored as little-endian, and the alignment follows the Intel specifications [Int99, Chap 2] to minimize memory access penalties. If two variables with different size—and thus different alignment—are adjacent, a gap between them is possible.

Pointers are heap addresses; references are either heap or stack addresses; references are allowed only on the stack. Pointers in the heap must be known to the garbage collector, whereas pointers and references on the stack are collected using a conservative collector. References are mostly used to implement the reference parameters in procedure calls.

[1]Another reason to avoid the garbage collector as conformance checker, is the non-graceful handling of data with invalid format, which usually results in the system crashing without warning nor information

Encoding	Size	Alignment
char (ASCII)	1	1
char (UNICODE 1.0)	2	2
char (UNICODE 2.0)	4	4
signed integer	1	1
signed integer	2	2
signed integer	4	4
signed integer	8	4
single precision IEEE-754	4	4
double precision IEEE-754	8	4
bit pattern	4	4
address (reference)	4	4
address (pointer)	4	4
boolean	1	1

Table 6.1: *Core Basic Type Sizes and Alignment*

Objects

Object types are user-defined structures that can be allocated on the stack and on the heap. Every object type has an *object descriptor* containing the size of the type, the pointer offsets for the garbage collector, the supertype table for type tests, and a method table.

Objects are used to implement both records (user defined structures) and classes with single inheritance.

Object instances in the heap carry an hidden pointer to the object descriptor and—if needed—space for other hidden fields used for concurrency and synchronization. Object allocated on the stack do not have those hidden fields.

Figure 6.3 shows an object instance allocated on the heap and its object descriptor.

User defined fields have a positive offset relative to the object base. When an object type is extended through sub-classing, the new fields are appended at the end of the object; this ensures that the offset of a field doesn't change in a subclass. Fields with a negative offsets are reserved for storing system information.

The first negative field is the pointer to the object descriptor. Objects that are used as monitors have also hidden fields to store the information about lock state and owner[2].

Object descriptors contain the size of the object instance for the memory allocator, a list of all the pointer fields for the garbage collector, and a method table containing the entry points of the methods bound to the object type. Also in part of the descriptor is a table containing references to the type descriptors of all supertypes of the current object; this table is hierarchically ordered to allow efficient type compatibility tests.

To simplify garbage collection, object descriptors are themselves normal heap objects; they

[2]A detailed description and explanation can be found in Muller [Mul02], stemming on the original idea of [DR97, Dis97]

contain their own descriptor, so that they can be garbage collected without special handling.

Aos has supports only for single class inheritance; this allows for simplifications that make the object descriptors and the operations on the object simple and efficient. All the informations are stored at fixed offsets, allowing their retrieval with a single memory access. Each object field can be accessed at a fixed offsets known at compile-time; methods are at a fixed offset relative to the object descriptor pointer. The garbage collector can exactly identify the fields containing pointers during the scan phase. Type equality tests can be performed with a direct memory access, while type compatibility tests[3] require a double memory access.

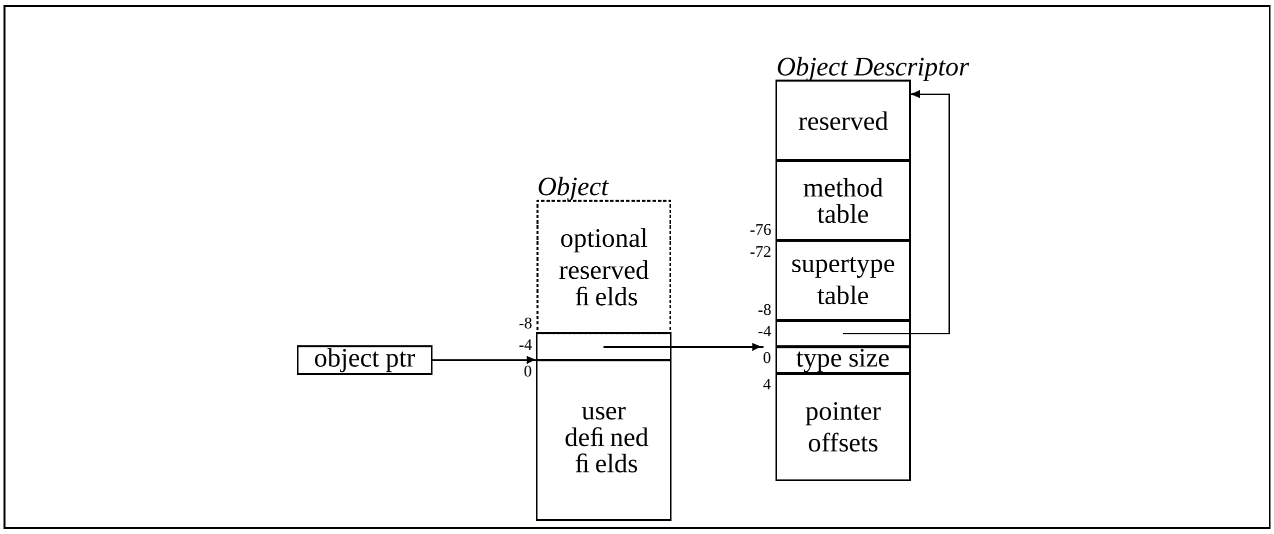


Figure 6.3: The Object Memory Layout

The Aos kernel provides the function `AosStorage.NewType` to create a new object descriptor.

To allocate heap objects, two kernel functions are available: `AosStorage.NewRec` and `AosStorage.NewProtRec`. Both take the object descriptor as parameter and return a pointer to a freshly allocated object instance on the heap. The latter function also allocate the hidden fields needed to handle concurrency in the object.

These functions are inlined by the Oberon compiler when `NEW` is called; Jaos translates the byte-code `new` to a call to the same functions.

Interfaces

An interface defines a set of method signatures; a class implementing it commits itself to supply an implementation to each method in the interface. It is a contract that the class must fulfill.

Interfaces are realized using an *interface method table* (IMT), similar to the *virtual method table* (VMT) used for the class methods. Multiple implementation of interfaces precludes a direct access to the IMT like is done for the VMT; the requested IMT must be located at run-time. All

[3]check if an object is a subclass of a given type

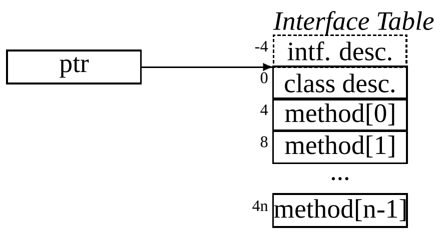


Figure 6.4: Interface Method Table

IMT are stored in a global hash table [Lai00] with the class and interface descriptor pointers as keys.

Besides the method table, the IMT contains a pointer to the descriptor of the implemented interface, and a pointer to the descriptor of the class implementing the interface. The interface descriptor is also used as object descriptor for the method table because it contains the size to be allocated for each table.

The kernel provides two operations on the global hash table: `Lookup` searches for the IMT of an object class implementing a give interface; `Register` adds an IMT to the table. Interface implementation check is performed using the lookup operation.

```
PROCEDURE Register(imt: PTR);
PROCEDURE Lookup(class, intf: PTR): PTR;
```

Listing 6.1: Interface Kernel Calls

The IMT contains a pointer to the type descriptor of the class; the interface information is implicitly stored in the type descriptor of the IMT (this is possible because interfaces are flattened to eliminate their hierarchy).

Figure 6.4 shows the memory layout of an IMT.

More efficient implementations[4] are known [ACF$^+$01], but their complexity and memory usage are large and thus not adapted to a system like Aos.

Dynamically Allocated Arrays

Dynamically allocated arrays are heap allocated structures containing an homogeneous array of elements. Each dynamic array carries type information for the garbage collector, and the array dimensions for performing boundary checks. Multidimensional dynamic arrays are supported as well as single dimensional ones.

Figure 6.5 shows an example of a dynamic array. Because every dynamic array has different dimensions, the garbage collection related informations are stored directly in the array structure, including a pointer to the object descriptor of for the array elements—if the element type is an

[4]obviously, less effi cient solutions are also known

object containing pointers. The padding field in the dynamic array is used to make the first element aligned. The offset of the first array element relative to the block begin is statically known and depends only on the number of dimensions.

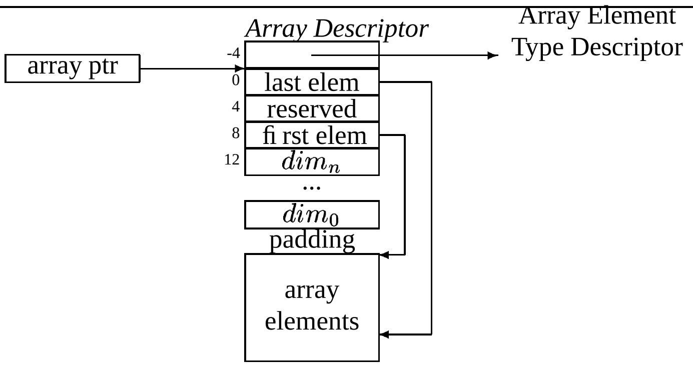


Figure 6.5: The Dynamic Array Memory Layout

Access to the elements of a 1-dimensional dynamic array is performed with a memory access relative to the array base. For multi-dimensional arrays, the array dimensions must be first read to compute the effective element offset; as an example, the offset of an element in a 3-dimensional array is

$$offset = padding + baseTypeSize * (index_2 + (dim_2 + index_1 * (dim_1 + index_0)))$$

Dynamic arrays are allocated by calling the kernel function `AosStorage.NewArr` or `Aos-Storage.NewSys` if the array contains no pointers.

```
  PROCEDURE NewArr (VAR p: PTR; elemTag, numElems, numDims: LONGINT);
  PROCEDURE NewSys (VAR p: PTR; size: LONGINT);
```

6.3.2 Concurrency Support

The Aos kernel provides an implementation for active object-based systems. In this paradigm, every object corresponds to an activity executed concurrently to the other activities in the system. Concurrency support can be divided in three main categories: activity, protection, and synchronization.

The following system calls—described with Oberon syntax—perform the basic functions required to support concurrency:

```
PROCEDURE NewThread(body: Body; prio: LONGINT; flags: SET; obj: Object);
PROCEDURE Await(cond: Condition; slink: LONGINT; obj: Object; flags: SET);
PROCEDURE Lock(obj: Object);
PROCEDURE Unlock(obj: Object);
```

`NewThread` associates a `body` (usually the body method of the object) with an object and schedules it for asynchronous execution. Because the object initializer has to be called before starting the activitity, and because an object may have more than one activity, this function is kept separated from the object allocation function.

The `Lock` and `Unlock` take and release the lock associated with an object. If the lock is already taken, the current activity is preempted until the lock is released. Locks are not reentrant; Section 7.2.2 describes how the reentrant Java locking mechanism is implemented using Aos locks.

`Await` synchronizes the current activity with a condition; checking and scheduling of the condition is delegated to the kernel. While the activity is preempted, its lock on the current object is released. `condition` is a boolean function evaluating to true whenever the condition is true; `slink` is the frame pointer of the procedure where await is called, and is used to evaluate the function in the proper context without having to perform a context switch. Discussion and explanation of the synchronization primitive can be found in Disteli [Dis97], Disteli and Reali [DR97], and Muller [Mul02].

6.3.3 Modules

Modules are compilation and deployment units. They are containers for code, data and type descriptors; additionally they carry administrative structures that permit the linking of newly loaded modules to them.

Figure 6.6 shows a loaded module structure, and all the section inside it.

The code section contains the machine code of the module.

The data section contains both variables and constants [5]. Variables and constants are accessed relative to the static base pointer (sb) that points to the boundary between the two sections. Variables have a negative offset relative to the static base, while constants have a positive offset. Both sections are kept together because Aos has a single globally shared address space, and rely on the language for type-enforcement, thus removing the expensive need for separate address-spaces for each process.

The *export* and *import* sections contain the information for the dynamic linking of modules. Entities are referenced by a fingerprint [Cre94, Kis95] computed over the name and signature or type. This makes the section shorted and allows to detect entities whose signature has changed. Each exported entry consists of a fingerprint and the address in the code or data section; each imported entry consists of a fingerprint and the address of the patch list.

6.3.4 Calling Convention

The calling convention is an implicit interface of Aos; the applications generating code to be executed on the system must enforce this convention.

Parameters are passed left-to-right; by-value and by-reference modes are possible. Functions can return only values. User defined types (structures and arrays) can be returned as values too. All generic purpose registers are caller-saved; the stack, frame pointer and program counter registers are handled by the calling convention.

[5] the literature usually refers them as data and text sections, see Levine [Lev00]

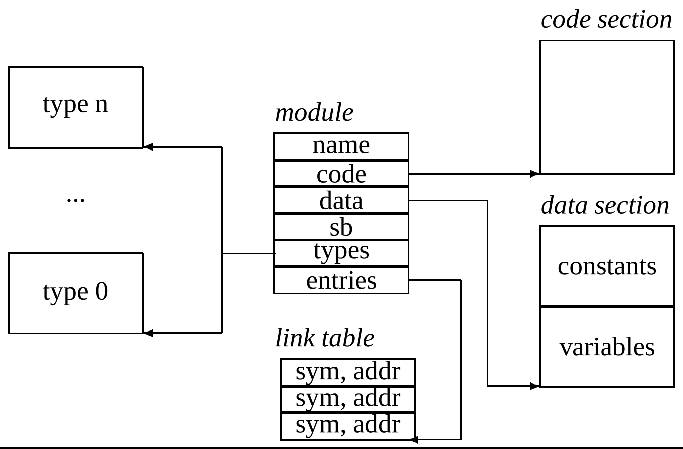


Figure 6.6: A Module

The frame pointer (FP) points to the current procedure active frame (PAF). The procedure parameters are at a positive offset relative to the FP, the local variables at a negative offset. The callee's program counter and PAF pointer (also called dynamic link) are saved on the stack too.

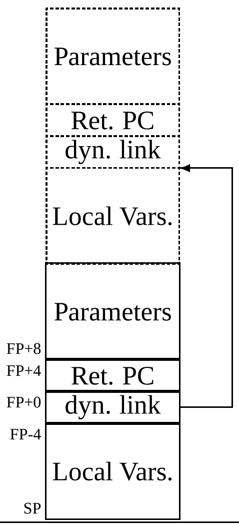


Figure 6.7: Procedure Allocation Frame Organization on the Stack

The calling convention explicitly supports the Oberon-X language extension [Jan98] defining operators; this extension requires the ability to return user defined types. Because values of those types usually do not find in a register, the caller has to pass an hidden return descriptor containing the destination of the value. This implementation is an improved version of [Mor97], and allows to return open arrays without using the heap.

Table 6.2 shows how parameters are pushed and values returned. Parameters are pushed left to right, according to the formal parameter list of the procedure signature. Value parameter of user-

```
save registers
push return descriptor
push parameters
call routine
restore registers
```

Listing 6.2: Caller's code overview

```
;
; enter procedure
;
push FP       ; store dyn. link
mov  FP, SP   ; set new FP
sub  SP, N    ; allocate local vars
...
;
; leave procedure
;
mov  SP, FP   ; deallocate local vars
pop  FP       ; restore FP
ret  M        ; return to caller and
              ; and remove parameters
; ret M is equivalent to
;   pop PC
;   add SP, M
```

Listing 6.3: Callee's code overview

defined types are passed by value[6], if their size is known at compilation-time. This is more efficient when operators are chained, as the return value of an operator is passed directly as parameter to the next operator.

For *nested procedure* calls, the static link[7] is passed as last hidden parameter. The compiler is allowed to omit this parameter , if it detects that it is not needed.

Method calls pass the object self-reference as last hidden parameter.

Returning Records

User defined structures like records do not fit into a processor register, and are thus handled specially. To minimize the number of times the value has to be copied, the caller gives the callee the address where the result has to be copied to.

The caller passes the record size and destination address (a record value-descriptor) as first hidden parameters. When leaving the function, the caller copies the requested number of bytes to the destination address and removed the record value-descriptor from the stack.

The size parameter is needed because of the type compatibility rules, which make a record compatible with all its supertypes. When the function value is assigned to a variable having superclass type, only the fields present in the superclass have to be copied.

[6]In the original Oberon system only a reference to the structure or array was passed and the callee had to make a local copy of them to work on

[7]the pointer to the stack frame of the procedure textually containing the current procedure

Type	Value Parameter	Reference Parameter	Return Value
CHAR SHORTINT	Value:1	Ref:4	AL
INTEGER	Value:2	Ref:4	AX
LONGINT SET POINTER	Value:4	Ref:4	EAX
HUGEINT	Value:8	Ref:4	EDX:EAX
REAL	Value:4	Ref:4	ST(0)
LONGREAL	Value:8	Ref:4	ST(0)
Record	Value	TD:4 Ref:4	RecSize:4 Ref:4
Static Array	Value	Ref:4	Ref:4
Open Array n-dim	Dim-n:4 Dim-1:4 Ref:4 copied by callee	Dim-n:4 Dim-1:4 Ref:4	on top of Stack

***Table 6.2**: Calling Convention Overview*

Table 6.3 shows the code to return a record.

caller	**callee**
`push SIZE(rec)`	
`push ADDR(rec)`	
parameters	
call	*routine code*
	`move ADDR(value), ADDR(rec), SIZE(rec)`
	`return`

***Table 6.3**: Returning a Record*

Returning Statically-Sized Arrays

Statically-sized arrays do not fit in a processor's register. To minimize the number of times the value has to be copied, the caller passes to the callee the array's destination address, and the callee eventually copies the result to this address. A size parameter is not needed, because statically-sized arrays are compatible only with themselves.

The caller passes a reference to the array destination (an array value-descriptor) as first hidden parameter. When leaving the function, the caller copies the returned array's value to the destination address and removes the array value-descriptor from the stack.

Table 6.4 shows the code to return a static array.

Returning Open Arrays

Open arrays as return values are the most complex case of user defined return values.

caller	**callee**
`push ADDR(arr)`	
parameters	
call	*routine code*
	`move ADDR(value), ADDR(arr), LEN(arr)`
	`return`

***Table 6.4**: Returning a Static Array*

The difficulties stem from open arrays own nature. First, their size is not known in advance, and thus they cannot be preallocated. Second, because of the type compatibility rules they are compatible only to open array parameters.

Open array return values always occour in the following code pattern:

```
PROCEDURE P(... x: ARRAY OF T ...);
PROCEDURE Q(...): ARRAY OF T;

... P(... Q(...) ...) ...
```

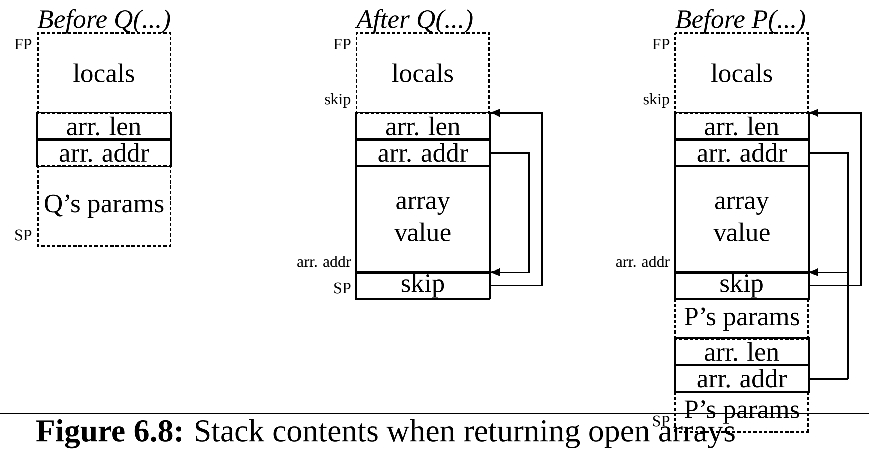


Figure 6.8: Stack contents when returning open arrays

Figure 6.8 shows the stack contents while returning an open array. Open array values are returned by `Q` on the top of the stack, and kept there while `P` consumes them. Because open arrays are passed by reference, it is not possible to pass them directly as parameters: they are temporarily allocated on the stack before the other parameters and then a descriptor consisting of the arrays' length and address is pushed as parameter.

The implementation of Q's `return` is particularly complex. In particular it must take care of saving the return address and frame pointer before overwriting its own procedure activation frame; it must also ensure that the whole result is on the stack to allow the garbage collector to keep the referenced objects alive[8]

[8]this requires to grow the stack under given circumstances

After procedure P has been called, the callee must remove the temporary values from the stack. This is done by popping the value on top of the stack into the stack register for each temporary array.

Optimized Cases

Returning user-defined types can be optimized in two cases. First, when the returned value is directly passed as parameter to a procedure (return-to-call). Second, tail-calls can be optimized to minimize the number of copies of the value.

In the return-to-call case, the parameter is copied directly to the stack location where it has to be pushed. The caller allocates the space for the returned value on the stack where the parameter has to be passed, then it pushes the return value-descriptor on the stack and invokes the function. Upon return, the function copies the value directly on the stack.

As an example, we define a type and two procedures and a call:

```
TYPE
  R = RECORD .... END;

  PROCEDURE P(c: LONGINT): R;
  PROCEDURE Q(a: LONGINT; r: R; b: LONGINT);

  Q(a, P(c), b);
```

The optimization is as follows:

```
  ;
  ; Q(a, P(c), b)
  ;
  ; parameters for Q(a, P(c), b)
  push a
  sub  SP, SIZE(R)    ; allocate space for P
  ;
  ; parameters for P(c)
  push SIZE(R)
  push SP-4           ; destination address for P
  push c
  call P
  ;
  ;
  push b
  call Q
```

Tail-calls are optimized by passing the return descriptor to the callee, instead of creating an own temporary variable; the callee copies the value directly to the final destination, and the current function can return without having to copy the value. The following example shows the tail call optimization:

```
  ; RETURN P(c)
  ;
  push Descriptor.size   ; pass hidden return value descriptor
```

```
push Descriptor.addr
push c
call P

; leave procedure
;
mov  SP, FP
pop  FP
ret  N
```

7 Case Study: The Jaos JVM

I've very often made mistakes in my phisics by thinking the theory isn't as good as it really is, thinking that there are lots of complications that are going to spoil it.

— R. Feynman

7.1 Introduction

7.1.1 Goals

The Jaos JVM has two goals:

- prove that the Aos kernel is generic enough to support another language and operating system
- investigate the interoperability between the Oberon and Java languages and systems

These goals are apparently far apart and unrelated, but in fact their realization is based on the same idea: using the kernel conventions as much as possible.

Jaos is to be as thin as possible, and thus rely on the kernel libraries and conventions for its implementation. The Object Model of Aos appears to be very similar with Java's one, and there is no need to implement every feature a second time, so most of the conceptual work consists in choosing an appropriate mapping from Java's model to Aos' model.

Following this strategy the second goal becomes quite obvious: as Java and Oberon code rely on the same implementation conventions, the interoperability between the two languages is simple to achieve without any glue logic, and requires only the sharing of the metadata information for the separate compilation and linking of the compilation units; interoperability is then only restricted by the lack of features in a language or in the common type system: for example, Java has no direct support for static structures.

A third unspoken goal is less technical but nonetheless to be ignored: this project opens the Java world to Oberon. Even if this is not a research relevant goal, it is nevertheless an important application for the Aos platform.

7.1.2 Java, the Java VM, and the Java Libraries

The Java Language [GJS96] is a recent object oriented language. It imposes type safety through strong typing; as a reaction against C and C++ misuse of pointer arithmetic, Java completely abolishes pointers—even abstract ones—, and makes every object dynamic. Cornerstones of the language are *classes*, containers for fields and methods; both can class members be either static (class bound) or dynamic (instance bound). Through single inheritance of type and implementation, classes build a hierarchy originating at the root class `java.lang.Object`. Interfaces are lists of abstract methods defining a contract; a class commits itself to implement zero or more interfaces, allowing thus a simple multiple type inheritance[1]. Java Arrays are always dynamically allocated, and behave like classes extending `java.lang.Object`.

A Java Virtual Machine (JVM) [LY99] is the preferred platform for the execution of Java programs. The JVM is a stack machine executing a typed and verifiable byte-code. Most of the byte-code operations are low-level ones (addition, multiplication, comparison), whereas the few high-level instructions (interface invocation, array allocation, field access) decouple and hide the allocation and layout information to the program. Defining a Virtual Machine as execution environment has the great advantage of making the programs portable and the ideal way to share implementations across a distributed medium like the Internet. To execute Java programs, programmers need to implement a byte-code interpreter on the host system[2]. Recent JVM implementations use a just-in-time (JIT) compiler to make the execution of the program faster. Dynamic optimization schemes have also been devised to speed up the methods that are heavily used, as the language is inherently designed to be slow, being every structure dynamic. Compilation has the drawback of creating possibly long delays during the machine startup.

An impressive amount of libraries [CLK98, CL97] are defined to be used with Java. The whole JVM is part of the libraries, and many features like the file system, user interface, network interface, and others are defined in the standard API. The number of libraries has been rapidly growing: Java 1.0 had 8 packages and 212 classes; Java 1.1: 23 packages, 504 classes; Java 1.2: 59 packages, 1520 classes. Since version 1.1, Sun defines three platforms—Enterprise Edition, Standard Edition, and Micro Edition—which differ in the number of packages and classes included, and are tailored to different application domains. In particular, the Micro Edition is targeted at the embedded systems market segment, and is highly configurable to allow to include only the bare minimum of the libraries in a system configuration.

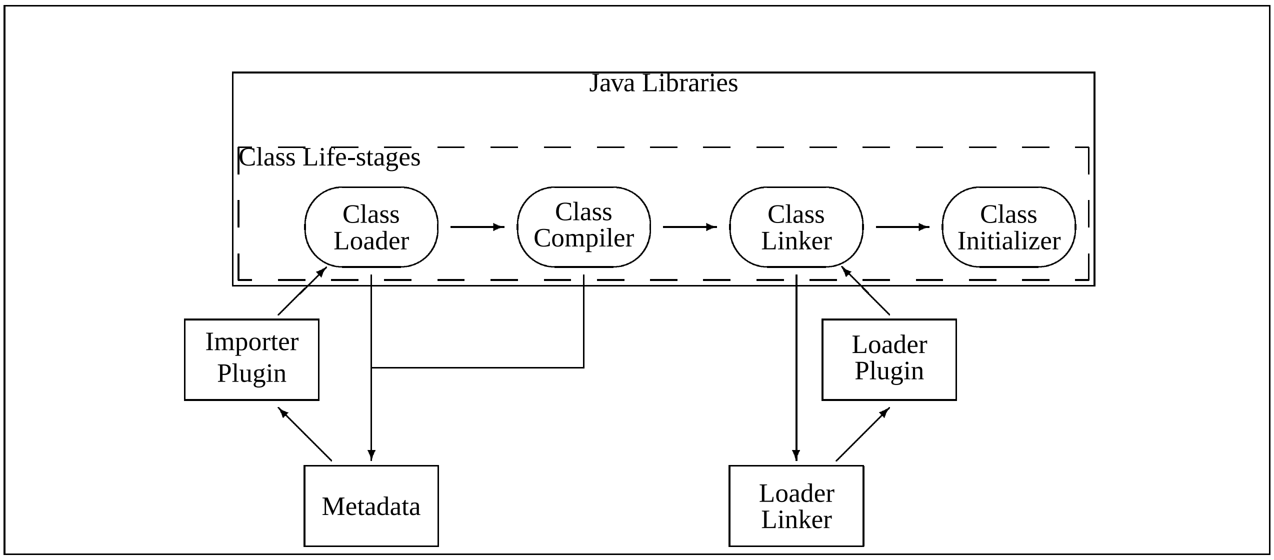
The definition of the Java Native Interface (JNI) [Lia99] allowed to break the isolation of the Java programs. The JNI allows to interface Java with code written in another language. The initial idea was to allow programmers to use their legacy code, or a more efficient language for time critical routines. In fact, the JNI opens the doors to a rudimentary language interoperability between Java and any other language. The JNI achieves a loose and expensive interoperability, as

[1]Being interfaces completely abstract, no conflict through multiple implementations of the same method are possible. Implementation of the language is also simplifi ed, since no fi elds are allowed in an interface

[2]once used to be a simple task, but the number and features of the mandatory libraries to be supported makes this a complex implementation

data has first to be retrieved through conversion routines. Main reason for this strong decoupling is to enforce the type safety outside of the JVM and a correct handling of those objects by the garbage collector, by forcing the native routines to work with copies of the data. This allows only for a rather coarse grain interoperability, usually on the level of software components, as the overhead for calling a procedure is quite high[3].

7.1.3 VM Overview



Jaos is composed by four parts that closely follow the a class' life-stages: the class loader, the class compiler, the class linker, and the class initialization. In addition, indispensable for the proper execution of the Java programs, are the Java Libraries; they are mostly implemented in Java itself and can be considered as an application for the JVM itself. Nevertheless we tried to integrate the JVM and the Libraries as much as possible, striving for functionality reuse. The only limit to integration and reuse is given by bootstrapping issues.

The *class loader* internalizes a Java class file, and performs the allocation of the fields and methods. It checks if a native implementation of the class is available, and performs consistency checks between the native and Java versions. The metadata of the class is stored in the common metadata repository.

The *class compiler* translates the Java byte code into Intel IA32 native code compliant with the Aos calling convention and data layout.

The *class linker* resolves external references to fields and methods, and initializes the method and interface tables.

The *class initialization* is performed by calling the class static initializer. The JVM specification and Java Language specify a lazy initialization of classes. This is usually before the first use

[3]Sometimes very simple routines have to be called through the JNI to perform low-level operations, which are not possible in the Java language

of a static field or static method. Special rules and exceptions apply to static final fields.

The *Java Libraries* are mostly implemented in Java and available in byte-code form to the JVM. The Library Classes are dynamically loaded by the JVM when needed. Some classes require a native implementation to interface with the underlying kernel for functionalities which cannot be supplied in Java itself; this includes the file access, object locking, thread operations, network operations, and textual and graphical IO. The loader and the linker provide support for native methods implemented in Oberon.

Jaos uses the system's common metadata repository (Section 5.2) for storing and retrieving symbol information through all the life phases of a class. This is a strategical choice: using the common repository for the symbol information makes the information available independently of the language it is defined in. Java and Oberon symbols can be accessed without further bridging or wrapping them; of course this requires that both compilers comply with the various kernel API and conventions.

7.1.4 Related Work

The number of Java related projects is growing at an exponential rate. The JVM implementations are very important for the Java language to spread around and be usable, in particular to defeat the inherent slowness of the Java programs. This section introduces some related work, mostly other JVM implementations or implementation related problems. Most of the research is currently focusing on compiler optimizations targeted at Java.

As original Java creator, Sun was the first company to have a running JVM. The first version—now called *Classic VM*—was constructed around a byte-code interpreter. Sun also provided for the first Java to byte-code compilers. Sun implementation is considered the reference for every other JVM and library implementation, in particular whenever the documentation is not exhaustive. Hot Spot [Hot99] is Sun's high performance JVM. There are a client and a server version, where the client is optimized for fast startup times and small memory footprint, while the server is optimized for operating speed, in particular with more aggressive compiler optimizations [Gri00]; HotSpot includes a state-of-the-art compacting generational incremental garbage collector.

JavaSoft's JavaOS [Jav96] is a native Java operating system based on a specialized Java kernel. Its target are small network devices; the byte-codes are interpreted. Parts of the Java Libraries can be leaved out if not needed, like the whole AWT package on devices with no screen. The similarity with Jaos is, that the whole JVM is build on a kernel instead of on top of another OS. This project seems to be discontinued now.

JOS [JOS] is an open source project to achieve the same goal as JavaOS: a Java Operating System running natively on the bare hardware.

IBM's Jalapeño JVM [AAB$^+$00] is targeted at SMP Server architectures; the current implementation runs on an AIX system, but the use of kernel services is limited to a minimum. The byte-code is directly compiled into native code with a simple compiler. An optimizing compiler is available for dynamic recompilation of the code. Jalapeño is written as much as possible in Java;

bootstrapping is done by creating a static image of the JVM core using another JVM. Most of the publications focus on compiler optimizations.

JBed [JBe] by Esmertec AG is a commercial JVM implementation of the Micro Edition platform for embedded systems. The core JVM implementation is written in Component Pascal.

The official Java API implementation comes from Sun, which also provides the documentation of the classes [CLK98, CL97]. The rapidly growing number of classes makes every alternate version almost impossible. Three platforms are also available (Enterprise, Standard, Micro) which differ in the number of packages and classes included. Apparently, the development seems to be driven by quantity, as some methods are often duplicated in different classes with different names. The design of the libraries is sometimes less than optimal and contains lots of legacy code. The greatest problem is the recursive import of classes, which creates a system whose hierarchy is not clear.

Classpath [Cla] is a GNU open source project to implement the Java API Classes independently from Sun and any particular architecture and system. The core classes are available, but Sun's constant creation of new API keeps this project in a eternal beta state. Classpath is in use in many JVM, in particular Japhar, Kaffe, SableVM and Intel's Open Runtime Platform (ORP).

Microsoft's .NET [Pla01] platform earns also a mention here. .NET start where the JVM stopped short, and adds interoperability and more flexibility to the platform. .NET defines its own intermediate language (MS-IL) based on a stack machine, and an explicit Common Type System (CTS) to store typing and allocation information. .NET object model is similar to Java and embodies the most common object oriented features. Main differences with the JVM are the always compiled code, and a richer object model including reference parameters and delegates.

The rapid expansion of the Java technology has created new research interests, or renovated some old ones. In particular the whole field of garbage collection is experiencing a new life, after having been mostly limited to functional language run-time systems. The fields of compiler optimization techniques tailored to the Java language has become very active to solve the inherent slowness of the language; as an example, pointer escape analysis is being deeply researched, because it allows to detect the scope of a pointer value, and thus makes possible to decide if substituting an object with a statically allocated structure is possible. Of course, it is possible to argue that extending the language and virtual machine to allow static structures would be a much simpler and efficient solution.

	HotSpot	Jalape˜no	Jaos
Target	Client / Server	Server	Client
Host System	Solaris, Linux Windows	AIX	Aos kernel
Garbage Collector	Generational, Compacting	?	Mark and Sweep
Compiler	Fast (Client), Optimizing (Server)	Optimizing	Fast, Pattern oriented

Table 7.1: *JVM Overview*

7.2 Mapping Java To Aos

This section describes the mapping of the Java Object Model to the Aos Object Model.

7.2.1 Java Object Model

In Java, classes are deployment units, compilations units, and namespaces at the same time. Classes are organized in a single inheritance hierarchy, and contain fields and methods that can be either class-bound (`static`) or instance-bound. interfaces define contracts for classes. Arrays are also classes. Packages are further namespaces kinds for grouping classes together, and can be used to limit class member's visibility.

Most of the mappings from the Java model to Aos' model are obvious. Problems arise whenever a Java primitive has a different semantic, like arrays, or doesn't exists in Aos and has to be translated to a composition of Aos' primitives, like classes and packages.

The main problem is the different use of namespaces and deployment-units. In particular, if Aos' modules correspond to Java's classes or packages. Both mappings are possible, although neither one completely preserves the Java semantic.

A class and its non-static members are always mapped to an object. Aos has no support for static members in an object, hence they must be moved inside the module that contains the class. Two mappings are possible: either a module for each class, or a module for each package.

If classes are mapped to modules, the deployment-units matches. An object is containing all non-static members is declared in the module, whereas static members are directly declared in the module. This mapping creates three problems. First, the symbol lookup differs from Oberon: the class and the module scopes are to be looked together, then the superclasses[4]. Second, this restricts the interoperability from Oberon to Java, as Java is considers every module as a class wrapper, and will thus be able to access only Oberon modules containing a single class. Third, packages have no mapping and are ignored[5], and in particular package visibility has to be mapped to public visibility.

Mapping packages to modules is even worse, making packages the deployment units and forcing to load a whole package at once every time a class is wanted: having to pay the cost of loading and just-in-time compilation of a whole package makes the booting time of the JVM extremely slow[6]. The class' static members still have to be mapped inside modules , with the additional problem that all static members of each class in a package would be declared in the same scope, thus requiring the use of a name mangling mechanism to tell entities of different classes apart. Class-wide visibility would have to be enforced by the compiler. On the other hand, Java would be able to understand Oberon modules declaring more than one object.

[4]in Oberon, the class hierarchy is traversed fi rst, then the local module

[5]in fact, they are implicitly present, as the group of classes having the same name prefi x

[6]the delay could be partly reduced by using a way-ahead-of-time compiler for the Java API, thus introducing more complexity in the design

Because the semantic loss in the first strategy is smaller than the loss in the second strategy, we thus choose to map classes to modules, aware of the limitations bound to this choice.

Java Primitive Types

Java Type	**Metadata Type**	**Description**
boolean	Bool	boolean type
byte	Int8	signed 8-bit integer
short	Int16	signed 16-bit integer
int	Int32	signed 32-bit integer
long	Int64	signed 64-bit integer
fbat	Float32	32-bit IEEE 754
double	Float64	64-bit IEEE 754
char	Char16	16-bit UNICODE

***Table 7.2**: Mapping of Java Primitive Types*

Table 7.2 shows the mapping of Java's primitive types to Oberon and Aos types. All primitive types find a direct mapping but `char`. In Java, the character type represent a 16-bit Unicode [Uni] character, while Oberon internally uses a proprietary 8-bit encoding close to ISO-8859-1 Latin. Great care must be taken when using characters defined in the other language, as the encoding may be different.

Java Classes

A Java class can contain both class-bound (`static`) and instance-bound entities. The Aos model clearly separates those entities into different declaration scopes: class-bound entities are part of a module, whereas instance-bound entities are part of a class. Because of this, for each Java class, a module and a class (in the same module) are created. Table 7.3 and Listing 7.1shows the various mappings in detail.

Java	*Oberon*
class x/y/Z	module x/y/Z object Class
fi eld	object fi eld
method	object method
static fi eld	module variable
static method	module procedure
compile-time constant	module constant
run-time constant	module variable

***Table 7.3**: Mapping a Java class to Oberon*

```
PROCEDURE ParseFields;
  VAR count, i: LONGINT; flags: SET; name: Name;
    type: Type; attr: Attributes; vis: Visibility;
BEGIN
  ReadU2(count);
  FOR i := 0 TO count-1 DO
    ReadFlags(flags); ReadName(name);
    ReadType(type); ReadAttributes(attr);
    vis := ConvertVisibility(flags);
    IF ~(Static IN flags) THEN
      c.object.CreateVar(name, vis, type, res);
    ELSIF ConstAttr IN attr THEN
      c.module.CreateValue(name, vis, GetConst(attr), res)
    ELSE
      c.module.CreateVar(name, vis, type, res)
    END;
    IF res # Ok THEN ...error... END
  END
END ParseFields;

PROCEDURE ParseMethods;
  VAR count, i: LONGINT; flags: SET: name: Name;
    sig: Signature; attr: Attributes; vis: Visibility;
BEGIN
ReadU2(count);
FOR i := 0 TO count-1 DO
  ReadFlags(flags); ReadName(name);
  ReadSignature(sig); ReadAttributes(attr);
  vis := ConvertVisibility(flags);
  IF Static IN flags THEN
    c.module.CreateProc(name, vis, sig, res)
  ELSE
    c.object.CreateProc(name, vis, sig, res)
  END;
  IF res # Ok THEN ...error... END
END;
END ParseMethods;
```

Listing 7.1: Parsing the class members

Java Interfaces

Java interfaces are mapped to a module containing a definition.

Mapping to a module is required, because interfaces can contains the code needed for the static initialization of the interface whenever static fields initialized at run-time are declared. These fields are usually arrays, which must be dynamically allocated by the class initializer `clinit`. Table 7.4 and Listing 7.1 show the mapping of interfaces; note that the parse for classes and interfaces is the same.

Java	*Oberon*
interface x/y/Z	module x/y/Z defi nition Class
method	defi nition method
compile-time constant	module constant
run-time constant	module variable

***Table 7.4**: Mapping a Java interface to Oberon*

Java Arrays

Java arrays are subclasses of `java.lang.Object`. Arrays contains a number of variables, referenced by integer index values. All variables have the same type. The number of variables in an array can be changed at any time.

Java	*Oberon*
array of T	module ”[T” object ”Class” fi eld: pointer to array of T

***Table 7.5**: Mapping a Java array to Oberon*

No code is generated for an array (in fact arrays are always implicitly declared inside a class); the module is simply a wrapper used to access the mapped type from Oberon.

The array is mapped to an object type extending `java.lang.Object` and containing a single field with dynamic array as type. Using a class as a wrapper for the dynamic array requires a double indirection for every array element access, but is the only way of implementing the class semantic for arrays, because Aos' arrays carry no method table.

This cost could be reduced by the use of escape analysis in the compiler, which would allow to use less expensive array types whenever possible. Another possible optimization is to handle the methods define in `Object` as system calls, but this requires them to be final, which is not the case[7].

[7] many classes redefi ne the methods `equals`, `clone`, and `hash`

Java Packages

Java packages have no mapping in Aos and are ignored. The package-wide visibility is mapped to Aos' public visibility. The enforcement of such visibility is nevertheless performed by the compiler translating the Java code to byte-code; security problems are still possible whenever a class is changed and recompiled without recompiling its clients.

Java Visibility Modes

Table 7.6 shows the mapping of Java visibility flags for the common metadata repository.

Java	*maps to*
private	internal
package (default)	public
public	public
protected	protected

***Table 7.6**: Java visibility flags mapping*

The package visibility is approximated by the public visibility, because there is no structure equivalent to a package in the Aos object model.

7.2.2 Java Concurrency Model

Java concurrency consists of thread control, thread synchronization, and data protection. The implementation for thread control is provided by the class `java.lang.Threads`; class `java.lang.Object` provides signals and the the wait, notify, notifyAll method to synchronize threads; instance based monitors are used for protecting data, and are implemented by the two byte-code operations `monitorenter` and `monitorexit`.

The Java design is somewhat inconsistent, as the support for concurrency is done in two classes and the byte-code. In fact, having the locking primitives in the language itself (the synchronized construct in the Java language and the two aforementioned byte-codes) makes no sense, as long as the language doesn't include concurrency too. Apparently, the design of the JVM is strongly determined by the underlying implementation, instead of being driven by a clean design.

Protection

Protection against concurrent execution is implemented by the `monitorenter` and `monitorexit` byte-code instructions. Java locks are reentrant

The Aos kernel provides non-reentrant locks, but it is possible to construct reentrant locks on top of non-reentrant locks. The locking information has to be extended with a counter to keep track of the number of times the lock has been takes; Aos' locks are used to protect and synchronize access to this structure.

Table 7.7 shows the details of the mapping. Appendix A.1.3 gives a full implementation of reentrant locks.

Java Code	Aos Implementation

`monitorenter`	`IF  LockedByMe(SELF) THEN TakeLock(SELF) END;`
	`INC(count);`

`monitorexit`	`DEC(count);`
	`IF count = 0 THEN ReleaseLock(SELF) END;`

Table 7.7: *Mapping of Java protection to Aos*

The different semantic—and thus implementation—of the locking mechanism, makes automated interoperability of code impossible. Developers wishing to protect objects defined in the other language must explicitly lock the object by calling the locking primitives in their code, by calling `jjlObject.Lock` and `jjlObject.Unlock`.

Threads

The class `java/lang/Threads` defines the operations on threads. The most relevant ones are `Threads.start()`, `Threads.stop()`, `Threads.suspend()`, `Threads.resume()`, and `Threads.yield()`. The semantic of those methods should be obvious.

Corresponding to the active object model, the Aos kernel provides primitives to start and yield a thread. No support is given to stop or kill a thread, as this would contradict the active object model; those operations are avoided, because they cannot be implemented to preserve the consistency of the program: the locks and invariants of the objects used by the thread to be stopped would be in an undefined state. Sun recognized this problem too, and declared those methods as deprecated.

Jaos implements only `Threads.start` and `Threads.yield`; programs using the other methods to stop a thread are not supported. Table 7.8 shows the details of the mapping. As every object allocated in the system is potentially an active object, no special handling of the object before the start system call is needed.

Java Code	Aos Implementation
`Threads.start()`	`SystemCall(AosActive.Start)`
`Threads.yield()`	`AosActive.Yield`

Table 7.8: *Mapping of Java thread operations to Aos*

Synchronization

Every Java object is associated with a signal. Synchronization of threads is achieved by waiting (`Object.wait()`) for a signal to be activated (`Object.notify()` and `Object.notifyAll()`). The waiting thread can also specify a delay, after which the thread should resume execution even if no notification happened.

The Java synchronization primitives are implemented in Aos using a ticket algorithm [And91]. Preempted thread are assigned a unique number (the ticket); activation is done by rescheduling the thread holding the smallest ticket.

Listing 7.2 shows the implementation[8] of the ticket algorithm.

```
TYPE
  Ticket = OBJECT
    VAR current, next: LONGINT;

    PROCEDURE Wait;
      VAR ticket: LONGINT;
    BEGIN {EXCLUSIVE}
      ticket := next;
      INC(next);
      AWAIT(ticket = current)
    END Wait;

    PROCEDURE Notify;
    BEGIN {EXCLUSIVE}
      IF current # next THEN
        INC(current)
      END
    END Notify;
  END Ticket;
```

Listing 7.2: The ticket algorithm

The ticket algorithm is first extended to allow the notification of all waiting threads required by the `notifyAll` method. The equality check is thus changed into a range check[9]. Listing 7.3 gives the algorithm.

```
TYPE
  Ticket = OBJECT
    VAR current, next: LONGINT;

    PROCEDURE Wait;
      VAR ticket: LONGINT;
    BEGIN {EXCLUSIVE}
      ticket := next;
      INC(next);
      AWAIT(ticket-current < 0)
    END Wait;

    PROCEDURE Notify;
    BEGIN {EXCLUSIVE}
      IF current # next THEN
        INC(current)
      END
    END Notify;

    PROCEDURE NotifyAll;
    BEGIN {EXCLUSIVE}
      current := next
    END NotifyAll;
```

Listing 7.3: The ticket algorithm modified for notifyAll

Second, the `wait` instruction has define a time-out delay, after which the thread must be rescheduled even if no `notify` or `notifyAll` did awake it. Besides the implementation of the deadlock, the ticket of the awaken thread must be invalidated, to avoid calling a thread which is already awake. This is implemented with a list of ”returned”tickets: a thread awaken by a time-out inserts its ticket in the list; notify must checks that the ticket to be called is not in the list (in which case it removes it from the list and proceeds to the next ticket). Listing 7.4 gives the complete algorithm. Although timed waits make the algorithmmuch more complex, they are quite seldom; the implementation is thus optimized for the common case.

[8]The careful reader has probably noted, that tickets are allocated without checking for the overflow condition `current = next`, which can only happen if 2^{32} clients are waiting at the same time for the same signal. Allocating that many processes would require more memory than a actual computer can possibly have.

[9]We use the subtraction here to cope with the case where the counters overflow and wrap around

```
TYPE
  Object = OBJECT
    VAR current, next: LONGINT;

    PROCEDURE Notify;
    BEGIN {EXCLUSIVE}
      IF current # next THEN
        INC(current);
        WHILE InList(current) DO
          Remove(current)
        END
      END
    END Notify;

    PROCEDURE NotifyAll;
    BEGIN {EXCLUSIVE}
      current := next
    END NotifyAll;

    PROCEDURE Wait;
      VAR ticket: LONGINT;
    BEGIN {EXCLUSIVE}
      ticket := next;
      INC(next);
      AWAIT(ticket-current < 0)
    END Wait;

    PROCEDURE Wait(delay: LONGINT);
      VAR ticket: LONGINT;
        t: Timer;
    BEGIN {EXCLUSIVE}
      ticket := next;
      INC(next);
      NEW(t, delay);
      AWAIT(t.ready OR
            (ticket-current < 0));
      IF t.timeout THEN
        AddToList(ticket)
      END
    END Wait;
  END Object;
```

Listing 7.4: Java Synchronization Implementation

Class Kind	**Java Signature**	**Module Name**	**Class Name**
primitive	*P*	*P*	*P*
	int	int	int
class	*package*/*Class*	*package*/*Class*	*Class*
	Java/lang/Object	Java/lang/Object	Object
array	[*C*	[*C*	Class
	[D	[D	Class
class array	[L*package*/*Class*;	[L*package*/*Class*;	Class
	[LJava/lang/String;	[LJava/lang/String;	Class

***Table 7.9**: Naming Conventions*

7.3 JVM Implementation

7.3.1 Introduction

Java classes go through many life stages, depending their usage. Table 7.10 lists those stages. The loaded stage is only transitory and is used as an intermediate step to detect and break recursion in classes.

Stage	**Description**
Loaded	Class fi le internalized
Allocated	Field and Methods addresses available
Compiled	Byte-code compiled
Linked	External references resolved
Initialized	Static initialized executed

***Table 7.10**: Life stages of a Java class*

The decision when to migrate a class to an higher stage is relevant. Early migration makes more information available to the system, but tends to cause a domino effect because every stage-change in a class can cause many more in other classes; this has an high cost in time and memory. A late migration reduces the loading time and memory usage, but requires to keep track of the missing information and to patch it at a later time whenever it is needed and becomes available.

To support Jaos' JIT-compiler, migration is forced whenever the compilation requires information about the class. This strategy can be described as early loading; otherwise most migrations are delayed to the latest moment. This strategy has the advantage of simplifying compilation and reducing the information to be fixed to a minimum; it also allows to perform constant inlining during compilation.

	Allocate(a)	Compile(a)	Initialize(a)
precondition	Allocate(Super(a))	Compile(Super(a))	Initialize(Super(a))
		Allocate(Interfaces(a))	
		Allocate(UsedClasses(a))	

***Table 7.11**: Static Dependencies for class stage changes*

Table 7.11 shows the static dependencies between classes. Migrations can also be dynamically triggered by a program's execution which requires a class to be initialized. The static class initializer (method `clinit`) must be called as described in [LY99, Sec. 2.16.4] the first time a class is instantiated, a static field is accessed[10], or a static method is called. To detect such events, Jaos uses the technique explained in [CLS00]: every `GetField`, `SetField`, and `InvokeStatic` instruction is preceded by a check if the referenced class is already initialized; JIT-compilation allows a small optimization: if the referenced class is already initialized, no check in added to the generated code.

[10]there are a few exception for fi nal static fi elds

For bootstrapping reasons, a few classes are loaded and initialized by the JVM at startup; these classes contains features that are needed by the JVM itself. These classes are Object, String, Throwable, Thread, and ThreadGroup. The initialization of those classes indirectly causes many more classes to be loaded and initialized; so that at startup eventually about 50 classes are initialized and further 90 loaded.

The rest of this chapter describes the implementation of Jaos, divided in the single stages: Loading and Allocation (Section 7.3.2), Compilation (Section 7.3.3), Linkage (Section 7.3.4) and Initialization (Section 7.3.5); Furthermore, Section 7.3.6 describes the system integration of Jaos into Aos, and Section refBenchmarks reports some benchmarks.

7.3.2 Loading and Allocation

The Jaos loader internalizes a Java class-file. The class-file is parsed and the information contained are inserted in the common metadata repository. Some information like the constant pool is also kept by the JVM, because it is later used by the byte-code to access constants and external references.

Allocation of the loaded class is automatically performed by the metadata repository when the class is committed; this information is further used by the JIT-compiler.

The loader also detects whenever a native implementation is available for the loaded class, in which case it loads the module, checks that the native implementation contains the same members and with the same type or signature, and controls that the allocation corresponds.

A class is loaded whenever its information is needed. Usually the class itself is to be executed, a subclass is being loaded, or it is referenced by the class being compiled.

7.3.3 Compiling

The Java byte-code to Intel IA32 code compiler is a pattern-oriented just-in-time compiler; every byte-code is translated to a fix IA32 pattern; this allows to perform the compilation in one pass. The only optimization performed is constant-inlining. The generated code reflects the stack-based byte-code of the JVM by keeping temporary values on the stack.

One core task of the compiler is to map the Java object model to the Aos object model according to the Aos calling convention and the memory layout. This affects in particular the byte-code operations accessing the memory.

The Aos *calling convention* differs from the Java one only in one aspect: for virtual method calls ,Java passes the object self reference as first (hidden) parameter, whereas Aos expects it to be the last parameter. Because no data-flow analysis is performed, this is detected when all parameters are already pushed on the stack. The obvious workaround is to push the first parameter again on the stack.

Parameters, local variables, and the operand stack are allocated directly on the thread stack. The class file contains the information about the number of parameters and local variables; in the byte-code there is no such distinction, both parameters and local variables belong to the local space of the method, and are accessed using the `load_<n>` and `store_<n>` instructions. The parameters are mapped from left to right beginning with index 0, followed by the local variables. Whenever present, the object self reference is allocated as last parameter. Figure 7.1 shows the procedure activation frame for each method and the mapping of the method locals in the the activation frame.

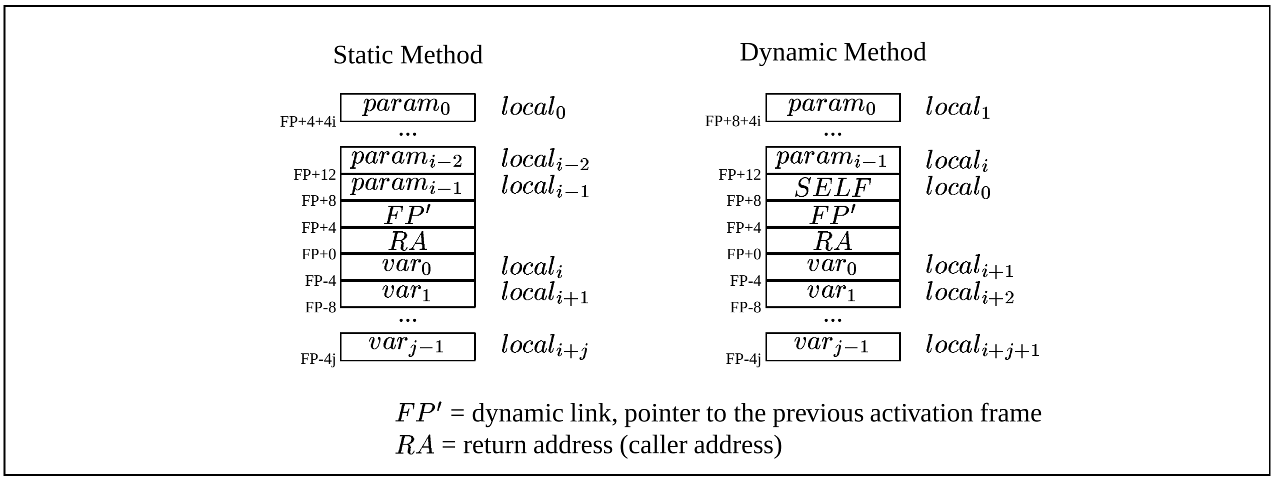


Figure 7.1: Procedure Activation Frame Layout for Java Methods

Because all operation's results are kept on the stack, no register allocation is needed.

Whenever an address is not available at compile time, a fixup list is created for later resolution. This can happen with static methods, whose address is known only when the referenced class has been compiled. The fixup list is embedded in the code, and anchored at the symbol to be resolved. Fixup lists are resulted by the linker when the addresses become available.

Some byte-code instructions perform complex operations that require information or interaction with the kernel. These instructions are implemented *system-calls* to the JVM, which will then perform the requested operation. The instructions implemented as system calls are `monitor-enter`, `monitorexit`, `new`, `newarray`, `anewarray`, `multinewarray`, `checkcast`, `instanceof`, `throw`, and `invokeinterface`.

The execution of some byte-code instructions must perform run-time checks, to ensure that no null pointer is dereferenced and array access are within their own bounds; test failures must result in an exception. `NullPointerExceptions` are handled as in Oberon and do not require any code: the first memory page is not mapped. Every attempt to dereference an address within the first memory page causes a page fault, which is caught by the exception handler and mapped to a `NullPointerException`.

Memory accesses are performed using the `load_<n>`, `store_<n>`, `*aload`, `*astore`, `getfield`, `putfield`, `getstatic`, and `putstatic` instructions. Table 7.12 shows the code patterns used when translating these instructions. In the table, `reg` refers to one of the registers `AL`, `AX`, `EAX`, `EDX:EAX`, or `ST(0)`, depending on the type of the operand.

The instructions `invokespecial`, `invokestatic`, `invokevirtual`, and `invoke-interface` transfer the execution to a subroutine. To comply with the Aos calling convention, before each virtual and interface invocation the object reference has to be copied to the top of the

	from memory	to memory
locals	`; load_<n>`	`; store_<n>`
	`push off[EBP]`	`pop reg`
		`mov off[EBP], reg`
statics fields	`; getstatic`	`; putstatic`
	`push @addr`	`pop reg`
		`mov @addr, reg`
instance fields	`; getfield`	`; putfield`
	`pop ebx`	`pop reg`
	`mov reg, off[ebx]`	`pop ebx`
	`push reg`	`mov off[ebx], reg`
array elements	`; *aload`	`; *astore`
	`pop ecx`	`pop reg`
	`pop ebx`	`pop ecx`
	`mov ebx, 12[ebx]`	`pop ebx`
	`IndexCheck`	`mov ebx, 12[ebx]`
	`mov reg, 16[ebx][ecx*scale]`	`IndexCheck`
	`push reg`	`mov 16[ebx][ecx*scale], reg`

Table 7.12: *Memory access patterns*

stack, making it the last parameter passed; after the call the original value on the stack has to be removed; the function result is available in a register (usually `EAX`, `EDX:EAX` for 64-bit values, and `ST(0)` for floating point values) and has to be pushed on the stack after the call, where Java expects it. Table 7.13 gives the code patterns used for the routine invocation.

On a 400 MHz Pentium-II, the JIT-compiler can process up to 690'000 byte-code instructions[11] per second for huge procedures (like those in the MpegAudio Spec JVM suite), with an average of 233'000 instructions per second. For every byte-code instruction is generates an average of 7.5 bytes of IA32 code. The compiler is optimized for a client JVM: compilation is very fast to allow for a quick startup, but the quality of the generated code is not high, it is nevertheless faster than interpreted code.

7.3.4 Linking

The linker resolves the external references that were not known at compilation time, method and interface tables are completed, and the module run-time structure is completed.

The first task of the linker is to resolve external references from and to a newly compiled Java Class. The compiler inlines all known addresses in the code, but because of the recursive class imports, some addresses are not available at compilation time; in this case fixup lists are created for the linker. Whenever the class is compiled, these addresses become available, and the linker has to resolve the fixup link and patch the address in the code using them.

Native implementations of a method override the Java implementation; the linker must ensure that the right method native method is called. Implementation overridingis very useful for boot-

[11]please note that we use byte-code instructions, not byte-code size, which is bigger

invokestatic	`syscall CheckInitialized`	
	`call routine`	
	`push result`	
invokevirtual	`mov eax, (nofPars * 4) [esp]`	`; load SELF`
	`push eax`	`; push SELF`
	`mov eax, -4[eax]`	`; get type descriptor`
	`mov eax, methodOffset[eax]`	`; get method address`
	`call eax`	`; invoke method`
	`pop ebx`	`; remove SELF`
	`push result`	
invokeinterface	`mov eax, (nofPars * 4) [esp]`	`; load SELF`
	`push eax`	`; push SELF`
	`push eax, -4[eax]`	`; type descriptor`
	`push interface`	`; interface descriptor`
	`syscall interfacelookup`	`; eax = vtable addr`
	`mov eax, mthOffset[eax]`	`; get method address`
	`call eax`	
	`pop ebx`	`; remove SELF`
	`push result`	

Table 7.13: *Routine invocation patterns*

strapping the JVM and thus provide code before the java code is available, or to disable some code which executes classes that are not available.

Native code does not use the JNI API, because the interoperability is ensured at source-code level; the generated code complies to the constraints imposed by Java (garbage collection and type safety).

The linker completes the module structure for each class. In particular, the export tables are completed to allow the current class to be linked to Oberon code by the Oberon linker. For each exported symbol, the export table has an tuple with the fingerprint representing the symbol and the address of the symbol. The module loader uses this information to resolve the external links between modules; Jaos does not use it, as the addresses are usually inlined by the compiler.

The last task of the linker is to complete the method tables with the entry points of the methods, and register the interfaces implemented directly or inherited by the class.

7.3.5 Initializing

The last stage in a class' life is the static initialization. The code to perform it is supplied by the class in method `clinit`; it initializalizes the final static fields that are not compile-time constants—like used-defined arrays—and does the user-defined static class initialization.

Java defines a lazy class initialization: initialization is triggered by the first access to a static field, a static method, or the first allocation of a class instance. This are all dynamic conditions, that must be checked at run-time, by letting the compiler insert checks before each static call or static field access.

The check is a system call to the JVM routine that initializes a class; after each call the branch is removed from the code to avoid useless calls. If the referenced class is already initialized at compilation-time, no run-time check is inserted in the code.

7.3.6 System Integration

Jaos has to register itself to the kernel and metadata repository to allow for automated loading of deployment-units and their metadata. For this, Jaos provides a module loader plugin and a metadata loader plugin.

The *module loader plugin* is called by the system whenever a new deployment unit shall be loaded. It locates and loads a Java class on request. This happens explicitly when a command of the class is invoked, or implicitly when the class is required (”imported”) by another deployment unit. This mechanisms makes Java classes equivalent to the Oberon Modules, as the are both dynamically loaded on-demand by the system; it makes no difference to the system, if code comes from Java or Oberon: in fact, this anonymity enables interoperability between both languages.

The *metadata loader plugin* fills on-demand the common metadata repository with metadata on a class. The repository requires the plugin to supply the information for the deployment-unit with a given name. This allows to import Java classes into an Oberon module, by supplying the metadata information required by Oberon to perform the separate module compilation.

7.3.7 Benchmarks

	Time	*Ratio*	*Time*	*Ratio*
201_compress	81.72 s	14.38	53.45 s	21.98
202_jess	113.39 s	3.35	85.52 s	4.44
209_db	207.88 s	2.43	164.84 s	3.06
228_jack	208.35 s	2.18	158.75 s	2.86
Machine	Dell Latitude 600		Dell Optiplex GX 200	
CPU	Pentium-III 750 MHz		Pentium-III 1 GHz	
RAM	256 MB		256 MB	

Table 7.14: *Benchmark Results*

Table 7.14 shows the results obtained with Jaos running the Spec JVM98 benchmark suite [SPE98]. Each measurement is the average of three test runs on a given machine. The test environment consisted of Aos, Jaos, and Classpath 0.03. Only successful tests are listed in the table.

Compared with other JVMs, the results may look pale. The reasons are quite simple: the pattern expanding compiler simply does not optimization at all. An optimizing compiler using the same technology as Intel’s ORP compiler [ATCL+98] is in preparation, but the result are not available at this time.

7.4 Interoperability with Oberon

7.4.1 Native Methods (stubs)

The Java language allows to declare a method as *native*; the implementation of such a method is provided in another language (in our case Oberon). Native methods are used to implement low-level features, or to interface the Java implementation with a component provided in another language. Native methods are often used in the standard class API to access the underlying kernel.

Native methods require a tighter integration than classes, because they belong to a class scope and shall be granted exactly the same privileges of the Java method for field access and method invocation. In some cases, a class may have to be extended to store additional information used by the native method. Last but not least, such methods are already used at bootstrap, when the JVM is not available and the normal interoperability features cannot be used.

Native methods are implemented in Oberon, by providing a skeleton of the class with the declaration of all methods and fields; the programmer can provide an implementation to any method, which overwrites Java's implementation (if any). This semantic addresses two problems at once: native methods and bootstrapping. During the bootstrap of the JVM, it allows to easily redefine or ”switch off”some methods, because they would perform functions that are not available at the that time.

Jaos's loader and linker detect if a class has a native implementation and merge the native code with the Java code.

Implementation

Three modules provide support for native implementations.

A *stub generator* translates the Java class definitions into an Oberon module and class declaration, by mapping the Java structures to Active Oberon code. Native methods are clearly annotated as such. The stub generator is basically a module interface browser with a few added functionalities for name mangling and code annotation.

The class-file *loader* checks if a native implementation is available, in which case the module containing the native implementation is loaded and checked for consistency[12]; the allocation data is taken from the native implementation. If no implementation is provided for a native method, a warning is issued.

The class-file *linker* patches the native methods entry points into the method table. For static methods, a jump to the native implementation is inserted in the Java code and the entry in the metadata repository is overwritten.

java/lang/Object	getClass, clone, notify, notifyAll, wait (all three variants)
java/lang/String	intern, clinit
java/lang/reflect/Field	get, getBoolean, getByte, getChar, getDouble, getFloat, getInt, getLong, getShort, getType, set, setBoolean, setByte, setChar, setDouble, setFloat, setInt, setLong, setShort
java/lang/reflect/Method	getModifi ers, getExceptionTypes, getParameterTypes, getReturnType, invokeNative
java/lang/reflect/Constructor	constructNative, getExceptionTypes, getModifi ers
java/lang/Class	newInstance, isInstance, isAssignableFrom, isInterface, isPrimitive, getName, getClassLoader, getSuperclass, getInterfaces, getModifi ers, getDeclaringClass, getClasses, getFields, getMethods, getConstructors, getField, getMethod, getConstructor, getDeclaredClasses, getDeclaredFields, getDeclaredMethods, getDeclaredConstructors, getDeclaredField, getDeclaredMethod, getDeclaredConstructor, getResource, forName
java/lang/Throwable	fi llInStackTrace, printStackTrace, writeObject, readObject, clinit
java/lang/Thread	countStackFrames, isAlive, isInterrupted, join, nativeDestroy, nativeInit, nativeInterrupt, nativeResume, nativeSetPriority, nativeStop, nativeSuspend, start, currentThread, sleep, yield
java/io/FileDescriptor	init, syncInternal, validInternal
java/io/File	canReadInternal, canWriteInternal, deleteInternal, existsInternal, isDirectoryInternal, isFileInternal, lastModifi edInternal, lengthInternal, listInternal, mkdirInternal, renameToInternal, setLastModifi edInternal, createInternal
java/io/InputStream	
java/io/OutputStream	
java/io/FileOutputStream	closeInternal, open, writeInternal
java/io/FileInputStream	closeInternal, getFileLength, open, read, readInternal, skip, skipInternal
java/lang/System	setIn, setOut, setErr, currentTimeMillis, arraycopy, identityHashCode, isWordsBigEndian
java/lang/Runtime	init, execInternal, exitInternal, gc, loadLibrary, nativeGetLibname, nativeLoad, runFinalization, totalMemory, freeMemory, traceInstructions, traceMethodCalls, getLibraryPath, runFinalizersOnExitInternal
java/lang/VMClassLoader	defi neClass, getPrimitiveClass, resolveClass
java/lang/Number	
java/lang.Float	floatToIntBits, floatToRawIntBits, intBitsToFloat, parseFloat, toString
java/lang/Double	doubleToLongBits, doubleToRawLongBits, initIDs, longBitsToDouble, parseDouble, toString
java/lang/Math	IEEEremainder, acos, asin, atan, atan2, ceil, cos, exp, floor, log, pow, rint, sin, sqrt, tan

***Table 7.15**: Implemented Native Methods*

Native Implementations

Table 7.15 resumes the methods with a native implementation. Some methods are not part of the standard Java API, but where defined in the Classpath project to ease the implementation by dividing between Java and native parts. Some classes had to be defined as stubs although no methods are implemented, because they are further imported by other native stubs.

7.4.2 Incompatibilities

This section presents the incompatibilities between Java and Oberon from the Java point of view; it completes the remarkspresented in Section 5.6.

The JVM hides most of the incompatibilities, as it presents a closed universe. Nevertheless, relying on Aos to implement the JVM causes some problems. The first part relates the problems encountered implementing the JVM, whereas the second part shows the language interoperability limitations.

Implementation Problems

Java has some features and conventions that differ from Aos. To be able to implement the JVM using the common metadata repository, the repository had to be extended to support such features. Java *naming conventions* differ from Oberon in two ways: first, Java allows names to be up to 256 characters; second, Java uses the Unicode character set. To cope with this, the maximal identifier length has been extended to allow names of any length by using a string pool in the repository. Identifiers are stored using UTF-8 encoding, which allows to have ASCII and UNICODE identifiers in the same table at the same time.

A second extension of the repository was made to support *method overloading*, which is present in Java.

Java *final fields* are also a cause of problems: the simplistic assumption that a final static field is equivalent to a constant is wrong. A final field is a field that can be assigned only once, independently of the time the assignment is done. It is possible to distinguish between compile-time constants, whose value is know at compilation-time, and other final static fields, whose value is computed during the initialization of the class. Such fields declared in a interface are a particularly nasty combination, as they force a completely abstract interface to have static initialization method, which is definitively counter-intuitive.

The *floating point* rounding mode is also source of some problems, as Oberon and Java use different rounding modes for their operations. In Aos, a different rounding mode can be set for each thread; threads started by Java use the Java rounding mode, and Oberon one's the other mode. Although fairly reasonable, this choice can cause some problems in code that is shared and used between by the two systems. An obvious example is the output to the Java Console, which displays

[12]all fi elds and methods declared in the Java class must be present, type and signatures must be the same

text aligned in a different way[13]

The Aos kernel provides non-reentrant object locks, but Java allows reentrancy[14]. For this reason, the JVM had to implement an own locking mechanism.

Interoperability Problems

The Oberon code written to be accessed by Java is allowed to use only a subset of the Aos object model, which is understood by Java. Oberon types follow a slightly lower-level approach than Java: types are defined in many different variants (e.g. arrays can be stack or heap allocated, have a compile-time size or not), whereas Java provides only one generic type (which is correspondingly expensive to implement). Records cannot be used, because Java does not support statically allocated structures.

The second main problem is subtler: the discrepancy in the way compilation-units, deployment-units, and namespaces are grouped and handled. The current mapping of a class to a module containing an object type is a huge limitation, as only modules declaring a single object type can be understood automatically. A workaround is possible, but it requires registering each class singularly in a list of special name mappings in the JVM.

[13]the code to implement the scrolling functionalities of the console runs under Oberon

[14]a good overview of the reasons for and against lock reentrancy is presented in [Szy98]

8 Conclusions

8.1 Achievements

8.1.1 Language Interoperability

The requirements and limitations of language interoperability among software components defined using object-oriented language, which interoperate through an interface built using a predefined type system have been investigated. In particular, a set of algorithms for mapping the various flavors of the object-oriented models (limited to the inheritance of classes and interfaces) has been proposed, and a few common languages (Active Oberon, C# , Eiffel, Java, Oberon, Oberon.NET) have been classified following a fine-grained criterion.

8.1.2 Active Oberon Language

This thesis explains the rationale, design, and considerations behind the Active Oberon Language. Many examples show the use of the concurrency features in the language and the transformation of the AWAIT synchronization primitive to and from a signal model.

8.1.3 Active Oberon Compiler

The Active Oberon Compiler provides a non-trivial example of using Active Oberon. The compiler has a scope-parallel parser, which concurrently parses and emits the scopes in an Active Oberon module. The scope-parallel parser also allows handling forward references in the source code as a synchronization problem among parsers instead of requiring a second pass on the code. The Intel back-end of the compiler contains a peephole optimizer that reconstructs the complex addressing modes for the CPU from an SSA-like low-level intermediate representation.

8.1.4 Jaos JVM

The Jaos JVM provides a Java run-time environment that is tightly integrated in the Aos / Bluebottle system. The Aos kernel was presented as an interoperability platform, by using the same conventions like the data-layout, and the same symbol-table as Oberon. This allows Java and Oberon programs to easily and transparently interoperate. The realization of interoperability uncovered all the incompatibility between the two languages, and their idiosyncrasies.

8.2 Future Directions

The work during dissertation did grasp many fields of computer science, and being time limited[1], many paths have been only partially walked. System construction is such a wide and interwinded field, that a huge amount of details must be learned before being able to move a single bit.

This section points out how this work could be continued and (obviously) improved. We also propose some implementation projects with would complete the current body of software, and possibly evolve into serious research projects.

8.2.1 Research Topics

Programming Models and Interoperability

This thesis gives a simple fine-grained classification of programming models. The classification of the various object oriented models flavors is still partial, and more work should be invested in it to create a better and more complete classification. The goal should be a better understanding of these languages, and a set of clear rules that allow defining what a language can and cannot do[2]. The classification should consider the less common features in object orientation, like parametric type polymorphism, method covariance, and subtyping vs. subclassing. This effort should obviously be extended to the non-object oriented models, in particular imperative, functional, and logic languages.

In parallel with this classification, the mapping among the different flavors of the programming models should also be extended. The final goal should be:

- understanding of which models are equivalent
- understanding of which features are primitives
- simplification of compiler development by providing the mapping between the various models

With the renewed interest in virtual platforms, in particular those providing a high-level programming model like the Java VM and .NET, compilation is turning into model mapping. We analyzed how to map a few flavors of the object-oriented model into each other, the work should be continued with different models.

One important result of the mapping would be to help the designers of virtual machines to better understand which primitives they should provide, and which ones are just redundant.

[1] although this work took quite a long time
[2] let's call this a checklist

Multi-level Machines

One interesting characteristic of Jaos is, that it is a system with two programming models: the Active Object model and use the Java model. We used the Active Object model to let Oberon and Java interoperate, but it is also possible to use Java's model to interoperate. This work shows, that it is possible to define an abstract machine having multiple models with increasing abstraction. Because each model builds upon the lower one, part of the information known at one level would be available at lower and higher levels too.

Instead of creating new virtual machines and systems for each model, it should be investigated how to provide multiple programming models on the same machine.

Parallelism

The current programming models provide a very low-level abstraction of concurrency. This has to change. Reasoning about locks and threads is perhaps very efficient, but the lack of abstraction makes concurrency a difficult matter. An important problem is the bad integration of concurrency in the programming languages; the same languages have otherwise reached a much higher abstraction-level in their models. With the active object model, we tried to shift the perspective from the synchronization of execution threads to the synchronization of object states. We think that this model is appropriated and well integrated in the object-oriented technology.

Obviously more work can be done in this field.

First, more experience should be collected on the Active Object model to validate it.

Second, concurrency in programming languages should be make simpler to understand. Currently, programs are very good at showing static structures, but not at showing the execution paths of a program; concurrency is even harder to read in a program. The question is whether human beings have a chance to perceive and reason about concurrency at all; if yes, how should programming languages be modified to make concurrency easier to model.

Third, as languages evolve, so should the concurrency model behind them evolve too. What is appropriate for an object-oriented language will probably be not appropriate for another programming model.

8.2.2 Drudgery

Every software system is like a living piece that evolves over the time. New trends and technological advances happen all the time, and make existing systems eventually obsolete. This is the case for Paco and Jaos too.

Paco could be improved in many ways:

- Adding Exception Handling capabilities to the language and compiler
- Adding more optimizations on the intermediate representation
- Using SafeTSA [ADvRF01] as intermediate representation

- Writing a back-end for Microsoft's IL

Jaos has also many unfinished parts:

- Bind Java's networking and GUI into Jaos
- Contribute to Classpath by writing some missing Java classes

A more challenging project is the implementation of the ECMA 335 standard on top of Bluebottle. Supporting the Microsoft's CLR would allow all the languages that run on .NET to run on Bluebottle too. This would provide a lean and small platform for the CLR (in particular if compared to the mammoth systems Windows and Linux).

8.3 Concluding Remarks

Just when everybody though that the last word about programming languages and platforms had been said, and interest (and money) started drifting towards other fancier fields of computer science, this field experienced a huge revolution, and created renewed interest.

The availability of a platform where language interoperability is one of the principal goals is also a huge shift in mentality, away from the almost religious belief in the true and only programming language. Every programming language has a clear scope and reason to exist, and it is important to be able to choose the most appropriate one. In fact, the modern platforms supporting language interoperability achieve two goals at the same time: providing interoperability and providing a higher-level computing platform.

Multilanguage platforms have a clear advantage over other platforms. They are able to attract a wider community, and thus develop themselves much faster. With language interoperability, the various communities sharing one platform can even take better advantage of each other's work. The requirement for such a platform is quite small: making metadata available, and providing a model that is general enough to support all the targeted languages.

High-level computing platforms, usually called virtual machines, have proved to be very useful. They are fast to develop and easy to deploy. Combined with a high-level programming model, they bear advantages that correspond to today's needs: safety, portability, and comfort. The managed environments are able to ensure a higher degree of safety in a run-time; when considering that the buffer overflows are the most common breach used to break into a system, migrating to a managed environment would automatically prevent all those attacks. The use of an intermediate representation makes the platform easier to port to different hardware environments; this is a very welcome property for a software system, because we're on the verge of a major change in the chip technology (the change from 32-bit to 64-bit processors). Comfort is always a welcome feature, and many characteristics of newer programming models like automated storage reclamation and object orientation concur in providing it.

Better concurrency models for the current programming languages are now needed, because systems are becoming increasingly distributed and the use of concurrency cannot be avoided. The use of an Active Object model in the language Active Oberon is a step in this direction, but the way to better models is still long. It wouldn't be surprising that the language research will attack this problem with increased vigor.

In conclusion, the past decade has produced many changes in the programming language and run-time environment panorama, and awakened a new the interest in this field of computer science; most of these fields are still in their infancy, and this promises many surprises to come in the next few years.

A
Active Oberon Examples

A.1 Synchronization Examples

A.1.1 Readers and Writers

```
MODULE ReaderWriter;

TYPE
    RW = OBJECT
       (* n = 0, empty *)
       (* n < 0, n Writers  *)
       (* n > 0, n Readers *)
       VAR n: LONGINT;

      PROCEDURE EnterReader*;
      BEGIN {EXCLUSIVE}
          AWAIT(n >= 0); INC(n)
      END EnterReader;

      PROCEDURE ExitReader*;
      BEGIN {EXCLUSIVE}
          DEC(n)
      END ExitReader;

      PROCEDURE EnterWriter*;
      BEGIN {EXCLUSIVE}
          AWAIT(n = 0); DEC(n)
      END EnterWriter;

      PROCEDURE ExitWriter*;
      BEGIN {EXCLUSIVE}
          INC(n)
      END ExitWriter;

      PROCEDURE & Init;
      BEGIN n := 0
      END Init;
    END RW;

END ReaderWriter.
```

The Readers - Writers paradigm regulates the data access in a critical section. Either a single Writer (activity changing the state of the object) or many Readers (activities that don't change the state of the object) are allowed to enter the critical section at a given time.

A.1.2 Signals

```
TYPE
  Signal* = OBJECT
    VAR
        in: LONGINT;    (*next ticket to assign*)
        out: LONGINT;    (*next ticket to service*)
        (* entries with (out <= ticket < in) must wait *)

    PROCEDURE Wait*;
    VAR ticket: LONGINT;
    BEGIN {EXCLUSIVE}
        ticket := in; INC(in); AWAIT(ticket - out < 0)
    END Wait;

    PROCEDURE Notify*;
    BEGIN {EXCLUSIVE}
        IF out # in THEN INC(out) END
    END Notify;

    PROCEDURE NotifyAll*;
    BEGIN {EXCLUSIVE}
        out := in
    END NotifyAll;

    PROCEDURE & Init;
    BEGIN  in := 0; out := 0
    END Init;
  END Signal;
```

`Signal` implements signaling primitives in Active Oberon, similar to those of Java and Modula-2. It uses a slightly modified ticket-algorithm. Like in some stores, every customer receives a numbered ticket, to ensure that the customers are serviced in order of arrival. This algorithm handles the wrap-around of `in` and `out` indexes too.

A.1.3 Re-entrant Locks

```
ReentrantLock* = OBJECT
  VAR
    lockedBy: PTR;
    depth: LONGINT;

  PROCEDURE Lock*;
  VAR  me: PTR;
  BEGIN {EXCLUSIVE}
    me := AosActive.CurrentThread();
    AWAIT((lockedBy = NIL) OR (lockedBy = me));
    lockedBy := me;
    INC(depth)
  END Lock;

  PROCEDURE Unlock*;
  BEGIN {EXCLUSIVE}
    DEC(depth);
    IF depth = 0 THEN lockedBy := NIL END
  END Unlock;

END ReentrantLock;
```

The `ReentrantLock` Object allows to re-lock an object by its owner more than once. Clients of this object must explicitly use `Lock` and `Unlock` instead of tagging their protected regions with `EXCLUSIVE`.

Care must be taken, whenever such an object also contains a synchonization primitive (e.g. wait) whose semantic requires to automatically release the lock, and to take it again when the thread is resumed. Because another thread may be resumed prior to the current one, and thus take the lock, the reentrancy counter must be saved in the current thread state. This is easily achieved using a variable local to the procedure causing the synchronization (the stack is private to each thread).

```
PROCEDURE Wait*;
VAR count: LONGINT;
BEGIN
  count := SELF.count;
  .. release lock ..
  .. suspend thread ..
  .. acquire lock ..
  SELF.count := count;
END Wait;
```

A.1.4 Binary and Generic Semaphores

```
MODULE Semaphores;

TYPE
  Sem* = OBJECT  (* Binary Semaphore *)
      VAR  taken: BOOLEAN

      PROCEDURE P*;  (*enter semaphore*)
      BEGIN {EXCLUSIVE}
          AWAIT(~taken); taken := TRUE
      END P;

      PROCEDURE V*;  (*leave semaphore*)
      BEGIN {EXCLUSIVE}
          taken := FALSE
      END V;

      PROCEDURE & Init;
      BEGIN  taken := FALSE
      END Init;
  END Sem;

  GSem* = OBJECT  (* Generic Semaphore *)
      VAR  slots: LONGINT;

      PROCEDURE P*;
      BEGIN {EXCLUSIVE}
          AWAIT(slots > 0); DEC(slots)
      END P;

      PROCEDURE V*;
      BEGIN {EXCLUSIVE}
        INC(slots)
      END V;

      PROCEDURE & Init(n: LONGINT);
      BEGIN slots := n
      END Init;
  END GSem;

END Semaphores.
```

The well-known synchronization primitive by Dijkstra [Dij68]. Note that the ability to implement semaphores shows that the Active Oberon model is also a synchronization primitive and is powerful enough to support protection and synchronization of concurrent processes.

A.1.5 Barrier

```
MODULE Barriers;  (** prk/pjm 12.6.97  **)
(*
  A barrier is used to synchronize N activities together.
*)
  TYPE
    Barrier = OBJECT
      VAR n, N: LONGINT;

      PROCEDURE Enter*;
        VAR i: LONGINT;
      BEGIN {EXCLUSIVE}
        i := n DIV N;
        INC(n);
        AWAIT (i < n DIV N)
      END Enter;

      PROCEDURE & Init (nofProcs: LONGINT);
      BEGIN
        N := nofProcs; n := 0
      END Init;

    END Barrier;
END Barriers.
```

Barriers are used to synchronize activities together. If activities are defined as

$$P_i = Phase_{i,0}; Phase_{i,1}; .....Phase_{i,n}$$

then the barrier is used to ensure that all activities will complete $Phase - i, j$ before starting $Phase_{i,j+1}$. One thread of execution would look like this:

```
  FOR j := 0 TO N DO
    Phase(i, j); barrier.Enter
  END;
```

A.1.6 Bounded Buffer

```
MODULE Buffers;

CONST
  BufLen = 256;

TYPE
  (* Buffer- First-in first-out buffer *)

  Buffer* = OBJECT
    VAR
      data: ARRAY BufLen OF INTEGER;
      in, out: LONGINT;

    (* Put - insert element into the buffer *)

    PROCEDURE Put* (i: INTEGER);
    BEGIN {EXCLUSIVE}
      AWAIT ((in + 1) MOD BufLen # out);  (*AWAIT ˜full *)
      data[in] := i;
      in := (in + 1) MOD BufLen
    END Put;

    (* Get - get element from the buffer *)

    PROCEDURE Get* (VAR i: INTEGER);
    BEGIN {EXCLUSIVE}
      AWAIT (in # out);  (*AWAIT ˜empty *)
      i := data[out];
      out := (out + 1) MOD BufLen
    END Get;

    PROCEDURE & Init;
    BEGIN
      in := 0; out := 0;
    END Init;

  END Buffer;

END Buffers.
```

Buffer implements a bounded circular buffer. The methods *Put* and *Get* are protected against concurrent access; they also check that a buffer slot , resp. data, is available, otherwise the activity is suspended until the until the slot or data become available.

A.2 Active Object Examples

A.2.1 Dining Philosophers

```
MODULE Philo;

IMPORT  Semaphores;

CONST
    NofPhilo = 5;  (* number of philosophers *)

VAR
    fork: ARRAY NofPhilo OF Semaphores.Sem;
    i: LONGINT;

TYPE
    Philosopher = OBJECT
      VAR
        first, second: LONGINT;
        (* forks used by this philosopher *)

      PROCEDURE & Init(id: LONGINT);
      BEGIN
          IF id # NofPhilo-1 THEN
              first := id; second := (id+1)
          ELSE
              first := 0; second := NofPhilo-1
          END
      END Init;

    BEGIN {ACTIVE}
        LOOP
            .... Think....
            fork[first].P; fork[second].P;
            .... Eat ....
            fork[first.V; fork[second].V
        END
    END Philosopher;

VAR
  philo: ARRAY NofPhilo OF Philosopher;

BEGIN
    FOR i := 0 TO NofPhilo DO
      NEW(fork[i]);
      NEW(philo[i]);
    END;
END Philo.
```

A.2.2 Sieve of Eratosthenes

```
MODULE Eratosthenes; (* prk 13.09.00 *)

IMPORT Out, Buffers;

CONST
  N = 2000;
  Terminate = -1;    (* sentinel *)

TYPE
  Sieve = OBJECT (Buffers.Buffer)

    VAR prime, n: INTEGER; next: Sieve;

    PROCEDURE & Init;
    BEGIN
      Init^;    (*call Buffer’s (superclass) initializer *)
      prime := 0; next := NIL
    END Init;

  BEGIN {ACTIVE}
    LOOP
      Get(n);
      IF n = Terminate THEN
        (* terminate execution *)
        IF next # NIL THEN next.Put (n) END;
        EXIT
      ELSIF prime = 0 THEN
        (* first number is always a prime number *)
        Out.Int(n, 0); Out.String(" is prime"); Out.Ln;
        prime := n;
        NEW (next)
      ELSIF (n MOD prime) # 0 THEN
        (* pass to the next sieve if not a multiple of prime *)
        next.Put (n)
      END
    END
  END Sieve;

  PROCEDURE Start*;
  VAR s: Sieve;  i: INTEGER;
  BEGIN
    NEW(s);
    FOR i := 2 TO N-1 DO s.Put (i) END;
    s.Put(Terminate)    (* use sentinel to indicate completion*)
  END Start;

END Eratosthenes.
```

`Eratosthenes` implements the sieve algorithm for finding prime numbers. Every sieve is an active object that passes the all the received values that are not a multiple of the first received value to the next sieve. The synchronization between sieves is encapsulated in the buffer.

B
Critique of the Oberon Language

This appendix discusses the experience gained using the Active Oberon language. We emphasize the weaknesses of the language and propose some improvements. Many of the problems are inherited from Oberon, and since Active Oberon is a backward compatible extension of it, we could not fix them.

We think that the Oberon Language is underspecified. The original report [Wir88] is extremely short, relying on in the reader's common sense and intuition; this has the great advantage of making the language simple to learn (less rules to learn, reuse of known rules), but on the other hand, it leaves some "grey zones"in the language.

The report specifies the language's syntax and semantic. The syntax is the backbone of the language, showing how to construct a program, while the semantic rules give a meaning to the program. Our interpretation of the report can be summarized like this: *what is allowed by the syntax and not forbidden by the semantic*. If the syntax allows it and the semantic doesn't forbid it, then it is legal.

Even if the list of criticized topics seems long, most of them are a source of troubles only for compiler implementors, not a real problem for language users. Compared with the criticism against other languages [Ker83, Thi99], our remarks can appear to be meager but have to be made as an aid to understand the language better. We can conclude that Oberon is well designed.

Unless explicitly specified, in the following sections we will use the term Oberon to refer to both the Oberon and the Active Oberon languages.

B.1 Syntax

B.1.1 Semicolons

The use of semicolons in the language is inconsistent; declarations are terminated by them, whereas statements are separated by them. As Oberon is usually criticized for its different use of semicolon from other mainstream languages, it should at least handle them in a consistent way.

We propose to use everywhere semicolons as separators, as it is the semicolon use that Oberon is usually associated with. In practice this is just a matter of taste.

B.1.2 Ambiguous Constant Type

Character constants and one-letter strings share the same notation (`"x"`). This proves to be a problem in the compiler, as the type of the constant depends on the context where it is used. When passed as actual parameter to an `ARRAY OF SYSTEM.BYTE` or type-casted with `SYSTEM.VAL`, which are compatible to all types, the constant type cannot be determined.

There are three possible solutions to this problem: 1) using a different notation for character constants and strings, 2) making CHAR constants compatible to strings, and 3) making one-letter strings compatible to CHAR constants. The first solution would be the best one, but appears to be extremely unpractical, because of all the legacy code; we think that ”x”should be considered a string, and that a string of length 1 should be compatible to CHAR.

B.1.3 Ambiguous Productions in the Grammar

The Oberon grammar contains ambiguous productions, which require semantic knowledge to be parsed correctly:

qualified identifiers It is impossible for the parser to distinguish between a qualified identifier `M.p` and a record field `r.f`, in the `designator` production[1]

type casts `x(y)` in the production `term` can be parsed to a type cast of designator `x` to type `y`, or to a function call of `x` with parameter`y`.

factor `CharConstant` and `string` both begin with same delimiter. Usually the scanner handles this case, but for one-letter strings the conflict remains.

constant expressions The `ConstExpression` production requires semantic informations to make sense; its parsing is not ambiguous.

These design impurities are likely to be the consequence of using top-down 1-pass parsers, where syntactic and semantic analysis are performed at the same time; the semantic information is readily available and the appropriate production can be chosen. Nevertheless, this can be a problem when syntactic and semantic analysis are separated, or when using parser generators like CoCo\\R [M¨os90] or Yacc [Joh75].

B.2 Semantic

B.2.1 Numeric Constants and Constant Folding

The type of a numeric constant is the minimal type to which the number belongs [Wir88, §3.2]. The result of an arithmetic operation is the operand's type which includes the other operand's type (except for division). The type-casting functions have clearly defined semantics.

[1]This problem already existed in Modula [Wir77].

These semantic rules are quite clear and make sense for the operations on variables; for operation on constants, this easily leads to misleading results: as an example, declaring a buffer of size $64*1024$ (in the programmer's intention 64 KB), the compiler must compute it to be 0, or throw an overflow exception; the first operand is a SHORTINT and the second one an INTEGER, thus the result must be of INTEGER type; as the result too big to fit into the result type, it is either truncated or an overflow exception is thrown. This is obviously not what a programmer would intuitively expect.

Applying the same rules to the type cast functions should also be clear: LONG and SHORT change the type of the value. This means, to hold 65536 as result, the previous expression should have been written as $64 * LONG(1024)$.

The language has a ConstExpression production used for defining constants and array sizes; the value of these expression is needed at compile-time and thus constant folding is implicitly required. We investigated the conformance to the semantic rules of various Oberon compilers[2]: all implement constant folding optimization to some extent; they compute the value of constant expressions at compile-time and use only the expression's result in the code. No compiler conformed to the language semantic whenever constant folding was applied.

In oo2c [Ooc] all numeric values are untyped; the type is set only when the constant is used in a non-constant expression [vH01]. This strategy allows to evaluate arithmetic operations in a type-independent way[3]; type casts are considered identity functions and have thus no effect on the values: writing a 32-bit memory location using the polymorph function SYSTEM.PUT with an arbitrary constant value is a difficult task, because the type of the constant depends on its value and SYSTEM.PUT adapts itself silently; as a workaround, the constant shall first be assigned to a variable, and then the variable written to the memory.

OP2 and Paco take a more schizophrenic approach. Numeric constants are typed, but arithmetic operations are applied to the most expressive type; the result type is set according to the result value. This allows to perform constant folding without incurring into overflow[4], and to use the casting functions to change the type of a constant as needed to implement low-level programs.

There are four possible solutions to this problem: 1) keep the current language semantic and enforce it in constant expressions too; 2) production `ConstExpression` shall be evaluated outside the type system; 3) explicitly typed constant values; 4) have only one integer type in the language.

We are convinced that proposal 1) is too confusing to be implemented: it goes against the user expectation of how constant expression are to be evaluated[5]. Proposal 2) introduces two different semantics for the same syntax, and is thus rather confusing, although this is the way ETH compilers are currently implementing constant folding. Proposal 3) would solve the problem in a pragmatic way; many other languages already have integrated the type and encoding information

[2]OP2, Paco, oo2c, and Wirth's ARM compiler

[3]In practice, constants are evaluated with the most expressive numeric type LONGINT

[4]a error is emitted, if the result cannot fi t into a LONGINT

[5]Writing `64*LONG(1024)` is not a pragmatic solution

into the constant values; Oberon does this in some cases: for the hexadecimal char constants, the hexadecimal numeric constants, and the real constants . This would solve the problem in an elegant way while fitting nicely into the language semantic. Proposal 4) is also appealing, as it would simplify the language, compiler, and remove this problem too; the Oberon-0 language used by Wirth for the compiler construction lecture, which relies only on the `INTEGER` and `BOOLEAN` types, is expressive enough for many applications. The only shortcoming would be in the low-level programming, where the hardware requires values of a predefined size. Mapping an hardware structure directly into an Oberon structure would become more difficult, as registers with size smaller than the machine word would have to be treated specially. A possible solution would be to substitute the `PUT` and `GET` polymorphic functions of module `SYSTEM` with a non-polymorphic functions (`PUT8`, `PUT16`, `PUT32`, `GET8`, `GET16`, `GET32`). We already introduced such functions to reduce the confusion created by the implicitly typed constants used in polymorph functions; the experience with them was very satisfying.

As a comparison, Java takes a different approach: every arithmetic operation is evaluated in the type `int`[6]; a sum of two short returns an `int`. To be assigned to a field of a lesser type, the value must be casted This removes the problem in an elegant way for Java, but Oberon makes an heavy use of all three integer types, and this would invalidate much of the existing code.

B.2.2 Scope vs. Block

In Oberon, no identifier may denote more than one object within its definition scope (§A.4). This can be misleading, as a definition scope does not correspond to a block: it extends from the definition itself to the end of the current block. This means that in a block, an identifier can have two meanings: between the beginning of the block and the declaration (inherited from the parent block), and another meaning between the definition and the end of the block.

Pointer declarations particularly emphasize this problem: if a type `T` is defined as `POINTER TO T1`, then `T1` may be declared textually later in `T`'s scope[7](§4.1). In the case where T1 is defined twice, once in a visible outer scope and once after T in the same block, the report doesn't specify which declaration has the precedence. Our interpretation is the definitions in the same block must have the precedence; our experience shows that the Oberon compilers usually implement the other variant. Reali [Rea00] first reported this problem and gave an example.

Active Oberon removes this problem by extending a declaration's scope to the whole block.

B.2.3 Error Handling and Exceptions

Errors and exceptions are a grey zone in the language report: they are often underspecified or left out. Obviously, dereferencing a NIL pointer should produce a run-time exception, and compiling

[6]There is an exception: if at least one of the operands is of type `long`, the operation and the result are also of type `long`

[7]This introduces another hole, as the declaration scope may have nested blocks and T1 could be declared in one of those; this is not explicitly restricted

a semantically incorrect program should be reported by the compiler.

In some cases the report states that an operation is illegal, but does not tell if a declaration is legal or not, and if the error can be emitted at compile-time or only at execution-time.

As an example, in an array only *elements with indices between 0 and length minus 1* can be accessed (§6.2). This implies that an array with a negative or null size is useless, as it is not possible to access its elements (nor does it make sense). The language semantic does not restrict negative sized arrays, so we assume they should be legal.

The various compiler implementations make different assumptions, making programs non-portable whenever using those types. Usually OP2 and Paco take a very rigorous attitude, emitting compilation errors for all these cases, while oo2c tolerates them and inserts a run-time exception in the code.

We found implementation differences for these constructs (the language leaves them unspecified):

- array declaration with zero or negative length
- dynamic array allocation with zero or negative length
- ASSERT(FALSE)
- overflow detection and handling

B.3 Design

B.3.1 Visibility Rules

Active Oberon forces a more object oriented programming style than Oberon; this stresses the visibility rules and makes the module-based export rules somewhat inadequate.

Because of subclassing, the fields and methods defined in the superclasses are visible without importing the modules defining them. Another problem is, that every procedure in a module can access and modify the object fields, although this should be allowed only to the methods of the object.

Active Oberon needs a new visibility modifier to allow fields and methods to be visible to the subclasses only.

Although not strictly necessary, the read-only visibility modifier introduced in Oberon-2 has proved to be very efficient. Reali [Rea00] advocates its use, and explains in detail the reasons for adopting it in Active Oberon.

B.3.2 Active Oberon Initializers

There are currently two notations for Active Oberon initializers, the one introduced previously in this chapter and the Oberon.Net one [Gut01a], where all methods named ”NEW”are automatically considered to be initializers.

Our definition promotes every method tagged with ”&”to an initializer; as we do not allow overloading in the language, overriding a method requires the new method implementation to have the same signature as the overridden one, and it is extremely unlikely that the initializer for a class will have the same parameters as the superclass’ one, thus every initializer must have a different name. On the other hand, initializers are can be used like normal methods.

In Oberon.Net, all class initializers are named ”NEW”. This is easier to read, and no conflict with the overriding rule is possible, because there is no subclassing in the language. A problem arises with the use of NEW, which is a language predefined procedure with a different meaning: this prevents the programmer from explicitly calling the initializer, and requires to make exceptions in the visibility rules by defining the built-in NEW always to be always visible, as it would be otherwise shadowed by the initializer.

As Active Oberon heavily relies on subclassing, using the Oberon.Net definition would require to introduce many exceptions in the language, creating confusion and incompatibility.

We think that initializers must be clearly identifiable (thus have a predefined name) and should be callable as normal methods. The issue with the non-matching signatures can be tolerated in a clearly defined and limited case like this one. An approach like in Java and C# , where the class initializer has the same name as the class seems a good idea.

B.3.3 Built-in Types and Constants

The built-in Oberon types and constants are predefined identifiers. The keyword `NIL` is the only exception. Like every identifier, predefined identifiers can be overridden by a local declaration; the programmer is allowed to redefine them. This is uncommon and quite dangerous. Wirth already recognized that, but for some reason he restrained from promoting those identifiers to keywords.

This is an example related by Wirth, about the havoc such a practice could cause:

```
CONST  TRUE = FALSE;
VAR b: BOOLEAN;
BEGIN
  b := FALSE;
  IF b THEN ... END;  (* condition is false *)
  IF b = TRUE THEN ... END; (* condition is true *)
END
```

We think that the built-in types and constants should be keywords of the language.

B.4 Other Proposals

B.4.1 Splitting SYSTEM

The module `SYSTEM` has been traditionally used as a repository for unsafe and low-level functions inlined by the compiler into the code. We think it would be beneficial to split this module in two parts: a module `SYSTEM` or `UNSAFE`, containing the unsafe but portable functions like rotations, shifts, and type-system overriding features; and a module containing the functionalities available only on a specific platform, which would have the platform name (e.g. `INTEL`, `MAC`, `UNIX`).

This would simplify the compiler portability and cross-compilation issues, as every back-end would define its own low-level module, instead of redefining SYSTEM like it is now. It would also simplify the porting of the system, by allowing to localize which modules are unsafe and which ones are not portable.

Bibliography

[AAB$^+$00] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P.Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Virtual Machine. *IBM System Journal*, 39(1), February 2000.

[ACF$^+$01] B. Alpern, A. Cocchi, S. Fink, D. Grove, and D. Lieber. Efficient Implementation of Java Interfaces: Invokeinterface Considered Harmless. In *Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'01)*, ACM Sigplan Notices, pages 108–124. ACM Press, October 2001.

[ADvRF01] W. Amme, N. Dalton, J. v. Ronne, and M. Franz. SafeTSA: A Type Safe and Referentially Secure Mobile-Code Representation. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI-01)*, 2001.

[And91] G. Andrews. *Concurrent Programming: Principles and Practice*. Addison-Wesley, 1991.

[App] Apple Computer, Inc. *Apple Technical Library, Inside Macintosh, Overview*.

[ATCL$^+$98] A-R. Adl-Tabatabai, M. Cierniak, G-Y. Lueh, V. Parikh, and J. Stichnoth. Fast, Effective Code Generation in a Just-In-Time Java Compiler. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI-98)*, ACM Sigplan Notices, pages 280–290. ACM Press, 1998.

[Bec99] K. Beck. *Extreme Programming Explained*. Addison-Wesley, 1999.

[BG96] T. Bergin and R. Gibson, editors. *History of Programming Languages - II*. ACM Press, 1996.

[BH72] P. Brinch Hansen. Structured Multiprogramming. *Communications of the ACM*, 15(7):574–578, July 1972. Reprinted in The Search for Simplicity, IEEE Computer Society Press, 1996.

[BJPW99] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making Components Contract Aware. *Computer*, 32(7):38–45, July 1999.

[BPSMM00] T. Bray, J. Paoli, C.M. Sperberg-McQueen, and E. Maler. Extensible Markup Language (XML) 1.0. Technical report, World Wide Web Consortium, 2000.

[Bre95] R. Brega. Real-Time Kernel for the Power-PC Architecture. Master's thesis, Institut f¨ur Robotik, ETH Z¨urich, 1995.

[Bro95] F. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, corrected reprint edition, 1995.

[CL97] P. Chan and R. Lee. *The Java Class Libraries*, volume 2: java.applet, java.awt, java.beans of *The Java Series*. Addison-Wesley, 2nd edition, October 1997.

[Cla] GNU Classpath. http://www.classpath.org/.

[CLK98] P. Chan, R. Lee, and D. Kramer. *The Java Class Libraries*, volume 1: java.io, java.lang, java.math, java.net, java.text, java.util of *The Java Series*. Addison-Wesley, 2nd edition, March 1998.

[CLS00] M. Cierniak, G-Y. Lueh, and J. Stichnoth. Practicing JUDO: Java under Dynamic Optimizations. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI-00)*, ACM Sigplan Notices, pages 13–26. ACM Press, May 2000.

[Cor01] Microsoft Corporation. *Microsoft C# Language Specifications*. Microsoft Press, 2001.

[COR02] Object Management Group (OMG). *Common Object Broker Request Architecture (CORBA/IIOP)*, 3.0.1 edition, November 2002.

[Cre90] R. Crelier. OP2: A Portable Oberon Compiler. Technical Report 1990TR-125, Department of Computer Science, ETH Z¨urich, 1990.

[Cre91] R. Crelier. OP2: A Portable Oberon-2 Compiler. In *Second International Modula-2 Conference*, Loughborough University of Technology, UK, 1991.

[Cre94] R. Crelier. *Separate Compilation and Module Extension*. PhD thesis, ETH Z¨urich, 1994.

[CW85] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[dI02] M. de Icaza. Experiences from the Mono Project. Talk at Z¨urich University, November 2002.

[Dij68] Edsger W. Dijkstra. The Structure of the THE-Multiprogramming System. *Communications of the ACM*, 11(5):341–346, May 1968.

[Dis97] A. R. Disteli. *Integration aktiver Objecte in Oberon am Beispiel eines Serversystems*. PhD thesis, ETH Z¨urich, 1997.

[DR97] A. Disteli and P. Reali. Combining Oberon with Active Objects. In H. M¨ossenb¨ock, editor, *Proceedings of the JMLC*, volume 1204 of *LNCS*, pages 221–235, 1997.

[Ebe87] H. Eberle. *Development and Analysis of a Workstation Computer*. Dissertation 8431, ETH Zürich, 1987.

[ECM01] ECMA. *ECMA 335 - Common Language Infrastructrure (CLI)*, December 2001.

[Egg01] B. Egger. Development of an Aos Operating System for the DNARD Network Computer. Master's thesis, Institut für Computersysteme, ETH Zürich, 2001.

[Fra94] M. Franz. *Code-Generation On-the-Fly: A Key for Portable Software*. PhD thesis, Institut für Computersysteme, ETH Zürich, 1994.

[Frö97] P. Fröhlich. Projekt Froderon: Zur weiteren Entwicklung der Programmiersprache Oberon-2. Master's thesis, Fachhochschule München, 1997.

[GC00] J. Gough and D. Corney. Evaluating the Java Virtual Machine as a Target for Languages Other than Java. In J. Gutknecht and W. Weck, editors, *Proceedings of JMLC*, volume 1897 of *LNCS*, Zurich, Switzerland, 2000. Springer.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, 1st edition, 1996.

[Gla99] R. Glass. *Computer Calamities*. Prentice Hall, 1999.

[Gri93] R. Griesemer. *A Programming Language for Vector Computers*. Dissertation 10277, ETH Zürich, 1993.

[Gri00] R. Griesemer. A Compiler for the Java HotSpot Virtual Machine. In *The School of Niklaus Wirth*. dpunkt.verlag, 2000.

[Gut01a] J. Gutknecht. Active Oberon for .NET. http://www.oberon.ethz.ch/oberon.net/whitepaper/, June 2001.

[Gut01b] J. Gutknecht. Active Oberon for .NET: An Exercise in Object Model Mapping. In N. Benton and A. Kennedy, editors, *BABEL'01 First International Workshop on Multi-Language Infrastructure and Interoperatibility*, volume 59.1 of *ENTCS*. Elsevier, September 2001.

[GW92] J. Gutknecht and N. Wirth. *Project Oberon - The Design of an Operating System and Compiler*. Addison-Wesley, 1992.

[Hal77] M. Halstead. *Elements of Software Science*. Elsevier North-Holland, New York, 1977.

[Hoa74] C. A. R. Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10):549–557, October 1974. Erratum in *Communications of the ACM*, Vol. 18, No. 2 (February), p. 95, 1975. This paper contains one of the first solutions to the Dining Philosophers problem.

[Hoa78] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science, 1985.

[Hot99] The Java Hotspot Performance Engine Architecture. White Paper, 1999. http://java.sun.com/products/hotspot/whitepaper.html.

[HP92] B. Heeb and C. Pfister. Chameleon: A Workstation of a Different Colour. In *Field-Programmable Gate Arrays: Architectures and Tools for Rapid Prototyping. Second International Workshop on Field Programmable Logic and Applications*, pages 152–161, August 1992.

[HP96] J. Hennessy and D. Patterson. *Computer Architecture, A Quantitative Approach*. Morgan Kaufmann Publishers, 2nd edition, 1996.

[Int] Intel StrongARM processors. http://developer.intel.com/design/strong/.

[Int95] International Organization for Standardization. *ISO/IEC 8652:1995: Information technology — Programming languages — Ada*. International Organization for Standardization, Geneva, Switzerland, 1995.

[Int99] Intel Corporation. *Intel Architecture Optimization; Reference Manual*, 1999. Order Number 245127-001.

[Int00] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual; Volume 2: Instruction Set Reference*, 2000. Order Number 245471.

[Jan98] P. Januschke. *Oberon-XSC- Eine Programmiersprache und Arithmetikbibliothek für das Wissenschaftliche Rechnen*. PhD thesis, Universit¨at Karlsruhe, 1998.

[Jav96] JavaOS: A Standalone Java Environment. White Paper, May 1996.

[JBe] JBed. http://www.esmertec.com/.

[Joh75] S. Johnson. YACC - Yet Another Compiler Compiler. Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, NJ, 1975.

[JOS] JOS: A Free Java Based Operating System. http://www.jos.org.

[JW74] K. Jensen and N. Wirth. *PASCAL - User Manual and Report*, volume 18 of *Lecture Notes in Computer Science*. Springer, 1974.

[Ker83] B. Kernighan. Why Pascal is not my Favourite Programming Language. Technical Report 100, Bell Labs, 1983.

[Kis95] T. Kistler. Smartest Recompilation. Master's thesis, Institut f¨ur Computersysteme, ETH Z¨urich, 1995.

[Knu83] S. E. Knudsen. *Medos-2: A Modula-2 Oriented Operating System for the Personal Computer Lilith*. Diss no. 7346, ETH Z¨urich, 1983.

[KP97] P. W. Kutter and A. Pierantonio. The Formal Specification of Oberon. *Springer Journal of Universal Computer Science*, 3(5):443–503, 1997.

[Lai00] R. Laich. A Java Virtual Machine for Aos. Master's thesis, Institut für Computersysteme, ETH Zürich, 2000.

[Lev00] J. Levine. *Linkers & Loaders*. Morgan Kaufmann Publishers, 2000.

[Lia99] S. Liang. *The Java Native Interface; Programmer's Guide and Specification*. The Java Series. Addison-Wesley, 1st edition, 1999.

[LY99] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, 2nd edition, 1999.

[Mar96] J. Marais. *Design and Implementation of a Component Architecture for Oberon*. PhD thesis, Institut für Computersysteme, ETH Zürich, 1996.

[McC76] T. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976.

[Mey97] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.

[Mey99] B. Meyer. Overloading vs. Object Technology. *Journal of Object-Oriented Programming*, pages 3–7, October 1999.

[Mik00] V. Mikheev. Design of Multilingual Retargetable Compilers: Experience of the XDS Framework Evolution. In J. Gutknecht and W. Weck, editors, *Proceedings of JMLC*, volume 1897 of *LNCS*, pages 238–249, Zurich, Switzerland, 2000. Springer.

[Mor97] R. Morelli. Intergration of Oberonx in Oberon. Master's thesis, Institut für Computersysteme, ETH Zürich, 1997.

[Mös90] H. Mössenböck. Coco/R: A Generator for Fast Compiler Front-Ends. Technical Report 1990TR-127, Department of Computer Science, ETH Zürich, February 1990.

[Mös98] H. Mössenböck. *Objektorientierte Programmierung in Oberon-2*. Springer, 3rd edition, 1998.

[Mös00] H. Mössenböck. Compiler Construction - The Art of Niklaus Wirth. In L. Böszörményi, J. Gutknecht, and G. Pomberger, editors, *The School of Niklaus Wirth*, pages 55–68. dpunkt.verlag/Copublication with Morgan-Kaufmann, 2000.

[MTG89] H. Mössenböck, J. Templ, and R. Griesemer. Object Oberon: An Object-Oriented Extension of Oberon. Technical Report 1989TR-109, Department of Computer Science, ETH Zürich, June 1989.

[Muc97] S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers, 1997.

[Mul00] P.J. Muller. A Multiprocessor Kernel for Active Object-based Systems. In J. Gutknecht and W. Weck, editors, *Proceedings of JMLC*, volume 1897 of *LNCS*, pages 263–277, Zurich, Switzerland, 2000. Springer.

[Mul02] P.J. Muller. *The Active Object System – Design and Multiprocessor Implementation*. PhD thesis, Institut für Computersysteme, ETH Zürich, 2002.

[MW91] H. M¨ossenb¨ock and N. Wirth. The Programming Language Oberon-2. *Structured Programming*, 12(4):179–195, 1991.

[Ohr84] R. Ohran. *Lilith: A Workstation Computer for Modula-2*. Dissertation 7646, ETH Z¨urich, 1984.

[Ooc] ooc, Optimizing Oberon-2 Compiler. http://ooc.sourceforge.net.

[Pla01] David Platt. *Introducing the Microsot .NET Platform*. Microsoft Press, May 2001.

[Pos80] J. Postel. RFC768: User Datagramm Protocol. RFC768, August 1980.

[Pos81] J. Postel. RFC793: Transmission Control Protocol. RFC793, September 1981.

[Rad95] A. Radenski. Introducing Objects and Concurrency to an Imperative Programming Language. *Information Sciences, an International Journal*, 87(1-3):107–122, 1995.

[Rad98] A. Radenski. Module Embedding. *Software - Concepts and Tools*, 19(3):122–129, 1998.

[Ray99] E. S. Raymond. *The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly & Associates, Inc., 1999.

[Rea00] P. Reali. Structuring a Compiler with Active Objects. In J. Gutknecht and W. Weck, editors, *Proceedings of JMLC*, volume 1897 of *LNCS*, pages 250–262, Zurich, Switzerland, 2000. Springer.

[Rog96] D. Rogerson. *Inside COM*. Microsoft Press, 1996.

[RvA] P. Reali and M. van Acken. Project Hostess: the Oberon Test Suite. Web site. http://hostess.sourceforge.net.

[Sea00] David Seal, editor. *ARM Architecture Reference Manual*. Addison-Wesley, 2nd edition, 2000.

[SL94] B. A. Sanders and S. Lalis. Adding Concurrency to the Oberon System. In *Proceedings of Programming Languages and System Architectures*, Lecture Notes in Computer Science (LNCS) 782. Springer Verlag, March 1994.

[SOA00] World Wide Web Consortium (W3C). *Simple Object Access Protocol (SOAP)*, 1.1 edition, May 2000.

[SPE98] SPEC. SPEC JVM98. http://www.spec.org/, 1998.

[SSB01] R. St¨ark, J. Schmid, and E. B¨orger. *Java and the Java Virtual Machine Definition, Verification, Validation*. Springer, 2001.

[SSW87] V. Seshadri, I. S. Small, and D. B. Wortman. Concurrent Compilation. In *Proceedings of the IFIP Conference on Distributed Processing*, Amsterdam, October 1987.

[SW91] V. Seshadri and David B. Wortman. An Investigation into Concurrent Semantic Analysis. *Software - Practice and Experience*, 21(12):1323–1348, December 1991.

[Szy92] C. Szyperski. Import is not Inheritance – Why we need both: Modules and Classes. In O. Lehrmann Madsen, editor, *Proceedings, ECOOP 92*, number 615 in Lecture Notes in Computer Science, pages 19–32. Springer-Verlag, 1992.

[Szy98] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 1998.

[Tan98] A. Tanenbaum. *Structured Computer Organization*. Prentice Hall, 4 edition, 1998.

[Thi99] H. Thimbleby. A Critique of Java. *Software - Practice and Experience*, 29(5):457–478, 1999.

[Tol] R. Tolksdorf. Languages for the Java Virtual Machine. Web page, http://flp.cs.tu-berlin.de/ tolk/vmlanguages.html.

[UDD02] Organization for the Advancement of Structured Information Standards (OASIS). *Universal Description Discovery and Imtegration (UDDI)*, 3.0 edition, July 2002.

[Uni] The Unicode Standard. http://www.unicode.org/.

[vH01] M. van Hacken, April 2001. private communication.

[Weg96] P. Wegner. Interoperability. *ACM Computing Surveys*, 28(1):285 – 287, March 1996.

[Wex81] R. L. Wexelblat. *History of Programming Languages*. Academic Press, New York, 1981.

[Wha] whatis?com. http://whatis.com.

[Wir77] N. Wirth. MODULA : A Language for Modular Multiprogramming. *Software Practice and Experience*, 7:3–35, 1977.

[Wir88] N. Wirth. The Programming Language Oberon. *Software Practice and Experience*, 18(7):671–690, July 1988.

[Wir96] N. Wirth. *Grundlagen und Techniken des Compilerbaus*. Addison-Wesley, 1996.

[WJ92] David B. Wortman and Michael D. Junkin. A Concurrent Compiler for Modula-2+. *ACM SIGPLAN Notices*, 27(7):68–81, July 1992.

[Wor90] David Wortman. A Concurrent Modula-2+ Compiler. In *Proc. First International Workshop on Parallel Compilation*, Kingston, Canada, May 6-8th 1990. Department of Computing and Information Science, Queen's University. University of Toronto.

[WR92] N. Wirth and M. Reiser. *Programming in Oberon - Steps Beyond Pascal and Modula*. Addison-Wesley, 1992.

[WSD01] World Wide Web Consortium (W3C). *Web Services Description Language (WSDL)*, 1.1 edition, March 2001.

[Zel] E. Zeller. Sourcecoder. cited in [Mar96].

Curriculum Vitae

11. July 1971	Born in Sorengo, Switzerland Son of Anne-Marie and Carlo Reali
1977–1982	Primary school in Pregassona
1982–1986	Secondary school in Viganello
1986–1990	High School: Liceo Lugano 2 Matura Typus C (scientific)
1990–1995	Studies in Computer Science Swiss Federal Institute of Technology, Zurich
1994	Internship at Alcatel STR, Zürich
1995	Diploma in Computer Science, ETH Zurich (Dipl. Informatik-Ingenieur ETH)
1995–2002	Research and teaching assistant at ETH Zurich, Institute for Computer Systems, Language and Run-time Research Group of Prof. Jürg Gutknecht
2002	Senior Software Engineer, ELCA Informatik AG Zürich