# DIPLOMARBEIT

# Implementation of the Coordination Language C&Co

ausgeführt am Institut für Computersprachen
der Technischen Universität Wien

unter der Anleitung von
Univ. Doz. e. Kühn

durch


Alexander Forst

Karl Föderl Straße 3
A-3032 Eichgraben


Wien, am 21. September 1995

The author can be reached at 'alex@complang.tuwien.ac.at'.

# Contents

# Acknowledgements

I want to thank my adviser, eva Kühn, for the opportunity to be part of her group, for the many discussions, for the honor to be one of her co-authors, and for the freedom she gave me to try out other directions. I thank the Austrian "Fonds zur Förderung der wissenschaftlichen Forschung" for funding me. I thank my colleagues Herbert Pohlai for proofreading and his computer magazines, Christian Wehrl who suffered with me, Thomas Hollunder for making coffee, and Günther Spörk for his arguments about anything else except this work. Anton Ertl and Manfred Brockhaus deserve special mention for encouraging me to use an attributed grammar. Thank you for your helpful comments.

I really want to thank my family for their support, patience, and love they have invested in me over the years. Very special thanks to Mom for her daughter, my beloved Doris.

**Abstract**

This thesis describes the implementation of a compiler which translates the language C&Co into C.

C&Co is the second representative of a coordination language that works with the so called coordination kernel. The language is based on C and uses shared variables which can be communicated to other processes anywhere in the network. These communication objects are written atomically within transactions.

This work describes the concepts of coordination and gives an overview of other languages and tools that aim at the coordination of distributed systems. After that, the language extensions of C&Co are presented in detail from the point of view of the programmer. Then follows a description of the necessary tools which were used to implement the compiler. The description of the implementation concludes this diploma thesis.

# Part I

# Motivation

# Chapter 1

# Introduction

Programming in heterogeneous environments is more difficult than sequential programming. The programmer has to distribute the computations on different sites. There must be some means to control their execution states and to communicate data between their sites.[1]

Distributed applications may be very large pieces of software and may involve long lasting computations. They consume much resource and time. A failure of one system or network link can be very costly. One would expect not only a well defined failure behaviour but also, in the best case, automatic failure recovery.

The "traditional" way to solve these problems is to build a set of library functions. In addition to the simplicity of this approach, libraries have a number of disadvantages, e.g.:

- The library functions can only be called from certain languages. If another programming language does not support extern libraries, there is no way to use their new functionality.
- Most libraries consist of many functions. A programmer is not able to know exactly the behaviour of every function in the library.
- The call to library functions has nothing in common with the language a programmer wants to use. What if the algorithm he has in mind is better suited for a language like Prolog, Lisp, or—for the bold—APL? Does the functionality of the library functions fit into the paradigm of my language, e.g., what about backtracking in Prolog?

Another matter to think of is, does the programmer really have to worry about network programming, transmission of messages, retransmission of lost messages, or dealing with network failures?

Most tools for distributed programming are based on message passing for communication and synchronisation purposes. However, the shared data paradigm is conceptually superior at the same level of expressiveness. The programmer

[1] The work was developed in the framework of project P09020-MAT of the Austrian "Fonds zur Förderung der wissenschaftlichen Forschung".

has the vision of a global accessible, shared object space and doesn't have to decompose his task into subtasks—which have nothing in common with the original task, most of the time—that send and receive messages.

The group led by my adviser, eva Kühn, has been developing a distributed coordination kernel, the CoKe [34]. The kernel is language independent, runs on different platforms with different word lengthes and endian types, and can be used as a library from any language supporting external linkage. It uses communication objects which are communicated to other sites in a reliable way without involvement of the programmer.

C&Co [33] is, after VPL [39, 38], the second coordination language which communicates directly with the coordination kernel. As might be expected, C&Co uses the same communication objects exactly as VPL because both languages use CoKe for communication [21].

The implementation of the C&Co compiler was the main part of my work. It is realised as a precompiler. It translates C&Co source code to C.

This master thesis consists of three parts. The first part gives a motivation for the need of coordination languages. It describes coordination concepts and gives a short overview of other languages and coordination tools.

The second part details the language C&Co. It gives a summary of all language extensions and their application in various programming situations and is the first complete user manual of the C&Co language.

The third part describes the implementation of C&Co. It starts with an explanation of the tools which were used to build the compiler and portrays the data structures and functions of the program.

# Chapter 2

# Motivation of Coordination Languages

The increasing number of computer systems in addition to the economic compulsion for rationalisation is of great importance to the integration of existing systems. This leads to a variety of new problems, e.g., the preservation of the autonomy of local systems or the reliability of the global system in case of local failures.

Parallel hardware is still far from being widely used, although it is accepted as the technology of the future. The main justification for parallel hardware architectures is the higher performance of existing programs. New compilers which support parallel architectures can do the job sufficiently. The source of the old program is not changed in any way.

Better results can be achieved when the programs are designed from scratch to implement a true parallel algorithm. No compiler is able to fully understand the semantics of a program in order to transform it automatically into another, semantically equivalent parallel algorithm. For this purpose, programming languages are extended by some constructs to express parallelism, to control communication, and to synchronise concurrently running processes.

On the other hand, it is very cheap to connect workstations to a local or wide area network. Availability of distributed computing power opens a lot of new application domains and thus poses new requirements on programming tools.

Distribution of computations makes a better use of resources and can be more reliable because it can provide more redundancy than a single computer system. A machine can offer its computation power, its memory, its data, and its services to other machines on the network. The use of data includes the retrieval of data from huge repositories, the use of functions from a specialised function library, or the use of a specialised system. Many applications run only on special hardware but it could be possible for other machines to access their data, services, and functionality.

The use of systems opens many new perspectives that are not yet fully understood. There is no adequate software support available to make their services accessible in general. There is a need for a new high-level programming tool that allows to easily specify parallel activities and to control the data flow between those processes.

## 2.1 Distributed Environments

The number of challenges that will arise when one wants to write programs in such a distributed heterogeneous environment include:

**Distribution of already existing programs.** The problem is to change programs which are written for sequential machines in such a way, that they can run in a distributed environment. The situation is quite different from a program, running on a parallel machine.

In a distributed environment, virtually anything can happen all the time. Network connections might slow down or fail, a new machine is connected to the network, or another one is removed. There might be the opportunity to get a job done by a different service, at least in a semantically equivalent way (the booking of a ticket at different airlines comes to mind). As a consequence, it is impossible to balance the task load on different machines because of the instability of the heterogeneous environment. The only solution is to make it easy to fine-tune the software in a flexible way.

A parallel architecture runs as an isolated island with little or no connection to the rest of the world. A failure effects the entire system.

**New application domains.** Programming will slowly change towards organising existing services which are offered by various sources. The programmer has to decompose a problem into pieces that can be solved autonomously by other systems. The main advantage is, that not only passive data are shared but also functions can be communicated to other sites. One part of the task of a programmer is to find the sites which offer the needed services and, e.g., to minimise the rental costs.

**Administration.** New programs are needed to administrate this newly created "electronical software supermarket". A centralised solution is not feasible for a widely distributed algorithm. Thus, any administrative program itself must be a distributed algorithm. The importance of its availability and correctness cannot be overestimated.

As a consequence of the above, programmers should work with adequate tools that support them to formulate distributed algorithms. This should not be done with low-level tools because the complexity of distributed programs is too high. Nobody will want to undertake responibility for programming errors or unmanageable problems. A high-level programming tool should have the following features:

- There have to be language constructs to start processes which run in parallel, either on the same site or on another site.

- There must be means to pass data to and from other processes and to synchronise parallel activities.
- The programmer should not have to deal with failures (in the best case). These failures have nothing in common with the task that has to be solved. The programmer should not have to know (conceptually) that the spawned processes run actually on different sites.
- Hardware abstraction should be as high as possible. Thus, it will be easy to port and adapt programs to different systems.
- Security is very important. Every site participating in some form of communication must agree upon a global security policy. For example, if many global transactions access the same databases, which are located on distributed sites, and there is no global access policy, it is obvious that conflicts may arise at the global level.

As we have seen from the necessity of a global policy it sounds logical, that the programming support for distributed systems will have to offer the functionality of a complete operation system. One isolated programming language or system is simply not sufficient to solve the addressed problems. On the other hand, one completely new—even optimal—distributed system is not the solution.

There are so many different programming paradigms that we have to find a more general solution. One way to solve this problem consists of the following two steps:

1. Development of a new and advanced paradigm for coordination.
2. Enhance existing languages by this coordination paradigm without destroying their language paradigm.

It is desirable to extend common programming languages by this concept of coordination in an orthogonal way without changing or destroying their semantics. Every paradigm should be supported. There is the nice side effect, that every language which is extended by such a coordination paradigm can automatically communicate with other languages of this sort. A second thought is, that it makes no difference if one develops a program with such an extended language for a distributed environment or a parallel architecture. The needs for concurrency, communication, and synchronisation are identical, even though at a different granularity. If such an abstraction can be achieved, porting software between these different architectures will become easy.

## 2.2 Current Software Support

Current software support for distributed programming comes from three directions:

**Database Systems.** The database research community has recognised the problems, which are caused by heterogeneous systems, to their full extent. Multi database systems have emerged that aim at the integration of existing systems. However, the problems, e.g., semantic data integration, are too complex to be solved by database technology alone. But, on the other hand, the need for global concurrency control strategies and advanced transaction models has been recognised clearly.

**Operating Systems.** Operation Systems and their numerous add-ons for parallel and distributed processing (sockets, RPC, threads) are (unfortunately) the state of the art. However, their use is quite cumbersome and does not necessarily lead to elegant solutions.

**Coordination Languages.** Numerous languages have been proposed recently which support distributed programming. Many of them use shared data instead of message passing because programming with shared data is more comfortable and less error-prone at the same expressiveness. Chapter 4 includes an overview of these languages.

Coordination languages are a new concept for the programming of distributed computer systems. They are an attempt to integrate various computations or processes.

A coordination language provides mechanisms to initiate computations and to communicate between newly established processes. The language is sort of a *mega-programming language* [54]. This notion was coined by Wiederhold, Wegner, and Ceri and results from the increasing need to combine existing applications (so called legacy systems) into larger functional units. This must be done with minimal intrusion into the existing application. The application might not be accessible or it is impossible to change it. In multidatabase theory, this is known as the *local autonomy* of the sites offering these applications [22].

# Chapter 3

# Coordination Concepts

Coordination requires language constructs to control local and autonomous software systems that are distributed in a network and may run concurrently [54]. A coordination language must be able to

- pass parameters to local and autonomous systems,
- initiate and control computations at those systems,
- pass results back and forth between systems,
- and combine those results.

Additionally, if a local system cannot fulfil its task, another system may be selected instead. This is known as *function replication* [6].

## 3.1 Shared Objects as Reliable Communication Media

Message passing and communication via shared memory are communication mechanisms with equal expressiveness. Approaches to distributed computing, that are operating system oriented, tend to use message passing. Imperative languages can call communication primitives which are implemented as library functions. In general, message passing primitives, such as synchronous or asynchronous message passing, RPC, and rendezvous, are low level communication concepts that do not necessarily fit well into the host language and thus do not lead to a clean integrated language concept which meets the requirements of distributed systems.

Recently, shared objects have been proposed as the communication medium for many advanced language paradigms (for example, shared logic variables [26, 49], shared data structures [13, 9], shared updateable objects that are written in atomic actions [5], and shared objects providing transactions and fault-tolerance [17]).

Shared object communication provides a higher level of abstraction. The details of the underlying hardware can be hidden to a high degree. Every site which participates in the communication has a common view of the shared memory locations which represents the shared data objects. The programs on those sites can access these objects as if they resided on their local computer. Therefore it is possible to embed the shared objects into languages in a way which makes them almost indistinguishable from the objects of this language.

We propose therefore to communicate by a mechanism that belongs to the class of shared memory communication, and introduce *communication objects* as communication media. The programmer has the vision of a possibly distributed, globally shared space, comparable with a distributed database. A communication object is just as a normal variable. It can hold the same values and can have the same types which are supported by C including structures and arrays. It can also contain other communication objects as components or point to them, e.g., to form infinite lists.

The main features of communication objects in this shared space are that they are written in transactions and they can be assigned a value only once (*rigorous single assignment property*). Each process which runs in parallel can rely on data read from the shared space, because the values of communication objects—once assigned—will not change and is recoverable in case of a failure.

Every process has its own window to the globally shared space. Only objects which are shared between communicating tasks are visible, i.e., accessible from the processes. The assembly of communication objects is also called the process's *shared space window*. The process maintains it's local data as usual. Data types between local and shared space are fully compatible and the semantics of the operators of C is extended to handle communication objects.

Depending on the underling architecture, a communication object is either held in shared memory or each process that shares this object owns a copy of it. One copy is designated as the *primary copy* which must be owned by the process in order to access it, the other copies are named *secondary copies*. The protocol and the application of this reliable communication mechanism in the domain of multi database systems are presented in [34].

Between processes, objects are shared by passing them as arguments (global names of communication objects are not required). Processes that have not been given a reference to a communication object cannot access it. This mechanism provides security as only authorised processes may access the data.

Communication objects provide one uniform mechanism for inter- and intra-process communication.

## 3.2 Concurrency

Concurrency in a coordination language is needed because the software systems it coordinates already exist and run in parallel on different sites. A language that provides concurrency is the adequate tool to reason about the parallelism that exists in the real world. Although concurrency may be emulated with sequential languages, the abstraction provided by concurrent language constructs is much higher.

We differentiate between inter- and intra-process parallelism. The first one is crucial for coordination languages and requires a language primitive (we call it `process`) that spawns and controls a process on another site, and to pass communication objects which become shared between the initiating site and the site where the new process is created.

The computer site at which the process is created can be specified. As a default the local site is assumed and the process is executed as another thread of the system by which it was called. This provides a sufficient form of intra-process parallelism for imperative languages.

Intra-process parallelism is not a necessary feature of coordination languages. However, if the language paradigm is chosen adequately, it may be provided also.

## 3.3 Advanced Transaction Model

All processes that share communication objects must have a consistent view of these objects. This can be achieved by ensuring that values can be written into the shared space by transactions only. Classical transactions provide the ACID (Atomicity, Consistency, Isolation and Durability) properties and thus the writing of the objects becomes serialisable and partial failure recovery can be guaranteed, as communication objects can be recovered after a system crash.

As these communication objects are used for the communication between autonomous systems, we propose to use an advanced transaction model, the Flex Transaction Model [18], that has been designed especially to meet the requirements found in heterogeneous environments. This model has been invented for multi database systems. It provides *function replication*, *semantic compensation* and *relaxation of the isolation property* of nested transactions [6, 19].

We need function replication to be able to substitute a failing service by another one that fulfils the same task in an equivalent way.

Relaxation of the isolation property is important if we assume that we coordinate local database systems. A subactivity at a local database system will require locks there and keep these locks until the global activity completes. The autonomy principle is heavily violated; in particular, long-living, and possibly ever-lasting activities are an unsolvable problem. Therefore, recent work on advanced transaction models for heterogeneous systems has dealt with allowing subtransactions to complete (in the database sense: commit) before the complete task is finished.

Semantic compensation is important if we allow the isolation property to be relaxed. In this case a completed action might not be needed (due to function replication) or be aborted (if finally the global task fails). A successfully completed transaction must not be rolled back, because other autonomous processes may already have seen its effects and may have based their computations on them. This concept can be seen to be in accordance with the rigourous single assignment property for communication objects. Thus, if a process later on decides that a communicated value is no longer valid, it may not simply over-write this value, but must *semantically compensate* it. This means that a user-defined compensate action (procedure, goal, function) may be specified, which is automatically activated and can write another communication object. In summary, only actions that are compensatable may complete early.

The necessary language extensions to support the Flex Transaction Model include constructs to express:

**Transactions** The start and completion (commitment) of a transaction in the programming language must be made visible. All communication objects are written upon commitment in an atomic step and made globally visible.

**write/test/read of communication objects** Write operations are collected until the transaction on the communication objects is committed. All the values of communication objects (instantiations/bindings) become visible in one atomic step. If a communication object cannot be written, e.g., because it has already been assigned a value by another transaction, the transaction commitment cannot be performed (the transaction fails). Test operations serve to examine the state of communication objects, i.e, whether they still are undefined (free, unbound) or defined (bound, instantiated, written). Read operations implicitly comprise synchronisation (compare with logic variables in concurrent logic languages [49]): if a communication object is read, which is still undefined, the read operation must wait until another parallel running process writes this object.

**Compensate actions** Transactions allow the run time selection of compensations, i.e., procedures that are to be activated, if the global transaction decides to abort and thus this completed transaction must be compensated (semantically undone).

**Prepare phases** are executed by the commit procedure of a transaction when it is sure that the assignment of communication object values will be possible, i.e., the transaction will succeed. In database terminology, the prepare phase may serve to signal other concurrent processes that this very transaction commits.

# Chapter 4

# A Survey of Languages and Tools

In this section I will summarise general purpose programming languages that offer coordination properties. The list of languages is by no means complete but I hope it will help to better understand the different approaches towards coordination.

The notion of a *coordination language* was coined first in [14], where a language is considered to consist of two disjoint parts: a computation part and a coordination part. The latter one can be added to the former one in an orthogonal way. The Linda model [13] is a coordination language in this sense. At almost the same time the term *mega programming* was coined by [54] with similar meaning. The task of a mega programming language is to coordinate preexisting, large software (mega) modules. It must initiate a computation at another site, pass parameters to the remote task, control it, and receive results in a reliable way. Furthermore, a mega programming language should provide partial failure recovery. Due to their autonomy, local systems may fail individually. The mega programming language should allow the substitution of failing services (this relates to function replication in the Flex Transaction model). As coordination of mega modules involves programming in the large, object-orientation is a must [53]. Not only is the modularization of data and names a necessity, but also the integration and modeling of autonomous data and schemes requires concepts like inheritance and polymorphism. Similarly, many object-oriented approaches can be found to integrate data semantically in multidatabase systems [10].

Many languages and systems exist that aim at one or more of these coordination properties. In particular, a large number of languages and systems that support distributed computing (i.e., communication, synchronization, and concurrency) have emerged over the last ten years [4]. [1, 15, 45, 51] describe parallel and concurrent systems that also aim at object-orientation.

In the following we will briefly classify languages that offer at least support for distributed computing. Such a list never can be complete, as there exist hundreds of such new languages [4]. For many of these languages the authors did not explicitly claim that their languages are coordination, mega programming, or work flow languages.

The design dimensions go from the invention of a completely new language to the creation of library functions that can be called from any language or software system. To extend an existing language by coordination properties stands somewhere in between. We may further differentiate languages by their paradigm, for example the imperative paradigm like C and Pascal, or logic, functional and object-oriented programming.

## 4.1 Languages and Language Extensions

**Argus** [44] is a fault-tolerant, object-based programming system that also can be considered an operating system. For recovery the programmer may provide a so-called recovery section that is automatically called by the system after a failure is encountered.

**Concurrent Eiffel** [45] extends the object-oriented programming language Eiffel by concurrency by introducing a few language constructs. As usual in concurrent and distributed object-based systems, objects communicate by invoking each other's methods.

**Concurrent logic programming languages** [49] change Prolog's semantics to support explicit parallelism. Representatives of concurrent logic languages are Concurrent Prolog, Guarded Horn Clauses, and Parlog, which use shared logic variables for communication. However, these languages do not transgress the borders of a system and cannot deal with failures, which makes them unsuitable for heterogeneous environments. The approaches to exploit parallelism in Prolog implicitly are not of interest for coordination languages that need explicit language constructs to reason about the data and control flow of activities.

**Orca** [3, 5] is a new object-based programming language that uses the shared data communication paradigm. Orca provides atomic actions on distributed objects. There exist very detailed performance analyses of Orca implementations in the Internet, on a multi-processor architecture and on the distributed Amoeba operating system, demonstrating that the shared data paradigm is not necessarily less efficient than message passing.

&Co**-languages** [21, 33, 38] are based on the coordination kernel. C&Co and $^{\mathrm{V}}\mathrm{P}_{\mathrm{L}}$ support multiple inheritance, encapsulation of data and behavior, and different forms of polymorphism ([11, 35]).

The coordination kernel integrates particularly well with Prolog, resulting in $^{\mathrm{V}}\mathrm{P}_{\mathrm{L}}$, because Prolog variables already offer the single assignment property. Like in C&Co, accessing communication variables is through the use of (unification) operators. Concurrency fits well to the logic programming model and can be realized by sequential/parallel AND/OR operators. Transactions are determined by the commit operator "|". The nesting of transactions corresponds to the possibility of nesting goal calls. Testing execution states is completely implicit: this is reflected by the success or failure of goals and causes either forward or backward execution. The process primitive for inter-process concurrency is supported in an analogous way as in C&Co.

## 4.2 Libraries

**CORBA** [16] supports the administration of distributed objects. Object data and methods must be described in the IDL (Interface Definition Language) that aims towards object-orientation, similar to the C++ language.

**PVM** [23] is a library that provides functions to receive and send messages and to spawn remote processes by taking care of the correct data conversions between different architectures. PVM's functions are more convenient to program with than sockets, for example.

**ISIS** [7] is a library that supports reliable broadcast utilities, and group based communication mechanisms based on RPC.

In summary, CORBA, PVM, ISIS, and operating system facilities [50, 52] (like remote procedure call (RPC), sockets, and threads) are too low level to be used as coordination languages by themselves. They can serve as the implementation tools for other work flow languages but do not exhibit satisfying linguistic support for coordination. These libraries introduce a type of thinking in the term of message passing which is non-problem related and does not provide a sufficient abstraction for the programmer. In contrast, the shared data paradigm offers—at the same level of expressiveness as the message passing paradigm—a much better conceptual model. The programmer sees a globally shared object space that serves for communication and synchronization purposes. All other mentioned tools support the shared data model.

**MeldC** [29, 30] aims at a uniform design so that the reflection about MeldC programs (i.e., the meta-reasoning about programs) can be supported. MeldC is implemented as a runtime object library and supports different kinds of message passing as well as different kinds of threads. It is dynamically expansible by means of dynamic composition, using shadow objects.

**Arjuna** [17, 51] provides transactions on distributed objects. Arjuna is an object-oriented programming system that itself is organized by means of classes. Replication techniques are used to provide fault-tolerance. Arjuna is a C++ library: its class definitions can be inherited from C++ programs. There are classes for concurrency control and the management of object locks, etc. Arjuna objects are persistent and can be recovered after system failures.

**Linda** [13, 14] provides blackboard communication and follows the shared data paradigm. More precisely, Linda is a new communication model that should be termed shared data structure paradigm. The basic idea of Linda is a globally shared *tuple space* into which concurrent processes can insert tuples, and from which they may read or delete tuples. Reading and deleting are blocking primitives: if no suitable (matching) tuple can be found, implicit synchronization takes place, analogous to the implicit waiting in C&Co.

## 4.3 The Origin of C&Co

This section describes the origin of the coordination language C&Co. It all began with the language VPL.

### 4.3.1 V*ienna* P*arallel* L*ogic*

VPL (Vienna Parallel Logic) [39, 37] has been defined at the department of computer languages. The language integrates computation and coordination. VPL is a superset of Prolog and has originally been designed as a query and transaction specification and execution language for multidatabase systems [41, 42]. It bases on the Flex transaction model ([18] and Section 3.3) and provides reliable communication, function replication, and semantic compensation.

VPL provides a clear distinction between ordinary logic variables and communication variables. Bindings of the former ones are undone on backtracking making them unsuitable for communication. The latter ones provide a rigourous single assignment property which means that once they have been bound by a transaction the binding must remain forever. A protocol has been described in [40] that implements communication variables reliably on different hardware architectures. Only communication variables can be shared between concurrent VPL processes. On backtracking over a transaction, the bindings of communication variables are not undone but instead a compensate action is called that semantically undoes the effects of the transaction as a whole. Thus, VPL unifies the two worlds of backtracking and communication. VPL also provides explicit language constructs for concurrency control as well as neutral control operators to achieve a maximum of hardware independence. These fit well into the logic programming framework as they are sequential, parallel, and neutral AND/OR operators. Explicit processes on other computer sites can be created by a process primitive that resembles Parlog's control meta call. Independently, whether intra or inter process communication is required, shared communication variables can be used as one uniform communication mechanism.

### 4.3.2 The Linda Model

Linda is one of the models which had the greatest influence on the development of C&Co or rather the idea to extend existing languages by a coordination paradigm.

Orthogonal language constructs like the Linda tuple space [13, 25] can be added to existing programming languages without changing or destroying their semantics. Linda has been grafted on languages such as C, Prolog, Fortran, and Lisp. The host language is responsible for computation aspects of the program and the coordination language for communication aspects.

This linguistically simple model is expressive and allows the elegant specification of most classes of concurrent problems. The limitations of Linda are: its original design did not consider transactions, i.e., it is not possible to perform several Linda operations in one atomic step, and tuples have global names thus introducing a name space problem. Security might become a problem, because any process that connects to the tuple space may "guess" the name of a tuple which may result in unauthorized access. Moreover, garbage collection is impossible, because there is no point in time when the Linda system can decide that a tuple will no longer be used. Fault-tolerance has not been considered, thus a system or network failure might compromise the tuple space.

Many attempts have been made to overcome these limitations [2, 24]. However, as more functionality is added to a sound model, its design becomes less pure.

### 4.3.3 The Coordination Kernel

CoKe is a successor of VPL in that sense, that it extracted the coordination features of VPL to make them available for other languages.

The Coordination Kernel [12, 34] offers the shared data paradigm by means of communication objects. It provides advanced transactions and thus reliability and software fault-tolerance. Inter-process concurrency is provided by the `process` primitive which can be seen as a reliable software contract. Efficient implementation is assisted by the read-once property of communication objects. Garbage collection is possible, because the kernel can determine, if a communication object is no longer referenced. The coordination kernel offers a set of library functions that can be called from any software system. Among them are the starting, aborting, and committing of transactions, the creation, reading, writing, and testing of communication objects, and the spawning of processes.

These functions can either be called directly from a language, or be integrated more smoothly into the host language, without violating the original language's programming style and by adding only a few language constructs. Influenced by multidatabase systems, it follows the principles of respecting the local system's autonomy which includes not to change or destroy a local system.

A communication variable in a programming language which does not provide object-orientation becomes a typed communication object of the coordination kernel; otherwise the declaration of an object as a communication variable makes the object a communication object which can then be used interchangeably in the host language and by the coordination kernel.

# Part II

# C&Co Manual

# Chapter 5

# Overview of C&Co

The contents of the chapters in this part are closely knotted together. They describe the language extensions to C and their applications which make the language C&Co.

In order for concurrently running processes to communicate with each other, one has to create communication objects, start the processes, write the communication objects within transactions, and perform operations to achieve synchronisation.

## 5.1 Keywords of C&Co

The syntax of C&Co is closely related to ANSI-C [47, 32]. The intention of the design of C&Cowas to keep the number of additional keywords minimal. One of the greatest concerns was to integrate these extensions as smoothly as possible into the language. One shall not have to learn many functions and shall not change his programming style.

The keywords of C&Co are given in Table 5.1 and are described in the following chapters. A few other novelties are important to mention:

| | |
|---|---|
| `comm` | Declaration of a communication object |
| `new` | Generate a local variable |
| `( ).` | Interface of a function |
| `process` | Start a new process |
| `signal` | Send a signal to a process |
| `trans` | Start of a transaction |
| `commit` | End of a transaction |
| `abort` | Abort a transaction |
| `undefined` | Test a communication object |
| `defined` | Test a communication object |
| `//` | Comment until newline |

Table 5.1: New Keywords of C&Co

- The use of function prototypes is strictly enforced. The full types of a function must be declared before the function can be used.
- Every function can be declared with an *interface*. See below for an explanation.

## 5.2 The Interface

The interface of a function or consists of the open parenthesis, the *interface definition list*, the close parenthesis and the period before the function name.

$$\underbrace{\texttt{(}\ \textit{interface-list}\ \texttt{)}.}_{\text{interface}}\textit{function}\texttt{(}\ \textit{parameter-list}\ )$$

In the current implementation of C&Co only communication objects are allowed in *interface-list*, separated by semicolons. In later implementations, we want to use the interface of a function for true object orientation. Every parameter in the interface can be omitted. If a parameter is left empty, the function substitutes a default value for it. Different parameter types might trigger the call of different functions.

An essential feature of communication objects in *interface-list* is that they are passed by reference (comparable with variable parameters in Modula-2) in contrast to the variables in *parameter-list* which are passed by value, as is the usual mechanism in C. This is also true for communication objects which are used as parameters in a function call. They "lose" their communication ability in the process.

The idea behind this is to move true reference parameters—which also serve to return variables of arbitrary types—from the parameter list to the interface list. Thus, references to data objects and value parameters are separated clearly.

A function still has its standard return value, if it is declared with a type other than `void`. A transaction (see Chapter 8) cannot have a standard return value.

Note: In the current implementation of C&Co only certain parameters from the interfaces of the builtin functions from Table 5.1 can be omitted.

# Chapter 6

# Creation of Communication Objects in C&Co

In order to communicate between processes running in parallel, the programmer has to define communication objects which are communicated between those processes. The next sections detail the necessary language extensions.

## 6.1 Static Creation of Communication Objects

Communication objects that can be shared between several processes are defined with the keyword `comm`. It can prefix any common C data type declaration, causing the creation of a communication object of the specified type. The keyword is used similarly to the *storage class* keywords, i.e., `auto`, `register`, `static`, `extern`, and `typedef`. On the other hand, it is possible to use `comm` within structures in contrast to, e.g., `extern` [47].

**Example 6.1.1** (*Static Creation of Communication Objects*)

```
comm int iobj;                          /* creation of integer object */
comm struct date                     /* creation of a structured object */
     {
     int dd, mm, yy;
     comm long time;
     } date_object;
struct node {
      char key[20];
      comm struct node *left, *right;
      };
comm struct node distributed_tree;
```

Example 6.1.1 shows some utilizations of `comm`. The variables `iobj`, and the structures `date_object` and `distributed_tree` are defined as communication objects.

It is possible to declare members of structures explicitly as communication objects. The reason for this lies in the implementation (see Section 10 and

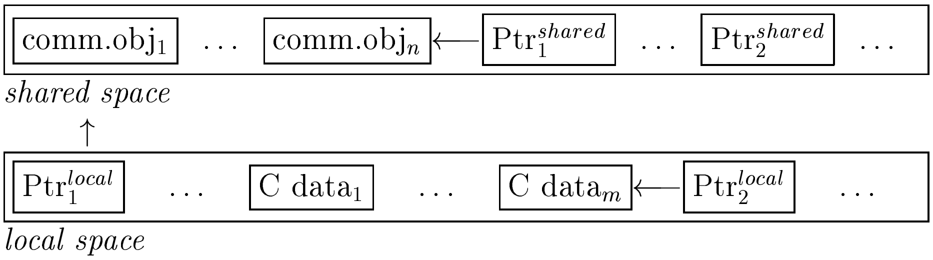


Figure 6.1: Memory Spaces and Referencing

[34, 35]) of communication objects. If two or more concurrent processes must work on the same structured variable in parallel, it makes perfectly sense to declare the members seperately with `comm`. The coordination kernel is then able to communicate the so declared members independently of each other to different processes which raises the availability and shortens the access times. The variable `date_object` is such an example. One process can work on the members `dd`, `mm`, and `yy` while another process—running at the same time at another site—is able to work on `time`.

A second case where it makes sense to declare members as separate communication objects is the declaration of the pointers `left` and `right` in the structure `struct node`. Independent processes can work on different parts of `distributed_tree` because each node of the tree can be communicated separately. Otherwise, a program, that traverses the tree from the root to the leaves, might slow down other ones because the entire structure is communicated automatically instead of just the pointers it follows.

`comm` automatically converts all members of a complex data type to communication objects. The advantage of this approach is that all C types remain, but some of them can be defined to be communication objects which causes their creation in a globally shared space.

## 6.2 Pointers

No differentiation for pointers is necessary whether they point to local data or to communication objects except that it is impossible to let a communication object point to local data. The address in physical memory of one machine cannot be communicated to another machine. The relation between pointers is shown in Figure 6.1

A *local pointer*, i.e., ordinary C pointer located in the local space, may point to any kind of data, objects, and pointers located at either local or shared space. A shared pointer must not point into the local space. It can only point to another shared pointer, to a communication object, or be undefined.

Every communication object that was declared as a pointer, e.g. the shared pointer $\mathrm{Ptr}_1^{shared}$, and every local pointer, e.g. $\mathrm{Ptr}_1^{local}$ can point to a another communication object.

## 6.3 Dynamic Creation of Communication Objects

The operator `new` makes it possible to declare not only a communication object but any C variable anywhere—where expressions are allowed—in the program. The type and name of the variable is given in the interface of `new`. One can also initialise this newly created variable at the same time. Example 6.3.1 shows some applications of this operator.

**Example 6.3.1** (*Dynamic Creation of Communication Objects*)

```
int foo( int bar, int gnu )
{
int a[10];

     for( (int i).new=0 ; i<10 ; i++ )
          {
          (int m).new = bar * i;
          a[i] = calc( m, gnu );
          }
     (comm int c).new = sum( a );
}
```

Note that the newly defined variables have the same scoping rules as local variables in C, i.e., they are visible only in the *compound statement* were they have been declared. The program of Example 6.3.1 is semantically identical to Program 6.3.2. See Section 12.1 for implementation details.

**Example 6.3.2** (*Equivalent Program*)

```
int foo( int bar, int gnu )
{
     comm int c;
     int a[10];

     {
     int i;
     for( i=0 ; i<10 ; i++ )
          {
          int m;
          m = bar * i;
          a[i] = calc( m, gnu );
          }
     }
     c = sum( a );
}
```

# Chapter 7

# Concurrent Processes in C&Co

A concurrent language must certainly provide means to express parallelism. The programmer must be able to spawn processes on the same site or on a remote site. Signals can be sent to already running processes to alter their execution behaviour.

## 7.1 Spawning New Processes

A new process is created with the function

( *where* ; *func* ; *pid* ; *type* )`.process`

It calls the function *func*, executed by the C&Co system at the site *where*. Every function can be used as an entry point for external processes. The process runs as a separate thread and may communicate with its caller only via communication variables that are passed through the interface of *func* and therefore cannot interfere with other programs running at that C&Co site. Thus, the `main()` function of C programs becomes superfluous.

The `process` construct does not wait and the program continues with the next statement. The parameters in the interface of `process` have the following meaning:

*where* is a string with the name of the software system and its Internet address (e.g. `"cco@some.site"`) and—depending on the architecture—the processor where the C&Co process is to execute. If *where* is `"LOCAL"` the process is started as another process at the local C&Co system, if *where* is `"REMOTE"`, the process is sent to any other C&Co system[1] (depending on the work load of the system).

*func* specifies the C&Co function or transaction (see Section 8.1) to be executed and its arguments. It is an expression of the form:

( *interface-list* )`.`*funcname*`(` *parameter-list* `)`

[1] This is currently not supported by the CoKe.

Communication variables which are specified in *interface-list* are shared between the concurrent processes.

*pid* is a communication variable that identifies the new process. It is automatically assigned the value `"SUCCEEDED"` after the completion of the process, if the called function was not a transaction. If `process` was used to call a transaction the assigned values are `"SUCCEEDED"` after a successful execution of the transaction or `"ABORTED"` otherwise.

*type* is either `DEP` (dependent) or `INDEP` (independent). The caller of independent processes does not rely on their termination state. An unfinished independent process is automatically restarted by the CoKe after a system failure. The kernel starts the same process again with its previous image. This can be considered as a software contract between the C&Co-language and the coordination kernel. An issued `INDEP` process will be eventually executed. Dependent processes must be started manually again.

**Example 7.1.1** (*Process Synchronisation*)

```
comm int a, b;

( "cco@other.site" ; (a).f1() ; (comm char *pid1).new ; DEP ).process;
( "LOCAL" ; (b).f2() ; (comm char *pid2).new ; DEP ).process;

/* Synchronisation */
if( !strcmp(pid1, "SUCCEEDED") && !strcmp(pid2, "ABORTED") )
     {
     // Code, if the process executing "(a).f1()" at another site has
     // successfully terminated and the process executing "(b).f2()" at the
     // local site has unsuccessfully terminated.
     }
```

Example 7.1.1 shows a very simple application of `process`. The program spawns two transactions—the first one at another site, the second one at the same site—and waits for the first transaction to succeed and the second one to fail. The code of the `if` statement is executed only when these two conditions are met.

The `process` primitive is more powerful than the `fork/exec` mechanism of UNIX for the following reasons:

- The other process can be executed also on another site.
- It may spawn a light weight process depending on the architecture of the target machine without the use of another function call, e.g., to a thread-library.
- It supports reliable process communication via shared objects.

- It provides symmetric communication without a father-son relationship. Communication objects can be shared beween an arbitrary large number of processes.
- It illustrates the data flow between the concerned processes. Every process can only exchange information via the communication objects that are given to it.
- Arbitrary data structures—not only streams—can be exchanged between different hardware platforms and programming paradigms.

## 7.2 Signals

The function

( *pid* ; *signal* ).signal

sends signals to the specified process. The library functions of the C&Co interface to the current version of the CoKe does not support signals to processes on the local system. Nevertheless, the C&Co compiler understands the keyword. The possible signals are:

`KILL` terminates the specified process and aborts running transactions. Transactions which were started by the process and already comitted are compensated (if possible).

`SLEEP` suspends the execution of the specified process. This might be useful for everlasting processes which have to be interrupted for a certain time period.

`WAKE-UP` continues with the execution of the suspended process.

`ABORT`, `SUSPEND`, `RESUME` are similar to `KILL`, `SLEEP`, and `WAKE-UP` respectively but they affect also all subtasks of the specified process [36].

# Chapter 8

# Transactions and Commitment in C&Co

Every communication object must be written within a transaction. That guarantees that only one process is able to write a value into a communication object at one time. It is also possible to perform more than one operation within a transaction, e.g. to write many communication objects.

## 8.1 Transactions and Commitment

Transactions can be defined in C&Co with the keyword `trans` in two possible ways.

1. A function can be statically defined as a transaction with

   `trans (` *interface-list* `).`*function-name*`(` *parameter-list* `)`
   `{`
   *statements*
   `} (` *retry-condition*`;` *prepare-phase*`;` *compensate-action* `).commit`

   The function has no return value, i.e., it is of type `void`.

2. Statements within functions can be grouped together with

   `trans {`
   *statements*
   `} (` *retry-condition*`;` *prepare-phase*`;` *compensate-action* `).commit`

`commit` corresponds to the end of the transaction and implicitly performs the *commit procedure*. All communication variables are written in one atomic step. The commit procedure waits for all dependent processes (with type `DEP`) to complete.

All arguments of `commit` are optional—comparable to the syntax of `for(;;)`. The interface of `commit` can be omitted entirely if the *interface-list* is empty.

*retry-condition* specifies if the transaction will be retried if it was not successful. More precisely, if the transaction cannot commit, the boolean expression *retry-condition* is evaluated. If it evaluates to `TRUE` (non-zero), then the transaction is retried. If *retry-condition* is omitted, `FALSE` is assumed as the default. A retry condition of `TRUE` means repetition until the transaction succeeds. To retry a transaction is reasonable because the environment might have changed in the meantime.

*prepare-phase* is a function or a process, that is called when it is certain that the transaction can commit. See Section 8.3 for a detailed description.

*compensate-action* specifies a function which is called *after* the transaction has committed but must be compensated. Only the compensate action of the outermost transaction is called. It is responsible to semantically undo the effects of the transaction and of all subtransactions. Thus, compensation is not a cascading operation. Note that if the signal `KILL` is sent to a process, that has already finished and did execute a transaction, then its compensate action (if existent) is started. In all other cases all subprocesses and subtransactions of the process are aborted.

All statements between "{" and "}" belong to the scope of the transaction. Global variables and pointer references to local memory are meaningless for functions called via `process`. Only the declarations of types and functions, the values of the variables in *parameter-list*, the communication variables in *interface-list*, and the locally defined ordinary and communication variables are in scope.

It is not possible to access the variable `globalvar` in line 54 of Example 8.1.1 and the pointer `p` in line 55 points to an undefined memory location because the transaction was started as a separate thread via `process` in line 8. If you call `alloc_node` without `process` the variables `globalvar` and `*p` are accessible. The subtransaction (lines 56–60) sees all objects which are visible in its parent transaction.

**Example 8.1.1** (*Transaction Scope*)

```
1  int globalvar;
2  comm struct node *root;
3  void some_function( void )
4  {
5       comm int pid;
6       int *ptr;
7       ptr = &globalvar;   // Pointer to local memory
8       ( "LOCAL" ; (root).alloc_node(globalvar, ptr) ; pid ; "DEP" ).process;
9  }
   ⋮
50 trans ( comm struct node *tree ).alloc_node( int var, int *p )
51 {
```

```
    int localvar;
    localvar = var;          // okay
    localvar = globalvar;    // ERROR
    localvar = *p;           // ERROR
    trans                    // subtransaction
        {
        (comm struct node nd).new;
        tree = &nd;
        tree->key = var;
        } commit;
} commit
```

The values of communication variables, that are not yet committed and are visible within the scope of the transaction, can be over-written there, because the variables are treated as local objects. Thus, the rigorous single assignment property only applies at the end of the transaction. The value which has been assigned to the communication variable in the scope of a transaction is exported upon commitment and can never be changed again.

## 8.2 abort

Transactions are aborted by calling the primitive `abort` which corresponds to a break from a block. This abort causes the compensate actions of all committed subtransactions, but not of sub-subtransactions to be executed. If a transaction fails, i.e., it cannot commit, and the *retry-condition* evaluates to `false`, execution continues immediately after the transaction.

**Example 8.2.1** (*Insert Node into Tree*)

```
struct node *nptr; nptr = &distributed_tree;
trans {
    if( equal( k, nop->key ) )
        abort;                              // key found; exit transaction

    if( greater(k, nop->key) )
        { /* search right */
        if( (nop->right).undefined )
            { (struct node nd).new;
            nop = nop->right = &nd;
            ( np->key ).insert_key( "mno" );     // try to insert key here
            }
        else if( (nop = nop->right) == NULL)
            abort;              // end of tree structure, exit transaction
        }
    else
        {
        /* search left */
        ⋮
```

```
        }
    } commit;
```

The program in Example 8.2.1 shows an application of `abort`. The key "mno" shall be inserted into a distributed tree. If the key is found or the end of the tree is reached, the transaction is aborted. The function `undefined` is explained later in Section 9.3.

## 8.3 Prepare Phase

The *prepare phase* of a transaction is an expression that is executed by the commit procedure when every `undefined` test succeeds (see Section 9.3) and it is sure that the assignment of values to communication objects will be possible, i.e., the transaction will succeed. In database terminology, the prepared phase may serve to signal other concurrent processes that this very transaction commits. If the execution of the prepared phase of transaction $\mathcal{A}$ activates another transaction $\mathcal{B}$, that fails to commit, this has no harm on the success of transaction $\mathcal{A}$.

If a function *f()* is specified as the prepared phase, the actual parameters of *f()* are those given by the definition with the actual values when the end of the transaction is reached (see Example 8.3.1). Please note, that `i` and `j` are passed by value to `pfunc`, i.e., the prepare phase is called with the same values, that will be assigned to the communication objects at the point of commit.

**Example 8.3.1** (*Prepare Phase*)

```
comm int i, j;

void pfunc(int a, int b)
{
    /* some code */
}

void foo(void)
{
    trans {
        i = 4;
        j = 5;
        } (; pfunc(i,j) ;).commit;
    // pfunc(4,5) is called as prepare phase
}
```

For example, in the prepared phase an explicit process on another computer site can be informed about this very transaction's end.

## 8.4 goto

Jumps with `goto` statements have the same meaning as in standard C, except that it is not possible to jump into or out of a transaction. This would overload the behaviour of `goto` because it is not clear if you want to commit or abort when you jump out of a transaction.

To specify a certain behaviour you must use `abort` in conjunction with flags (see Example 8.4.1).

**Example 8.4.1** (*Early Exit Without* `goto`)

```
int jumpflag = 0;

trans {
     ⋮
     if( disaster )
          {
          jumpflag = 1;
          abort;                       /* exit transaction, no commitment */
          }
     } commit;
if( jumpflag ) goto out;
/* code, if transaction was successful */
⋮
out:  /* exit-label */
```

Note that a transaction defines a new lexical scope—like a normal block does in standard C—including a new scope for labels so it is of course possible to use `goto` statements within a transaction.

# Chapter 9

# Synchronisation in C&Co

Synchronisation is done implicitly by the use of communication objects in expressions. In short, a process has to wait if it wants to read an undefined communication object.

## 9.1 Extended Semantics of Assignment Operator

The semantics of `x = y` must be extended for communication objects (see Table 9.1). The `=` operator can be interpreted as "read the value of `y` and then write this value into `x`". Synchronisation can intervene by waiting until all objects that appear on the right side of `=` are defined.

The notion "$x \Leftarrow y$" is used in Table 9.1 to explain the semantics of "=". It sets x to the value of y locally, i.e., within the scope of the current transaction. If type compatibility is violated, an error is raised. Locally, communication objects can be re-assigned values as long as they are not committed by the actual transaction, i.e., they are undefined. The current, local assignments of all variables that have been made within the transaction become globally visible at the end of the transaction, provided they have not received any value yet by another transaction. If x has already been committed, even if y equals the value of x, the assignment fails, due to the rigorous single assignment property of communication objects. In this case, the programmer must use `==`.

**Example 9.1.1** (*Writing of Communication Objects*)

```
trans {
     date_object.dd = 25;
     date_object.mm = 9;
     date_object.yy = 1995;

     ( comm struct node nd ).new;
     strcpy( nd.key, "abc");
     distributed_tree->left = &nd;
     } commit;
```

## 9.2 Extended Semantics of Other Operators

If an undefined communication object is involved in an expression, using logic, arithmetic, relational, or bit operators, where it is set into relation with other C&Co variables or types, then the system waits until a value is written into the communication object. The operators of Table 9.2 can be savely used with communication objects.

Table 9.3 shows the operators which should not be used in conjunction with communication objects. The reason for this lies in the implementation of communication objects (see Section 12.1). In short, every communication object in an expression is read before the *statement* which includes the expression. If you use one of the operators of Table 9.3 it might be the case, that the expression which includes a communication variable is not executed. Nevertheless, the program waits for the communication object which might lead to wrong behaviour.

**Example 9.2.1** (*Unsave Situation*)

```
if( a == 3 && b > cobj )
     {
     ⋮
     }
```

Take a look at the `if` statement in Example 9.2.1. A normal C compiler might skip the second part of the logical expression if the variable `a` is not equal to 3. The C&Co compiler checks the communication variable `cobj` *before* the `if` statement. This might lead to an unwanted synchronisation. The programmer is advised to change the structure of the program to make clear the desired behaviour.

Table 9.4 shows the operators which cannot be used in connection with communication objects because they imply, that the variable in question has already a value. The *write once* semantic of communication objects doesn't allow that.

## 9.3 `undefined` Test

The function

( *comm-obj* ).`undefined`

tests whether a communication object is still without a committed value. If the value of *comm-obj* is already defined, `undefined` evaluates to `FALSE` immediately.

`undefined` is valid only in the scope of a transaction and checks whether the communication object is still undefined also at the point of commitment of the transaction, i.e., it is tested twice:

1. immediately in the expression where it appears, and

| x | y | semantics of "x = y;" |
|---|---|---|
| non-comm | non-comm | same as in C |
| non-comm | comm | = waits, until the value of y becomes defined |
| comm | non-comm | $x \Leftarrow y$ |
| comm | comm | = waits, until the value of y becomes defined |

Table 9.1: Semantics of the Assignment Operator (=)

| | | | |
|---|---|---|---|
| + | Addition | < | Less Than |
| - | Subtraction | > | Greater Than |
| * | Multiplication | <= | Less or Equal |
| / | Division | >= | Greater or Equal |
| % | Modulus | == | Equal |
| & | Bitwise AND | != | Not Equal |
| \| | Bitwise OR | << | Shift Left |
| ^ | Bitwise XOR | >> | Shift Right |
| ~ | One's Complement | ! | Logical Negation |

Table 9.2: Save Operators for Communication Objects

| | |
|---|---|
| && | Logical AND |
| \|\| | Logical OR |
| ?: | Conditional Expression |

Table 9.3: Unsave Operators for Communication Objects

| | | | |
|---|---|---|---|
| ++ | Pre/Post Increment | %= | Modulus Assignment |
| -- | Pre/Post Decrement | &= | AND Assignment |
| += | Plus Assignment | \|= | OR Assignment |
| -= | Minus Assignment | ^= | XOR Assignment |
| *= | Multiplication Assignment | <<= | Shift LeftAssignment |
| /= | Division Assignment | >>= | Shift Right Assignment |

Table 9.4: Forbidden Operators for Communication Objects

2. at the point of commitment.

Note that a second `undefined` test could return a different result because another process might have written a value to the communication object.

**Example 9.3.1** (*Test Whether a Communication Object is Undefined*)

```
trans
     {
     if( undefined( nop->right ) )
          { ( comm struct node nd ).new;
          nop->right = &nd;
          }
     /* guarantees that nobody else writes something into nop->right */
     } commit;
```

The transaction in Example 9.3.1 assures that no other process writes something into the pointer `nop->right`.

Note: `undefined` does not immediately set locks on the object, because this would lead to deadlocks.

## 9.4 `defined` Test

The primitive

`( ` *comm-obj* ` ).defined`

is similar to `undefined` but not equal to `!(` *comm-obj* `).undefined`. It tests whether its argument is defined at the place where it is used and immediately returns `FALSE` or `TRUE` (zero or not-zero).

`defined` can be used anywhere, i.e., not only within a transaction.

# Part III

# Implementation

# Chapter 10

# Implementation of C&Co

The sections in this chapter give a short introduction to the tools and steps which were used to implement the C&Co compiler. Chapter 11 describes the important functions and data structures which are necessary to parse C&Co source and to handle attributes. Chapter 12 describes the idea and process of code generation.

C&Co follows closely the ANSI-C Standard [47, 32]. Nevertheless, there are a few differences from C in the current implementation of C&Co besides the language extensions of Chapter 5.

- Comments starting with `//` until newline are permitted, similar to C++.
- The type qualifiers `const` and `volatile` are ignored. We are currently thinking of the semantic properties of these qualifiers for communication objects.
- `typedef`s and `enum`s are not recognised by the parser and thus cannot be used.
- Macros and `define`s are not recognised by the parser. Use the C preprocessor to expand the symbolic names.
- Function prototypes are strictly enforced. Thus, an old-style function [31, 32] like

  ```
  foobar( a, b, c )
  int a, b, b;
  {
  ⋮
  }
  ```

  is not allowed in C&Co.

## 10.1 First Steps

C&Co is implemented with a combination of C and Ox [8] which itself generates code for Flex and Bison, the GNU equivalence of Lex [43] and Yacc [28]. I had to use Flex and Bison because I was unable to compile the quite complicated grammar of the language with Lex and Yacc. The reason why I started to write the compiler from scratch is, that I have to output C source instead of machine code. Additionally, most of the C compiler which are in the public domain have either grammars with many conflicts or no grammar at all. But for my work it is essential to have a good C grammar which can be extended easily.

I started with the well known Roskind Grammar [46]. This grammar accepts ANSI-C code and is conflict free[1]. At first, I tried to build a program which reads C source, analyses the code, and writes syntactically equivalent source to another file.

The next step was to enhance the grammar to accept C&Co source. The grammar is still conflict free. The first problems showed up when I tried to "do something" with the parsed code, e.g., use the symbols from the symbol table at other places in the grammer, recognise types, or generate code at places "above" or "below" the current token.

I found out, that it was not possible to solve the emerging problems with Lex/Yacc alone. At this point, a colleague at the institute, Anton Ertl, pointed me in the direction of a tool named "Ox" which manages attributed grammars.

The next move was to extend the grammar by attributes. This is described in detail in Chapter 11. One of the greatest challenges was to use as much "attributed grammar" as possible. Lex and Yacc were against it.

Lex must look ahead one token to determine the grammar rule to use. This is somehow unfortunate in situations like this:

```
struct name a;
struct name { ... };
```

The parser gets the token `STRUCT` from the scanner and knows that the following identifier (if any) must be inserted in the symbol table as a "struct identifier". Therefore, a global variable is set to indicate a different insert mode for the symbol table functions. The scanner reads the token `IDENTIFIER` with its value "name" and inserts/looks up the symbol table entry for a `structdef` (see Section 11.3.2).

But, in order to determine the right grammar rule, the parser has to read the *next* token too. It can either be a '{' or another identifier, which is disastrous because it is then inserted into the wrong symbol table.

The only possible way to solve this dilemma was to change the type of the symbol in the symbol table at a later time. Right now, when the scanner reads a string of characters, it has to determine if this string is a valid keyword, an identifier, or a `typedef` name. The scanner searches for the name in the symbol

[1] The well known shift/reduce conflict of the `if-else` ambiguity is the only one.

table and inserts the name if it finds no entry. Another possibility would have been to move the insertion of symbols to other points of the grammar and thus let the parser insert symbol names but this would have changed the grammer at delicate places. A mixed solution is probably the best one: The Scanner searches for symbol table entries and the parser inserts new ones at the appropriate places.

## 10.2 What is Ox?

Ox [8] is an attribute-grammar compiling system based on Yacc [28], Lex [43], and C. The name "Ox" comes from an attempt to pronounce an acronym for "An Attribute-Grammar Compiling System". An ordinary Yacc and Lex specification can be expanded with definitions of synthesised and inherited attributes written in C syntax. From these specifications, Ox generates an evaluator written in Yacc, Lex, and C. For a given input, the evaluator builds a parse tree, determines an order of evaluation for attributes of the tree, and performs, for each attribute, the semantic action required to evaluate it.

Additionally, the Ox user can easily specify parse-tree traversals that are performed after evaluation of the tree's attributes and that refer to those attributes.

Ox is a preprocessor that translates into files suitable to Lex and Yacc. An Ox input specification consists of at least two files: a syntactic specification (which resembles a Yacc input specification) that Ox translates into a Yacc input specification, and one or more lexical specifications (which resemble Lex input specifications) that Ox translates into Lex input specifications.

## 10.3 Attribute Declarations and References

An attribute declaration consists of the reserved word `@attributes` followed by '{', an `attribute` declaration list, '}', and a list of grammar symbols.

For example, the token `IDENTIFIER` has the following attribute declaration:

```
@attributes { SYMTAB *p_sym; char *output; } IDENTIFIER
```

Then the Ox-generated evaluator allocates storage for a pointer to a symbol table entry named `p_sym` and a pointer to the output string, when building a parse-tree node labeled `IDENTIFIER`.

Each Yacc/Ox grammar rule (production) is viewed as a sequence of grammar symbols. The object of each `return` statement in a Lex action is a sequence consisting of a single grammar symbol. The leftmost symbol of a rule is called the *left-hand side (LHS)*. The *right-hand side (RHS)* comprises the rule's other symbols. A symbol's position in a rule together with an attribute of that symbol constitutes an *attribute occurrence* in that rule.

Attribute occurrences are named by a grammar symbol, followed by a period, followed optionally by a non-negative decimal integer and another period, followed by the name of an attribute of that symbol. The integer and the second period are needed only when a given grammar symbol appears more than once in the rule. In that case, those distinct appearances are numbered from left to right with consecutive increasing integers starting with 0 (see Program 10.3.1, lines 4 and 5).

**Program 10.3.1** (*References to Attributes*)

```
1 paren.identifier.declarator:
2      IDENTIFIER
3      | '(' paren.identifier.declarator ')'
4           @{ @i @paren.identifier.declarator.0.output@ = strbuild( "010",
5                "(", @paren.identifier.declarator.1.output@, ")" );
6           @}
7      ;
```

The attribute occurrence output of the leftmost appearance of `paren.identifier.declarator` is denoted in Ox code as paren.identifier.declarator.0.output, while the attribute occurrence output of the rightmost appearance of `paren.identifier.declarator` is denoted paren.identifier.declarator.1.output.

### 10.3.1 Definition Modes in the Yacc-File

For each grammar rule, the programmer may provide an *attribute reference section*, delimited by `@{` and `@}`, and optionally containing *attribute definitions* of attribute occurrences of the given rule. Conceptually, an attribute definition has a *dependency part* and an *evaluation part*. There are three modes of expression of attribute definitions. Different modes may also be used within a single attribute reference section. Each attribute definition begins with a *definition mode annunciator* (`@e` *explicit mode*, `@i` *implicit mode*, or `@m` *mixed mode*) and is terminated by another mode annunciator or by `@}`.

**Program 10.3.2** (*Definition Modes*)

```
1 declaring.list:
2      declaration.specifier declarator initializer.opt
3           @{ @i @declaring.list.output@ = strbuild( "111",
4                @declaration.specifier.output@, @declarator.output@,
5                @initializer.opt.output@ );
6           @e declaring.list.sp:  declaration.specifier.sp declarator.p_sym;
7                @declaring.list.sp@ = @declaration.specifier.sp@;
8                add_specifier(@declaration.specifier.sp@,@declarator.p_sym@);
9           @}
```

Only the explicit and implicit modes are used in Program 10.3.2. The dependency expression in line 6 makes explicit the constraints on the order of execution of the evaluation expressions in lines 7–8. The occurrences to the non-empty list to the left of the colon are said to depend upon the possibly empty list to the right. Thus, the attribute sp of declaring.list depends upon the attribute sp of declaration.specifier and the symbol table entry p_sym of declarator.

An implicit mode is used in line 3 of Program 10.3.2. The mode annunciator @i informs Ox that the definition has a single dependent, namely the first attribute occurrence referenced in the evaluation expression, i.e., declaring.list.output. The dependees in the definition consist of all other attribute occurrences referenced in the evaluation expression, i.e., declaration.specifier.output, declarator.output, and initializer.opt.output.

Mixed mode attribute definitions consist of the reserved word @m, one or more dependents, a semicolon, and an evaluation expression. The occurrences referenced in the evaluation expression which are not dependents are taken to be the dependees in the definition.

The Ox-generated evaluator chooses an evaluation order such that the evaluation expressions for all of the dependees in the definition are executed before those of the dependents.

### 10.3.2 Attribute References in the Lex-File

Lexical rules are associated with return statements in Lex actions. After the terminating semicolon of each such statement, there may appear a possibly empty *attribute reference section*, delimited by @{ and @}, in which are defined all of the synthesised attributes of the returned token (Program 10.3.3).

**Program 10.3.3** (*Attributes in Lex-File*)

```
"L"?["]{s_char}*["] {
     return(STRINGliteral);
          @{ @STRINGliteral.output@ = strsave(yytext); @}
     }
```

The text must contain the C reserved word return. In particular, a return statement must not be hidden within a C macro unless the Lex-file is passed through the C preprocessor prior processing by Ox.

## 10.4 Postdecoration Traversals

Postdecoration refers to any time after the final decoration of the parse tree, which follows parsing of a correct input. The sequence

```
@traversal @lefttoright @postorder LRpost
```

specifies that a left-to-right postorder traversal of the parse tree be performed by the evaluator after the final decoration, and that the traversal be identified as LRpost.

Note that LRpost is programmer-defined, and is not an Ox reserved word.

**Program 10.4.1** (*Postdecoration Traversals*)

```
function.definition:
     declaration.specifier identifier.declarator compound.statement
     @{ @LRpost fprintf( outfp,"%s %s\n%s\n",@declaration.specifier.output@,
          @identifier.declarator.output@, @compound.statement.output@ );
     @}
     | type.specifier identifier.declarator compound.statement
     @{ @LRpost fprintf( outfp,"%s %s\n%s\n", @type.specifier.output@,
          @identifier.declarator.output@, @compound.statement.output@ );
     @}
     | declaration.qualifier.list identifier.declarator compound.statement
     @{ @LRpost fprintf( outfp,"%s %s\n%s\n",
          @declaration.qualifier.list.output@,
          @identifier.declarator.output@, @compound.statement.output@ );
     @}
     | type.qualifier.list identifier.declarator compound.statement
     @{ @LRpost fprintf( outfp,"%s %s\n%s\n", @type.qualifier.list.output@,
          @identifier.declarator.output@, @compound.statement.output@ );
     @}
     ;
```

Each attribute reference section in Program 10.4.1 contains a *traversal action specifier* starting with the *traversal mode annunciator* `@LRpost`, which is defined in the above-mentioned `@traversal` specification.

Following the final decoration, the parse tree is traversed once for each traversal specification. The order of performing the traversals corresponds to the order of appearance of the traversal specifications. In Program 10.4.1, e.g., when the `LRpost` traversal reaches a node at which the first rule is applied, the attribute `output` of `declaration.specifier` and `identifier.declarator` followed by a newline character, the `compound.statement`, and another newline character is written to the output file.

A traversal specification consists of the reserved word `@traversal`, followed by a *traversal specifier sequence* and a non-empty list of identifiers. The traversal specifier sequence may contain the following traversal specifiers (in any order):

- `@postorder` or `@preorder`
- `@lefttoright` or `@righttoleft`
- optionally `@disable` which may be useful for debbuging

If the traversal specifier sequence is left empty, the default is `@postorder @lefttoright`.

## 10.5 Automatic Generation of Copy Rules

Many attribute definitions function only to copy an instance belonging to one node to a like-named instance belonging to the node's parent or child. Large attribute grammars tend to have many such definitions, which are sometimes called *copy rules*. Ox provides two constructs for automatic generation of definitions of synthesised and inherited occurrences:

**`@autosyn`** Ox tries to supply missing definitions of synthesised occurrences by searching the RHS for occurrences with the same name. If exactly one such RHS occurrence is found, Ox generates a definition to copy it to the LHS.

**`@autoinh`** Ox tries to supply missing definitions of inherited occurrences by searching the LHS for occurrences with the same name. If exactly one such LHS occurrence is found, Ox generates a definition to copy it to each RHS occurrence.

The declaration `@autosyn output` supplies the following copy rule to the grammar rule in line 2 of Program 10.3.1:

```
@{ @i @paren.identifier.declarator.output@ = @IDENTIFIER.output@; @}
```

# Chapter 11

# The Making Of An Attribute Grammar

The following chapters describe the necessary functions and data structures which are needed to parse a C&Co source.

## 11.1 Attributes for Output

Since the grammar of C&Co is very similar to the grammar of C it sounds logical to copy as much code as possible from the C&Co source to C source. I have added an attribute `output` to nearly every non-terminal, for that reason. The terminal tokens assign `output` an ASCII representation of their value, e.g., the representation of the token `FOR` is the character string "`for`". The `output` attribute of every non-terminal is constructed according to the rules of the grammar.

A typical example is shown in Program 11.1.1. A `cast.expression` is either a `unary.expression` or a `type.name` in parentheses followed by another `cast.expression`.

**Program 11.1.1** (*Attribute* `output`)

```
1 cast.expression:
2      unary.expression
3      | '(' type.name ')' cast.expression
4           @{ @i @cast.expression.0.output@ = strbuild( "0101",
5           "(", @type.name.output@, ")", @cast.expression.1.output@ );
6           @}
7      ;
```

According to the automatic copy rule (see Section 10.5) of the attribute `output` it is not necessary to supply code for the first rule in line 2. In the second rule in line 3 the new value of `output` is an open parenthesis, the value of `output` of `type.name`, a close parenthesis, and the value of `output` of another `cast.expression`.

Nearly every grammar rule has a similar output evaluation expression. The other occurrences are handled with postdecoration traversals (see Section 10.4 and Program 10.4.1).

Other attributes analogous to output are blkout, ifout, pre, and post. The attribute blkout is used to move information from statements located inside a compound statement to the beginning of the enclosing block. See Section 6.3, Program 6.3.2 and Section 12.1.

The attribute ifout handles argument expressions in the interface of a C&Co function. Those are moved to the standard argument expression list of the C function in the final output. See Chapter 5 and Section 12.1.

The attributes pre and `post` of statements handle the occurrences of communication objects in expressions. Every read request of a communication object is moved to the attribute pre. Every write request is moved to the attribute post.

## 11.2 Auxiliary functions to handle attributes

In order to generate C source code it is necessary to build character strings out of other character strings. Two utility functions are `strsave()` and `strbuild()`.

### 11.2.1 `strsave`

Prototype: `char * strsave( char *s )`

This function copies the input string `s` to a new memory location which is allocated with `malloc`. The parser calls this function, e.g., to store the names of identifiers and other symbols.

### 11.2.2 `strbuild`

Prototype: `char * strbuild( char *fmt, ...  )`

This function concatenates a variable number of strings. It allocates memory for this new string, and returns a pointer to it. The number of strings is determined by the number of characters where `fmt` points to. If a character at position $n$ in `fmt` is '1', `strbuild` deallocates the $n^{th}$ string from memory. If the $n^{th}$ character in `fmt` is '0' or the $n^{th}$ string is empty, the memory location is not deallocated. See Program 11.1.1, lines 4–5 and Example 11.2.1.

**Example 11.2.1** (*Build a String*)

```
char *str1, *str2, *str3;

str1 = strsave( "Alpha" );
str2 = strsave( "Beta" );
str3 = strbuild( "0110", "This is ", str1, str2, "!" );
```

After the execution of the code of Example 11.2.1, the variable `str3` points to the string `"This is AlphaBeta!"`. The memory locations where `str1` and `str2` pointed to were freed.

## 11.3 The Symbol Table

I adapted the structure definitions of the symbol table from [27, 48] and built my own set of functions. Every symbol is represented as a structure which is given in Program 11.3.1

**Program 11.3.1** (struct symtab)

```
struct symtab
    {
    char *s_name;
    int s_type;
    int s_blknum;

    struct link *type;              /* first link in declarator chain */
    struct link *etype;                                /* last link */

    union
        {
        struct symtab *args;        /* If a function declaration, the */
                                                    /* argument list */
        struct structdef *sd;          /* If a struct, the structdef */
        } t;

    struct symtab *s_next;
    };
```

`s_name` points to the name of the symbol.

`s_type` is used to distinguish between names of structures, names of variables, and names of parameters of functions.

`s_blknum` is the level of indentation of the symbol. Global variables and functions start at an indentation level of 1. Every parameter type list of a function and every compound statement increases this level. The number is used to know which symbols are in what scope.

`type` points to the first link in the declarator chain. See below in Section 11.3.1 for a detailed description.

`etype` points to the last link in the declarator chain. This is usually the specifier of the variable (see Section 11.3.1).

`t.args` is used only if this symbol designates a function. It points to a linked list of symbols which specifies the parameters to this function. Note that the parameters have additional entries in the symbol table during parsing of the function. However, these entries are discarded after leaving the function declaration. The types of the parameters are saved in this list for future references.

`t.sd` is used only if this symbol designates a `struct` declaration (not a variable of type `struct`). It points to a structure `structdef` which is described in Section 11.3.2.

`s_next` points to the next symbol.

The symbol table is logically organised as a stack and implemented as a linked list. This has the following advantages:

- It is easy to implement.
- Local symbols are found before global ones.
- Local symbols are discarded easily.

The major disadvantage is that one can only perform a linear sort in the direction beginning from the local symbols to the global ones which is very time-consuming. Future implementations will probably use hash tables.

### 11.3.1 Declarators and Specifiers

It is quite complicated to represent types in an unconstrained typing system that permits virtually unlimited complexity in a variable declaration. A C/C&Co variable's type must be represented by a system of data structures working in concert.

Variable declarations have two parts: a *specifier* part which is a list of various keyword (`int`, `long`, `struct`, and so forth) and a *declarator* part that is made up of the variable's name and an arbitrary number of stars, array-size specifiers, and parenthesis (used both for grouping and to indicate a function).

The specifier is constrained because there is only a limited number of legal combinations of keywords that can be used here. It can be represented by a single structure.

The declarator is not constrained. Any number of stars, brackets, or parenthesis are permitted in any combination. Because of this organisation, a type can be represented using two kinds of structures, one representing the specifier and another representing the declarator. The type definition is a linked list of these structures. As already explained, the `type` field in the `symtab` structure points at the head of the list (`etype` points at the end). All type representations have exactly one specifier structure but there can be any number of declarators (including none).

Let's look at some examples. A simple declaration of the form `long foobar;` is represented like this:

| symtab | | | specifier |
|---|---|---|---|
| `s_name` | foobar | $\longrightarrow$ | `long int` |

A pointer declaration, like `short *value;` adds a declarator structure, like this:

| symtab | | | declarator | | specifier |
|---|---|---|---|---|---|
| `s_name` | value | $\longrightarrow$ | pointer to | $\longrightarrow$ | `short int` |

An array of pointers to chars such as `char *args[10];` has an additional "array" declarator:

| symtab | | | declarator | | declarator | | specifier |
|---|---|---|---|---|---|---|---|
| `s_name` | args | $\longrightarrow$ | array of 10 of | $\longrightarrow$ | pointer to | $\longrightarrow$ | `char` |

The foregoing system of declarators and specifiers is implemented with the set of structures given in Program 11.3.2.

**Program 11.3.2** (*Structures for Declarator and Specifier*)

```
typedef struct declarator
     {
     int type;                      /* POINTER, ARRAY, or FUNCTION */
     int num_ele;                   /* If class==ARRAY, # of elements */
     } declarator;

typedef struct specifier
     {
     int noun;              /* one of INT, CHAR, VOID, FLOAT, STRUCT, GOTO */

     unsigned _long :1;             /* 1=long 0=short */
     unsigned _unsigned :1;         /* 1=unsigned 0=signed */
     unsigned _static :1;           /* 1=static keyword found in declaration */
     unsigned _extern :1;           /* 1=extern keyword found in declaration */
     unsigned _comm :1;             /* 1=communication object */

     struct structdef *_struct;     /* Used only, if this is a struct */
     } specifier;

typedef struct link
     {
     unsigned class :1;             /* DECLARATOR or SPECIFIER */
     union
          {
          specifier s;              /* If class == SPECIFIER */
          declarator d;             /* If class == DECLARATOR */
          } select;

     struct link *next;             /* Next element of chain */
     } LINK;
```

Each link in the type chain can be either a declarator or a specifier. This is represented with the member `class` and the union `select` of the structure `LINK`.

A declarator can be either a pointer, an array, or a function. The number of elements of an array is stored in `select.d.num_ele`. This member is not used for pointers and functions.

The last link in the type chain is always of type `SPECIFIER`. The members of `struct specifier` have the following meaning:

`noun` stores the basic type of the symbol. It has one of the values `INT`, `CHAR`, `VOID`, `STRUCT`, or `GOTO`. If `noun` has the value `STRUCT`, the component `_struct` points to the definition of the structure. The value `GOTO` denotes a label.

`_long` is set if the variable is of type `long`. Thus, `short int i;` and `int i;` have identical representations.

`_unsigned` is set if the variable is of type `unsigned`.

`_static` is set if the keyword `static` is found in the declaration list of the variable.

`_extern` is set if the keyword `extern` is found in the declaration list of the variable.

`_comm` is set if this is a communication object.

`_struct` points to the definition of a structure. See Section 11.3.2 for the definition of `struct structdef`.

### 11.3.2 Representing Structures

Another structure is needed to represent `struct` declarations. It is called `structdef` and is shown in Program 11.3.3.

**Program 11.3.3** (*struct structdef*)

```
typedef struct structdef
    {
    char *tag;          /* Tag part of structure definition */
    int num_ele;        /* # of declarations in this structure */
    SYMTAB *fields;     /* Linked list of field declarations */
    } STRUCTDEF;
```

`tag` is the name of the structure. An arbitrary name is assigned to an otherwise unnamed structure.

`num_ele` is the number of declarations in this structure.

`fields` points to a linked list of field definitions. Each of which is a `symtab` structure—one symbol for each field. The `s_next` field of the `symtab` structure links together the fields of this structure. This means that you must use a linear search to find a field, but it lets you have an arbitrary number of fields.

**Example 11.3.4** (*Example of a Structure*)

```
struct foo
     {
     int a;
     int b;
     };

struct foo bar;
```

The representation of the structure and variable declaration of Example 11.3.4 looks like this:

| symtab | | | structdef | |
|---|---|---|---|---|
| `s_name` | foo | ⟶ | `tag` | foo |
| `type` | `NULL` | | `num_ele` | 2 |
| `t.sd` | → `structdef` | | `fields` | → `a` |

While parsing the declaration of `struct foo` a new entry is made in the symbol table with the name `foo` and the type "structure". For this reason, there is no need for a declaration chain—`type` points to `NULL`. On the other hand, a new `structdef` is created and `t.sd` points to it.

The `structdef` gets the same name as the symbol table entry. This is redundant on first thought but necessary when this `structdef` is pointed to from a variable specifier (see below). The number of elements is 2 (a and b) and `fields` points to an element of type `symtab` which is *not* in the symbol table. Otherwise, the parser might look up the *structure field* `a` where it should find a *variable* `a`.

| symtab | | | symtab | |
|---|---|---|---|---|
| `s_name` | a | ⟶ | `s_name` | b |
| `type` | declaration chain | | `type` | declaration chain |
| `s_next` | → `b` | | `s_next` | `NULL` |

The fields of a structure are organised as a simple linked list. The last element of the list points to `NULL`. Thus, the field `num_ele` of `structdef` is redundant. It is used only for easier reference. The standard declaration chain which was specified in Section 11.3.1 is also used to describe the types of these fields.

| symtab | | | specifier | | | structdef | |
|---|---|---|---|---|---|---|---|
| `s_name` | bar | ⟶ | `noun` | `STRUCT` | ⟶ | `tag` | foo |
| `type` | → specifier | | `_struct` | → `structdef` | | `num_ele` | 2 |
| | | | | | | `fields` | → `a` |

The symbol table entry for `bar` designates a variable. Therefore, `type` points to a link of the declaration chain. The link is a specifier and `noun` of this specifier has the value `STRUCT`. The field `_struct` points to the same `structdef` element which was described above.

This is the reason why it is necessary to store the name of the structure also in the `structdef`: The specifier of a variable of type `struct` points to the `structdef` and not to the symbol table entry of the name of the structure.

## 11.4 Attributes for Symbols and Types

The type of a variable is recognised with the help of attributes during the parsing process. The process of type recognition involves various steps. A declaration in C&Co consists basically of a *specifier* and a *declarator*. The specifier part is a combination of various *basic types* and one *storage class* (including `comm`). The declarator part consists of an *identifier* and an arbitrary combination of stars, parenthesis, and brackets.

These parts are connected only by their reference to the symbol table entry of the identifier. The various grammar rules synthesise information which is eventually assigned to the symbol's declaration link list.

There is a number of attributes which are needed for type recognition, namely:

bt is the basic type of a variable, e.g., `long`, `short int`. The attribute is synthesised in the grammar rule `basic.type.name`. The attribute value of every type name consists of a single bit set at different bit positions. Thus, a combination of more than one basic type is managed easily.

sc is the storage class of a variable, e.g., `static`. The attribute is synthesised in the grammar rule `storage.class`.

sp contains a complete specifier (see Program 11.3.2) of a variable. The value of this attribute is a combination of bt and sc. This attribute is later assigned to the specifier part of the declaration chain of the appropriate symbol.

link points to the start of a declarator chain of a variable. A link is added to this chain at each pointer indirection or array declarator.

field points to the beginning of a list of field declarations of a structure. A new field is added to this list at every field declarator.

sd points to a `STRUCTDEF` (see Program 11.3.3) of a structure. The attribute is synthesised during the declaration of a structure in the grammar rule `struct.specifier`.

## 11.5 Auxiliary Functions to Handle Symbols and Types

The functions which are described in this section perform various tasks on variables. They manage the symbol table, perform operations on attributes and connect attributes with symbols.

### 11.5.1 `s_lookup`

Prototype: `SYMTAB * s_lookup( char *text )`

This function is called by the scanner to look up a symbol from the symbol table. A global variable indicates, if we are currently parsing inside a `struct` declaration or an expression with a structured variable. In these cases, we must not insert/look up any symbols from the normal symbol table.

If the symbol `text` does not exist, it is inserted into the symbol table with a call to `s_create()`.

When we are parsing a `struct` declaration, we have to insert `text` as a field of the structure with a call to `create_struct_field()`.

### 11.5.2 `s_find`

Prototype: `SYMTAB * s_find( char *name )`

This function looks up the symbol `name` from the symbol table dependant of the parser status. If the parser reads past the token `STRUCT` we are parsing a `struct` specifier and must search only for the names of structures. Any other case searches for the name of a variable.

### 11.5.3 `s_create`

Prototype: `SYMTAB * s_create( char *name )`

This function creates a new symbol `name` and inserts it into the symbol table. As already mentioned, the symbol table is logically organised as a stack and implemented as a linked list. This is necessary to build a stack with no restriction in size.

The “stack” consists of two empty symbols at first. The first symbol is always empty, the second one, which is called “blind”, marks the end of the list and is pointed to by the first. Every new symbol is inserted at the end of the list just before the symbol “blind”.

This data structure makes it easy to unlink unnecessary symbols at the end of the list, i.e., pop the symbols from the stack.

### 11.5.4 `create_struct_field`

Prototype: `SYMTAB * create_struct_field( char *name )`

This function allocates memory for a field in a structure. The symbol name is not stored in the symbol table but added to a list of other fields. The list is added to the symbol table entry of the structure.

### 11.5.5 `blk_push`

Prototype: `void blk_push( void )`

This function increments a global block counter and is called whenever a new program scope is opened, i.e., at the following places:

- At the beginning of a compound statement.
- At the beginning of a parameter type list. The reason for this is that the name of a function must be visible also outside of the declaration block of the function. On the other hand, the names of the parameters of the function—given in the parameter type list—are visible only in the function.
- At the beginning of an interface definition list. This is a tricky situation because the interface is parsed *before* the identifier, i.e., the symbol table entry, of the function. The block number of the communication objects in the interface definition list is therefore adjusted to reflect the correct scope.

Note that technically the declaration of a function defines two new blocks: The first is opened after the '`(`' before the parameters. The second after the '{' of the compound statement of the function.

### 11.5.6 `blk_pop`

Prototype: `void blk_pop( void )`

This function decrements the global block counter and removes symbols from the symbol table whose block number is greater than the current block, i.e., all symbols which were defined at an inner scope and are not visible anymore.

### 11.5.7 `check_basic_type`

Prototype: `void check_basic_type( long bt, specifier *sp )`

This function validates the combination of basic types in `bt` and writes the appropriate values into the fields of `sp`. The following basic types must not be specified together:

- `long` and `short`
- `signed` and `unsigned`

### 11.5.8 check_storage_class

Prototype: `void check_storage_class( long sc, specifier *sp )`

This function checks the storage class in `sc` and writes the appropriate values into the fields of `sp`. Only one storage class can be specified for a given symbol.

### 11.5.9 add_declarator

Prototype: `void add_declarator( int decl_type, SYMTAB *p_sym )`

This function adds a declarator of type `decl_type` to the declarator chain of the symbol `p_sym`. The new declarator is appended to the end of a possibly existing list.

Note that in the current implementation this function is only used for the declaration type `POINTER`. Arrays are handled with `make_array_link`. Future implementations might add other declarator types.

### 11.5.10 add_specifier

Prototype: `void add_specifier( specifier sp, SYMTAB *p_sym )`

This function adds the specifier `sp` to the declarator chain of the symbol `p_sym`. The new link is appended to the end of a possibly existing list.

### 11.5.11 add_link_list

Prototype: `void add_link_list( LINK *first, SYMTAB *p_sym )`

This function adds a complete list of links to the declarator chain of the symbol `p_sym`. The list starts with `first` and is appended to the end of a possibly existing list. This function is needed for the declaration of arrays. In contrast to a pointer declaration, which produces only a single declarator link, a declaration like

```
int array[5][10][15];
```

produces this declarator list which must be added to the declerator chain:

| declarator | | declarator | | declarator |
|---|---|---|---|---|
| array of 5 of | $\longrightarrow$ | array of 10 of | $\longrightarrow$ | array of 15 of |

### 11.5.12 make_label

Prototype: `void make_label( SYMTAB *p_sym )`

This function creates a specifier link for the symbol `p_sym`. The type `GOTO` is assigned to the `noun` field of the specifier which indicates a label.

### 11.5.13 make_structdef

Prototype: `STRUCTDEF * make_structdef( SYMTAB *p_sym, SYMTAB *fields )`

This function is called while parsing the declaration of a structure. It creates a `structdef` element for the structure with the symbol table entry `p_sym`. `fields` points to the first element of a list of symbols which identify the fields of this structure.

Important: If `make_structdef` is called with the parameter `fields` pointing to `NULL`, nothing is changed in the list of—the already existing—fields. If there already exists a `structdef` element for the symbol `p_sym`, nothing is created. This is important for a list of declarations like this:

```
struct s1;
struct s1 { int a; int b; };
struct s2 { int c; int d; };
struct s2 var;
```

The first declaration of `struct s1` just identifies the name `s1` with a structure. Thus, `make_structdef` is called with the second parameter being `NULL` which creates a new `structdef` but doesn't assign anything to the list of fields.

The second declaration adds the now available list of fields to the already existing `structdef`.

The third and fourth cases do just the opposite. The third one builds a complete `structdef`, including a list of fields but the fourth case doesn't do anything because the structure `s2` is already defined.

### 11.5.14 make_array_link

Prototype: `LINK * make_array_link( char *constant )`

This function creates a declarator link with the type `ARRAY`. The field `num_ele` gets the value of the integer representation of `constant`.

### 11.5.15 push_parse_struct

Prototype: `void push_parse_struct( void )`

This function is called whenever the first "{" of a declaration of a structure is reached. It pushes the symbol table entry of the current structure onto a stack and switches to a mode which prevents the parser from inserting the following identifiers to the symbol table. The identifiers must be added to the `fields` element of the `structdef` instead.

The stack of symbol table entries is needed because structures can be nested.

### 11.5.16 pop_parse_struct

Prototype: `void pop_parse_struct( void )`

This function removes from the statck the symbol table entry of the structure which was pushed onto it. It is called after the "}" of a structure declaration.

### 11.5.17 `new_symbol`

Prototype: `SYMTAB * new_symbol( void )`

This function creates a new symbol with an arbitrary name (which has not been used before). Unnamed structures need an entry in the symbol table because variables must have access to their `structdef` entry for code segments like:

```
struct { int a; int b; } var;
```

The declarator list of the variable `var` contains a pointer to the `structdef` of the unnamed structure.

### 11.5.18 `string_declaration`

Prototype: `char * string_declaration( SYMTAB *symbol )`

This function builds a type description from the declarator link list of `symbol`. For example, it builds the string `"char *args[10]"` from the declarator list of a variable named `args`:

| declarator | | declarator | | specifier |
|---|---|---|---|---|
| array of 10 of | $\longrightarrow$ | pointer to | $\longrightarrow$ | `char` |

The result string is allocated with `strsave`. It is needed for references to communication objects in the source code of the output. See the next chapter for details.

# Chapter 12

# How To Generate Code

The following sections describe the process of code generation by means of examples[1]. The basic idea is the following:

- For every occurrence of a declaration of a communication object declare a new variable which holds an object identifier (OID). The OID is needed as a reference for the CoKe.
- For every occurrence of a communication object in a C&Co statement, perform all read requests before, all write requests after the statement.
- For every occurrence of a C&Co keyword which has a direct equivalent to a CoKe function, write appropriate function calls.

## 12.1 Communication Objects

It was very tricky to implement communication objects in a satisfactory way. I made several attempts.

### 12.1.1 Mapping for `comm`

My first consideration was the following:

1. Translate every declaration of a communication object into its OID, e.g.,

   `comm int i;` ⟶ `CoKe_OID i;`

2. Define a function `read(OID)` and substitute every occurrence of an OID on the right side of an assignment with it, e.g.,

   `a = 2 * i;` ⟶ `a = 2 * read( i );`

[1] Many of the examples make calls to the CoKe or use type definitions of the library. These functions have the characters "CoKe" in their name.

3. Define a function `assign(OID, value)` and substitute every occurrence of an assignment to an OID with it, e.g.,

   `i = 2 * a;` $\longrightarrow$ `assign( i, 2 * a );`

Fine, but you would have to define many functions which return and accept every possible type of your communication objects and assignment values, even for arrays and structures. It doesn't work!

The solution that I implemented uses a combination of OIDs and standard C variables. I don't translate communication objects to something else, however, I declare an OID in addition to a standard C variable. The OID gets the same name as the communication object with the prefix `_OID_`, e.g,

**Example 12.1.1** (*Mapping for* `comm`)

```
comm int i;
```

$\longrightarrow$

```
int i;
CoKe_OID _OID_i;
```

There are functions in the interface library [36] for the CoKe which make it easier as before to send variables of arbitray types to the kernel. Every transmit operation starts with a call to the function `CoKe_INIT_BUF()`. After that, the programmer has to call similar functions for different basic types (integers, characters) and structures, e.g.,

```
CoKe_PUT_INT(long i);
CoKe_PUT_RAW(char *data);
CoKe_PUT_STRUCT(char *functor_name, long arity);
```

and finally a call to the function

```
coke_write(CoKe_TID tid, CoKe_OID oid, CoKe_IC *icptr, long ic_len)
```

to write the communication object. The parameter `tid` is the transaction identifier because communication objects can only be written within transactions. A TID is always defined in a transaction.

Similar functions exist to read communication objects from the kernel. These functions need an additional variable of type `CoKe_IC *`, thus, it has to be declared too.

### 12.1.2 Mappings for Read and Write Requests

The algorithm for communication objects now looks like this:

1. Remember the communication objects $c_1 \ldots c_n$ in any given statement $\mathcal{S}$ of the C&Co code.

2. Perform all read requests for all communication objects $c_r$ *before* $\mathcal{S}$.

3. Use the standard C variables for the communication objects $c_1 \ldots c_n$ in statement $\mathcal{S}$.

4. Perform all write requests for all communication objects $c_w$ *after* $\mathcal{S}$.

Example 12.1.2 shows a simple application of the above algorithm. The variables `i` and `j` are declared as `comm int i, j;`, thus their OID counterparts `_OID_i` and `_OID_j` are also defined.

**Example 12.1.2** (*Mappings for Read and Write Requests*)

```
i = 2 * j;
```

$\longrightarrow$

```
coke_b_read( _OID_j, &icptr );
icptr = CoKe_SCAN_INT( icptr, &j );
i = 2 * j;
CoKe_INIT_BUF();
CoKe_PUT_INT( i );
coke_write( tid, _OID_i,
     CoKe_BUF(), CoKe_BUF_SIZE() );
```

Please note, that the statement $\mathcal{S}$ remains unchanged—the line "`i = 2 * j`" in Example 12.1.2 is the same as before. You don't have to worry about different types. But there are also a few drawbacks. As already mentioned in Chapter 9, some operators are unsave to use in conjunction with communication objects. But this is only a minor drawback in comparison to the advantages.

I implemented these mappings with the help of two attributes: pre and post. The appropriate CoKe functions are appended to these attributes whenever the parser encounters a communication object in an expression. After the parser completes one statement, it writes the contents of pre, output, and post (in that order) to the output file.

### 12.1.3 Mapping for the Interface

The interface of a function is implemented in a similar fashion. Every communication object in the interface list of a function is represented by an OID in the standard parameter list of the corresponding C function. Additionally, a local variable with the name of each communication variable in the interface is created in the C function.

The declaration of the function

```
int ( comm int i ).function( long a, char *b )
{
⋮
}
```

is transformed into

```
int function( long a, char *b, CoKe_OID _OID_i )
{
int i;
⋮
}
```

## 12.2 process, trans, and Other Keywords

These features of C&Co are trivial to implement in comparison to communication objects because they map directly to equivalent functions of the kernel library routines. For example, the begin of a transaction is indicated with a call to the function

```
coke_top_trans_begin( CoKe_TID tid )
```

`tid` is the newly created transaction identifier which is used for all subsequent calls to transaction functions, e.g.,

```
coke_trans_commit( CoKe_TID tid )
```

which commits a transaction.

Every occurrence of such a keyword is handled with the appropriate output in either one or more of the pre, output, or post attributes.

# Chapter 13

# Experience and Future Work

I really cannot say that the implementation of C&Co was easy. Most of the problems I encountered originated

- from the grammar of C, which is simply overwhelming at first,
- from my lack of experience with Lex and Yacc (which changed drastically in the meantime),
- from the need to extend the grammar without introducing additional conflicts,
- from the need to switch to an attributed grammar, including the time expense to learn the new tool,
- and from the enforced coexistence of Lex, Yacc, and Ox.

From the current point of view there is a list of thing to follow in the future.

- My personal ambition is to build a better parser, without the need to make context switches at various points in the grammer in order to tell the scanner what to do next.
- Enhancement of the library functions for the CoKe. This includes new types for communication objects, e.g., a communication object which is actually a stream of many similar objects. I started to write a paper about a new protocol between the CoKe and C&Co already [20].
- Porting the compiler to other machine architectures. The compiler runs currently only on a DECstation and an Alpha AXP microprocessor. The next target machine is a Sun Sparc because the CoKe runs also on this computer.
- Rewrite of the compiler to generate machine code. This is also necessary to implement true object orientation, e.g., selection of a function according to the types of the parameters or operator overloading.

- Implementation of threads. Right now, every new process is started by the coordination kernel as a heavy weight process. A new C&Co compiler might generate a run time system which itself starts any C&Co process as a light weight thread.

# Chapter 14

# Conclusions

Instead of defining a completely new language or system, our group had the idea to extend traditional programming languages by coordination properties. This resulted in the development of the coordination kernel CoKe and the languages VPLand C&Co.

The result from this thoughts is a coordination toolkit which can be used to enhance virtually any programming language by coordination. One of the most important aspects is, that the programming style of the host language is not changed. It is very easy to reuse already existing programs. The coordination extensions include a high-level, reliable communication mechanism, concurrency, and an advanced transaction model.

Languages that are enhanced by this coordination toolkit communicate via the coordination kernel, are interoperable, and can cooperate by crossing hardware and language borders.

I have programmed a compiler for the coordination language C&Co which generates standard C source. The compiler was implemented with Ox, a compiler generator which can use attributed grammars. The C&Co compiler runs currently on a DECstation with a 32Bit Mips processor and on an Alpha AXP which has a 64Bit architecture.

Future work include the building of a better parser and porting the compiler to other architectures.

# Bibliography


[1] G. Agha, P. Wegner, and A. Yonezawa, editors. *Research Directions in Concurrent Object-Oriented Programming.* The MIT Press, 1993.

[2] D. E. Bakken. *Supporting Fault-Tolerant Parallel Programming in LINDA.* PhD thesis, University of Arizona, Department of Computer Science, August 1994. TR 94-23.

[3] H. Bal. *Programming Distributed Systems.* Prentice Hall, New York, 1990.

[4] H. Bal, J. Steiner, and A. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[5] H. Bal and S. Tanenbaum. Distributed programming with shared data. *Computer Languages*, 16(2):129–146, 1991.

[6] Ph. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[7] K. P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53, December 1993.

[8] Kurt M. Bischoff. *Ox: User Reference Manual*, 1993.

[9] A. Brogi and P. Ciancarini. The concurrent language, Shared Prolog. *ACM Transactions on Programming Languages and Systems*, 13(1):99–123, January 1991.

[10] O. Bukhres and A. Elmagarmid, editors. *Object-Oriented Multidatabase Systems.* Prentice-Hall, 1994. in print.

[11] O. Bukhres, A. Elmagarmid, and e. Kühn. Advanced languages for multidatabase systems. in [10], 1994. to appear.

[12] O. Bukhres and e. Kühn. Highly available and reliable communication protocols for heterogeneous systems. *Information Sciences*, 1995. to appear.

[13] N. Carriero and D. Gelernter. How to write parallel programs: A guide to the perplexed. *ACM Computing Surveys*, 21(3):323–358, September 1989.

[14] N. Carriero and D. Gelernter. Coordination languages and their significance. *Communications of the ACM*, 2(35):96–107, 1992.

[15] R. Chin and S. Chanson. Distributed object-based programming systems. *ACM Computing Surveys*, 23(1):91–124, March 1991.

[16] Digital Equipment Corporation, Hewlett Packard Company, HyperDesk Corporation, NCR Corporation, Object Design, Inc., SunSOft, Inc. *The Common Object Request Broker: Architecture and Specification*, December 1991. Draft 10, Revision 1.1, OMG Document Number 91.12.1.

[17] Graeme N. Dixon, Graham D. Parrington, Santosh K. Shrivista, and Stuart M. Wheater. The treatment of persistent objects in arunja. In *Proceedings of the Third European Conference on Object-Oriented Programming ECOOP89*, pages 169–189, July 1989.

[18] A. Elmagarmid, Y. Leu, W. Litwin, and M. Rusinkiewicz. A multidatabase transaction model for InterBase. In *Proceedings of the 16th International Conference on Very Large Data Bases*, August 1990.

[19] A. K. Elmagarmid, editor. *Database Transaction Models for Advanced Applications.* Morgan Kaufmann Publishers, 1992.

[20] A. Forst. Protocol between CoKe and C&Co systems. Technical report, University of Technology, Vienna, 1995.

[21] A. Forst, e. Kühn, H. Pohlai, and K. Schwarz. Logic based and imperative coordination languages. In *Seventh International Conference on Parallel and Distributed Computing Systems*, Las Vegas, 1994.

[22] H. Garcia-Molina and B. Kogan. Node autonomy in distributed systems. In *Proceedings of the International Symposium on Databases in Parallel and Distributed Systems*, pages 158–166, Austin, Texas, December 1988. IEEE Computer Society Press.

[23] G. A. Geist and V. S. Sunderam. Network-based concurrent computing on the PVM system. *Concurrency: Practice and Experience*, 4(4):293–311, June 1992.

[24] D. Gelernter. Multiple tuple spaces in Linda. In M. Rem E. Odjik and J. C. Syre, editors, *PARLE'89*, pages 20–27. Springer Verlag, 1989.

[25] D. Gelernter and N. Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2), February 1992.

[26] S. Gregory. *Parallel Logic Programming in PARLOG. The Language and its Implementation.* Addison-Wesley, England, 1987.

[27] A.I. Holub. *Compiler Design in C.* Prentice Hall, Englewood Cliffs, 1990.

[28] Stephen C. Johnson. Yacc: Yet another compiler compiler. Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hills, New Jersey, 1975. Reprinted as PS1:15 in *UNIX Programmer's Manual*, Usenix Association, 1986.

[29] G. E. Kaiser and B. Hailpern. An object-based programming model for shared data. *ACM Transactions on Programming Languages and Systems*, 14(2), April 1992.

[30] G. E. Kaiser, W. Hseush, J. C. Lee, S. F. Wu, E. Woo, E. Hilsdale, and S. Meyer. MeldC: A reflective object-oriented coordination language. Technical Report CUCS-001-93, Columbia University, January 1993.

[31] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, New Jersey 07632, 1 edition, 1978.

[32] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, New Jersey 07632, 2 edition, 1988.

[33] e. Kühn. Development of software for the coordination of distributed systems. Technical Report TR Project P7773-PHY, University of Technology, Vienna, September 1992. (in German).

[34] e. Kühn. Fault-tolerance for communicating multidatabase transactions. In *Proceedings of the 27th Hawaii International Conference on System Sciences (HICSS)*, Wailea, Maui, Hawaii, January 4–7 1994. ACM, IEEE.

[35] e. Kühn. A universal model for the coordination of distributed systems. Technical Report TR Project P9020-PHY, University of Technology Vienna, 1994. submitted for publication.

[36] e. Kühn. *Principles of Coordination Systems*. University of Technology Vienna, 1995. A Library Approach for Distributed and Reliable Programming in C and Prolog.

[37] e. Kühn, H. Pohlai, and F. Puntigam. Communication and transactions in $\mathrm{V}^{ienna}\mathrm{P}^{arallel}\mathrm{L}_{ogic}$. *Computers and AI Journal*, 1993.

[38] e. Kühn, H. Pohlai, and F. Puntigam. Concurrency and backtracking in $\mathrm{V}^{ienna}\mathrm{P}^{arallel}\mathrm{L}_{ogic}$. *Computer Languages*, 19(3), July 1993.

[39] e. Kühn and F. Puntigam. The VPL Vienna Parallel Logic language. Technical Report TR-185-91-1, Technical University of Vienna, 1991.

[40] e. Kühn and F. Puntigam. Reliable communication in VPL. In *Proceedings of the Parallel Architectures and Languages Europe (PARLE-92)*, pages 769–782, Paris, June 15–18 1992. Springer Verlag, LNCS.

[41] e. Kühn, F. Puntigam, and A. K. Elmagarmid. Multidatabase transaction and query processing in logic. In A. K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, chapter 9. Morgan Kaufmann Publishers, 1991.

[42] e. Kühn, F. Puntigam, and A. K. Elmagarmid. An execution model for distributed database transactions and its implementation in VPL. In *Proceedings of the International Conference on Extending Database Technology, EDBT'92*, Vienna, March 1992. Springer Verlag, LNCS.

[43] M. E. Lesk and E. Schmidt. Lex — a lexical analyzer generator. Computing Science Technical Report 39, AT&T Bell Laboratories, Murray Hills, New Jersey, 1975. Reprinted as PS1:16 in *UNIX Programmer's Manual*, Usenix Association, 1986.

[44] B. Liskov. Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.

[45] B. Meyer. Systematic concurrent object-oriented programming. *Communications of the ACM*, 36(9):56–80, September 1993.

[46] James A. Roskind. The Roskind Grammar. Can be downloaded via ftp from ftp://ftp.edvz.uni-linz.ac.at/pub/unix/sun/c++/c++grammar2.0.tar.Z.

[47] H. Schildt. *The Annotated ANSI C Standard*. Osborne MacGraw-Hill, Berkeley, 1990.

[48] A.T. Schreiner and G. Friedman. *Compiler bauen mit UNIX*. Carl Hanser Verlag, München, Wien, 1985.

[49] E. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):412–510, September 1989.

[50] J. Shirley. *OSF Distributed Computing Environment: Guide to Writing DCE Applications*. O'Reilly & Associates, Inc., 1992.

[51] S. K. Shrivastava, G. N. Dixon, and G. D. Parrington. An overview of the Arjuna distributed programming system. *IEEE Software*, pages 66–73, January 1991.

[52] A. S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, 1992.

[53] P. Wegner. Concepts and paradigms of object-oriented programming. In *Proceedings of the OOPSLA-89*, June 1990. Expansion of Keynote Talk.

[54] G. Wiederhold, P. Wegner, and S. Ceri. Toward megaprogramming. *Communications of the ACM*, 35(11), November 1992.