Diss ETH No. 17480

A Component Language for Pointer-Free Concurrent Programming and its Application to Simulation

A dissertation submitted to
ETH Zurich

for the degree of
Doctor of Sciences

presented by
Luc Bläser
MSc. ETH in Computer Science, ETH Zurich
born 3 October 1979
citizen of Luxembourg

accepted on the recommendation of

Prof. Dr. J. Gutknecht, examiner
Prof. Dr. K. Nagel, co-examiner

2007

Abstract

Today's programming languages are still noticeably underdeveloped for the construction of well-structured concurrent software systems. They typically impose many unnecessary and unacceptable restrictions and inconsistencies due to a multiplicity of different counterproductive concepts. In particular, pointers generally counteract structured and hierarchical program relations at runtime, while the support of concurrency remains largely neglected as a second-class feature in current programming languages.

To improve this adverse situation, we have developed a new programming language, which directly integrates a general component notion. Components are only managed by three fundamental relations: (1) hierarchical composition without use of explicit pointers, (2) symmetric connections with a dual concept of offered and required interfaces and, (3) communication-based interactions. As a result of these concepts, the new language enables general hierarchical encapsulation, client-individual statefull communications, inherent and race-free concurrency, symmetric polymorphism separated from code reuse, as well as hierarchical lifetime dependencies of the components.

The programming language has been implemented with a runtime system that comprises a small kernel. **It** facilitates high-performance concurrency, clearly surpassing existing systems in the scalability and efficiency of parallel processes. For this purpose, the system features innovative concepts such as light-weighted processes with fine-granular stacks and low-cost software-based preemption, safe memory management without need of automatic garbage collection.

In order to offer a proof of concepts and demonstrate the practical suitability, we have applied the new programming language to an archetypical kind of application: simulation. At the example of a traffic simulation, we can

demonstrate that the new language enables a more natural mapping to the program, with the potential to be distributed. At the same time, our language offers a substantially higher perfonnance compared to other sequential or concurrent simulation systems. More concretely, vehicles are modelled in this case study as self-active instances that all drive autonomously and concurrently in a virtual time. In particular, the component structures also support the flexible and accurate representation of dynamic hierarchical compositions with vehicle transports. With regard to the realistic traffic simulation of Greater Zurich, our simulation runs faster by a factor of three than a traditional simulation system.

Kurzfassung

Was die Konstmktion strukturierter nebenHiufiger Software betrifft, sind heutige Programmiersprachen noch erheblich unterentwickeit. Aufgrund der Vieizahl unterschiedlicher und oft unntitzer Konzepte unterliegen die Sprachen meist vielen unnotigen Einschrankungen und Inkonsistenzen. Die Verwendung von Zeigern (Pointers) erschwert im Allgemeinen die Beschreibung klar strukturierter und hierarchischer Programmbeziehungen zur Laufzeit. Ferner bieten heutige Programmiersprachen nur unzureichende Untersttitzung für Parallelität, da NebenHiufigkeit Iediglich mit schwerfalligen Zusatzkonzepten ermoglicht wird.

Um diese Schwachpunkte zu beseitigen, haben wir eine neue Programmiersprache entwickeit. Diese Programmiersprache basiert auf dem aligemeinen Begriff der *Komponente.* Die Verwaitung von Komponenten ist durch drei grundiegende Beziehungen gekennzeichnet. (1) *Hierarchische Komposition* ohne explizite Verwendung von Zeigern, (2) symmetrische Verkntipfung mit einem dualen Konzept von *angebotenen* und *eiforderten Schnittstellen,* (3) Interaktion durch *Kommunikation.* Durch die Verwendung dieser Konzepte ergeben sich verschiedene vorteilhafte Moglichkeiten. Dazu gehoren die allgemeine hierarchische Einkapselung, separate zustandsbehaftete Kommunikation zwischen Komponenten, inhärente NebenHiufigkeit unter Ausschluss so genannter Races, symmetrischer PoIymorphismus getrennt von der Code-Wiederverwendung, sowie hierarchische Existenzabhangigkeiten unter den Komponenten.

Zur Programmiersprache wurde ein Laufzeitsystem implementiert. Dieses System, welches tiber einen eigenen Kern verfügt, macht besonders Ieistungsstarke Ausfuhrung von nebenläufigen Programmen moglich. Im Vergleich zu

anderen Systemen weist das neue Laufzeitsystem eine deutlich hahere Skalierbarkeit und eine wesentlich schnellere Ausfuhrungsgeschwindigkeit fur parallele Prozesse auf. Zu diesem Zweck wurden beim Bau des Systems weitere innovative Konzepte verfolgt. Dazu zählen unter anderem der Einsatz von *Leichtgewichtsprozesse mit beliebig kleinen Stacks* und *kosteneffiziente, softwarebasierte Preemption* sowie eine *sichere dynamische Speicherverwaltung* ohne Bedarf eines *Garbage-Collectors.*

Zur Demonstration der praktischen Einsetzbarkeit der Konzepte haben wir die neue Programmiersprache fur eine archetypische Art der Anwendung eingesetzt, nämlich zur Simulation: Anhand des Beispiels einer Verkehrssimulation hat sich gezeigt, dass die neue Programmiersprache eine natiirlichere Abbildung des Realitätsmodells zum Programm ermaglicht. Damit geht ein erhebliches Potenzial zur Verteilbarkeit einher. Zusatzlich erzielen wir eine substanziell hahere Ausfiihrungsgeschwindigkeit als andere sequenzielle und nebenHiufige Simulaticmssysteme. In dieser Fallstudie werden die Fahrzeuge als selbstaktive Komponenten dargestellt, so dass sie aIle autonom und parallel in einer virtuellen Zeit fahren. Insbesondere erlauben es die Komponentenstrukturen, dynamische hierarchische Kompositionen für die Fahrzeugverladung realitatsnah und flexibel zu modellieren. Eine realistische Verkehrssimulation vom GroBraum Zurich wird mit unserer Simulation urn Faktor drei schneller ausgefiihrt alsmit einem traditionellen Simulationssystem.

Acknowledgments

I am very grateful to several people who supported me during this work and helped me to complete this dissertation.

First of all, I would like to sincerely thank Prof. Dr. Jurg Gutknecht for giving me the opportunity to follow and realise my ideas within this doctoral thesis. I am indebted to his continuous support, helpful advice, constructive feedback and kind encouragement throughout my work. I also greatly appreciate the comfortable work environment that was offered to me as well as the freedom given regarding my working practice.

I am also very grateful to Prof. Dr. Kai Nagel who kindly accepted to be the co-supervisor of this dissertation and gave me valuable feedback, support and encouragement during my thesis and especially during the case study on traffic simulation. It was a rewarding experience to apply my programming language in this research field.

My sincere thanks also go to my colleagues at the ETH Zurich, namely Dr. Felix Friedrich, Dr. Svend Knudsen, Daniel Keller, Roman Mitin, Sven Stauber, Thomas Kägi, Ulrike Glavitsch-Eggler, Dr. Thomas Frey, Dr. Emil Zeller, Dr. Stefan Muller, Dr. Simon Schubiger-Benz, André Fischer, Raphael Guntensperger, Mazda Mortasavi, Michael Szediwy, Florian Negele, Alexey Morozov, Dr. Dennis Majoe, Karel Skoupy and many others. It was an absolute pleasure to work together with them. We had a lot of interesting discussions and chats about research and other topics, not only during the numerous coffee breaks. I greatly appreciate the feedback on my work that I received from my colleagues.

During my visit to the TU Berlin and the subsequent e-mails which followed, I received valuable advice and feedback from Martin Rieser, David Strippgen and Gunnar

Flötteröd. I am very appreciative of their help, whether it was in explaining the details of traffic simulation techniques, providing the simulation data, and discussing interesting ideas for my work.

I would like to also express my gratitude to Ruth Hidalgo, who helped me with various administrative questions and complex issues that were often difficult to resolve. I am also grateful for the administrative support from Hanni Sommer and Franziska Mader.

In addition, the Swiss Federal Department of the Environment, Transport, Energy and Communications (UVEK) kindly granted permission to use the data for the traffic simulation.

Furthermore, I am particularly thankful to Dr. Felix Friedrich, Dr. Svend Knudsen and Nadja Beeli who proofread and scrutinised my dissertation before I submitted it to my supervisors. Roman Mitin also read the dissertation and provided valuable comments and suggestions for improvement.

Lastly (but of no less importance), I would like to thank Nadja Beeli, my family and all my colleagues for their constant encouragement and support.

Luc Blaser

Zurich, September 2007

Contents

Chapter 1

Introduction

1.1 Motivation

During recent decades of computer science, we can observe a predominant trend towards programming languages, which possess an increasing number of features of ever increasing abstraction and complexity. However, programmers are often poorly supported with suboptimal concepts and have to fmd workarounds to the various artificial restrictions and contradictions, which inevitably arise from the growing amount of features in programming languages.

One of the gravest deficiencies and the root cause of many problems in current languages is, according to our analysis, the fact that *pointers* (or *references)* still establish the main concept for describing dynamic program structures. Their expressiveness is much too weak to accurately organise well-structured relations between objects, such that programs are commonly based on underspecified and vulnerable structures. In fact, the presence of references forms the main reason for the missing support of *hierarchical encapsulation* in today's languages.

We also observe that the increasing need of concurrency can not be adequately satisfied by current programming models. They typically offer only poor and unsafe support of concurrency, which has been added with

hindsight to the classical procedural model, instead of constituting a primary property of the programming paradigm.

Unfortunately, attempts at a sustainable solution to such fundamental problems have become increasingly seldom in our time, whereby prevalent main-stream languages are often considered as "definitive" foundations and "innovation" is becoming more and more limited to adding new features of questionable value and patches on a suboptimal model.

Our motivation is hence to overcome this situation, by rigorously revising the conceptual basis of current programming languages. We believe that with the right choice of a sufficiently general but still simple paradigm, the artificially caused problems of the prevalent programming systems could be inherently abandoned.

The aim of this dissertation was therefore to develop a new programming model, which enables more powerful and structured construction of concurrent software systems. The specific goals of the project were:

1. The invention of programming concepts, which solve the existing structural problems of pointers and improve the support of concurrency.
2. The design of a concrete programming language which integrates this model with appropriate concepts.
3. The implementation of the new language with an innovative runtime system.
4. The demonstration of the advantages of the new language and runtime system. An extensive case study should give evidence of the suitability of the language.

To fulfil these goals, we have designed and implemented a new programming language, which directly integrates a general and substantially novel concept of components. Three fundamental relations govern components in this language:

1. Hierarchical composition without use of explicit pointers
2. Symmetric connections with a dual concept of *offered* and *required* interfaces
3. Communication-based interactions

Designed for fully-fledged programming, the new language is solely based on components and features only high-level concepts for their implementation. Using a clear separation of interface and implementation, we also allow *terminal components* (which are not composed of other components) to be implemented in any other programming language. We thus gain the flexibility to also support any special (e.g. machine-close) implementations in our programming model.

1.2 Contributions

The main contributions of this dissertation can be summarised as follows:

1. *The description of a refined programming model and a new language for the structured construction o.fmodern concurrent programs.* Using a general notion of components, the language advances the state of the art in programming, by enabling (a) general hierarchical encapsulation, (b) accurate dynamic structuring without use of explicit pointers, (c) hierarchically controlled component lifetimes, (d) fIrst-class and safe concurrency, and (e) flexible polymorphism without loss of expressiveness and practicability.
2. *The implementation of an innovative runtime system for the efficient support of the new programming language.* The system excels in (a) a very high degree of concurrency, (b) low-cost parallel processes, (c) safe hierarchical memory management without use of a garbage collector,

and (d) the rigorous elimination of classical system artefacts, which have become redundant in our model (this includes virtual memory management). The system comprises a micro kernel that clearly outperforms existing systems with regard to execution speed and scalability of concurrency.

3. *The report on a case study with the new component language, demonstrating the practicability of the concepts and the system.* By means of a traffic simulation, we can show how the language conveys more natural modelling and can offer a higher performance than a classical solution. Thereby, we can also show that the new pointer-free structural concepts of the language provide sufficient expressiveness and flexibility for real programmmg.

1.3 Outline

The main part of this dissertation is organised in the following chapters:

Chapter 2 analyses the current state in programming languages and identifies the most fundamental problems.

Chapter 3 describes the new programming language and its underlying component notion.

Chapter 4 shows the conceptual advances of the new language by means of various examples.

Chapter 5 presents the new runtime system for the programming language.

Chapter 6 explains the technical advances of the runtime system, supported by experimental evaluations.

Chapter 7 reports on the case study with the language and system.

Chapter 8 concludes this work and identifies open research directions.

Chapter 2

State of the Art

The need for developing a new programming language is usually something that arises in circumstances of (1) a new type of application or (2) an unsatisfactory state of the art. This is also true in our case, where we find that today's predominant programming languages lack the adequate support for structured and parallel programming. In a thorough examination of the current programming languages, we have identified various fundamental problems that are summarised in this chapter. We thereby not only focus on object-orientation but also consider other important programming models as well as existent solution proposals for the recognised problems.

2.1 Object-Orientation

Without doubt, object-orientation [DN66] is currently the most popular programming paradigm in industry and research, being incOlporated in nearly all modem programming languages, such as C# [CS06], Java [GJS+OO], c++ [Strous98], Zonnon [GZ05], Active Oberon [Gut97, Mul02, Reali04], Eiffel [Meyer97], or Smalltalk [ST98]. Other interesting but less popular object-oriented languages were developed: SELF [US87] for example does not engage class-based object creation but only allows replicating objects from previous instances (so-called *proto-*

types). In fact, object-orientation was originally invented with Simula I [DN66] for the purpose of simulation programming. Due to this focus, the model proved to be very general, as it encouraged programmers to directly represent the concrete or abstract units of our natural world as analogous instances (with an intrinsic memory and behaviour) within a program. Unfortunately, the fundamental concept of self-active objects of Simula I and Simula 67 [DMN68] was not incorporated in succeeding popular object-oriented languages, limiting these languages to primarily sequential programming. Though widely regarded as a high-level paradigm, object-orientation is in general strongly influenced by the classical procedural model. Many concepts have been directly adopted from the old model though not always making sense in the new context of objects. In the following analysis, we identify (1) *references,* (2) *methods* and, (3) *inheritance* as three such elementary concepts that are too low-level and cause many fundamental problems in object-orientation:

2.1.1 References

References (or *pointers)* form semantically very weak constructs for describing relations between dynamically created object instances. Similar to "goto" jumps, arbitrary interlinking of object instances is promoted with references, leading to an object graph of non-hierarchical shapel. Though typed languages may impose restrictions on the type of referenced objects, it is possible to arrange arbitrary structures among different instances with a compatible object type. Clear program structures and general encapsulations remain unsupported: any abstraction that consists of a dynamic structure of sub-elements is not adequately representable as a hierarchically composed

---

1 C.A.R. Hoare also unequivocally criticises the unstructured nature of references and calls their introduction in high-level programming languages a step backwards [Hoare73, page 20].

object. Instead, this has to be forcibly modelled as a reference-linked conglomerate of elementary object instances, constituting an undifferentiated part in the overall and flat object graph.

As a consequence of these underspecified object relations, incautious reference copying may quickly lead to incorrect program dependencies (also known as *aliasing* problems [Hogg91, Alm97, NVP98, CPN98, MP01, AC04]). In turn, object exchangeability and reusability are also decisively impaired due to the implicit dependencies of outgoing object references which are unspecified in object interfaces[2]. A further negative artefact of references constitutes the need for automatic (and heavy-weighted) garbage collection for memory-safe runtime support[3]. Rather designed as a provisional solution to prevent dangling pointers and memory leaks, a garbage collector can naturally not relieve the programmer from caring about object disposal: The programmer still has to keep in mind that references to unneeded objects have to be explicitly set to *null,* if they are transitively reachable from a static root[4]. The programmer's control over the object lifecycle is artificially limited to the initialisation and main use phase, since the finalisation of objects can only be started by the collector and entails various conceptual problems[5].

2.1.2 Methods

Methods fail the realisation of a true message passing paradigm, as they in fact only constitute conventional pro-

---

2 Every element of public visibility in the object may be considered as part of the object's interface.

3 Garbage collection already involves particularly high complexity and costs in an application with somewhat time-critical execution [JL03].

4 Static roots are represented by *static fields* in Java, C#, or C++ and by *modules* in Active Oberon.

5 The invocation time and order of finalisation is undefined. Moreover, conceptual paradoxes such as *resurrection* have to be considered [RichOO).

cedures (with an implicit dynamic link[6] to the containing object). Contrary to a message passing abstraction, methods are directly executed by the invocating process. With regard to concurrency, methods are severe obstructions as they unnecessarily block the invocator during their entire execution, instead of running at the expense of the actual containing object. To diminish these limitations, "asynchronous method calls" are often introduced in languages. However, this concept is not more than a bulky patch on the classical procedural model instead of establishing a natural and general notion of message passing.

With (synchronous and asynchronous) method calls, an object is also incapable of maintaining an arbitrarily long statefull interaction with multiple clients individually. Instead, an object can only hold a client-specific context during a method invocation. The pattern of a method for a client-specific interaction is however oversimplified, having only one parameterised input followed by one possible output (with generally only one value). To work around this, one has to separately store the client state outside the object in the case of more complex interactions, if the object should not be unreasonably limited to one client interaction at the same time. For this purpose, an extra helper object has to be usually assigned to a client or the state has to be explicitly re-established by passing it as a parameter to the methods[7].

2.1.3 Inheritance

The main object-oriented mechanism for type polymorphism, which is known as inheritance and is strongly influenced by classical record extension, enforces an unnecessary (and often unjustified) hierarchisation and

---

6 The dynamic link is also known as the implicit SELF- or this-reference.

7 For example, the iteration over a data collection has to be usually realised by a client-associated *iterator* object, or a synthetic *rider* object that is exchanged between server and client.

classification of object types at compile-time. Unlike a non-hierarchical polymorphism, objects can not be represented by a set of equally important facets, without artificially preferring some facets as sub-types of others. Inheritance also unsuitably combines the two antagonistic concerns of polymorphism and code reuse, often resulting in mutual imports of different classes. A special object class, which needs to be inherited from a general class for the purpose of type polymorphism, should not be obligated to also inherit the general implementation of the super-class, as the special class' code is naturally more specific than that of the general class[8].

Moreover, inheritance is either confined to single-inheritance [CS06, GJS+OO] or may entail cumbersome conflict resolution in the case of multiple-inheritance support [Strous98, Meyer97]. Keeping in mind that types were initially invented to separate between disjoint sets of data values and the thereon applicable operations, it is unclear why multiple-inheritance should allow to specialise multiple base types, that were initially classified to be separate. It seems that this rather promotes imprudent classifications and unifications of types, without really gaining any conceptual benefit. Inheritance is also susceptible to dangerous encapsulation breaches. By permitting a sub-class to override methods of the base class[9], the implementation of a class may be altered (with or without intention) or even corrupted by the implementation of any existing or future sub-class (known as fragile base class problem[10]). Generally, the correct development of a sub-

---

8 The example of a rectangle and square shows this contradiction: a square is a geometrical special-case (modeled as a *sub-class)* of a rectangle but on the other hand, should not inherit the general rectangle implementation (with the two variables length and width).

9 To avoid accidental method overriding, C# requires a double-side agreement by the virtual and override keywords in the super- and sub-class respectively.

10 See also [Szy98, Section 7.2].

class requires a total knowledge of the base class implementation, which should however, not be exposed.

2.1.4 Concurrency

Besides the abovementioned problems, another significant shortcoming of object-orientation is the inadequate support of concurrency. While objects originally featured an intrinsic *activity* in Simula I [DN66] and Simula 67 [DMN68J, this important characteristic unfortunately disappeared over time and objects became reduced to only passive instances. This step backwards to sequential programming no longer allows programmers to model daily-life abstractions, which feature their own concurrent lifetime scenario (such as persons or vehicles)l1. Though modern languages like Active Oberon [Gut97, Reali04] and Zonnon [GZ05] revive the concept of self-active objects, the object notion was, since its invention, not designed to convey truly disentangled interactions between the autonomous and self-active instances. Instead, ordinary procedure calls are generally employed to describe object interactions, an old mechanism that was renamed *method invocation* or *message passing*. With this mechanism, the internal process of one active object to synchronously enter into the execution domain of arbitrarily other objects, such that the execution of a process is generally not only located within one instance.

These historical legacies are responsible for the rather poor support of concurrency encountered in today's mainstream languages. Instead of supporting a well-structured concurrency, current languages only offer *threads* that operate (from outside) on the passive object instances without any clear restriction or control. The fact, that the set of potentially (directly or indirectly) accessed objects of a thread are not explicitly specified in a program, is

---

[11] Not even Beta [MMN93], the most recent language from the same origin as Simula, incorporates the concept of object-centric activities.

particularly harmful since threads can only interact implicitly via shared objects. With such an approach, concurrency is obviously doomed to be unsafe, inherently prone to uncontrolled concurrency overlapping (race conditions) or hard-to-detect deadlocks. Due to the conceptual weakness, it is not surprising that languages often do not integrate threads as a fIrst-class programming concept but rather hide them as an extra feature in a separate programming library.

2.1.5 Existing Solution Approaches

Naturally, many researchers (and practitioners) have addressed the aforementioned problems of object-orientation and have proposed corresponding solutions. However, references, methods and inheritance are generally not considered as the root problems but these approaches rather focussed on improving specific negative implications of these concepts. In this section, we review the most important existing solution approaches in this research field.

2.1.5.1 Object Encapsulation Models

The missing support of hierarchical encapsulation of object structures and the associated problems of uncontrolled referencing represent an acute matter in current research.

A variety of *ownership models* [Hogg9l, Alm97, NVP98, CPN98, MPOl, BROl] have been invented, to allow the specification of ownership relations in classical reference-linked object structures. More concretely, an object may aggregate a set of objects as its ownership and prohibit external accesses to these instances. While earlier ownership models define this relation implicitly [Hogg9l] or associate it with types (Balloons [Ahn97]), modern ownership models [NVP98, CPN97, MPOI, BROI] use reference modifiers and type parameterisation, to specify and propagate ownership dependencies in a network of objects. Figure 2-1 to 2-3 give samples of such techniques but also indicate inconsistencies in certain ownership

models. The main problem of the ownership models is however that the mechanisms are too technically intricate, involving numerous different modifiers (such as rep, norep, readonly etc.) and many hard-to-understand and unnatural rules (mostly defined as a type system). The encapsulation of object structures is also not directly visible from the program code but only implicitly defined by the modifiers along a chain of references. As the models are often too restrictive, exceptions may alleviate the encapsulation. For example, read-only references [MPOl] may be allowed to break into an encapsulated structure and read internal data (though this data is probably encapsulated not to be read). Alternatively, references in procedures of external objects may be allowed to temporarily access protected objects (dynamic aliasing [Hogg9l, Alm97]). The most decisive disadvantage is however, that conventional unrestricted references still constitute the standard constructs in all these models, such that the majority of objects may nevertheless be exposed as part of the system-wide flat object graph.

A somewhat more flexible but still fairly similar method is engaged by ownership domains [AC04]. With this method, objects are allocated in openly declared domains, which own the contained objects and may grant explicit access rights to a selective set of other domains. However, this technique is also relatively cumbersome and optional. For instance, objects can still be allocated without policy in the globally shared region.

To enable at least a certain level of encapsulation, some programming languages permit the static aggregation of objects with value semantics. For this purpose, object types are classified into value and reference types (such as in C# [CS06], or Eiffel [Meyer97]) or the value or reference semantics are defined at the variable side (such as in Beta [MMN93] or C++ [Strous98]). This approach is naturally not general, since dynamically created objects or structures of reference-linked objects can not be encapsulated.

The ownership model of Noble, Vitek and Potter [NVP98]

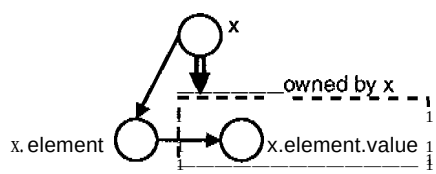
Summary:
A rep reference specifies that the target object is owned by the reference holder. This is however only true, if the rep mode is not associated with a class parameter (e.g. rep ElementType). Otherwise, the declaration denotes an external owner, which is defined by the argument of the class parameter. An owned instance can only be accessed by its owner and instances that also belong to this owner (ownership context). Owned instances can only refer to instances outside its ownership context with the arg mode, meaning that the instances are immutable for the referencing object.
Inconsistency:
It is for example not possible that the nodes of a linear list can be owned by the same list object. The template mechanism requires that the owner is specified for all nodes which potentially follow the first node.

```
class Elemenkrep ElementType> {
  rep ElementType next; /* ... */
}
class Collection {
  rep Element<rep Element<rep Element <...1...>>> first;
}
var Collection c;
```

The ownership model of Clarke, Potter and Noble [CPN98]

Summary:
Similar to the previous model, the rep attribute is used to specify that the current object figures as the owner of a referenced object. By using class parameters (e.g. dataOwner), the owner information can be propagated. The keyword norep declares a normal reference without ownership relation.
Inconsistency:
The exterior object x.element holds a reference to the encapsulated object x.element.value, although it is not the owner. The complicated type rules however ensure that x.element is only accessible via the owner x.

```
class ElemenkdataOwner> {
  dataOwner Data value;
}
class Intermediate {
  norep Element<rep> element;
}
norep Intermediate x;
```



Legend: 0 object → reference ⇒ owns-relation owned structure

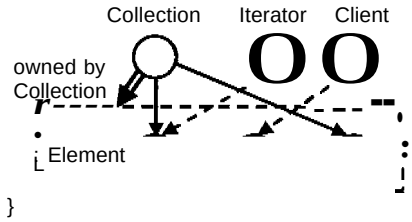
Figure 2-1. Examples of ownership models

The object universe model [MPOl]

Summary:
A rep reference denotes that the target object is owned by the current reference holder. The owner also indirectly possesses all objects that are transitively reachable via non-attributed references (peer references) from this directly owned instance. In this model, the set of objects that are directly or indirectly owned by an instance is called a *universe.* It is ensured that only the owner can have normal references going inside a universe, while normal references cannot exit a universe. In addition, readonly references are introduced to cross universe boundaries without restrictions. The semantic of such references is that they prevent modifying accesses to the referenced instances.

Inconsistency:
The model does not offer true encapsulation, as read-only references allow reading information from arbitrary objects, even if they are owned by another instance.

```
class Collection {
  rep Element first;
  Element last;
}
class Iterator { readonly Element current; }
class Element {
  Element next; // peer
  int value;
}
class Client { readonly Element lastelement; }
```



Legend: object reference owns-relation owned structure
read-only reference

Figure 2-2. Example of an ownership model

Ownership types for race-free programming [BROlI

Summary:
This model aims to economize synchronization for objects that can be only referenced by a single thread. This is the case if an object is directly or indirectly owned by a thread. For this purpose, the first class parameter at the variable declaration side defines the owner of the referenced object. The keyword self states that the object has no owner and is globally shared. The remaining class parameters are used to propagate ownership information.
Inconsistency:
The thread-local object system.device.config is referenced by the globally shared object system.device. However, the type system ensures that the object is only accessible by the same thread.

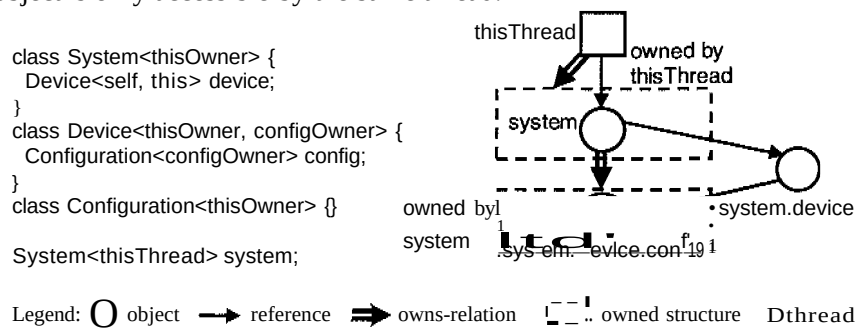


Figure 2-3. An ownership model for multi-threading

2.1.5.2 Object Lifetime Control

Other research focuses on the problem that objects can not be explicitly deleted in a program (without risk of memory errors), meaning that not the entire object lifecycle is under the programmer's control. As a resulting technical problem, the needed automatic garbage collection generally causes too high and unpredictable disruptions, which are unacceptable for time-critical applications [JL03].

A real-time garbage collector [Baker78, AEL88, Baker92, N093, CBOO, BCR03] could help in such a situation, even though the price in terms of execution performance, memory space reserve, and implementation complexity is substantial and the gained time guarantees

are relatively small[12]. Nevertheless, a programmer remains unable to clearly control the end of an object's lifetime.

As an alternative, *region-based programming* [BSB+03] can be engaged to reduce garbage collection overheads. The idea is to define explicit regions in a program, wherein the objects are allocated and managed. Regions (and their lifetimes) may be nested or hierarchically ordered, such that the objects can be directly deallocated at the end of a region's lifetime. This naturally requires that inter-region references do not lead to regions with shorter lifetimes. Region-based models have the same disadvantages as ownership models, which may also be used to control the lifetime of aggregated objects (supposing that the exceptions of read-only references and dynamic aliasing do not exist). On the one hand, the model is quite technically intricate for a programmer (see Figure 2-4). On the other hand, there are predefined global object spaces (typically a managed and an unmanaged one), where a lot of objects may be accumulated due to the restrictive rules of regions. As a particular danger, the unmanaged (uncollected) space may contain temporary objects which are not reclaimable anymore and thus produce memory leaks.

---

12 In the best case, the guaranteed worst-case execution time (so-called mutator utilisation rate) is about half as fast as in a system without collector [BCR03].

The unified ownership and region model [B8B+03]

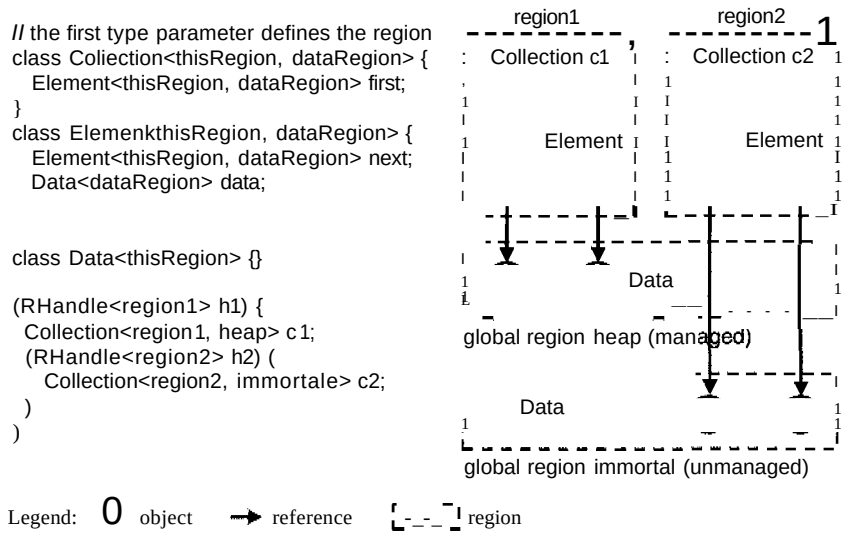


Figure 2-4. Region Model

2.1.5.3 Concurrency Improvement

A number of promising programming models could enable much better support of concurrency in object-orientation. For instance, the *active object model* [Gut97] of Active Oberon [AOS06, MuI02, Reali04] directly embeds concurrency in the object concept. Contrary to the classical approach of running separate threads on passive objects, active objects feature an intrinsic activity that is automatically started on the object creation and allows objects to act autonomously. This idea is similar to the original notion of objects as introduced in Simula I, in which objects were represented as processes [DN66]. To synchronise the object-centric processes, Active Oberon offers an inbuilt *monitor* concept [BH73, Hoare74] by supporting exclusive regions and system-monitored await conditions [BH73]. However, the processes can only interact implicitly by operating on shared objects. With the method-based execution, the dependencies between processes are generally not sufficiently specified, as they may (directly or indirectly) operate via method calls on arbitrary other objects.

A natural and clear description of interactions between active objects can be achieved by a real message passing paradigm, as known by CSP [Hoare78] or actors [Agha86]. In these models, *processes* (or *actors)* directly communicate with each other by sending and receiving messages according to a protocol[13]. The common problem of these models is however that an object *(process* or *actor)* can not communicate with each client individually, but has to handle the time-multiplexed communications of all clients simultaneously. The dialog concept of Zonnon [GZ05] gives a solution to this shortcoming: a client can explicitly invoke a new dialog with an object, such that a concurrent service agent runs at the server side and individually communicates with the client. However, Zonnon does not incorporate the dialog concept as the only possible model of object interaction. Instead, method calls are still supported as a fundamental concept, such that the aforementioned problems of methods and concurrency remain existent. Other languages have also integrated the communication paradigm, such as Occam [Occ88], Active C# [0004], Sing# [FA+06] or JCSP [Welch04]. A special case is the rendezvous-based communication in Ada95 [Barnes95] that is also based on CSP, though the notation of communication rather resembles method calls. Instead of sending and receiving messages in a bidirectional way, a client needs to call *task entries,* which have to be accepted at the server side (the *task).* Thereby, the client has to wait until the task entry has been accepted *(rendezvous)* and the body of the *accept* statement in the server has been completely executed. The latter is particularly unnatural for a message passing model as it is only necessary because a task entry may feature output parameters (similar to procedures). Alike other models, an Ada task does also not maintain a client-specific context in the presence of multi-

13 Contrary to the actors system, cSP uses synchronous communication channels, i.e. the send statement waits until the message has been received by the other side.

pIe concurrent clients, such that a task usually has to continuously accept any task entry in order to deal with all possible interleaving of client interactions.

The concurrency model of Polyphonic C# [BCF04] is based on a concept of matching synchronous and asynchronous method calls. More specifically, synchronous and asynchronous method declarations can be combined to a *chord*, such that the associated program body is only executed when all of these methods have been invoked. The typical scenario is that a synchronous method is declared in a chord together with one or multiple asynchronous methods[14]. In this case, the call of the synchronous method blocks as long as the corresponding asynchronous methods have not been called (and not been matched with another chord). Apparently, the asynchronous methods may be regarded as *events* that represent preconditions to run the procedure body of the chord. Consequently, the *signalling* of these events is realised by invoking the asynchronous methods. As the arguments of the asynchronous methods are directly passed to the chord, a chord allows direct information exchange similar to message passing without explicit synchronisation. However, a model of explicit wait & signal (such as monitors) as well as message passing would be more natural and clearer than implementing this paradigm by means of the rather technical concepts of procedure invocation and matching. As the precondition of a method execution is not defined with regard to the state of variables, the handling of a chord usually requires the launching the asynchronous methods for enabling the next chord matching. Therefore, the programmer often has to think in terms of a state machine (the asynchronous procedure calls enabling the transitions), usually leading to complex program logic.

---

14 However, a chord may also consist of only asynchronous or only synchronous method declarations.

To reduce concurrency problems such as race conditions and deadlocks in object-oriented programs, specific ownership models have been employed [BRO1, BLR02]. The idea is to require a lock for each globally shared object, if a procedure potentially accesses the object. As these models tend to accumulate objects in the conventional globally shared space, locking may be too coarse-grained and conservative (besides the complexity of these techniques). To avoid deadlocks, static lock levels have to be assigned and only acquired in a linear order. In such a model, fine-granular synchronisation of dynamic object structures remains unsupported (except specific object structures, for which static analysis may conservatively confirm an acyclic shape).

2.1.5.4 Inheritance Alternatives

The best (and most sustainable) solution to the problems of inheritance would certainly be the replacement of inheritance by a more elaborate concept. An interface support like in COM[15] [Box98, COM06] could enable much more flexible polymorphism for objects, without enforcing unnecessary type hierarchies. An object may implement an arbitrary set of interfaces, which all define equivalent external representations (often used like a type) of the object[16]. This approach also permits a clear separation of polymorphism and code reuse. An interface can be provided, without having to inherit a specific implementation of a base class.

In COM, code reuse can be directly realised by the general relation of aggregation. A component can expose interfaces of an aggregated class as interfaces of its own. The technical difficulty of this model is that not every

---

15 The fundamental ideas of COM are originated by A. Williams [Wil88, Wil90].

16 In COM, the designated home interface IUnknown however forms a *primus inter pares* with regard to the provided interfaces of a component.

component may be aggregated, since a component has to be explicitly prepared as such by its implementation[17]. Alike COM, Zonnon supports no inheritance but only interface polymorphism. For the purpose of reuse, Zonnon has an extra concept of *implementations* that define default interface implementations and can be aggregated as part of a class implementation.

Another mechanism of code reuse has been proposed by *mixins* [BC90], which provide partial implementations of interfaces within abstract classes. A class can then be mixed together by inheriting these abstract classes (assuming the support of multiple-inheritance). Compared to COM, mixins just form technical helper constructs for reuse that have no runtime meaning. Recently, *traits* [SD+03] have been advertised as another alternative for code reuse. As a supplement to inheritance, classes can be composed of traits, with each trait providing a set of methods by requiring the existence of other methods. As traits do not necessarily reflect a coherent implementation of a defined interface and the provided and required methods are automatically matched on name equality[18], this approach promotes rather unstructured merging of code fragments.

---

17 This is because the exposed interface must be implemented in such a way, that it can give information about all interfaces of its outer aggregating component.

18 For instance, if a method is added to a class, the required trait methods with the same name are automatically rebound to this new method. This happens even if the names are only accidentally equal.

2.2 Other Existing Programming Paradigms

2.2.1 Procedural Programming

A remarkable level of programming power has already been reached in classical procedural languages (like Algol60 [Naur60], Pascal [Wirth70], or Ada [Barnes80]). Based on a clear separation of data structures (stored in variables) and algorithms (procedures), these languages offer a powerful and simple programming paradigm that principally permits solving any programming task (although there may be a lack of a structuring concept at a larger program granularity).

For efficiency, concepts were typically chosen with a rather direct and simple representation to the underlying machine model. However, this approach also carried the risk that some concepts were selected as too low-level. Already in that time, pointers were considered as problematic [Hoare73], often leading to unstructured and error-prone interlinking of dynamically allocated data blocks and frequently provoking severe memory errors, if not managed by automatic garbage collection.

At the same time, many innovative concurrency concepts were invented for imperative programming languages [BH75]. One such elementary concept is the *parallel statement block* [Dij65], containing a set of statements that can all be executed in parallel but have to be completed at the end of the statement block. Using this construct, programmers do not have to over-specify a sequential execution for time-independent statements but can directly exploit fine-granular concurrency. Thereby, the constituents of a parallel block may interact over *critical regions* [Hoare?1], operating on explicitly declared shared resources *(shared variables* [BH72]).

However, hierarchically nested statement-level parallelism is not always suited as general notion of concurrency. Natural modelling (e.g. real-world simulations) may rather need *decentralised* concurrency, driven by

autonomously running and concurrently interacting *agents* (processes or objects). To describe such scenarios, *monitors* [BH73, Hoare74] (based on a module or object notion) and *communication-based interactions* [Hoare78] constitute fundamental concepts.

2.2.2 **Modular Programming**

For a clear structuring of programs at the level of the system, *modules* [Parnas72, Wirth77a, Wirth82, Wirth88] have proved to be a general and practical concept [WG89]. Statically, a module forms a container of logically coherent program declarations, grouped together to a system unit of compilation and deployment. The hierarchical *import*-relation thereby enables modules to reuse exported declarations of imported modules. At runtime, a module represents a singleton instance (with an own variable state) that is automatically loaded and managed by the system.

Whereas modules establish the sole statefull components in classical modular programming languages, modern such languages [AOS06, GZ05] typically institutionalise both modules and objects. This is because objects enable abstraction units at a smaller scale than modules. In contrast to modules, objects can have multiple instances and do not represent units of compilation, deployment, and system-managed loading. Due to the conceptual similarities of objects and modules, it remains however open as to whether modules and objects do not represent special cases of a common general *component* notion.

As for the discussed problem of underspecified reference structures, the hierarchical module structure can unfortunately not help. Even though an object class is declared in a certain module, its instances may be just as well be created (and used) in any other importing module. Therefore, an object does not necessarily belong to the declaration location of its class such that the object graph remains non-hierarchical. As a result of type polymorphism by sub-classes and generic pointers, objects can even cycli-

cally refer to each other, even though their classes may be declared in different and hierarchically ordered modules.

2.2.3 Functional and Logical Programming

Functional programming [MT+97, PJ02], in its pure sense, facilitates powerful descriptive programming by using hierarchical data values, functional composition, pattern matching, and type inference. The general type system allows the use of arbitrary (also recursive) hierarchical tree-like data values, constructible as a tuple, an alternative, or a list of another base type. A particular quality of the programming paradigm is the inherent ability for concurrency, as for the evaluation of function arguments. In accordance with the principle of descriptiveness, concurrency is not an explicit feature of language but can be internally exploited by the runtime system of the functional language.

As another example of descriptive programming, logical languages like Prolog [CR93, Pro195] allow the implicit specification of logical predicates which can be derived as true from a set of initial facts and rules, both defined in *horn clauses* (according to the *closed world assumption*, everything that can not be derived is considered as false).

Unfortunately, functional and logical programming languages have not become that widely accepted in practice. Their applicability is often considered as restricted, since a program only describes a solution without explicitly determining the computation process or the use of a storage space. This also forms the reason why such descriptive languages have often been diluted to factual imperative programming models, by introducing imperative elements such as side-effecting I10-operations[19]

---

19 As an alternative to side effects in functional languages, *monads* [Wad92] have been suggested to *describe* a sequential execution in terms of functions, i.e. the result value of a monad function is an executable imperative program. Though the descriptiveness of the language may be

or the *cut-operator* in Prolog. As a regrettable consequence, the specific underlying evaluation process (e.g. *eager* or *lazy evaluation* in functional languages or the concrete *unification algorithm* of Prolog) becomes decisive for the development of such programs and is frequently used to control the execution order of side effects.

2.3 Existing Component Models

Software components have been invented as a fundamental approach towards better software reusability and thus faster development of new programs [McIl68]. In recent decades, component-orientation has not only been acknowledged as a method for reusability but rather as a general abstraction for structured programming [Szy98]. The model conveys the continuous decomposition of program complexity in smaller building blocks called *components*, which have openly specified (and thus desirably minimised) dependencies. In fact, procedures, modules, and objects have evolved as most important concrete kinds of components. Conceptually, a component forms a reusable software unit, whose implementation should be completely encapsulated *(black box)* and whose external dependencies should all be defined in explicit *interfaces.* In general, components can be composed of other ones and can be instantiated (or replicated) multiple times by the use of a *builder tool* (based on a programmatic description or an interactive graphical system). As we have analysed before, not all species of components support this ideal equally well. For example, procedures may have implicit dependencies with the outer context (implicit use of variables of the outer static scope); objects may entail unspecified external dependencies by outgoing references; and modules are designed as singleton instances.

---

saved in theory, the character of a monad-based program remams imperative (namely describing a sequential execution).

2.3.1 **Component** Systems

For large-scaled industrial software construction, a lot of component systems have become popular. Microsoft COM [Box98, COM06], Java Beans [JB98], Enterprise Java Beans (ETB) [MHOO], CORBA [OMG98], as well as the Microsoft .NET framework [NET06] are but a few examples of such systems. In all these models, the components are not directly supported as an inbuilt language concept but have to be modelled by the use of specific programming frameworks or conventions. The component notion is rather defined by a standardised component representation and runtime infrastructure on the binary level (COM), on an intermediate language level (ETB and .NET), or simply, by naming conventions and explicit metadata on the programming language level (Java Beans). The main focus is thereby directed on universal deployment and interoperability, in particular the support of multiple languages, multiple platforms, or distributed computing.

In all of these models, the component notion corresponds to the conventional object concept, such that the same fundamental deficiencies with regard to references, methods, inheritance and concurrency exist. The only exception is COM, which does not integrate inheritance but instead engages symmetric interface polymorphism and component aggregation. Moreover, the COM wiring mechanism[20] with ingoing and outgoing interfaces could be seen an alternative to references. However, wiring is unfortunately only engaged for dual call directions of methods, as used for representing asynchronous events in the procedural model. Therefore, conventional references still establish the typical inter-component relations in COM and in fact, even a wire has to be represented by an explicit reference[21].

---

20 See [Szy98], Section 10.3.

21 A wire is established by registering a reference to an IConnection interface.

2.3.2 Architecture Description Languages

Architecture description languages (like UML[22], VHDL[23], and many others [GA094, BE+94, MD+95, LK+95, MQR95, SD+95, GMW97, MRT99, A1l97]) are commonly known to incorporate a more general component notion, where external dependencies of components indeed have to be rigorously specified in interfaces. Instead of using ordinary references, architecture description languages permit the wiring of component interfaces and hierarchical compositions. These languages do not form executable programming languages but are rather only designed for the formal description and specification of software and hardware architectures.

Unfortunately, all these languages have a common decisive limitation for practical use, namely that dynamic component structures can not be described. The number of components, as well as the entire wiring topology, is always static (like a single picture) and can not be defined at runtime. In some cases [MK96], the number of components may be fixed by a parameter, which also has to be statically defined on the (direct or indirect) use-side. Other architecture description languages [Med96, OMT98] allow scripts to perform dynamic updates on a statically specified compositions, by operating from outside on the internal structures of components. However, such an approach is not only awkward because it entirely differs from the usual notation of static compositions but also because it promotes the exposition of the encapsulated inner structure of a component.

Moreover, there is no agreement on a clear interface and wiring concept. Interfaces are generally represented as either collections of *exported* and *imported* methods (operations and events) [LK+95, SD+95 GMW97, MRT99,

---

22 See [OMG04], Sections 8.3 and 8.4.

23 See [VHOO], Section 5.

OMG04] or alternatively, as low-level communication *ports,* which are in most cases only primitive unidirectional data streams [GA094, BE+94, MQR95, SD+95, VHOO]. In the case of bidirectional communication ports [A1l97], each client typically requires a separate interface port for individual interaction, while a constant or a parameter fixes the number of ports. In addition, the wiring mechanism often necessitates a special construct of connectors to bind a set of ports by a particular "glue logic" [GA094, LK+95, SD+95, GMW97, A1l97, MRT99]. The concept of (implementable) connectors in these languages just unnecessarily raises the complexity, as a connector also forms a kind of a component which again maintains connections with components. In fact, these second-level connections between connectors and components are then not again represented as connectors.

2.3.3 Dataflow Languages

Dataflow languages [MSA+85, CPHP87, LabView, IW+88] have been invented to describe programs as an implicit or explicit network of data processing nodes. Descriptive dataflow languages such as SISAL [MSA+85] are typically based on the functional programming paradigm, where a program can be internally represented as a data flow graph. Thereby, any independency in the data flow graph inherently enables parallelism in the program. According to the principle of descriptiveness, a program does neither specify an explicit computation process nor does it feature an explicit variable state (that can be changed by assignments). However, this is also considered as a restriction by many practitioners, who like to directly control the execution process of their programs. Lustre [CPHP87] provides a concept of equations and functions (called *nodes)* with an inbuilt time axis. As a result, the programs describe a network of nodes that have a defined value in each time step. However, the functional nodes often remain quite low-level due to the elementary input and output values, such that programs often lack a

sufficient level of abstraction. Other data flow languages are visual [LabView, IW+88] and allow the construction of programs by means of interactive composition and wiring of processing nodes. Of course, such visual programs can only reflect a static picture, meaning that the program is forcibly limited to a static structure.

2.3.4 Component-Oriented Languages

Besides architecture description and dataflow languages, a few other, more practical programming languages [ACN02, GL+03, LS05] directly integrate a more general component notion than objects. The common problem of these languages is however that the inbuilt component concept still lacks sufficient elaboration.

While some languages [GL+03] can only describe static component structures, other languages [ACN02, LS05] remain too strongly influenced by object-orientation and still rely on ordinary references to express dynamic component structures. For instance, ArchJava [ACN02] can not guarantee encapsulation of hierarchically contained components, since a component may easily pass out references that lead to its sub-components. The support of multiple clients for an interface is also poor, requiring the programmer to explicitly request a reference to a new *port* from a so-called *port interface.* The recently presented language Classages [LS05] artificially distinguishes between three types of interfaces *(connectors, mixers, pluggers),* such that a component can only be hierarchically composed of sub-components which must not have connections among each other. Moreover, the interfaces in component-oriented languages are generally too low-level, featuring exported and imported methods for "inverse" event-driven programming.

Chapter 3

The New Component Language

In order to improve the current situation in the field of programming, we have developed a new component language for structured construction of concurrent software systems. The new programming language is called *Composita*[24] and has already been previously published in [Bläser06]. By means of innovation, the language features a directly integrated general component notion that is exclusively supported by expressive concepts. The highlights of the new language can be summarised as follows:

- *Hierarchical encapsulation*
  A component is able to *hierarchically encapsulate* any static or dynamic structures of components or program logic of arbitrary complexity, without the use of explicit references (or pointers).
- *Well-controlled structures*
  With a dual concept of *offered* and *required* interfaces, components can be connected to networks, whose structures are always exclusively controlled by the hierarchically surrounding component.

---

24 Latin: *lingua composita* - the well-structured language. In Latin, the adjective *compositus* is an established metonym for deliberate structuring. The noun Composita (neuter nominative plural of *compositus)* also characterises the programs written in this language, namely the well-structured / composed ones.

Components never have unspecified external dependencies.

- *Inherent concurrency*
  All components run fully autonomously and concurrently, only interacting via message communication.
- *Symmetric polymorphism*
  Components are represented by an arbitrary set of independent interfaces, each establishing an equally important external characterisation of the component. In total separation of implementation reuse, a new type system ensures the consistent handling of polymorphic components.
- *Implementation independence*
  Due to a clear separation of interfaces and implementation, the language also permits terminal components, which do not contain sub-components, to be implemented in any other programming language. This enables flexible and safe interoperability and leaves the programming model open for special-purpose implementations (such as machine-specific code).

Three principles led the design of the new programming language:

- *Simplicity*
  Provide a minimum set of most general concepts. The quality of the language is equally determined by what it provides and what it does not provide.
- *Expressiveness*
  Employ meaningful high-level concepts instead of limited machine-close abstractions. Do not unify different concerns in one concept and do not classify the same concern in different concepts.
- *Clarity*
  Use a syntax notation that is widely accepted for its clarity and readability.

Important inspirations for the new programming language came from COM [Wil88, Wil90, COM06] and CSP [Hoare78], as well as from Zonnon [GZ05], Active C# [GG04] and Oberon [Wirth88]. With this background, we introduced a series of new concepts (cf. Section 3.8), in particular the dynamic pointer-free structuring with components, the guaranteed and general hierarchical encapsulation, as well as the component interaction paradigm which is solely based on high-level message communication. The following presentation gives an overview of the programming language. The concepts are intentionally described in an informal way, to clearly focus on the essential ideas. For a more detailed and technical specification, the complete language report can be found in Appendix A.

3.1 The Component Concept

In the new language Composita, a program always is a component which can be constructed again from an assembly of components. The hierarchical composition can be arbitrarily continued in a recursive way. With this paradigm of stepwise refinement, complex systems can be built of program units that hide detailed logic from a higher abstraction level.

A *component* constitutes a closed program unit at runtime that encapsulates state (data values and components), as well as behaviour (interactions and functionality). It may be used to represent any kind of abstraction, like a subject (e.g. a person), an active object (e.g. a car), a passive object (e.g. a road) or an abstract notion (e.g. a route). Components are only allowed to have external program dependencies over explicitly defined interfaces. An *interface* represents an external facet of a component and thus establishes an explicit interaction point between the component and its outer environment. Each component *offers* an arbitrary number of own interfaces and also *requires* an

arbitrary number of foreign interfaces that belong to other external components.

By way of a first example, let us consider a standard house, which has the external facets of a residence and a parking space, requiring both electricity and water supplies from outside. The house may be represented as a component, which offers both a Residence and ParkingSpace interface. In addition, the house requires the foreign Electricity and Water interfaces from other external components. The house component is described by a *component template* called StandardHouse, that is shown in the program code below. The template specifies the offered and required interfaces, as well as the implementation of a house component. An arbitrary number of house *components* (called *component instances)* may be created from the same component template. One such possible instance of a house component is depicted by the diagram in Figure 3-1.

```
INTERFACE Residence; (* ... *)
INTERFACE ParkingSpace; (* ... *)
INTERFACE Electricity; (* ... *)
INTERFACE Water; (* ... *)

COMPONENT StandardHouse
  OFFERS Residence, ParkingSpace
  REQUIRES Electricity, Water;
  (* implementation *)
END StandardHouse;
```

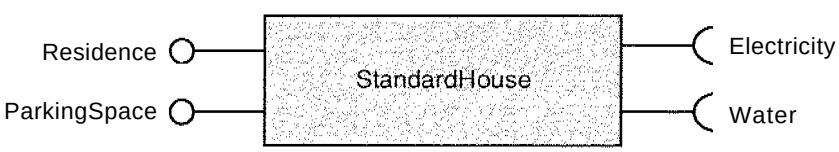


Figure 3-1. A component instance

Clearly, all interfaces of the component have equal rights, i.e. there is no artificially preferred interface. With regard to the example, this means that the characterisations of a

residence and parking space represent equally important facets of the house.

The component language supports three fundamental relations between components:

- *Hierarchical composition*
  Each component can contain an arbitrary assembly of other component instances, which are hierarchically encapsulated by the surrounding component.
- *Interface connections*
  An arbitrary network of components can be built by connecting the required interfaces of components to corresponding offered interfaces of other components. Each component exclusively governs the connections between its internal components.
- *Communication-based interactions*
  Components can interact via interfaces by bidirectional message exchange, based on symmetric sending and receiving of messages. An individual message communication channel is automatically maintained between a component, which offers an interface, and each component, which uses the interface.

As the component notion is designed to cover any conceivable abstraction unit and give higher generality than the classical component abstractions of objects and modules, components form the sole building units in the language.

3.2 Component Instances

All components potentially existing at runtime have to be declared in the program together with an identifier and a signature. The signature specifies permanent properties of the component, such as the offered and required interfaces.

As a *concrete signature,* one can specify the specific template of which the component is created. For example, house1 and house2 may be declared as two instances of the StandardHouse component template:

```
house1, house2: StandardHouse
```

In many cases, it is however necessary to declare component instances without statically fixing a specific template. Therefore, a component instance can be also declared by only postulating a set of offered and required interfaces for the component. The example below declares a building component with an *abstract signature* that lists the offered interfaces Residence and ParkingSpace and the required interfaces Electricity and Water.

```
building: ANY(Residence, ParkingSpace | Electricity, Water)
```

Using this declaration, the component instance can be of *any* component template that fulfils the following requirements:

1. The component template *offers at least* the interfaces which are postulated as offered by the declaration (i.e. Residence and ParkingSpace). These interfaces are always guaranteed to be provided by the declared component instance,
2. The component template *requires at I1Wst* the interfaces which are postulated as required by the declaration (i.e. Electricity and Water). These interfaces have to be provided by the environment of the declared component instance, before the component's offered interfaces can be used.

This dual rule of interface conformance statically ensures that interfaces can be only used if they are also offered by the component and, that a component does not maintain hidden external dependencies via undeclared required interfaces. Conversely, it does neither harm if the component offers more interfaces than declared (and used) nor, if fewer interfaces are required than effectively declared (and provided by the environment). Therefore, the following townHouse component may well be of the StandardHouse template. Conversely, the oldHouse component can not represent a StandardHouse as no required Electricity interface is postulated.

```
townHouse: ANY(Residence | Electricity, Water, CentralHeating);
oldHouse: ANY(Residence | Water)
```

A static declaration of component instances is not always applicable as in some cases, the number of component instances may be determined only at runtime. Hence, it is also possible to declare a dynamic *collection* of component instances with the same signature. An *index,* qualified by a list of comparable data values, thereby allows the dynamic identification of a component within the collection. For example, the following declaration defines a collection of components of the StandardHouse template, requiring a street number and name to identify an instance.

```
house[number: INTEGER; street: TEXT]: StandardHouse
```

With this declaration, the following component instances may be accessed for example.

```
house[12, "Market Street"]
house[3, "First Avenue"]
house[100, "Grand Boulevard"]
```

Of course, we may also define a collection of components without fixating their specific component template:

```
house[postalAddress: TEXT]: ANY(Residence | Water, Electricity)
```

3.3 Hierarchical Composition

A component can be hierarchically composed, by containing an arbitrary static or dynamic number of sub-components. The sub-components are fully encapsulated and exclusively managed by the sUlTounding component, such that the inner components are completely invisible and inaccessible outside the super-component.

The program below delineates a hierarchical composition with the example of a StandardHouse component, which contains a garage and two floors as sUb-components (see Figure 3-2). Variables enable hierarchical compositions by representing *separate containers,* within which a component instance with a compatible signature can be stored.

```
COMPONENT StandardHouse
  OFFERS Residence, ParkingSpace
  REQUIRES Electricity, Water;
VARIABLE
  garage: StandardGarage;
  groundFloor, firstFloor: ANY(Rooms | Electricity, Water);
BEGIN
  NEW(garage); NEW(groundFloor, Floor); NEW(firstFloor, Floor)
END StandardHouse;
```

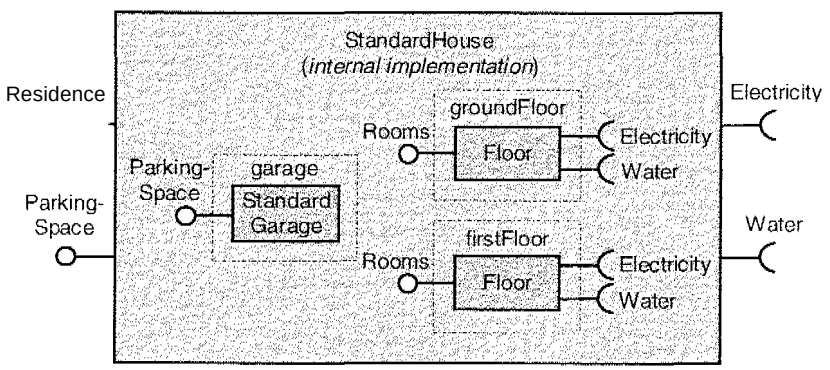


Figure 3-2. Hierarchical composition of components

As a variable is empty by default, a component has to be first installed within it before the variable can be used. This can be done by the NEW-statement, which creates a new

component within a variable. If an abstract signature is declared for the variable (ANY-construct), the component template has to be explicitly specified as second parameter of the NEW-statement (cf. the last two statements in the example above). In any case, a possibly pre-existent component in a variable is automatically deleted before a new component is installed in it.

Naturally, a variable is also capable of storing a dynamic collection of component instances:

```
VARIABLE room[number: INTEGER]: HotelRoom;
BEGIN
  FOR i := 1 TO N DO NEW(room[i]) END
```

The FOREACH-statement allows iterating over the elements of a collection, by assigning a valid index to the specified variable(s)in each iteration step. This is especially useful in a dynamic situation where it is not known at development time, which components all reside within a collectioll.

```
FOREACH i OF room DO use room[i] END
```

As variables are only locally defined in a program scope, they directly imply a hierarchical lifetime dependency between the surrounding component and the internal component instances. This means, if the surrounding component is disposed, the contained components are recursively deleted as well. A component in a variable may be also explicitly deleted before the automatic disposal on the end of the surrounding component. In addition, the language offers the EXISTS-function, which allows to determine whether a variable is empty.

```
VARIABLE oldHouse: ANY(House | Electricity, Water);
BEGIN
  IF EXISTS(oldHouse) THEN DELETE(oldHouse) END
```

It should be noted that variables really contain components and that there are no pointers (or references) involved at the level of programming language. Of course, dynamic com-

ponent structures need to be appropriately organised in the memory behind the scenes but this is no concern at the abstraction level of the programming language.

3.4 Component Networks

Components systematically decompose programs into separated logical parts, which only have explicitly defined dependencies in the form of offered and required interfaces. Networks of component instances can be built by connecting each required interface to one with an identical name which is offered by another component. The following example of a small city demonstrates the construction of such a network of component instances. Figure 3-3 visualises the resulting component network of the program.

```
COMPONENT HydroelectricPowerPlant
  OFFERS Electricity REQUIRES Water; (* ... *)
END HydroelectricPowerPlant

COMPONENT River OFFERS Water; (* ... *)
END River;

COMPONENT SmaliCity;
VARIABLE
  house1, house2: StandardHouse;
  powerPlant: HydroelectricPowerPlant;
  river1, river2: River;
BEGIN
  NEW(house1); NEW(house2); NEW(powerPlant);
  NEW(river1); NEW(river2);
  CONNECT(Water(house1), river1);
  CONNECT(Electricity(house1), powerPlant);
  CONNECT(Water(house2), river2);
  CONNECT(Electricity(house2), powerPlant);
  CONNECT(Water(powerPlant), river2)
END SmaliCity;
```

By means of the CONNECT-statement, the required Water interface of house1 is for example connected to the offered

Water interface of river1. The offered interface is thereby implied by the first argument of the statement.

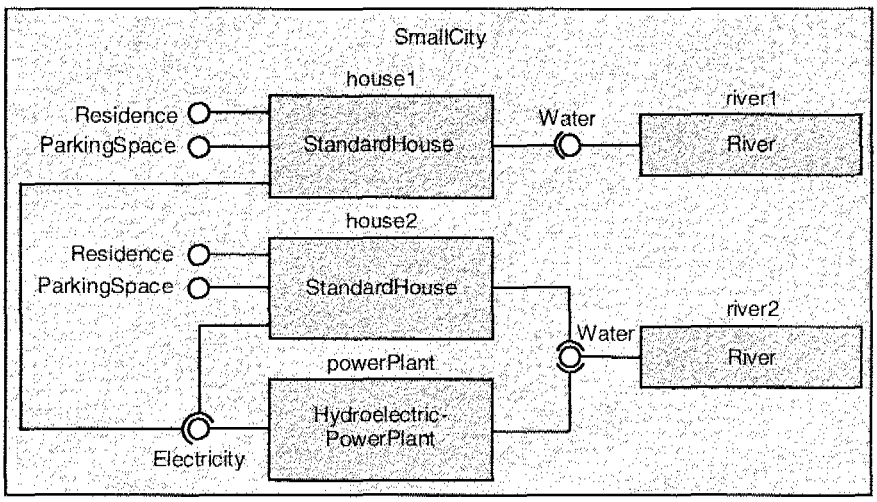


Figure 3-3. A static component network

Component networks can of course also be constructed with a dynamic number of component instances, as illustrated by the following program fragment and Figure 3-4.

```
COMPONENT City;
VARIABLE
 house[postalAddress: TEXT]: StandardHouse;
 powerPlant: HydroelectricPowerPlant;
 river[number: INTEGER]: River;
BEGIN
 FOR n := 1 TO N DO NEW(river[n]) END; (* N>=1 *)
 NEW(powerPlant); CONNECT(Water(powerPlant), river[1]);
 REPEAT
  location := postal address of the new house;
  NEW(house[location]);
  CONNECT(Electricity(house[location]), powerPlant);
  n := number of nearest river,
  CONNECT(Water(house[location]), river[n])
 UNTIL no free building site available
END City;
```

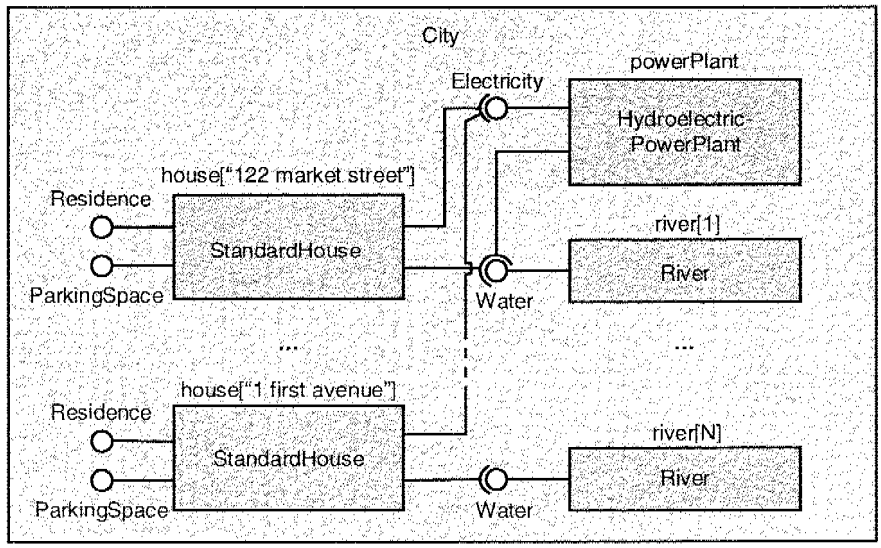


Figure 3-4. A dynamic component network

Furthermore, a component may also directly delegate the implementation of an own offercd external interface to one of its sub-components. For this purpose, an offered external interface (e.g. ParkingSpace of the StandardHouse below) can be connected to an offered interface with the same name that belongs to a sub-component (e.g. garage). This is possible because an interface that is externally offered by a component is regarded to represent a required interface plug inside the component. Analogously, a required interface of a sub-component (e.g. the Water interface of the groundFloor) is also connectable to a corresponding interface, which is required by the super-component from outside. The externally required interface of a component here represents an offered interface plug inside the component.

```
COMPONENT StandardHouse
  OFFERS Residence, ParkingSpace
  REQUIRES Electricity, Water;
VARIABLE
   garage: StandardGarage;
   groundFloor, firstFloor: ANY(Rooms | Electricity, Water);
BEGIN
  NEW(garage); NEW(groundFloor, Floor); NEW(firstFloor, Floor);
  CONNECT(parkingSpace, ParkingSpace(garage));
  CONNECT(Electricity(groundFloor), Electricity);
```

```
  CONNECT(Water(groundFloor), Water);
  CONNECT(Electricity(firstFloor), Electricity);
  CONNECT(Water(firstFloor), Water)
END StandardHouse;
```

Figure 3-5 depicts the corresponding redirected interfaces for the example above. As can be seen, hierarchical composition inherently enables implementation reuse. The StandardHouse component can be Oexibly built by integrating the existing StandardGarage implementation as a sub-component and by redirecting the ParkingSpace interface correspondingly.

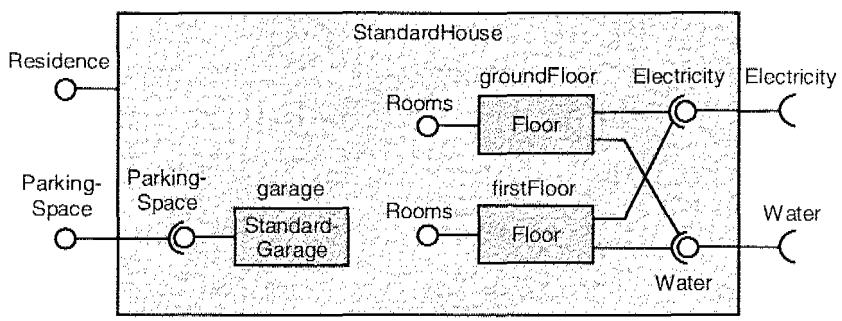


Figure 3-5. Redirected interfaces

By default, the declaration of a required interface means that a component requires an interface with that name of exactly one other component. In general, a component is however capable to also require interfaces with the same name from multiple other components. To do this, the number of required interfaces has to be specified with the corresponding declaration. The number is either static or defined in a dynamic range, where the maximum may be unbound using the star symbol. For example, a CompanyBuilding could require two or more Electricity interfaces for fault-tolerance, and exactly two Water connections.

```
COMPONENT CompanyBuilding
  OFFERS OfficeSpace, ParkingSpace
  REQUIRES Electricity [2..*], Water [2];
  (* ... *)
END CompanyBuilding;
```

As a consequence, we may integrate the new company building as follows in the city network (see also Figure 3-6):

```
COMPONENT City;
VARIABLE
  building: CompanyBuilding;
  powerPlant[number: INTEGER]: HydroelectricPowerPlant;
  river1, river2: River;
BEGIN
  construct the two rivers and N powerplants;
  NEW(building);
  FOR i := 1 TO N DO
    CONNECT(Electricity[i](building), powerPlant[i])
  END;
  CONNECT(Water[1](building), river1);
  CONNECT(Water[2](building), river2)
END City;
```

The different required interfaces have to be identified by an index, such as Water[1] and Water[2] and Electricity[1] to Electricity[N], where the number N of Electricity connections can be also determined at runtime by the function COUNT(Electricity(building)).

2

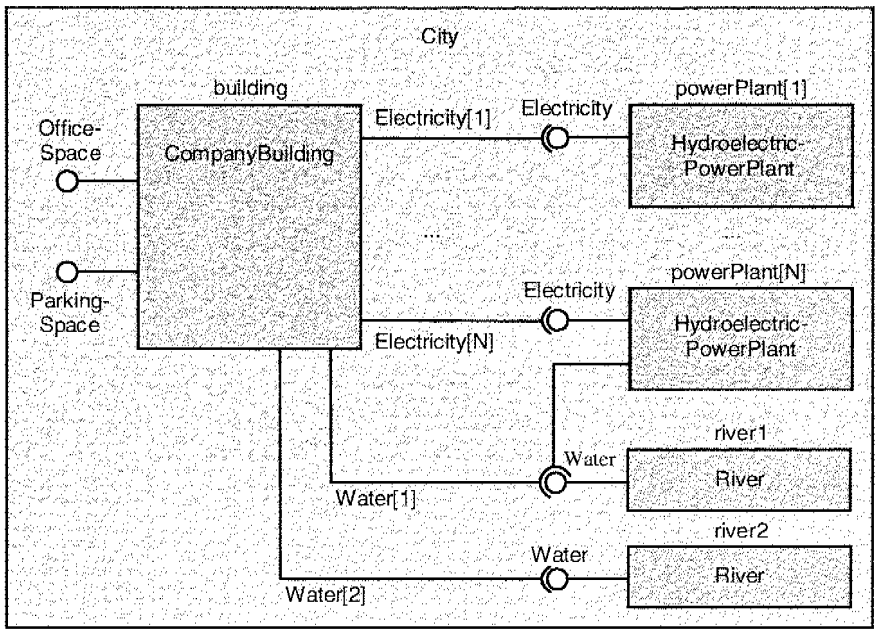


Figure 3-6. Multiple required interfaces with the same name

If a component is deleted explicitly, its *offered* interfaces have to be disconnected explicitly beforehand. As for the *required* interfaces, the DELETE-statement automatically disconnects them for the deleted component.

```
DISCONNECT(Electricity[N](building));
DELETE(powerPlant[N])
```

Although we can arrange arbitrary acyclic or cyclic component networks, the language ensures that a component network is always fully encapsulated by its surrounding component. The language hence supports hierarchies of arbitrary component networks, while classical pointer-oriented languages only have a flat runtime structure with a single object graph (cf. Section 1.1). This is due to the following two important conceptual distinctions between interface connections and classical references:

1. An interface connection constitutes a link which is only set and managed by the surrounding component, whereas a pointer (and a reference) forms

a data value that can be freely copied from one to another object.

2. An interface connection establishes a symmetric link between a required and an offered interface, whereas a pointer directly addresses a target object from the reference holder and may even not be visible outside the holder.

As will be shown, the pointer issue of ordinary programming languages is now overcome without loss of programming expressiveness. The examples in Section 3.7 show how the various topologies of component structures may be accurately described in our language.

3.5 Communication-Based Interactions

Interfaces enable general communication-based interactions between components. Two components, which are connected by a required and offered interface, can communicate over the interface by bidirectional message exchange. The feasible sequences of message transmissions during the communication have to be explicitly defined by a protocol in the interface. For example, the HotelService interface below describes the protocol for the communication between a component, which offers this interface, and an external component, which uses it (see the scenario in Figure 3-7).

```
INTERFACE HotelService;
 {
   IN CheckIn
   (
     OUT AssignedRoom(number: INTEGER)
     { IN EnterRoom IN ExitRoom }
     IN CheckOut OUT Bill(price: INTEGER)
     [ IN DirectPayment(m: Money) ]

     OUT FullyBooked
   )
 }
END HotelService;
```

A protocol has to be specifIed by a regular expression in the Extended Backus Naur Form (EBNF) lWirth77bJ25. Thereby, the symbols in the protocol denote messages exchanged during the communication. Each message has a declared transmission direction (either IN or OUT), an identifier (e.g. CheckIn), as well as an optional list of parameters (e.g. number). The IN-direction defines that a message is sent to the component offering the interface, whereas the OUT-direction characterises the opposite direction of transmission. According to this, the communication protocol of the HotelService interface can be understood as the temporal series of messages outlined in Figure 3-7. The parameters of a message represent data values or component instances that are transmitted within the message.

---

[25] In EBNF, a concatenation of expressions rcpresents a sequence, square brackets [1indicate an optional expression, curly brackets ( ) describe a repetition of zero or arhitrary times, and a vertical bar I denotes an altcrnative between two expressions. By default, concatenation has a stronger binding than an alternative. The default binding order can be explicitly changed with round brackets ( ).

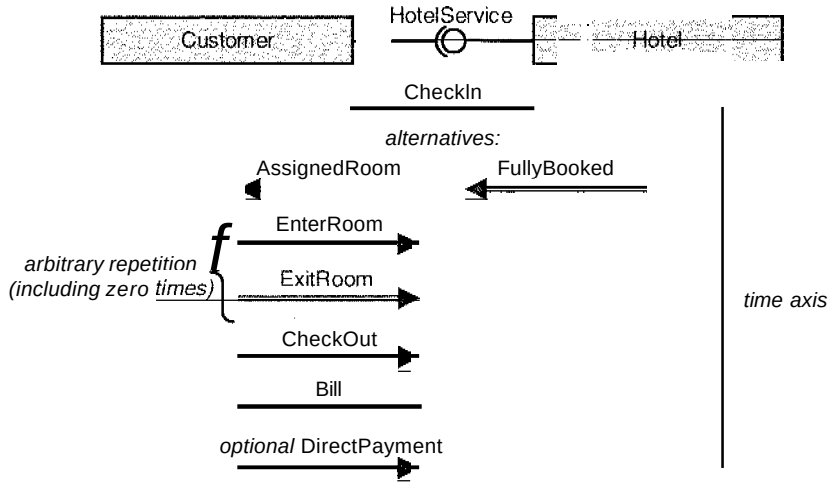


Figure 3-7. Message communication via an interface

An offered interface of a component can be used in parallel by all the components which are connected to the corresponding interface, as well as by the containing supercomponent itself. The component which offers the interface plays the role of the *server* of the interface, whereas the other components which use the interface act as *clients* of this interface. For each client of an interface, the server automatically maintains a separate communication channel. The server component automatically saves the state of the interaction for each client individually. Hence, multiple Customer components may simult<meously perform their individual hotel interactions, while they can be in different stages of communication with the same Hotel instance (see Figure 3-8). For each client, the hotel knows whether the hotel customer is already checked-in, what its assigned room number is, or whether he or she already left the hotel (with direct payment or not).

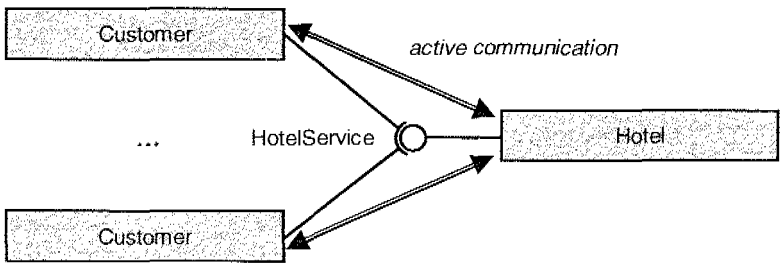


Figure 3-8. Multiple parallel client communications

The following program code sketches the implementation of a communication between a Customer and a Hotel component. The Customer component may directly communicate via its required interface, by sending and receiving messages. The Hotel component contains an implementation block for the offered HotelSeNice interface. This block is automatically incarnated as a separate *service process* for each client. It runs as an independent light-weighted process inside the server component and performs the server-side communication with the specific client. Variables may be associated with the implementation block, saving the individual client context in the service process. As multiple such processes may concurrently run within the same component, they may need to be mutually synchronised inside the component by means of monitor protection (explained later in Section 3.6).

```
COMPONENT Customer REQUIRES HotelService;
VARIABLE n: INTEGER;
BEGIN
  HotelService!CheckIn; (* send message *)
  IF HotelService?AssignedRoom THEN  (* receive-test *)
    HotelService?AssignedRoom(n) (* accept message *)
    (* ... *)
  ELSE (* fully booked *)
    HotelService?FullyBooked (* accept message *)
  END
END Customer;
```

```
COMPONENT Hotel OFFERS HotelService;
  IMPLEMENTATION HotelService; (* seNice process *)
  VARIABLE n: INTEGER;
  BEGIN
    WHILE ?CheckIn DO (* receive-test *)
      ?CheckIn; (* accept message *)
      IF free room THEN
        !AssignedRoom(n) (* send message *)
        (* ... *)
      ELSE !FullyBooked
      END
    END
  END HotelService;
END Hotel;
```

The *send statement,* denoted with"!", transmits a message to the other communication side, filling the message with the specified arguments. The message transmission is asynchronous, i.e. the send statement does not need to wait for the reception of the message by the other side. Hence, the execution can (but does not have to) immediately continue after sending. Whereas data values (of type INTEGER, TEXT etc.) are always sent *by copying,* component instances are always transmitted *by rnoving.* This means that the components are removed from the sender side and are delivered as disconnected instances to the receiver side. This enables dynamic exchange of component such as the plug-in scenarios described in Section 4.3. If a component is specified as message argument of a send statement, the statement first awaits the termination of the component's communications, then disconnects its required interfaces, and eventually removes the component from its variable.

Conversely, the *receive statement,* denoted with "?", awaits the arrival of a specific message from the other communication side and accepts the message on alTival. The contained component instances and data values of the received message are eventually assigned to the corresponding variables, which are specified as parameter arguments. (A possible previous content of a variable is thereby automatically deleted.) A receive statement blocks the execution as long as the message is not received.

Furthermore, the *receive-test function,* an expression denoted with '"1", allows testing whether any or a specific message can be received from a specific interface by first awaiting any message input. The function hence blocks the execution until the arrival of any message from the interface but does not yet accept the message nor assign the message parameters. It should be noted that a receive-test function is uniquely distinguishable from a receive-statement, because it forms a syntactical expression and not a statement. In addition, there is also a non-blocking INPUT-function to check the arrival of any or a specific message.

Within the implementation block, the send- and receive-statements without specified interface directly refer to the corresponding client, which is served by the block. Conversely, for the communication in the role of a client, the interface has to be specified (see Table 3-1).

effect	*syntactical construct*	*server-side communication*	*client-side communication*
send message	statement	!message(x, ...)	interiace!message(x, ...)
receive messaae	statement	?message(x, ...)	interface?message(x, ...)
blocking receive test	expression	?message	interiace?message
non-blocking receive test	expression	INPUT(message)	IN PUT(interface, message)

Table 3-1. Communication statements and functions

Obviously, a required interface of a component has to be connected before communication can be initiated via it. T'herefore, the client-side communication commands first await the establishment of the corresponding connection. Analogously, an interface can only be disconnected when it has no open communications.

During a communication between a client and server, all messages have to be sent and received according to the defined protocol[26]. The runtime system is in charge of monitoring the fulfilment of the protocol. When a client is disconnected from a component, the implicit **FINISH** message (without parameters), is automatically delivered to the server side and optionally accepted by the server.

3.6 Concurrency

Components figure as autonomous instances in the program that feature their own intrinsic behaviour by means of the internally running processes[27]. The language predefines a simple pattern of a lifecyc1e for a component, consisting of three st.ages: initialisation, main activity, and finalisation[28]. For initialisat.ion and finalisation, processes can be defined in the component body, running after t.he creation and before the disposal of the component, respectively. The service processes for the offered interfaces only run during the main activit.y, i.e. after initialisat.ion and before finalisation. The TERMINATED-funct.ion may be used during the main activity to decide whether the component should be finalised and the main activit.y should stop. This is the case when the component is to be deleted (on decision of t.he

---

[26] There exist protocols according to which both sides could bc senders at a certain time of communication, for example "[IN Request] OUT Response" or "{ IN Order} OUT Delivery". It is the task of the programmer to design the communication in such a way, that the sender side can be decided at runtime.

[27] Throughout this thesis, the term *process* means a parallel execution instance in its gencral sense. No spccific implementation (like threads or isolated UNIX processes) is prejudiced with this term.

[28] The time of finalisation is dcfined by the hierarchical lifetime dependency of composition. Before a componcnt is de leted, its own finaliser is first executed and then, all inner components are recursively finalised in parallel and deleted. Object-oriented problems like resurrection or undetermined finalisation times and orders IRichOO] are thus abandoned.

hierarchically surrounding instance) and all its internal service processes are terminated.

```
COMPONENT Hotel
  OFFERS HotelService
  REQUIRES Electricity, Water;
VARIABLE
  room[number: INTEGER]: Room; i: INTEGER;
BEGIN (* initialisation *)
  FOR i := 0 TO 100 DO NEW(room[iJ) END
ACTIVITY (* main activity *)
  WHILE ~TERMINATEDO DO
    FOR i := 0 TO 100 DO room[i]!Cleanup END
  END
FINALLY (* finalisation *)
  Water!StopConsumption; Electricity!StopConsumption
END HotelService;
```

Of course, the parallel execution of serVIce processes generally necessitates appropriate concurrency control within a component instance, to synchronise accesses to shared interior resources. For this purpose, monitor-oriented concurrency control is supported inside a component instance. Clcarly, monitor synchronisation is only applied within a component's implementation, whereas all interactions between different components are inherently synchronised due to the communication paradigm. Specifically, the EXCLUSIVE and SHARED attributes can be associated to any compound statement block (BEGIN, DO, THEN, or REPEAT) or expression, in order to establish an exclusive or shared *monitor lock* on the component for the execution of the corresponding program region. Whereas only one exclusive region can be executed at the same time in the same component, shared regions can run in parallel and are only mutually barred against exclusive regions of the component. All code regions operating on the component state have to be protected by a shared or an exclusive attribute. Exclusive protection is required if the code may modify the component state. The compiler ensures that all processes are sufficiently synchronised inside the component scope, see Section 4.6.2.

```
INTERFACE HotelReservation;
 { IN AskAvailability OUT FreeRooms(number: INTEGER)
 | IN Book(nofRooms: INTEGER) ( OUT Done IOUT Failed) }
END HotelReservation;

COMPONENT Hotel
 OFFERS HotelService, HotelReservation;
 (* ... *)
 VARIABLE freeRooms: INTEGER;

 IMPLEMENTATION HotelReservation;
 VARIABLE n; INTEGER;
 BEGIN
  WHILE ?AskAvailability OR ?Reserve DO
   IF ?AskAvailability THEN {SHARED}
    ?AskAvailability; !FreeRooms(freeRooms)
   ELSE ?Reserve THEN {EXCLUSIVE}
    ?Reserve(n);
    IF freeRooms >= n THEN
     freeRooms := freeRooms - n; !Done
    ELSE!Failed END
   END
  END
 END HotelReservation;

 BEGIN freeRooms;= 100
END Hotel;
```

Similar to Active Oberon tGut97, Reali04], an AWAIT-statement [BH73] can be used within a protected (exclusive or shared) region, in order to define a Boolean condition that has to be true before the execution continues (see the example in Section 3.7.1). For this purpose, the AWAIT-statement suspends the execution of the current process as long as the condition is not fulfilled. As usual, the AWAIT-statement temporarily releases the monitor lock while waiting, letting other processes fulfil the condition. The waiting process is automatically reactivated when the condition is fulfilled.

3.7 Examples

The following examples demonstrate how typical *design patterns* of concurrency29 can be described in the new programming language.

3.7.1 Producer-Consumer

The first example is a *producer-consumer* scenario, where producers and consumers concurrently interact with a common buffer. Producers thereby create data elements and put them into the buffer, whereas consumers need these items and take them out again.

```
COMPONENT Producer REQUIRES DataAcceptor;
  VARIABLE i: INTEGER;
  BEGIN
    FOR i := 1 TO 100000 DO DataAcceptor!Element(i) END
END Producer;

INTERFACE DataAcceptor;
  { IN Element(x: INTEGER) }
END DataAcceptor;

COMPONENT Consumer REQUIRES DataSource;
  VARIABLE i: INTEGER;
  BEGIN
    WHILE DataSource?Element DO DataSource?Element(i) END
END Consumer;

INTERFACE DataSource;
  { OUT Element(x: INTEGER) }
END DataSource;
```

29 Naturally, we do not show how to model classical data structures (linear lists, trees, hash tables etc.), as this is already covered by the inbuilt concept of collections (cL 3.2). A programmer can hence directly address the solution of a real problem, without having to deal with such low-level memory structures.

```
COMPONENT BoundedBuffer
 OFFERS DataAcceptor, DataSource;
 CONSTANT Capacity = 10;
 VARIABLE
  a[position: INTEGER]: INTEGER; (* circular buffer *)
  first, last: INTEGER; finished: BOOLEAN;

 IMPLEMENTATION DataAcceptor;
  BEGIN
   WHILE ?Element DO {EXCLUSIVE}
    AWAIT(last - first < Capacity);
    ?Element(a[last MOD Capacity]); INC(last)
   END;
   BEGIN {EXCLUSIVE} finished := TRUE END
 END DataAcceptor;

 IMPLEMENTATION DataSource;
  BEGIN
   REPEAT {EXCLUSIVE}
    AWAIT((first < last) OR finished);
    IF first < last THEN
     !Element(a[first MOD Capacity]); INC(first)
    END
   UNTIL finished
 END DataSource;

 BEGIN first := 0; last := 0; finished := FALSE
END BoundedBuffer;
```

The consumer-producer program may now be set up as follows (see Figure 3-9):

```
COMPONENT Simulation;
VARIABLE
  buffer: BoundedBuffer;
  producer: Producer;
  consumer: Consumer;
BEGIN
 NEW(buffer); NEW(producer); NEW(consumer);
 CONNECT(DataAcceptor(producer), buffer);
 CONNECT(DataSource(consumer), buffer)
END Simulation;
```

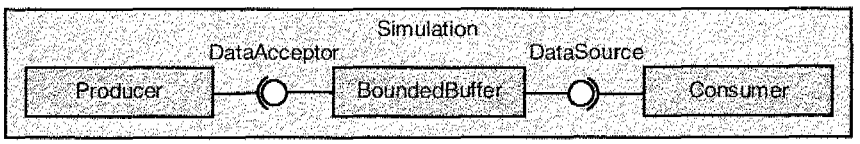


Figure 3-9. Simple producer-consumer scenario

Producer and consumer immediately begin to interact with the buffer, when the Simulation component is created and the internal components have been properly connected. Of course, one can also connect multiple producers and multiple consumers to the same buffer (see Figure 3-10):

```
COMPONENT Simulation;
VARIABLE
  buffer: BoundedBuffer;
  producer[number: INTEGER]: Producer;
  consumer[number: INTEGER]: Consumer;
  i, N, M: INTEGER;
BEGIN (* set N and M *)
  NEW(buffer);
  FOR i := 1 TO N DO
    NEW(producer[i]);
    CONNECT(DataAcceptor(producer[i]), buffer)
  END;
  FOR i := 1 TO M DO
    NEW(consumer[i]);
    CONNECT(DataSource(consumer[i]), buffer)
  END
END Simulation;
```

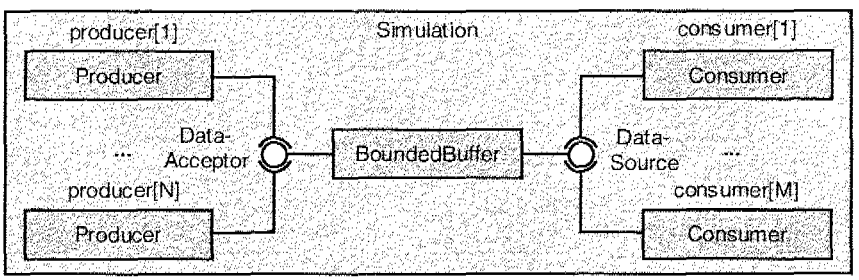


Figure 3-10. General producer-consumer scenario

3.7.2 Pipeline

Another elementary pattern is a *parallel pipeline,* in which data is delivered through a series of connected concurrent processing units. A popular example is the *Sieve of Eratosthenes,* in which prime numbers are computed via a series of Sieve components. Each Sieve component concurrently filters out multiples of an initial prime number, obtained from a preceding Sieve instance. At the beginning of the parallel pipeline, a special NumberGenerator component produces natural numbers starting from 2.

```
INTERFACE NumberStream;
 { OUT Number(x: INTEGER) } OUT Finished
END NumberStream;

INTERFACE Prime;
 OUT PrimeNumber(x: INTEGER) IOUT Finished
END Prime;

COMPONENT Sieve
  OFFERS NumberStream, Prime
  REQUIRES NumberStream;

  VARIABLE prime: INTEGER; finished: BOOLEAN;

  IMPLEMENTATION Prime;
  BEGIN {EXCLUSIVE}
    IF ~finished THEN !PrimeNumber(prime) ELSE !Finished END
  END Prime;

  IMPLEMENTATION NumberStream;
  VARIABLE i: INTEGER;
  BEGIN {EXCLUSIVE}
    WHILE NumberStream?Number DO
      NumberStream?Number(i);
      IF i MOD prime # 0 THEN !Number(i) END
    END;
    IF NumberStream?Finished THEN
       NumberStream?Finished
    END;
    !Finished
  END NumberStream;

 BEGIN
  finished := NumberStream?Finished;
```

```
    IF finished THEN NumberStream?Finished
    ELSE NumberStream?Number(prime)
    END
END Sieve;

COMPONENT NumberGenerator OFFERS NumberStream;
 IMPLEMENTATION NumberStream;
 VARIABLE i: INTEGER;
 BEGIN FOR i := 2 TO N DO !Number(i) END; !Finished
 END NumberStream;
END NumberGenerator;
```

The parallel pipeline may now be dynamically constructed, by extending the chain of Sieve components whenever a new prime number is obtained. The corresponding component structure of the Sieve of Eratosthenes is sketched in Figure 3-11.

```
COMPONENT Eratosthenes;
VARIABLE
  generator: NumberGenerator;
  sieve[i: INTEGER]: Sieve;
  i, p: INTEGER;
BEGIN
 NEW(generator);
 i := 1; NEW(sieve[i]);
 CONNECT(NumberStream(sieve[i]), generator);
 WHILE Prime(sieve[i])?PrimeNumber DO
   Prime(sieve[i])?PrimeNumber(p);
   WRITE(p); WRITE(" ");
   INC(i); NEW(sieve[i]);
   CONNECT(NumberStream(sieve[i]), sieve[i-1])
 END;
 Prime(sieve[i])?Finished
END Eratosthenes;
```

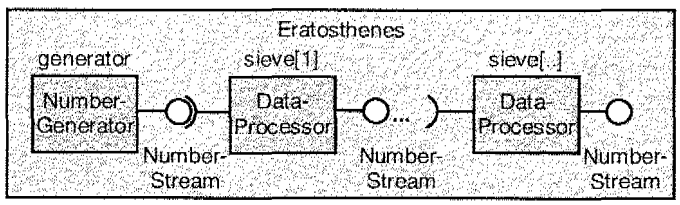


Figure 3-11. Parallel Sieve of Eratosthenes

3.7.3 Client-Server

A *client-server* model is a frequent software pattern, where an arbitrary number of clients generally use a common server in parallel, as sketched below and by Figure 3-12:

```
COMPONENT Server OFFERS Service;
 IMPLEMENTATION Service;
  BEGIN WHILE ?Request DO !Response END
 END Service;
END Server;

COMPONENT Client REQUIRES Service;
 ACTIVITY
  REPEAT Service!Request; Service?Response
  UNTIL TERMINATEDO
END Client;

INTERFACE Service;
 { IN Request OUT Response}
END Service;

COMPONENT System;
 VARIABLE
  server: Server;
  client[i: INTEGER]: Client;
  i, N: INTEGER;
 BEGIN (* set N *)
  NEW(server);
  FOR i := 1 TO N DO
   NEW(client[i]); CONNECT(Service(client[i]), server)
  END
END System;
```

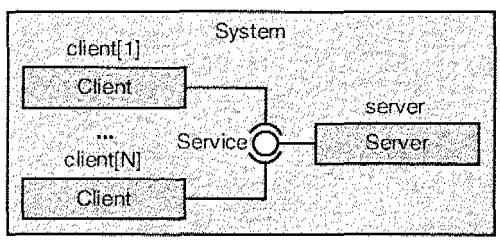


Figure 3-12. Client-server architecture

Apparently, a server component supports the concurrent and individual service of all clients, regardless of the com-

plexity of the protocol. In more sophisticated systems, a client may also use different servers (see Figure 3-13):

```
COMPONENT Client REQUIRES Service [1 .. *];
 VARIABLE k, m: INTEGER;
 ACTIVITY
  REPEAT
   k := RANDOM(1, COUNT(Service));
   (* random selection of a service interface *)
   Service[k]!Request END;
   Service[k]?Response END
  UNTIL TERMINATEDO
END Client;

COMPONENT System;
 VARIABLE
  server[m: INTEGER]: Server;
  client[n: INTEGER]: Client;
  i, k, N, M: INTEGER;
 BEGIN (* set Nand M *)
  FOR k := 1 TO M DO NEW(server[k]) END;
  FOR i := 1 TO N DO
   NEW(client[i]);
   FOR k := 1 TO M DO
    CONNECT(Service[k](client[i]), server[k])
   END
  END
END System;
```

It is noteworthy that client components immediately start to run after their creation, although the interface connections are dynamically added later. Therefore, the number of required Service interfaces may dynamically grow during the activity of a Client. However, the COUNT-function always returns at least the specified minimum number of connections, which is one in this program. In situations, where the number of connections should not dynamically grow, the exact number of connections may be explicitly specified from the outer instance by sending a message to the cOlTesponding sub-component.

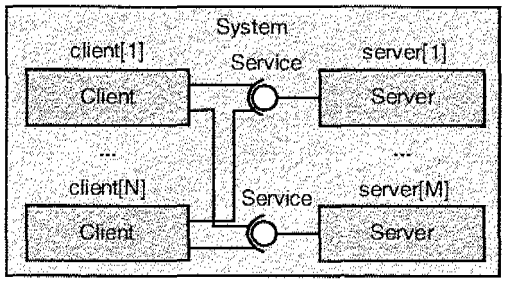


Figure 3-13. Clients using multiple servers

3.7.4 Divide-and-Conquer

The metaphor of *divide-and-conquer* is a useful computing pattern, especially in concurrent systems. The idea is to solve a complex task by splitting it into different sub-tasks, which are then solved in parallel. The results of the sub-tasks are eventually combined to yield the final result. A possible example is the computation of the *Mandelbrot fractal*, for which the fractal plane can be split into disjoint regions that are computed independently and concurrently.

```
INTERFACE MandelbrotSet;
  IN Start(N, M: INTEGER; xO, yO: REAL)
  {OUT Value(x, y: INTEGER; zreal, zimag: REAL) } OUT Finish
END MandelbrotSet;

COMPONENT MandelbrotTask OFFERS MandelbrotSet;
  IMPLEMENTATION MandelbrotSet;
  VARIABLE
    zreal[x, y: INTEGER]: REAL; zimag[x, y: INTEGER]: REAL;
    N, M, x, y, k: INTEGER; xo, yO: REAL;
    zreal2, zimag2, zri, creal, cimag, a, b: REAL;
  BEGIN
    ?Start(N, M, xO, yO);
    FOR x := 0 TO N-1 DO
      FOR y:= 0 TO M-1 DO
        zreal[x, y] := 0.0; zimag[x, y] := 0.0
      END
    END;
    FOR k := 1 TO Iterations DO
      FOR x := 0 TO N-1 DO
        FOR Y := 0 TO M-1 DO
          zreal2 := zreal[x, y] * zreal[x, y];
          zimag2 := zimag[x, y] * zimag[x, y];
          IF SQRT(zrea12 + zimag2) <= 2.0 THEN
```

```
        creal := xO + Resolution * x; cimag := yO + Resolution * y;
        zri := zreal[x, y] * zimag[x, y];
        a := zreal2 - zimag2 + creal; b := 2 * zri + cimag;
        zreal[x, y] := a; zimag[x, y] := b
       END
      END
     END
    END;
    FOR x := 0 TO N-1 DO
     FOR Y := 0 TO M-1 DO
      !Value(x, y, zreal[x, y], zimag[x, y])
     END
    END;
    !Finish
  END MandelbrotSet;
END MandelbrotTask;
```

The partitioning of the plane and the execution of Mandelbrot sub-tasks can then be organised as below.

```
COMPONENT Mandelbrot;
CONSTANT
 Iterations = 5000; H = 10; V = 10; Resolution = 0.05;
 Left = -2.0; Top = -1.0; Right = 2.0; Bottom = 1.0;
VARIABLE
 task[i, j: INTEGER]: MandelbrotTask;
 zreal[x, y: INTEGER]: REAL; zimag[x, y: INTEGER]: REAL;
 N, M, i, j, x, y: INTEGER; L, T, a, b: REAL;
BEGIN
 N := INTEGER((Right-Left) / (Resolution * HorizontaITasks));
 M := INTEGER((Bottom-Top) / (Resolution * VerticaITasks));
 FORi:=OTOH-1 DO
  FORj := 0 TO V-1 DO
   L := Left + i * N * Resolution; T := Top + j * M * Resolution;
   NEW(task[i, j]); MandelbrotSet(task[i, j])!Start(N, M, L, T)
  END
 END;
 FOR i := 0 TO H-1 DO
  FORj :=OTOV-1 DO
   WHILE task[i, n?Value DO
    MandelbrotSet(task[i, j))?Value(x, y, a, b);
    zreal[i * N + x, j * M + y] := a;
    zimag[i* N + x, j * M + y] := b
   END;
   MandelbrotSet(task[i, j])?Finish
  END
 END
END Mandelbrot;
```

Figure 3-14 illustrates the corresponding hierarchical task stTucture.

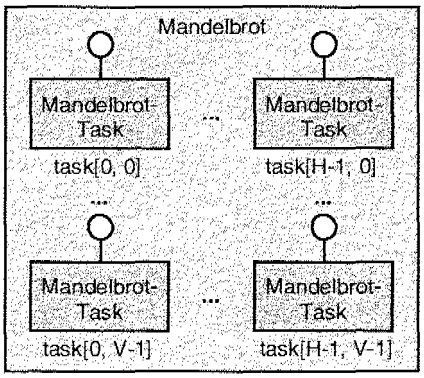


Figure 3-14. Two-dimensional partitioning of the Mandelbrot fractal

3.7.5 Token **Ring**

Another interesting concurrent pattern is a *ring* of connected components, which let a designated item circulate around. The item, typically called *token,* represents a specific status that only one participant ought to own at the same time. Due to the cyclic passing, the system inherently guarantees the fairness that every instance receives the token equally often. The code for such a ring topology goes as follows (see also Figure 3-15):

```
INTERFACE Partner;
 { IN Token} IN Finish
END Partner;

COMPONENT Player OFFERS Partner REQUIRES Partner;
 VARIABLE finished: BOOLEAN;

 IMPLEMENTATION Partner;
  BEGIN
   WHILE ?Token DO {EXCLUSIVE}
    ?Token; IF ~finished THEN Partner!Token END
   END;
   BEGIN {EXCLUSIVE}
    ?Finish; IF -finished THEN Partner!Finish END;
    finished := TRUE
```

```
    END
  END Partner;

  BEGIN finished := FALSE
END Player;

COMPONENT Ring;
  VARIABLE
    player[number: INTEGER]: Player;
    i, N: INTEGER;
  BEGIN (* set N *)
    FOR i := 1 TO N DO NEW(player[i]) END;
    FOR i := 1 TO N DO
      CONNECT(Partner(player[i]), player[i MOD N + 1])
    END;
    Partner(player[1])!Token (* inject token *)
  FINALLY
    Partner(player[1])!Finish
END Ring;
```

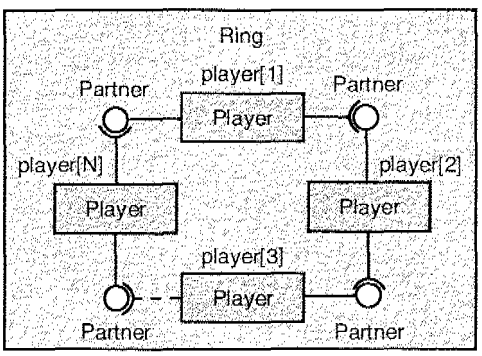


Figure 3-15. Token ring

In the program above, the outer Ring component initiates the cyclic flow by handing the token to one of the Player components. It is furthermore noteworthy that the terrnination of the cyclic flow is also clearly planned. On deletion of the Ring component, the finalisation process causes an arbitrary Player to stop passing and to transitively inform the other players as well.

3.7.6 Peer-To-Peer

The last example outlines a system of interacting equivalent components, often called *peers*. Organised in a completely connected network (cf. Figure 3-16), each component is able to interact with each other. With such a scenario, we can demonstrate that the language allows modelling component networks of arbitrary shape.

```
INTERFACE Colleague;
 { IN Ask(question: TEXT) OUT Response(answer: TEXT) }
END Colleague;

COMPONENT Person
 OFFERS Colleague
 REQUIRES Colleague [1..*];

 IMPLEMENTATION Colleague;
  VARIABLE question, answer: TEXT;
  BEGIN
   WHILE ?Ask DO
    ?Ask(question);
    (* think about answer to the question *)
    !Response(answer)
   END
 END Colleague;

 VARIABLE i: INTEGER; question, answer: TEXT;
 ACTIVITY
  REPEAT {EXCLUSIVE}
   (* think about interesting question *)
   i := RANDOM(1, COUNT(Colieague));
   Colieague[i]!Ask(question);
   Colieague[i]?Response(answer);
  UNTIL TERMINATEDO
END Person;

COMPONENT World;
 VARIABLE person[i: INTEGER]: Person; i, k, N: INTEGER;
 BEGIN (* set N *)
  FOR i := 1 TO N DO NEW(person[i]) END;
  FOR i := 1 TO N DO
   FOR k := 1 TO N DO
    CONNECT(Colieague[k](person[i]), person[kJ)
   END
  END
END World;
```

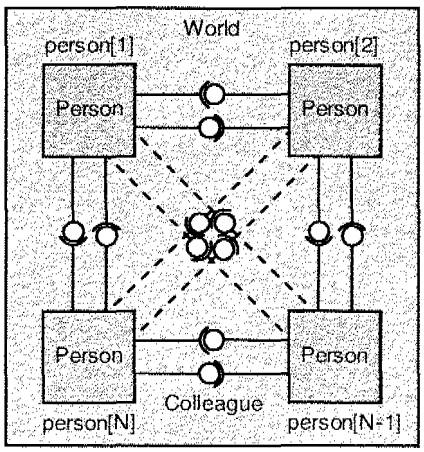


Figure 3-16. Interacting peers

3.8 Related Work

The programming language Composita has certainly profited from various sources of inspirations.

Component relations. Our component concept (including the diagram notation) is certainly influenced by COM rWiI88,WiI90, COM06]. In particular, this concerns the relations between components, namely *aggregation* and *wiring* of *ingoing* and *outgoing* interfaces. We have developed these relations towards first-level programming concepts, which do not involve explicit references anymore and also enable guaranteed hierarchical encapsulation (cf. Section 2.3.1). In a novel way, component structures can be dynamically built in our language, unlike architecture description languages [GA094, BE+94, MD+95, LK+95, MQR95, SD+95, GMW97, MRT99, A1l97], which only support *static* structures (cf. Section 2.3.2). The possibility of transmitting components within messages between others is also a new concept, allowing dynamic and safe plug-in scenarios without passing references.

Interface polymorphism. The idea of using *symmetric interface polymorphism* as a replacement for inheritance has been primarily motivated by COM and was also

inspired from Zonnon [GZ05]. **In** contrast to Zonnon, our language supports flexible reuse without an extra concept of default implementations (cf. Sections 2.1.5.4 and 2.3.1), as reuse is inherently enabled by hierarchical composition and interface connections. The abstract signature thereby ensures the correct use of polymorphic components.

Message communication. As for the fundamental communication paradigm (cf. Section 2.1.5.3), the model of component interactions is principally influenced by CSP [Hoare78]. The use of asynchronous message exchange with non-blocking send statements goes back to the Actors model [Agha86]. The dialog concept of Active C# [GG04] and Zonnon motivated us to also support client-individual communications and use EBNF for the protocol description. More innovatively, our language does not require explicit invocations of dialogs and also engages high-level messages (with arbitrary content), instead of directly exchanging tokens and single data values. According to our design principles, message communication is indeed provided as the sole concept for interactions between components.

Component-intrinsic activities. The idea of equipping components (or objects) with an intrinsic activity was originally introduced with Simula I [DN66]. A more elaborated version of this concept was realised in Active Oberon [Gut97, Mu102, Reali041, which also influenced our model of autonomous components. The concept of monitors [BH73, Hoare74], which is also integrated in Active Oberon, serves in our language as a mechanism to synchronise the processes within a component. However, a process only runs within its enclosing component and therefore, remains subject to synchronisation within only one component instance. As all component interactions are rigorously based on message communication, the monitor concept only forms a construct for the component implementation and is not of fundamental importance for the component model.

Syntax Notation. The style of the syntax, especially for the elementary statement constructs **(IF, WHILE** etc.), is largely adopted from Oberon [Wirth88], emphasising clear readability and avoiding counterproductive abbreviations.

Chapter 4

Conceptual Advances

As a result of the simple but powerful component notion, the new language abolishes various problems that exist in ordinary programming languages and for which no satisfactory solution has been found to date. This chapter discusses the most important conceptual improvements introduced by the new programming model. For each of these issues, we first discuss the shortcomings of existing languages before presenting the new solution achieved by our programming model.

4.1 Hierarchical Encapsulation

The key feature of the new programming language is the general ability to hierarchica.lly encapsulate arbitrary static or dynamic structures of components within others. By using general hierarchical composition in combination with interface connections, the complete encapsulation of inner components and their structures is continuously guaranteed. This is in contrast to ordinary programming languages, which are based on pointers and do not support general encapsulation and protection of logical sub-structures. This typically leads to fragile programs with potentially arbitrary referential dependencies.

In the subsequent two sections, we discuss the problem by means of a concrete example of a *library* that contains a

dynamic collection of books. The library should be usable by an arbitrary number of customers, which independently borrow and return books and which may also list tlle book catalogue. Tn the following section, we first demonstrate the difficulties involved using this example in classical pointer-oriented languages. After this, we focus on how the issue is overcome with the hierarchical composition of the new component language.

4.1.] The Classical Problems

In classical programming languages, such a task has to be modelled as a *flat structure:* both the library and books have to be allocated as usual objects in the heap and need to be appropriately linked together by pointers. Thereby, pointers commonly represent all kinds of relations, such as the *contains-relation* between library and books, as well as the normal *textual references* listed in books. Due to the missing support of hierarchical composition, encapsulation of books can not be guaranteed. Instead, very cautious programming is required to prevent passing out pointers to books that ought to be internal to the library. The object-oriented program below (written in C# [CS06l) demonstrates this situation. For illustration, an array is employed here for the implementation of the dynamic book collection. The use of a more sophisticated data structure would not change anything with regard to the following discussIOn.

```
class Book {
  public string isbn; string content;
  Book[] references; /* refers to other books */
  public string GetContentO { return content; }
  public void Annotate(string note) { content += note; }
}
```

```
class Library {
  Book[] books;
  public Book BorrowBook(string isbn) {
    for (int i = 0; i < books.Length; i++)  {
      if ((books[i] != null) && (books[i].isbn == isbn)) {
        return books[i];
      }
    }
    return null; /* null means unavailable */
  }
  /* ... */
```

Although the program fragment might look correct at the first glance, the BorrowBook method returns in error a reference to an internal book of the library, thus breaking the encapsulation of the library structure. A customer could then accidentally access an internal book of the library, as shown in the code fragment below and in Figure 4-1.

```
class Customer {
   Library library;
   void IncorrectUse {
      Book book = library.BorrowBook(13-46S-11124-2");
      read(book.GetContentO);
        /* forbidden reading of an internal library book */
      book.Annotate("personal note");
       /* forbidden modification of an internal library book */
   }
}
```

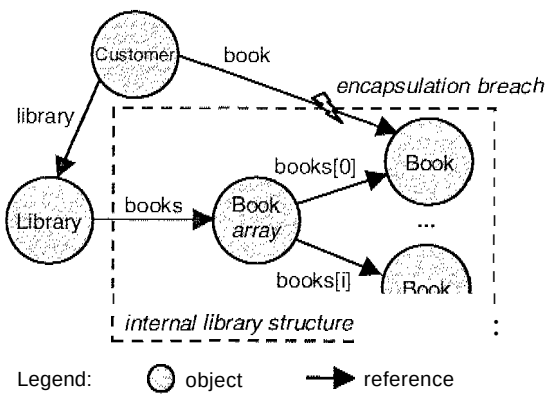


Figure 4-1. Encapsulation breach by incorrect referencing

To avoid this problem, one would decide to either return the book as copy or better, to properly remove the book from the bookshelf (array) before handing it over to the customer. The BorrowBook method may therefore be adjusted to this:

```
Book x = books[i]; books[i) = null;
return x;
```

However, despite the taken precaution, the directly or indirectly referenced books in the library still remain incorrectly accessible by external customers. The subsequent fragment and Figure 4-2 delineates an example of such a situation:

```
class Customer {
  Library library;
  void IncorrectUse {
    Book book = library.BorrowBook("3-468-11124-2");
    Book x = book.references[O);
    /* read and modify x */
  }
}
```

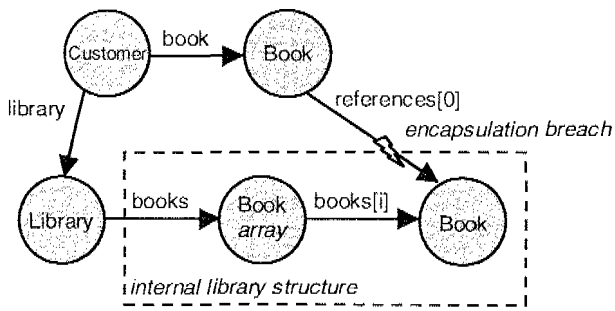


Figure 4-2: Another situation of incorrect referencing

The abovementioned scenarios should demonstrate how vulnerable object-oriented and other classical programs are due to the fact, that pointers can conceptually link arbitrary objects and can be copied without restriction in the system. Of course, it may be argued that pointers ought not to represent book references in this example (which would be actually a capitulation of this concept). Another approach

of only passing *read-only* references [MPOll, does unfortunately not give a sustainable solution either, since books may still be read without permission.

Allowing customers to list the library catalogue also entails problems with the encapsulation. Generally, the client-individual iteration logic is outsourced to an external *iterator* object[30] which stores a reference leading into the internal library structure (see Figure 4-3). Due to the potential encapsulation breaches, this scenario is usually considered as a counter-example for the proposed object-oriented ownership models [CPN98, MPOl]. However, we think that this problem is rather caused by the artificial externalisation of the iterator logic within a separate helper object.

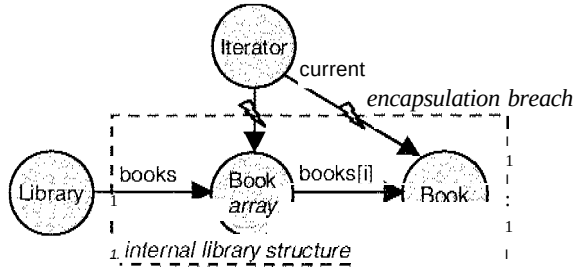


Figure 4-3. Iterator object

4.1.2 The New Solution

In our programming language, the hierarchical structure of the library example can be clearly reflected in the program. The PublicLibrary component contains books as sub-components (see Figure 4-4) and thus guarantees that the internal books are fully encapsulated. As a result, any direet and uncontrolled accesses from outside to the internal books are prevented. The PublicLibrary is generic, i.e. any component with the offered interface Book can be stored

30 Although the iterator has become a popular object-oriented design pattern [GH+951, it rather forms an artificial solution to enable long-term client-individual interactions.

within the dynamic collection (see Section 3.2) of the library component. We first focus on the implementation of the library, before we demonstrate how book components may be programmed.

```
INTERFACE Library;
  {
    IN BorrowBook(isbn: TEXT)
    (OUT Result(b: ANY(Book)) | OUT Unavailable)

    IN ReturnBook(b: ANY(Book))

    IN ListCatalogue
    { OUT Entry(isbn: TEXT) }
    OUT EndOfCatalogue
  }
END Library;

COMPONENT PublicLibrary OFFERS Library;
 VARIABLE book[isbn: TEXT]: ANY(Book);

 IMPLEMENTATION Library;
  VARIABLE isbn: TEXT; b: ANY(Book);
  BEGIN
    WHILE ?RequestBook OR ?ListCatalogue DO
      IF ?RequestBook THEN {EXCLUSIVE}
        ?RequestBook(isbn);
        IF EXISTS(book[isbn]) THEN
          !Result(book[isbn]) (* variable book[isbn] is now empty! *)
        ELSE !Unavailable
        END
      ELSIF ?ReturnBook THEN {EXCLUSIVE}
        ?ReturnBook(b); b!GetISBN; b?ISBN(isbn);
        MOVE(b, book[isbn])
      ELSE {SHARED}
        ?ListCatalogue;
        FOREACH isbn OF book DO !Entry(isbn) END;
        !EndOfCatalogue
      END
    END
 END Library;

 BEGIN (* initialise library *)
END PublicLibrary;
```

To identify the books in the collection, international standard book numbers (ISBNs) serve as indexes. If

present, the appropriate book is transmitted to the corresponding exterior customer. Note that the components are always sent *by moving* and thus, the book is removed from the library when sent within a message. Furthermore, the absence of a book can be appropriately communicated by the alternative message named Unavailable, whereas in object-orientation, an artificial null reference is typically employed to indicate this case. Moreover, the language-inbuilt MOVE-statement is engaged in the example above, to move a component from one variable to another. (A possible previous content of the target variable is in turn automatically deleted.)

Multiple customers may now be connected to the library and interact with it in parallel. The statefull process of listing the book catalogue is directly supported by the concept of long-term client-individual communications.

```
COMPONENT Customer REQUIRES Library;
 VARIABLE isbn, interested: TEXT; book: ANY(Book);
 BEGIN
  Library!ListCatalogue;
  WHILE Library?BookReference DO
    Library?Entry(isbn);
    IF Interesting(isbn) THEN interested := isbn END
  END;
  Library?EndOfCatalogue;
  (* determine an isbn of interest *)
  Library!RequestBook(interested);
  IF Library?Result THEN
   Library?Result(book);
   (* use book *)
   Library!ReturnBook(book)
  ELSE Library?Unavailable
  END
END Customer;
```

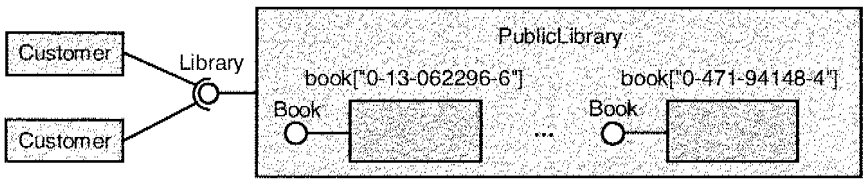


Figure 4-4. Digital library with encapsulated books

The book components may be implemented as below. **In** this program, book references are always represented as what they actually are in reality, namely *globally unique identifiers* (GUIDs) in the form of international book standard numbers (ISBNs). These real book references do not involve any specific language concept but only form programmer-defined globally unique identifiers of component instances. In contrast to ordinary pointers, such GUIDs do neither imply a direct access link nor an existence guarantee. Like in reality, a book reference does not yet enable a person to directly access and read a corresponding book without obtaining it first from a library, book store or another source. And evidently, the knowledge of a book reference does not yet guarantee that the corresponding book is still existent (and available).

```
INTERFACE Book;
 {
   IN Read OUT Content(isbn, content: TEXT)

   IN Annotate(note: TEXT)

   IN ListReferences
   { OUT Reference(isbn: TEXT) }
   OUT EndOfList
 }
END Book;

COMPONENT LibraryBook OFFERS Book;
 VARIABLE
  isbn, content: TEXT; reference[no: INTEGER]: TEXT;

 IMPLEMENTATION Book;
 VARIABLE note: TEXT; i: INTEGER;
 BEGIN
  WHILE ?Read OR ?Annotate DO
```

```
    IF ?Read THEN {SHARED} ?Read; !Content(isbn, content)
    ELSIF ?Annotate THEN {EXCLUSIVE}
     ?Annoate(x); APPEND(content, note)
    ELSE {SHARED}
     ?ListReferences;
     FOREACH i OF reference DO !Reference(reference[i]) END;
     !EndOfList
    END
   END
  END Book;

  BEGIN (* initialise book *)
 END LibraryBook;
```

4.2 Structured Networks

As a further improvement, networks of components are always clearly described and exclusively controlled by the hierarchically surrounding component. Components are not allowed to create and change exterior relations on their own initiative. In object-oriented systems, objects can not hierarchically manage and supervise such structures, potentially resulting in an object graph of unstructured topology with many implicit dependencies.

To demonstrate the gained structural control, we consider a concrete example. For this purpose, we reuse the City example from Chapter 3 and model buildings which are connected to different water and electricity suppliers. In the following section, we first describe how this example would be represented in a classical language and analyse the resulting structural problems. Subsequently, we show how these problems are eliminated if we realise the same scenario in our language.

4.2.1 The Classical Problems

The following code fragment sketches a possible solution of the example using an ordinary object-oriented language. The City object thereby creates a set of house objects, a

power plant and two rivers. The city is also responsible for arranging the supplies of the buildings (see Figure 4-5).

```
class House {
  public PowerPlant electricitySupply;
  public River waterSupply;
}

class PowerPlant {}

class HydroelectricPowerPlant: PowerPlant {
  public River waterSupply;
}

class River {}

class City {
  House[] houses;
  PowerPlant powerPlant;
  River northRiver, southRiver;

  public CitYO { // initialiser
    houses = new House[N];
    northRiver = new RiverO; southRiver = new RiverO;
    powerPlant = new HydroelectricPowerPlantO;
    powerPlant.waterSupply = northRiver;
    for (int i = 0; i < N; i++) {
      houses[i] = new HouseO;
      houses[i].electricitySupply = powerPlant;
      if (i <= N/2) { house[i].waterSupply = northRiver; }
      else { house[i].waterSupply = southRiver; }
    }
  }
}
```

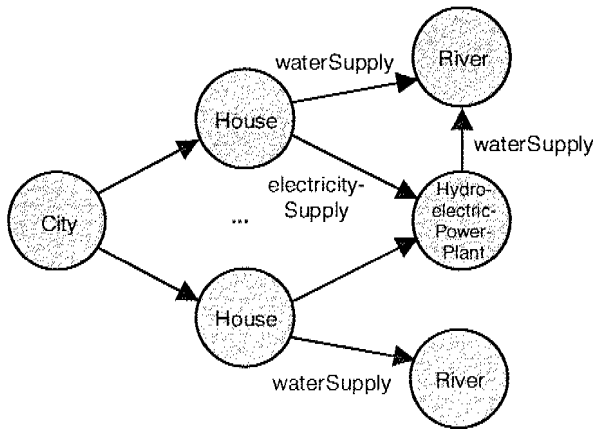


Figure 4-5. Object-oriented city

Though the city defines a particular organisation, house objects may freely rearrange the structure of the city (by accident or intention). With regard to the example, the intended city structure can be easily subverted by a specific implementation of a House object, as outlined below. On the one hand, the house-internal logic may change references among the houses and on the other hand, new buildings may be created without knowledge of the city (see also Figure 4-6).

```
class House {
  public PowerPlant electricitySupply;
  public River waterSupply;

  public void InteriorWorkO {
    waterSupply = electricitySupply.waterSupply;
    // unallowed change of the water supplier
  }

  House sommerHouse;
  public void StartInhabitationO {
     sommerHouse = new HouseO;
     sommerHouse.electricitySupply = electricitySupply;
     sommerHouse.waterSupply = waterSupply;
     // build a new house that is not controlled by the City
  }
}
```

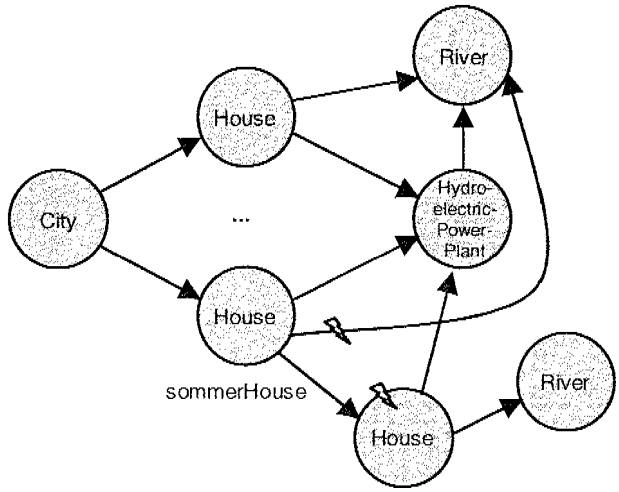


Figure 4-6. Uncontrolled change of the city structure

An implication of such uncontrolled object dependencies means the city is even unable to replace a power plant, since unknown third-party objects might still have a pointer to the former power plant. Figure 4-7 shows an example of such a situation, caused by the illegally built summerhouse that is only reachable via a private reference from another house.

```
class City {
  House[] houses;
  PowerPlant powerPlant;

  public void ReplacePowerPlantO {
    powerPlant:= new WindPowerPlantO;
    for (int i = 0; i < N; i++) {
      house[i].electricitySupply = powerPlant;
    }
  }
}
```

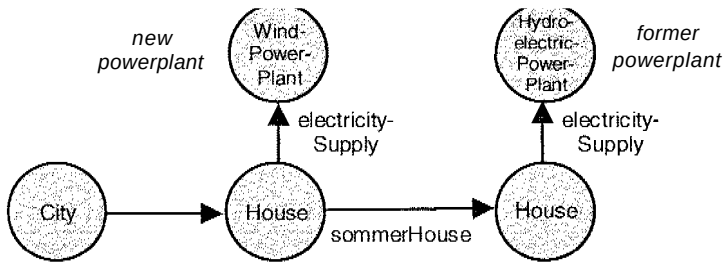


Figure 4-7. Replacing a power plant

4.2.2 The New Solution

Below, the same example is developed in our programming language. Apparently, the city possesses **full** control over its inner organisation and the relations between houses, water and electricity suppliers. None of the internal components may rearrange or establish connections on its own with other sub-components of the city.

```
COMPONENT City;
 VARIABLE
  house[number: INTEGER]: ANY(House | Electricity, Water);
  powerPlant: ANY(Electricity | Water);
  northRiver, southRiver: River;
  i: INTEGER;
 BEGIN
  NEW(northRiver); NEW(southRiver);
  NEW(powerPlant, HydroelectricPowerPlant);
  CONNECT(Water(powerPlant), northRiver);
  FOR i := 1 TO N DO
   NEW(house[i], StandardHouse);
   CONNECT(Electricity(house[i]), powerPlant);
   IF i <= N/2 THEN
    CONNECT(Water(house[i]), northRiver)
   ELSE
    CONNECT(Water(house[i]), southRiver)
   END
  END
END City;
```

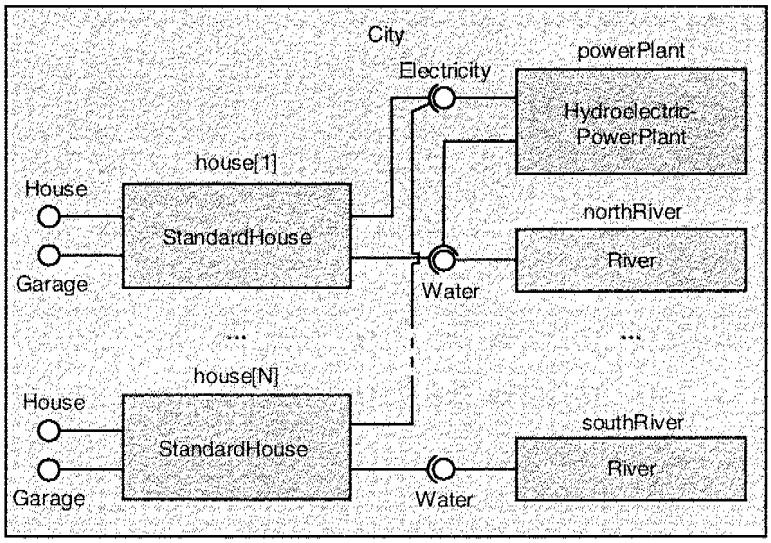


Figure 4-8. Structured city organisation

Of course, the City component may safely replace a power plant if need be. This is because the City component has absolute control over the structure of the sub-component and thus has complete knowledge of all the interface connections among the sub-components. Using the following statements, all the connections to the offered Electricity interface of the old power plant are first deleted, before the new power plant component is installed and the new connections are established. The **NEW**-statement automatically deallocates the previously existing component in the variable. In addition, the *required* interfaces of the deleted component are automatically disconnected.

```
FOR i := 1 TO N DO
  DISCONNECT(Electricity(house[i]))
END;
NEW(powerPlant, WindPowerPlant);
FOR i := 1 TO N DO
  CONNECT(Electricity(house[i]), powerPlant)
END
```

4.3 Dynamic Plug-Ins

Our component model offers the flexibility to dynamically install components within others, by transmitting such plug-in components within messages. Naturally, a plug-in component does not have to be known at the development time of their host component but the host accepts any plug-in component that fulfils the specified interfaces. In contrast to conventional object-oriented models, we require the explicit declaration of the offered and required interfaces of the plug-in component, such that unspecified external dependencies of the plug-in instances can be excluded.

By way of an exemplary scenario, the conceptual difference between the object-oriented approach and our language should be contrasted. For this purpose, an elevator ought to be dynamically installed in a house component *(plug-in).* We first discuss how this would be addressed in an object-oriented language and what kinds of difficulties are therewith involved. After that, we show how our language enables an adequate and safe realisation of this scenario.

4.3.1 The Classical Problems

Using an object-oriented language, the host object of a plug-in generally prescribes a specific base class, from which the concrete plug-in classes must inherit. As a result of the sub-type relation, the host object can accept a reference to any specific plug-in object and is able to use the plug-in as defined by the base class.

```
class Elevator {
  public abstract void StartRunningO;
}

class ModernHouse {
  Elevator elevator; // reference to a plug-in
  public PowerPlant electricity;
```

```
  public void InstaliElevator(Elevator x) {
    elevator = x;
    elevator.PutIntoOperation();
  }
}
```

We may now define and create a concrete elevator object and install it in the house object. This leads to the runtime structure depicted in Figure 4-9.

```
class ElectricalElevator: Elevator {
  PowerPlant electricity;

  public ElectricalElevator(powerPlant p) { electricity = p; }
  public override void PutintoOperation() {
    /* serve corridor calls */
  }
}

class City {
  PowerPlant p; ModernHouse h;

  public void Setup {
    p = new PowerPlantO;  h = new ModernHouseO;
    h.electricity = p;
    Elevator e = new ElectricalElevator(new PowerPlant());
    h.InstaliElevator(e);
  }
}
```

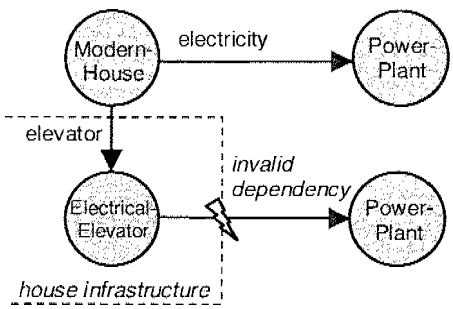


Figure 4-9. Object-oriented plug-in structure

Obviously, the host object has no information about external dependencies that may be induced by the plug-in object. In our setting, the electrical elevator maintains a direct reference to a specific electricity supplier, which is

potentially different to the supplier of the house. Due to the (synchronous) method invocations, the execution of the plug-in implementation also goes at the expense of the host object. This may block, delay or even corrupt the execution of the house logic in an undefined manner.

4.3.2 The New Solution

The same scenario can be safely realised in the new component-based language, as shown below. TIle abstract component signature with the ANY-construct (see Section 3.2) thereby states all necessary offered and all possible required interfaces of a possible plug-in component. This means that the elevator component in the house requires at most the Electricity interface, which is always connected to the required Electricity interface of the house (see also Figure 4-10). The concrete elevator may be eventually created outside the house and transmitted within a message to the inner domain of the house component.

```
INTERFACE Elevator;
  IN Start
END Elevator;

INTERFACE HouseConfiguration;
  IN InstallElevator(x: ANY(Elevator | Electricity))
END HouseConfiguration;

COMPONENT ModernHouse
  OFFERS HouseConfiguration
  REQUIRES Electricity;

  VARIABLE elevator: ANY(Elevator | Electricity); (* ... *)

  IMPLEMENTATION HouseConfiguration;
  BEGIN {EXCLUSIVE}
    ?InstallElevator(elevator);
    CONNECT(Electricity(elevator), Electricity)
  END HouseConfiguration;
END ModernHouse;
```

```
COMPONENT ElectricalElevator
 OFFERS Elevator
 REQUIRES Electricity;

 IMPLEMENTATION Elevator;
 BEGIN ?Start; (* serve corridor calls *)
 END Elevator;
END ElectricalElevator;

COMPONENT City;
 VARIABLE
  house: ModernHouse; powerPlant: PowerPlant;
  elevator: ANY(Elevator);
 BEGIN
  NEW(house); NEW(powerPlant);
  CONNECT(Electricity(house), powerPlant);
  NEW(elevator, ElectricalElevator);
  HouseConfiguration(house)!InstaliElevator(elevator)
END City;
```

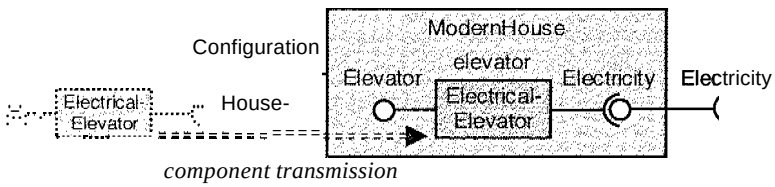


Figure 4-10. Dynamic component plug-in

By reason of the clear separation of the interfaces and the implementation of components and the communication-based interactions, the plug-in and the host component have shielded execution domains with separate inner processes. Hence, the execution speed and potential errors in the plug-in object do not necessarily impair the execution of the h(mse logic.

4.4 Symmetric Polymorphism

Due to the ability of offering an arbitrary number of interfaces, components are inherently polymorphic: each interface represents an independent external facet, without obligating the programmer to define any hierarchy among the

facets. For the same purpose, object-oriented languages generally require programmers to define sub-class relations by means of inheritance. Apart from the unfounded need of such type classifications, this approach also often entails unpleasant technical troubles, such as ambiguities or conflicts.

These problems can be illustrated by way of an example, in which different types of vehicles should be modelled, namely cars, boats and amphibian mobiles. Again, we first analyse the problems for classical object-orientation before we focus on how this is approached in our language.

4.4.1 The Classical Problems

In an object-oriented model, a programmer would usually classify the vehicles in cars and boats. An extract of the corresponding program could look like this:

```
class Vehicle {
  public int maximumLoad;
  public abstract void DriveO;
}

class Car: Vehicle {
  public void DriveO { ... }
  public override int MaximumSpeedO { ... }
}

class Boat: Vehicle {
  public void DriveO { ... }
  public override int MaximumSpeedO { ... }
}
```

The program should now be extended to additionally support amphibian mobiles, which feature both the aspects of a car and a boat. In object-oriented languages, which only support single-inheritance (and this is the majority), the programmer may be forced to define one of the following absurd sub-class relations (see also Figure 4-11):

```
// First work-around:
class Boat: Car { ... }
  // should a boat really be a sub-class of a car?
class AmphibianMobile: Boat { ... }

// Second work-around:
class Car: Boat { ... }
  // should a car really be a sub-class of a boat?
class AmphibianMobile: Car { ... }
```

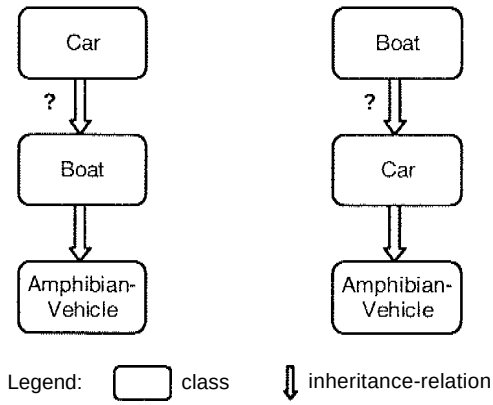


Figure 4-11. Hierarchisation dilemma of single-inheritance

It seems obvious that single-inheritance is not suited for such cases. Therefore, languages like Java and C# supplement inheritance with a secondary polymorphism concept, namely interfaces.

With multiple-inheritance (in C++ or Eiffel), the program above can be more adequately extended, such that the amphibian mobile forms a sub-class of the two equivalent base classes Car and Boat (see Figure 4-12). However, ambiguities and naming conflicts inevitably come up if the base classes have features with the same names.

```
class AmphibianMobile: Car, Boat {
  public override void DriveO { ... }
}
```

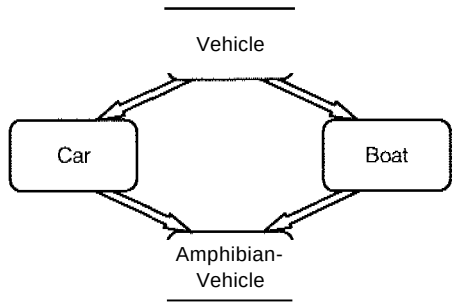


Figure 4-12. Multiple-inheritance

As for the *Drive* method, the conflict could be resolved by overriding the method by a new combined implementation. However, the two different versions of MaximumSpeed should be retained, such that they should be renamed to the methods MaximumCarSpeed and MaximumBoatSpeed in the sub-class[31]. Still, it remains open how variables of the multiply inherited base class Vehicle ought to be handled[32]: should different instances of the maximum Load variable be provided for the Car and Boat functionality; or should the amphibian mobile rather have only one instance of this variable[33]?

From a conceptual point of view, multiple-inheritance may be also put into question in generaL Initially, the programmer has classified the vehicles into different types of Car and Boat, and therewith has intended to describe separate sets (classes) of objects. However, by introducing the AmphibianMobile, a sub-class (or sub-set) of both the class Car and Boat is introduced, such that the prior sepa-

---

[31] Eiffcl supports renaming as one way of conflict resolution for multiple-inheritance [Meyer97, Section 15.2j.

[32] A situation like this, in which the same class is indirectly inherited in multiple ways, is also called *repeated inheritance* and involves quite complicated rules [Meyer97, Section 15.4J.

[33] In Eiffel, this difference is implicitly determined by whether the feature is renamed or not [Meyer97, Section 15.4J. llowever, renaming may be easily forgotten (especially if the features arc non-public), such that the variable is accidentally shared by the base classes (e.g. maximum Load of the Car and the Boat).

ration of Car and Boat is relaxed. However, a classification would only make sense if it also results in the definition of *disjoint* sets. Otherwise, it is simply an ad-hoc definition of sets that can be disjoint or may overlap. The type hierarchy here obviously serves two purposes, splitting a class into various sub-classes and merging parts of various classes into a common sub-class. This hierarchical class splitting and merging only involves unnecessary intricacy for enabling type polymorphism.

4.4.2 The New Solution

The program below sketches the solution in our language. The AmphibianMobile component offers both the equivalent RoadVehicle and WaterVehicle. Notably, there is no inbuilt mechanism of interface refinement provided, because interfaces may just as well textually refine the functionality of others (e.g. the interfaces RoadVehicle and Vehicle). Hence, a programmer can flexibly refine an interface by augmenting *or* reducing the functionality.

```
INTERFACE Vehicle;
 { IN Drive IiN GetMaximumLoad OUT Load(tons: INTEGER) }
END Vehicle;

INTERFACE RoadVehicle;
 { IN Drive IiN GetMaximumSpeed OUT Speed(kmh: INTEGER)}
END RoadVehicle;

INTERFACE WaterVehicle;
 { IN Drive IiN GetMaximumSpeed OUT Speed(kmh: INTEGER)}
END WaterVehicle;

COMPONENT Car
 OFFERS RoadVehicle, Vehicle;
END Car;

COMPONENT Boat
 OFFERS WaterVehicle, Vehicle;
END Boat;
```

```
COMPONENT AmphibianMobile
 OFFERS RoadVehicle, WaterVehicle, Vehicle;
 (* own implementation of interfaces *)
END AmphibianMobile;
```

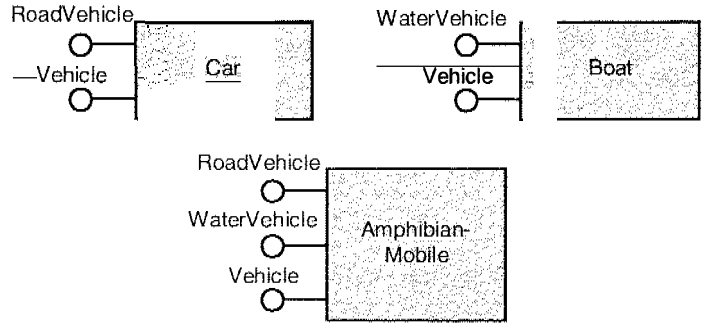


Figure 4-13. Interface polymorphism

As each component features its own independent implementation of the offered interfaces, conflicts such as with inheritance do not occur.

4.5 Flexible Reuse

Our component model enables flexible reuse of implementations by the relation of hierarchical composition. A component can use existing functionality of components by containing the corresponding components as parts of its own implementation. On the contrary, object-oriented inheritance only offers static code reuse in combination with type polymorphism. 'rhis unfortunate unification of two substantially different concerns in one concept often causes logical contradictions and hinders reusability[34].

We resume the vehicle example of the previous section to show these problems and eventually compare it with our

[34] To overcome this problem, *mixins* ISe90] or *traits* [SD+031 have been advertised as a special mechanism that separates polymorphism from code reusc (ef. Section 2.1.5.4). In our language, no such artificial helper concept is needed.

programming language. More concretely, the vehicles should be implemented with an internal motor, gear box, as well as wheels or a propeller, depending on whether it is a car or boat. The amphibian mobile should thereby reuse implementation parts of both vehicle types. The following sub-sections discuss the implementation of this scenario, first for an ordinary object-oriented language and then for the component language.

4.5.1 The Classical Problems

According to the stated requirements, we assume that the Car and Boat classes are implemented as follows:

```
class Car {
  public Motor carMotor = new MotorO:
  public NormalGears carGears =new NormalGearsO:
  public Wheels wheels =new WheelsO:

  public CarO { // initialiser
    carMotorJorce =carGears; carGears.transmission =wheels;
  }
  int CurrentPowerO { return carMotor.power; }
}

class Boat {
  public Motor boatMotor = new MotorO:
  public NormalGears boatGears = new NormalGearsO:
  public Propeller propeller = new PropelierO;

  public BoatO { // initialiser
    boatMotor.force = boatGears;
    boatGears.transmission = propeller;
  }
  int CurrentPowerO { return boatMotor.power; }
}
```

As the amphibian mobile should be a sub-class of Car and Boat (for the purpose of polymorphism), we are forced to also inherit the implementation of these base classes (see Figure 4-14). Clearly, the amphibian mobile requires a more specific implementation than just the combination of all features. For example, just one motor should be integrated instead of reusing both the motor of the Car and of

the Boat. **In** addition, the gear box should be replaced by a more specific hybrid model, enabling transmissions to both wheels and propeller.

```
class AmphibianMobile: Car, Boat {
  // inherit two motors and two gear boxes
  public HybridGears hybridGears = new HybridGearsO;

  public AmphibianMobileO {
    carMotor.force = hybridGears;
      /* or boatMotor.force = hybridGears? *j
    hybridGears.transmission[O] = wheels;
    hybridGears.transmission[1] = propeller;
     /* carGears and boatGears are redundant *1
  }
}
```

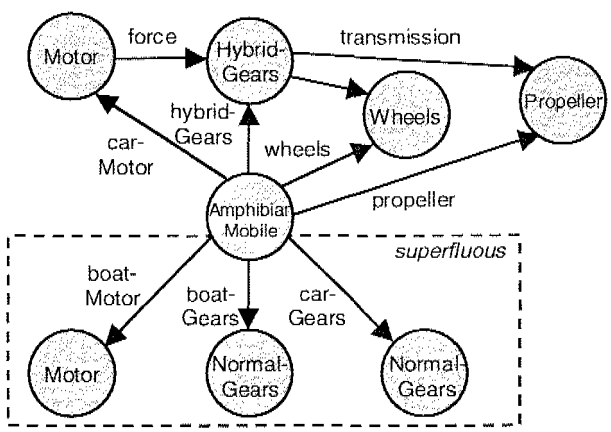


Figure 4-14. Inheritance of undesired Cunctionality

The preceding example also illustrates how easily the implementation of a class may be broken by inheritance: the CurrentPower methods still refer to their default motors, even though only one of the two motors is engaged in the amphibian mobile. To avoid such problems, the programmer should rather employ inheritance from *abstract classes* or use interfaces, such that implementation can be selectively reused by means of delegation.

4.5.2 The New Solution

The same vehicles can be programmed in our language independent of polymorphism and without risk of encapsulation breaches. The corresponding code is delineated as follows, leading to the runtime structure visualised in Figure 4-15:

```
COMPONENT Car OFFERS RoadVehicle, Vehicle;
 VARIABLE
   motor: Motor; gears: NormalGears; wheels: Wheels;
 (* ... *)
 BEGIN
  NEW(motor); NEW(gears); NEW(wheels);
  CONNECT(Gears(motor), gears);
  CONNECT(Transmission(gears), wheels)
END Car;

COMPONENT Boat OFFERS WaterVehicle, Vehicle;
 VARIABLE
   motor: Motor; gears: NormalGears; propeller: Propeller;
 (* ... *)
 BEGIN
  NEW(motor); NEW(gears); NEW(propeller);
  CONNECT(Gears(motor), gears);
  CONNECT(Transmission(gears), propeller)
END Boat;

COMPONENT AmphibianVehicle
 OFFERS RoadVehicle, WaterVehicle, Vehicle;
 VARIABLE
  motor: Motor; gears: HybridGears;
  wheels: Wheels; propeller: Propeller;
 (* ... *)
 BEGIN
  NEW(motor); NEW(gears); NEW(wheels); NEW(propeller);
  CONNECT(Gears(motor), gears);
  CONNECT(Transmission[1](gears), wheels);
  CONNECT(Transmission[2j(gears), propeller)
END AmphibianVehicle;
```

The offered interfaces of the vehicles can be either implemented for each component template individually or be redirected to matching offered interfaces of the vehicle's sub-components (cf. Section 3.4).

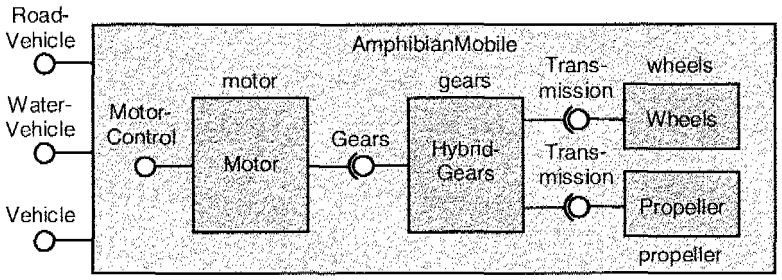


Figure 4-15. Flexible reuse by composition

4.6 Safe Concurrency

Concurrency forms a cornerstone of the new programming language, where components feature their own internal processes. Due to the structured component relations our language can offer safe concurrency, excluding unnecessary concurrency errors (such as data races and various kinds of deadlocks). In contrast, classical programming languages do not encourage well-structured concurrency but simply provide threads that can operate on arhitrary objects. Though languages like Simula fDN66, DMN68] or Active Oberon [Gut97, Mu102, Reali04] provide a much better integrated concurrency with object-contained concurrency and inbuilt language support for monitor synchronisation, object activities may still operate on arbitrary shared objects by invoking methods. Therefore, classical progrmns generally lack a clear specification of what objects are potentially directly or indirectly accessed by a thread or active object, such that these concurrent programs are inherently susceptible to data races and hard-to-detect deadlocks.

By means of an example, we demonstrate how the component language improves the safety of concurrency. For this purpose, we first concentrate on the classical object-oriented programming model and discuss the involved concurrency problems. Subsequently, we solve the same example in our language and discuss the gained advantages.

4.6.1 The Classical Problems

Most object-oriented languages only offer thread-based concurrency as an optional feature on top of the classical sequential execution model. This means that the language does not necessarily encourage the programmer to write thread-safe program code, since the default programming model is still non-concurrent. For instance, if we implement a collection data structure, we may assume that the collection is only used by classical sequential code and therefore, may not prepare the implementation for thread-safety. For the sake of simplicity, the collection could be realised as a doubly linked linear list, although the subsequent discussion would also apply to other implementations.

```
class ListNode {
  public ListNode prev, next;
  public int key; public object value;
}

class Collection {
  public ListNode first = null;
  public ListNode last = null;

  public object Get(int key) {
    ListNode x = first;
    while ((x != null) && (x. key != key)) {x = x.next; }
    return x;
  }

  public void Add(int key, object value) {
    ListNode x = new ListNodeO; x.key = key; x.value = value;
    x.next = first; x.prev := null;
    if (first != null) { first.prev = x; } else { last = x; }
    first = x;
  }

  public void Remove(int key) {
    ListNode x = Get(key); Assert(x != null);
    jf (x.prev == null) { first = x.next;}  else { x.prev.next = x.next; }
    if (x.next == null) { last = x.prev; } else { x.nextprev = x.prev; }
    x.next = null; x.prev = null;
  }
}
```

As nothing is specified about the concurrency assumptions in the public part of the class above, the user does not necessarily know that the collection is not thread-safe. Even if this is known, it is still probable that the collection is accidentally shared by multiple threads. This is because a thread potentially accesses an arbitrary set of objects via method calls. As a consequence, the programmer is principally unaware of the possible dependencies that might occur between threads, such that the programs frequently suffer from unexpected errors, such as data races. Notably, the ordinary compiler does not even identify potential unsynchronised parallel accesses on the collection. With regard to the example, race conditions may lead to inconsistent states of the collection (see Figure 4-16) or sudden runtime errors (see Figure 4-17).

```
using System.Threading;
class Test {
  static Collection collection =new ColiectionO;

  public static void RunO {
    int key =…; object x =…;
    cOliection.Add(key, x);
    1* computation *1
    collection. Remove(key)
  }

  public static void MainO { // thread incarnation
    Thread x =new Thread(new ThreadStart(Run));
    Thread y = new Thread(new ThreadStart(Run));
    x.StartO; y.StartO;
  }
}
```

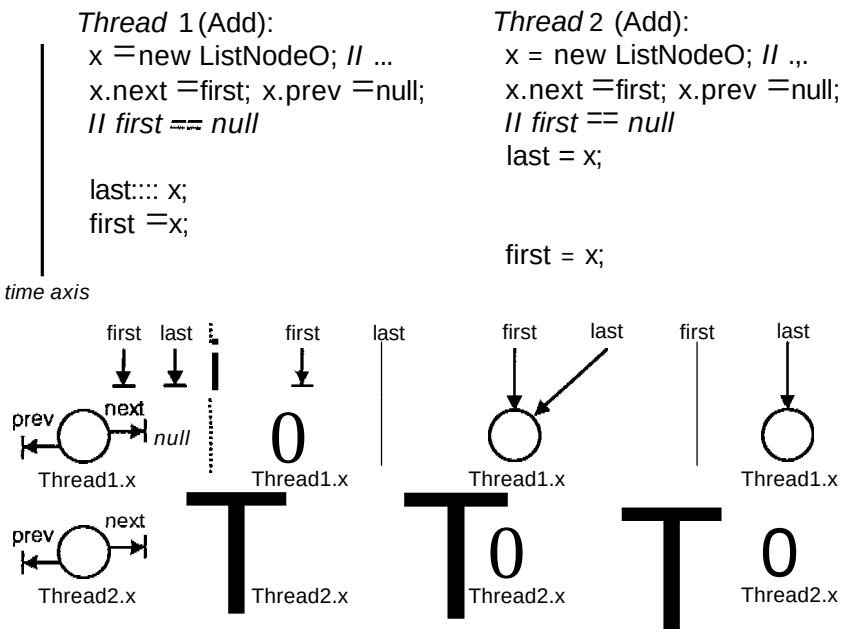


Figure 4-16. A race condition during concurrent Add-methods

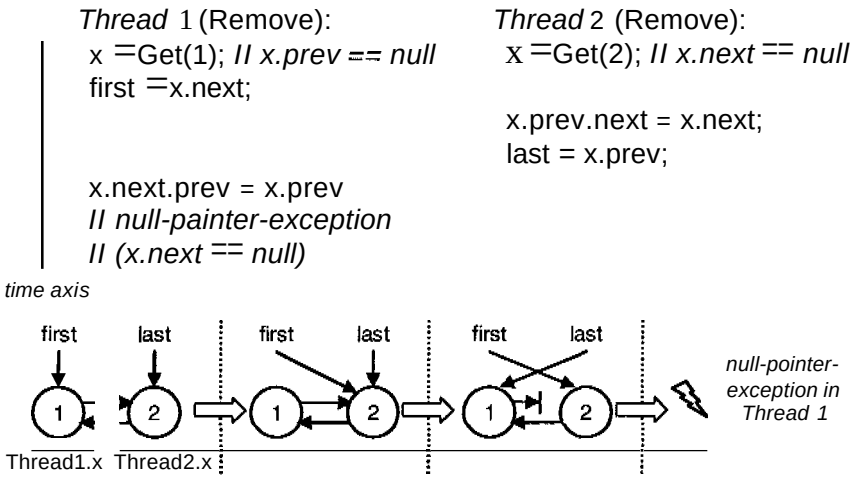


Figure 4-17. A race condition during concurrent Remove-methods

Moreover, the iteration over the collection is also not stable in the presence of multiple threads. If another thread interacts concurrently with the iterating process, the iteration may not consistently list all entries of the collection (see Figure 4-18).

```
class Iterator {
  ListNode current;
  public Iterator(Collection c) { current =c.first; }
  public bool HasNextO { return current != null; }
  public void Next(out int key, out object value) {
     key =current.key; value =current.value;
     current = current.next;
  }
}

using System.Threading;
class Test {
  static Collection collection = new CollectionO;

  public static void IterationRunO {
    int key; object x; Iterator iterator = new Iterator(collection);
    while (iterator.HasNext()} { iterator.Next(key, x); // use x  }
  }

  public static void OtherRunO {
     int key =...;
     collection. Remove(key);
  }

  public static void MainO {
    Thread x = new Thread(new ThreadStart(IterationRun));
    Thread y = new Thread(new ThreadStart(OtherRun));
    x.StartO; y.StartO;
  }
}
```

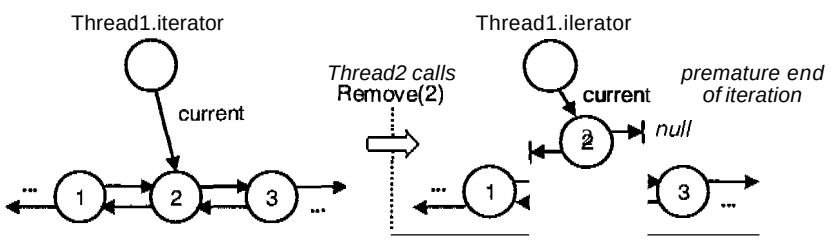


Figure 4-18. Unstable iteration

To prevent uncontrolled concurrency overlapping, the programmer has to very carefully identify all program parts which could be subject to possible parallel accesses and thus, have to be synchronised. As for our Collection class, we could make its implementation thread-safe by employ-

ing monitor locks as below[35]. It should be also noted that the collection must be particularly protected during iteration, since other threads could otherwise incorrectly interfere with the iterator logic. For this purpose, the programmer has to indicate when the iteration starts and stops, in order to lock or unlock the collection for the iteration process.

```
class Collection {
  public ListNode first;
  int iteratorLocks = 0;

  public object Get(int key) {
    lock(this) {/* Get implementation logic */}
  }

  public void Add(int key, object value) {
    lock(this) {
      while (iteratorLocks > 0) { Monitor.Wait(this); }
      /* Add implementation logic */
    }
  }

  public void AcquireIteratorLockO {
    lock(this) { iteratorLocks++; } // shared locks
  }

  public void ReleaseIteratorLockO {
    lock(this) {
      iteratorLocks--;
      if (iteratorLocks == 0) {Monitor.PulseAll(this); } }
  }
}
```

35 The pattern with the while-loop around the wait condition is crucial in Java and C#, because the wait condition may already be falsified by a third-party thread before the woken threads are resumed. Nested monitor locks on the same object are allowed in Java and C#, i.e. the Get procedure can be called by Add without leading to deadlock.

```
public Iterator {
  Collection coli; ListNode current;
  public Iterator(Collection c) { collection::: c; }
  public void StartO {
    collection.AcquirelteratorLockO;
   current::: collection.first;
  }
  public void 5topO { collection.ReleaselteratorLockO; }
}
```

Even in the case of single user thread, the collection is not safe against deadlocks. The following simple deadlock scenarios are often encountered in practice[36]:

```
public void TestCase10 {
  int key =...; object value = ...;
  Iterator it = collection.GetiteratorO;
  it.StartO;
  coliection.Add(key, value); // deadlock
}

public void TestCase20 {
  int key; object value;
  Iterator it = collection.GetIteratorO;
  it.StartO;
  if (it.HasNextO) { it.Next(key, value); }
  // it.StopO forgotten => deadlock
}
```

To detect data races and simple kinds of deadlocks in object-oriented or classical pointer-oriented programming languages, complicated and expensive analysis tools are needed [ABOl, EA03, VG03, VP04]. However, these static analysis tools are principally conservative, frequently reporting correct situations as problems *(false positives),* while often missing out real concurrency mistakes *(false negatives)* [ABOl, VP04 (Chapter 6)].

---

36 As for the second scenario, the deadlock risk also remains if the iterator would automatically release the lock at the end of the list. This is because the list does not necessarily have to be completely traversed.

4.6.2 The New Solution

In the new language, components are inherently designed for parallel use by multiple external clients. Due to the communication-based interactions, the execution between different components is inherently disentangled and synchronised. A process is always encapsulated in one component and can not directly operate on other components. As each client communication is handled by a separate service process within the server component, a component may naturally have multiple processes that run at the same time within its scope. These processes may also concurrently access the shared resources of the local component state and thus need to be explicitly synchronised by means of monitor protection. In contrast to classical languages, access dependencies of concurrent processes are always confined to the scope of a component instance, such that a programmer has to only concentrate on the synchronisation between the processes *inside* the same component instance. Thereby, the compiler of our language always checks that the processes are indeed sufficiently synchronised and that any kind of data races are excluded. To illustrate this by way of an example, we implement the generic data collection in the component language as below. In fact, an iterator here runs as part of the service process for the communication, such that no extra helper object or explicit context saving on the client side is needed.

```
INTERFACE Collection;
 {
   IN Add(key: INTEGER; x: ANY)
   ( OUT Added |OUT KeyNotFree )
 |
   IN Get(key: INTEGER)
   ( OUT Result(x: ANY) |OUT NotPresent )
 |
   IN Iterate { OUT Entry(key: INTEGER; x: ANY) } OUT Done
 }
END Collection;
```

```
COMPONENT GenericContainer OFFERS Collection;
 VARIABLE value[key: INTEGER]: ANY;

 IMPLEMENTATION Collection;
  VARIABLE key: INTEGER; x: ANY;
  BEGIN
    WHILE ?Add OR ?Get OR ?Iterate DO
      IF ?Add THEN {EXCLUSIVE}
        ?Add(key, x);
        IF EXISTS(value[key]) THEN !KeyNotFree
        ELSE value[key] := x; !Added
        END
      ELSIF ?Get THEN {SHARED}
        ?Get(key);
        IF EXISTS(value[key]) THEN IResult(value[key])
        ELSE !NotPresent
        END
      ELSE {SHARED}
        ?Iterate;
        FOREACH key OF value DO !Entry(key, value[key]) END;
        !Done
      END
    END
  END Collection;
END GenericContainer;
```

Due to the clear encapsulation, It IS very simple for the compiler to check the absence of data races within a component instance. As mentioned before, direct accesses from external processes to the local component state are not possible, such that the compile-time analysis can be performed for each component scope independently. For this purpose, the compiler requires that all modifying accesses to component-local variables are protected within an exclusive region. The same has to be ensured for the communications via the required interfaces of the local component. Conversely, all other accesses to shared variables of the component need to be surrounded by at least a shared lock.

Nested locks in the same process are not allowed in the language. This is also checked by the compiler by analys-

ing the compound statements in a process block[37]. If nested locks would be permitted, deadlocks of the following kind may occur. As illustrated in Figure 4-19, multiple processes may hold a shared lock and would then all try to acquire a nested exclusive lock.

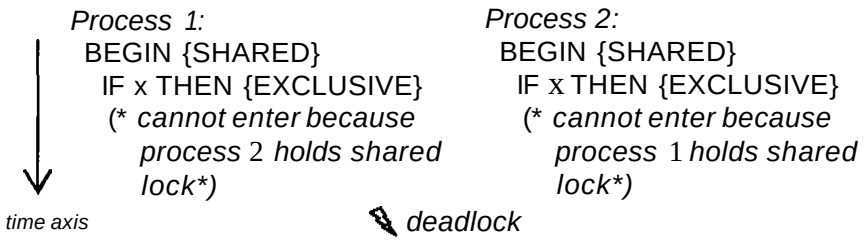


Figure 4-19. Deadlock risk of nested locks

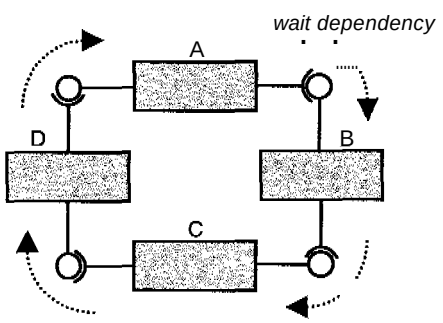
Moreover, other types of deadlocks can be eliminated in our programming language. For instance, it excludes interaction deadlocks between two components that communicate over one interface. Regarding the example of the collection, a shared lock can be safely maintained during the iteration process without danger of deadlocks. This is because the communication protocol ensures that only valid interactions are made during the iteration, Le. no element can be added in the meantime by a client and a complete iteration has to be done. Hence, the deadlock scenarios of the previous section are banned, since they would violate the protocol, as illustrated below:

```
COMPONENT User REQUIRES Collection; (* ... *)
  BEGIN
    Collection!Iterate;
    Coliection!Add(key, value) (* protocol violation *)
END User;
```

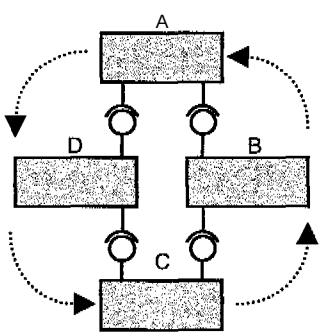
37 If procedures are supported inside a component, the following rules must hold for synchronisation: (1) A procedure, which contains a synchronisation attribute, cannot be called from another protected region. (2) A procedure may modify and read shared resources without lock, if it is only (directly or indirectly) called from a sufficiently protected region. As procedures can only be called from inside a component, the static analysis is limited to the scope of a component instance.

```
COMPONENT User REQUIRES Collection; (* ... *)
BEGIN
  Collection!Iterate;
  IF Collection?Done THEN Collection?Element(key, value) END;
  (* protocol violation: communication not properly terminated *)
END User;
```

However, the new programming language does not prevent all kinds of deadlocks because this would require us to restrict the structural flexibility of the language. As outlined in Figure 4-20, component structures with a cyclic or a split-and-join topology may indeed lead to deadlocks. For programmers willing to limit themselves to acyclic component structures, Appendix C presents three simple rules to exclude deadlocks. However in this language, a component always has the ability to control the interface connections of its contained components and therewith, the structural dependencies. Due to the formal communication protocols, a programmer also has a clear knowledge about the possible dependencies of connected and communicating components. This helps to more carefully plan cyclic relations between components and to avoid deadlocks, which are usually promoted in classical languages by insufficiently specified program dependencies.



A is locked and waits for a message from B, B waits for C, C for 0, and 0 for A, which is however locked.



C is locked and waits for a message from B, B waits for A. A awaits a message from 0 and 0 from C. which is however locked.

Figure 4-20. Potential deadlocks

4.7 Implementation Independence

Although our programming language supports the implementation of components to the smallest granularity, we also permit that *terminal components* (components which do not contain sub-components) can be implemented in any conceivable language. This is a particular flexibility resulting from the clear separation between the interfaces and the implementation of the components. Hence, a safe way of interoperability with other programming languages can be enabled. For example, this allows us to implement low-level components (e.g. device drivers) with machine-specific code[38]. One of the benefits of this approach is that our programming language can be kept general and flexible, without having to incorporate any low-level or special-purpose programming features.

It is left open to the concrete runtime system, how the interoperability with other languages is regulated. The runtime system may for example provide a specific component development interface or framework for the other languages. Typically, the following basic functionality has to be offered by such an interoperability mechanism:

- *Registration* of new types of terminal components that are implemented in the specific other language.
- Handling of external lifetime events such as *creation* and *deletion* of components.
- Support of component *cloning* and *introspection*[39].
- Generic *communication* commands for sending and receiving messages via a component's offered or required interface.

---

38 We ourselves make use of this flexibility for the runtime system which is implemented as a terminal component itself (see Section 5.1.4).

39 In this programming model, *introspection* (or *reflection)* is the ability to query the offered and required interfaces of a component at runtime.

Chapter 5

The Runtime System

Our programming language does not only offer a new paradigm of abstraction, but also poses new challenges for its implementation. To demonstrate that the language can be implemented with high efficiency on today's computer machines, we have built a new runtime system that supports the concepts of our language particularly well. Thereby, the primary use of concurrency and the pointer-free components set new requirements for our runtime system:

- *High degree of concurrency*
  The system should support a very large number of parallel processes[40], as many as possible. It should radically surpass ordinary operating and runtime systems with regard to the degree of concurrency.
- *High-performance concurrency*
  Concurrency should be very efficient, with particularly fast execution and high reactivity of all processes. All kinds of context switches have to be performed at very low costs. With regard to concurrent programs, the execution performance of

40 The tenn *process* is used in its general sense and does not mean an isolated UNIX-like process.

the runtime system should be significantly higher than in classical systems.

- *Compactness and robustness*
  With a simple design and careful implementation, the runtime system should be very compact and reliable. It has to guarantee full memory safety and should discard any superfluous artefacts, such as garbage collection and virtual memory management, which are no longer needed for our programming model.

It turned out that these requirements can not be adequately met if we use an ordinary operating system as a basis of our runtime system. This is because current operating systems only support very limited concurrency, with a maximum value of about 10,000 processes *(threads)* and with inefficient parallel execution, based on expensive synchronous and asynchronous context switches. Classical systems also impose unnecessary and obstructive infrastructural elements for our language, such as automatic garbage collection.

Due to this situation, we built a new small multi-processor kernel for our runtime system, such that the system can directly run on a conventional computer machine (off-the-shelf personal computers). As a result, our system offers the following highlights:

- The system supports *millions* of parallel processes.
- The *execution speed* of concurrent programs is by an order of magnitude faster than conventional languages and systems.

The abovementioned high scalability and performance is mainly due to the following technical innovations:

- *Fine-granular stacks*
  Processes are extremely light-weighted, with stacks that can have arbitrarily small size. In general, stack sizes do not need to grow and shrink at runtime, because the programming model uses communication instead of method

calls. Due to the fonnal protocols, the size of a communication buffer can be statically determined and pre-allocated with a defined capacity.

- *Preemption without timer interrupts*
  Preemptive execution of processes is realised by instrumented code that is automatically inserted by the compiler at required points. Processes can therefore share the processors with guaranteed time slices of arbitrarily small size. Due to the "synchronous" process switches, no unnecessary register backups have to be taken for preemption.
- *No garbage collection*
  The system ensures memory safety without need of automatic garbage collection. This is because the time for the memory deallocations is exactly defined by the hierarchical compositions. Hence, unexpected system disruptions by a garbage collector can be excluded.
- *No virtual memory*
  Virtual memory management (paging) is also no longer used, such that a general speedup on memory accesses is gained.

The entire runtime system (with the kernel) has a very compact size of less than 200KB in total and should be correspondingly reliable. We used the *ADS* kernel [MuI02] with its institutionalised *active object* [Gut97] and *monitor* [BH73, Hoare74] concepts as inspiration for the design and implementation of the kernel. Some code parts for device drivers and machine initialisation were adopted from *ADS,* though with very careful revision (see Section 5.4). The substantial part of the kernel is however new, in particular the heap and concurrency infrastructure with the above-mentioned innovations. In this chapter, we explain the design and implementation of the system and give a rationale for it. Empirical results in the form of experimental measurements and comparisons with other systems are presented in the next chapter.

5.1 Overview

Our system establishes a runtime environment for components, by representing itself also as a component that already pre-exists as a first instance (see Figure 2-3). The system component thereby runs together with application components in a network and offers necessary system interfaces, such as for loading, execution, and runtime introspection of components. The internal architecture of the system consists of a generic micro kernel and the component runtime support.

In the following sections, we give an overview of the system by describing the model of compilation and execution as well as the target platform and implementation language of the system.

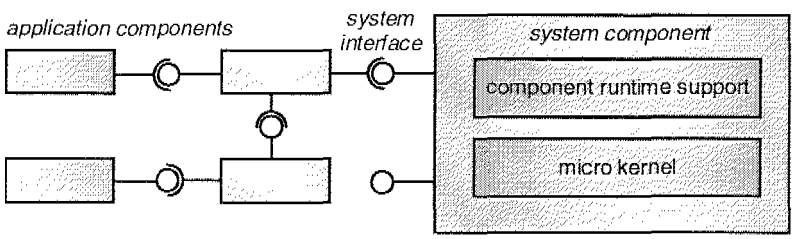


Figure 5-1. System structure

5.1.1 User Interactions

Unlike classical programming languages or systems, the component model does not engage the metaphor of "starting a program". Instead, the system enables users to directly create components in the global system scope and to connect their interfaces. With appropriate connections, a component automatically begins to run and interact. This continues as long as the component is not deleted. Figure 5-2 demonstrates how one can for example create and delete components via a user interface. The interface has been deliberately kept simple, as we focus in this work on the runtime system. For this purpose, the screen is divided into two sections, one for input and one for output. Both

sections run independently, Le. output can be written concurrently while the user is typing input commands. An initial configuration of components may be also defined by a script of such commands, executed on system start-up. The user commands (cf. Appendix B) are largely equivalent to the corresponding statements in the component language, except that the identifier for a created component does not have to be declared as a variable. For example, the identifier l is implicitly associated to the created instance of the PublicLibrary component. As a result, the user can connect the interfaces of l to other components and may also directly communicate with the component, by sending and receiving messages.

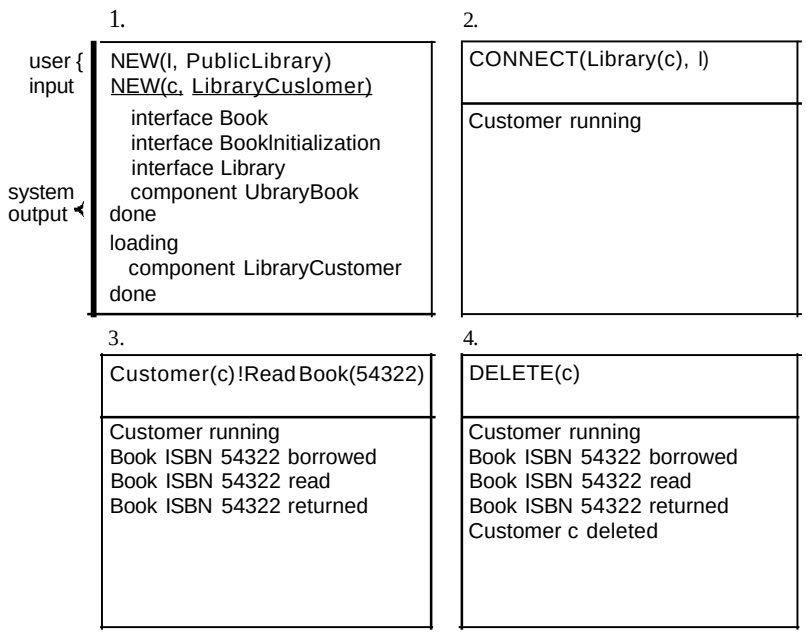


Figure 5-2. Sequence of user interactions with the system

5.1.2 **Compilation**

In our setting, the runtime system is not identical to the development system. Programs of the component language are written and compiled on the development system, while the code can be directly loaded and executed in the runtime system. As the compiler of the component language is pro-

grammed in Oberon, the programmer may use Native Oberon [WG89], AOS [Muller02] or WinAOS [Fried07] as the development system.

In order to have a compact and efficient runtime system, the compiler translates the component programs directly into machine code that can be loaded and executed by the runtime system. The internal structure of the compiler is outlined in Figure 5-3. To support portability, the compiler has been designed with a generic module interface for the code generation, such that different implementations for the specific machine platforms can be used. To port the compiler to a different platform than IA32, it suffices to only replace the two modules CCIx86CG and CCIx86A. The main code generator in module CCGenerator does not produce specific machine code and can be reused for other platforms.

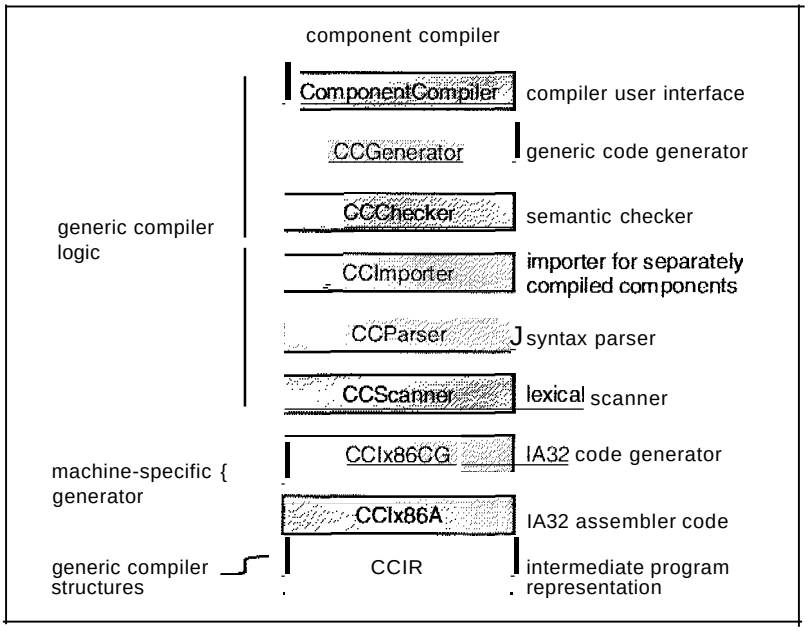


Figure 5-3. Modular structure of the compiler

5.1.3 Target Machine

We have implemented the runtime system on a conventional PC-compatible IA32 machine [IA32], with the

support of multiple processors. This platform has been chosen because it currently is one of the most prevalent computer machines and thus allows us to use our language on most machines in operation.

However, we also looked to see whether other machine architectures could offer us substantial advantages for the implementation. This is not really the case for the today's machines, as they all have approximately the same poor support for fine-granular concurrency. This is because the machine design is mainly optimised for classical procedural execution, while light-weighted processes and efficient concurrency control remain largely neglected. As a result, the operating system designers have to arrange the process structures, context switches, mutual exclusion, waiting queues, fine-granular stacks, communication channels, memory coherency etc. on their own, with no or very little help from the underlying machine and with the correspondingly poor performance. In the case of more advanced architectures, such as computer clusters, the implementer has to additionally address the explicit distribution of processes among the different machines in the cluster, which is particularly inefficient for fine-granular and frequently synchronised processes. Instead, it would be desirable that future computer machines rather offer improved support of concurrency, by providing adequate functionality for fast process switches, flexible stack management as well as efficient process synchronisation and communication. In particular, synchronisation should be realised as fast as normal sequential instructions. To achieve this, an extensive use of cache hierarchies would no longer be reasonable.

With regard to our requirements, the IA32 machine is obviously not an optimal architecture but at least, it is one of the most widespread platforms.

5.1.4 Implementation Language

In our system, normal application components are programmed in the new component language. The system

component itself has been implemented in another language, namely a rather machine-close programming language that is a reduced version of Oberon [Wirth88]. Here, we intentionally make use of the freedom of the component model, which permits any language for the implementation of terminal components (the system indeed forms such an instance). Obviously, our language would be inappropriate for implementing the runtime system on the chosen computer platform, as the primitive machine concepts can not be directly mapped to the high-level component abstractions, in order to express the system in terms of itself. Therefore, a language like Oberon is a much better choice for this purpose, seeing that the language concepts mostly correspond one-to-one to the underlying low-level representation, such as for example the defined memory layout of data types and the procedural execution. The used Reduced Oberon distinguishes itself from traditional Oberon insofar that explicit deallocation of heap blocks is required, using the newly introduced DELETE-statement. Since automatic garbage collection has become superfluous for our component-oriented programs, it would be unreasonable to only employ it for the implementation of the runtime system. Instead, the runtime system has been very carefully programmed and tested to be memory safe.

5.2 Component Support

The runtime infrastructure for the component support may be schematically described as in Figure 5-4. The compiled metadata and machine code of the component language can be directly loaded into the runtime system. The code only performs the primitive expression evaluation and execution branching, while more complex operations (such as component creations, message sending and receiving) are realised by invoking procedural system calls. Hence, the compiler does not need to make any assumptions about the specific implementation of the component structures, the process management and the message communication

inside the runtime system. As a result, the intemal representation of the component structures may be flexibly replaced or varied in the runtime system without affecting the compiler.

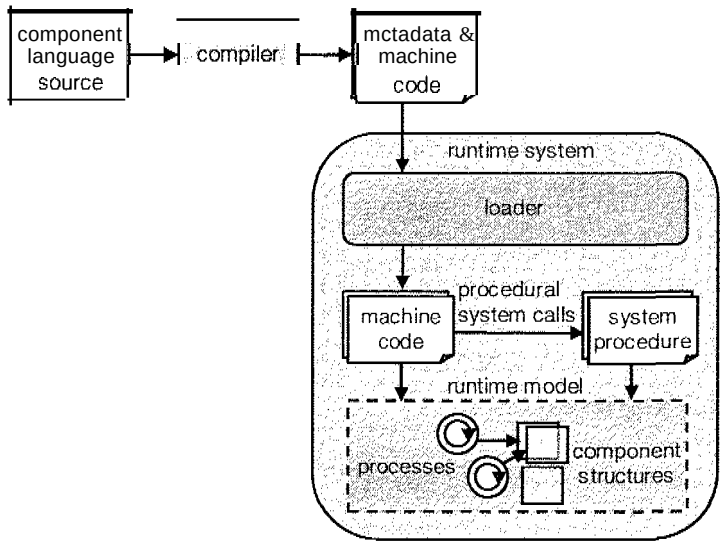


Figure 5-4. Schematic runtime architecture

The runtime system for components only involves an implementation with a code size of 160KB. The system runs on top of the elementary runtime model that is offered by the underlying micro kemel. The component-specific runtime support is organised in the Oberon modules depicted in Figure 5-5, which are linked as part of the dedicated system component (see Section 5.1).

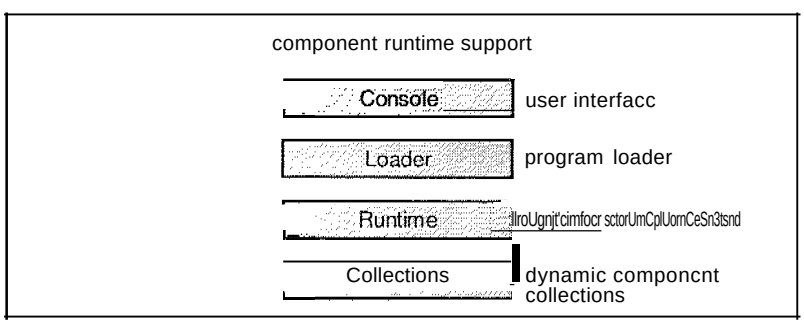


Figure 5-5. Structure of the componenI runIime support

5.2.1 Program Loader

The intermediate code of a component template or interface specification is automatically loaded "on demand", when the corresponding component or interface is used for the first time by the system user or another loaded code. Notably, the loaded units of intermediate code may have cyclic dependencies, as a component or interface can be described by using the identifiers of other component templates or interface specifications. Therefore, the system supports automatic loading of intermediate code units, even with cyclic use-dependencies. For this purpose, the loading algorithm involves two phases: a first phase collects the set of all component templates and interface specifications that are directly or indirectly used by the code to be loaded. In a second step, the system loads and atomically activates all these collected program units.

5.2.2 Memory Model

The runtime system features a very simple and uniform memory model. All kinds of program structures, such as components, processes, stacks, collections and communications are organised as memory blocks in the heap, as illustrated in Figure 5-6. The dynamic relations between these blocks are internally represented by memory addresses, such as for the interface connections, hierarchical compositions, entries in dynamic collections, and communication channels. The block of a component instance contains a slot for each offered and required interface, which is used to store the necessary interface connections. Such a connection either refers to another interface slot or to the program code of a service process, if it forms an implemented offered interface. Process blocks only need small memory space for context switch backups. In the case of a service process, the process block also includes the space for the communication buffer that is used between the service process and its individual client. The specific implementation of the processes, collections and communication is described in the following sections.

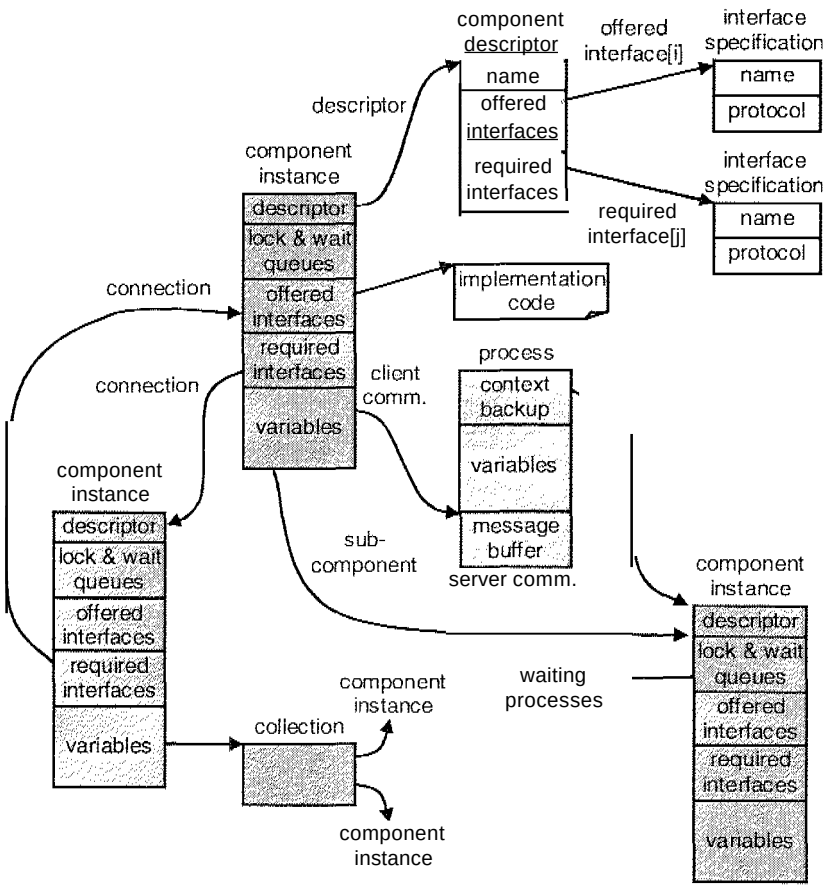


Figure 5-6. Memory representation of the component structures

5.2.3 Process Stacks

The use of very fine-granular process stacks is one of the main reasons, why our system offcrs such a high degree of concurrency. More specifically, a stack is no longer regarded as a large contiguous pre-allocated memory block of bound maximum size but rather represented as a linear list of heap blocks of arbitrary size, such that the stack can dynamically grow or shrink. Fortunately, the new programming language docs not require frequent dynamic stack extensions, as the programming language only uses communication-based component interactions in place of method calls. **In** our language, procedures can only be used *inside* a component instance, such that the processes

usually involve much shorter stacks than in classical object-oriented languages.

For each process, the backend compiler determines an individual initial stack size with a small reserve up to 640 bytes, to support the most frequent procedural system calls on the initial stack. For more complex system calls (operations on large dynamic collections), the stack size has to be dynamically increased and decreased during execution (see Figure 5-7). The stack may naturally also grow during the procedural execution, which is only allowed within a component. For this purpose, a small runtime check is instrumented at the entrance of a procedure, to determine whether sufficient stack space is present. If not, a new stack block with compiler-calculated size has to be allocated in the heap and linked to the previous stack block. The local data of a procedure however remains accessible via the base pointer, since the calling procedure *(caller)* always keeps sufficient stack reserve for the local variables of its directly invoked procedures *(callees)*[41]. This saves expensive copying of the procedure arguments in the case of dynamic stack extensions. When the procedure eventually returns, its extra stack block is deallocated in the heap.

---

41 As no virtual procedure calls are needed in our language, the set of potentially called procedures can be easily determined within each component scope individually.

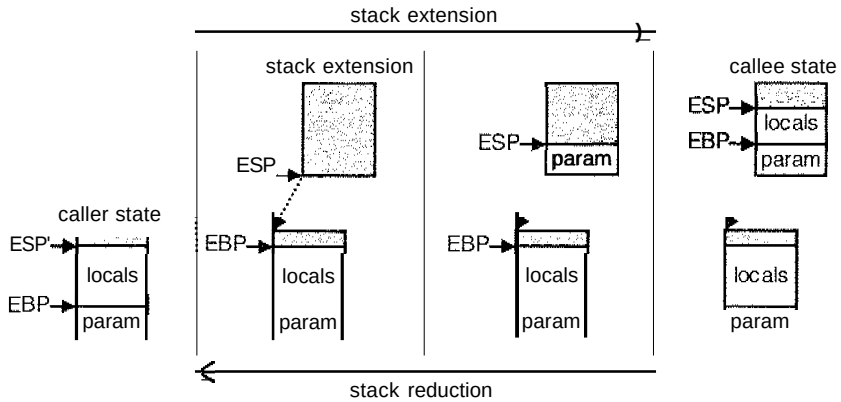


Figure 5-7. Dynamic stack extension and reduction

5.2.4 Dynamic Collections

As for the implementation of the language-inbuilt dynamic collections, the most important criteria for a corresponding data structure are both compact memory representation and fast element accessibility. For this purpose, we have implemented the collections as adaptive hash tables with a hash function that is defined on the indexes of the dynamic collection. Each entry of the hash table has sufficient preallocated space to store an clement of the collection. Therefore, elements can be inserted without requiring a dynamic allocation, which would be relatively expensive due to the necessary synchronisation of the heap. Elements with colliding hash function value are stored in other free entries and are linked together in a linear collision list (see Figure 5-8).

The size of the hash table is increased by a factor of approximately two, as soon as two thirds of its space is occupied. The size is always chosen such that it is not divisible by a small prime number. This ought to prevent frequent collisions of the hash function which is computed from the value of the clement index modulo the hash table size. If less than a third of the entries are occupied, the hash table is again reduced to approximately half its size.

Besides the dynamic collections, the runtime system also employs collections behind the scenes, if a component

reqUITes a dynamic number of interfaces with the same name.

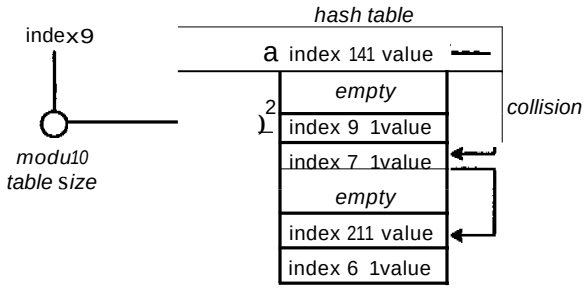


Figure 5-8. Data structure of a dynamic collection

5.2.5 Communication Mechanism

As the internal runtime model of the component structures is independent of the compiler and programming model, any implementation of communication channels could be supported, regardless of whether for local or remote communication. The currently implemented local communication channels use a conventional bounded FIFO message buffers with small size. By analyzing the largest message declaration in the communication protocol, the maximum size of the message entry in the buffer can be determined statically. Therefore, it is possible to pre-allocate a buffer with a defined buffer capacity, where the buffer capacity may vary for each communication. Measurements have shown that small sizes of about four elements are mostly sufficient, because processes can be scheduled in relatively small time slices due to the fast context switches and therefore, a message typically only remains buffered for a short time.

The fulfilment of the protocol is dynamically monitored for each channel. For this purpose, the backend compiler encodes the protocol of each interface specification as a finite state machine. This allows efficient runtime checking of the protocol, by only involving one read- and write-operation per send-statement. By using an example, Figure 5-9 shows how a protocol is translated in a finite

state machine. This can be efficiently represented in the memory as a transition table per interface specification.

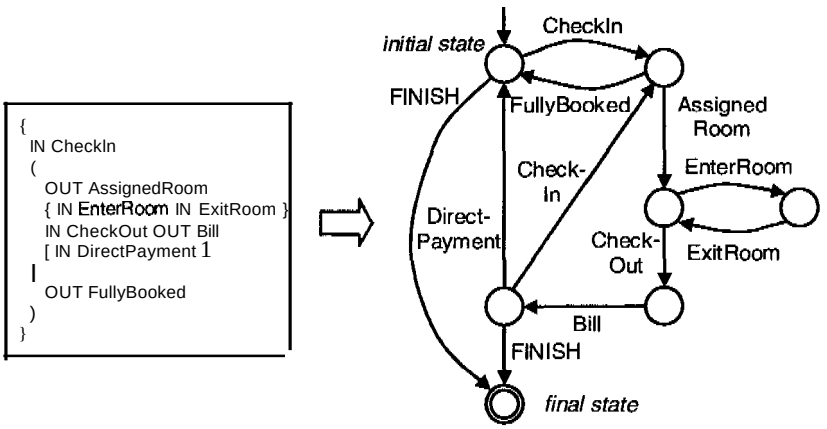


Figure 5-9. Encoding a protocol as a finite state machine

A service process is automatically incarnated in a component as soon as an external client component performs the first command of communication for the corresponding interface. When a process waits for the reception of a message or a non-full communication buffer (in the case of sending), it is internally registered at the communication buffer, such that it is immediately reactivated when the communication partner has changed the buffer state. The system ensures that at most one process is waiting at the same time for the same communication. Otherwise, the server and client process would be deadlocked, which is an error in the component program. A communication can be terminated when the client component is finalised or has sent the last message to the server side.

5.2.6 Memory Deallocations

Thanks to the concept of hierarchical composition, all run-time structures have a precisely defined deallocation time (ef. Section 3.3). The deletion of a component instance immediately leads to the deallocation of all inner components and connections among them, as well as the deletion of inner collections. A component can only be

deleted after all communications with it have been terminated. When a service process reaches the end of its statement sequence, it first awaits the termination of the client-side communication and then frees the associated communication channel. Of course, components or collections belonging to the scope of a process are also deallocated at the end of a process. Therefore, heap memory is explicitly managed by the runtime system without need of an institutionalised garbage collector to ensure memory safety. External fragmentation of the heap is possible but remains low in practice, because of the fast merging facility that is implemented in the underlying heap system, see Section 5.3.4.

5.2.7 Concurrency Model

The language abstraction of components and their internal processes are mapped to shared resources and light-weighted processes in the runtime system. Thereby, synchronous context switches of the processes involve very low costs, as only three registers (program counter, stack pointer and frame pointer) have to be saved and restored by the runtime system. Wherever a synchronous switch is generated in the code, the compiler already ensures that the other registers are not in use. Naturally, this requirement must be also fulfilled by any compiler optimisation. In our system, preemptive scheduling of processes is not implemented by conventional hardware interrupts but with a new technique of code instrumentation, which is discussed in the next section.

Processes operate directly on the corresponding shared resource of their enclosing component, by using the monitor protection (with both exclusive and shared locks). The monitor-like shared resources are implemented with two local waiting queues (see Figure 5-10), one for processes waiting for an exclusive or shared lock and one for all processes that are waiting on a Boolean condition (by the AWAIT-statement). The prioritisation among processes follows the so-called eggshell model [MuI02]. In

order of priority, processes with a fulfilled await-condition gain access to the resource, then processes waiting for the entrance with an exclusive or shared lock. This is implemented by checking the waiting queues when a process releases the monitor lock (AWAIT-conditions can only be satisfied at the release of an exclusive lock). If the condition is fulfilled for a waiting process, the corresponding process directly obtains the corresponding lock. In order to avoid potential starvation problems among reader or writer processes, a first-come-first-served strategy is used for the processes entering into an exclusive or shared regIOn.

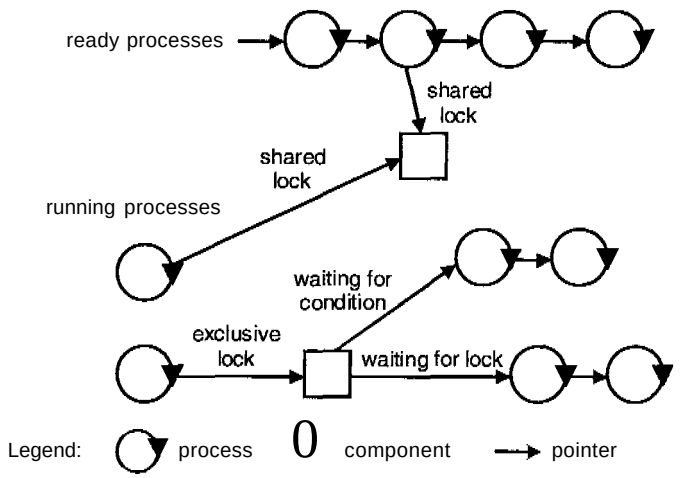


Figure 5-10. Internal representation of concurrency structures

In addition, components feature extra waiting lists for processes, which wait for a specific virtual time (cf. Section 7.2.1), or for external events such as finalisation of a sub-component. As for the processes waiting for a certain virtual time, the processes are sorted in the waiting list by their virtual time. In the case that the virtual time of component should be kept synchronous with the super-component, the passive process is directly registered at the corresponding super-component.

5.2.8 Software-Based Preemption

As a further innovation, the concurrency model does not prejudice a specific preemption mechanism. It may be realised by hardware interrupts or by code instrumentation. The latter is a novel technique which has been engaged for our component language. The compiler directly inserts *small* checks in the machine code, which continuously monitor the runtime of the process and initiate the preemption after a defined time interval. It should be noted that processes are *not* programmed for cooperative scheduling. The runtime system always guarantees time-sliced scheduling, which seems preemptive from the programmer's standpoint. Beneficially, the software-based preemption permits substantially cheaper context switches. In particular, no unnecessary state backup of all registers has to be taken on preemption because the compiler exactly knows which registers are in use and have to be spilled on the stack before preemption. As a consequence of this optimisation, processes do not require extra reserved memory for storing the backup. This design decision is another decisive reason for the high number of processes supported in the system.

In detail, the software-controlled preemption works as follows. For each process, the machine code is instrumented within extra instructions, which test how long the current process is running since the last context switch. If the process has run for a certain maximum period, the code only stores the necessary state and immediately launches the preemption of the process. To guarantee that the limits of a time slice are fulfilled, the checks need to be continuously executed in small time steps. Therefore, the checks are inserted at the following places:

- in the body of each loop statement (WHILE, REPEAT, FOR and FOREACH)
- at each procedure entry
- after each statement sequence of a maximum (worst-case) runtime

To determine when a process needs to be preempted, the runtime is measured by the timer interrupt and a reserved register (EDI[42] in our system) is set when the process should be preempted at the next check. It would be also possible that the process directly maintains a counter for the maximum worst-case execution time which is appropriately incremented by the instrumented code.

The current implementation of a check only requires a few simple instructions (see Figure 5-11) and we measured that the total cost of preemption checks is on average less than 0.5% of the total program runtime. In most cases, no registers are to be stored at the point of a preemption check, because the checks are inserted at the end of a language statement. Therefore, the preemptive context switch usually remains as efficient as a synchronous context switch for waiting on a monitor entrance or on an AWAIT-condition.

```
check = CMP EDI, 0
        JE continue
        ,save used registers
        CALL Preempt
        ,restore used registers
continue:
```

Figure 5-11. Preemption check (in IA32 assembler)

5.2.9 Smart Scheduler

It is well known that process synchronisation is quite expensive on today's computer machines due to the required synchronisation of the processor caches. In fact, the costs of synchronisation grow with the number of involved processors (see Table 5-1). Therefore, one can only gain from parallelism on current machines if the system uses long-running nearly independent processes, which need no or only very occasional mutual synchro-

42 The register may nevertheless be used in short-lasting execution (with a limited worst-case runtime). In this case, the compiler has to backup and restore the register (e.g. on the stack).

nisation. However, our programming language encourages a model of fine-granular interacting processes. Therefore, we equipped our system with a smart scheduler that only schedules processes in parallel, if they indeed run faster on multiple processors. To determine this, the system continuously measures the amount of synchronisations per time period. If the synchronisation frequency exceeds a certain threshold, the overheads of synchronisation outweigh the gain of parallelism and it is more efficient to temporarily schedule the corresponding processes in a serial but time-sliced way on a single processor.

execution time (in nanoseconds)	1 CPU	6CPUs	slowdown factor
atomic increment	32	620	19
atomic exchanae	32	620	19
spin lock	73	4000	55

Table 5-1. Costs of synchronised instructions, in nanoseconds (rounded to two figures), 6 Intel Xeon 700MHz processors, the instructions operate on the same memory word

5.2.10 Interoperability

Currently, the runtime system supports both the Component Language and Reduced Oberon as implementation language of terminal components. More languages could be added but the current selection of languages is quite appropriate for our needs. Naturally, we want to promote the new language as the standard implementation language for components. Only for very specific low-level tasks (such as implementation of device drivers) may the implementation in Reduced Oberon be necessary. The assurance of the memory safety remains the responsibility of the corresponding implementation language. For example, memory safety has to either be ensured by the language (such as for the Component Language) or has to be explicitly trusted after careful analysis (such as for Reduced Oberon). The runtime system offers a predefined programming interface in the module Runtime for the implementation of terminal components in Reduced Oberon.

5.3 Micro Kernel

The micro kernel manages the elementary machine resources (processor power and memory space) and also provides elementary runtime support (memory heap and process scheduling) for the implementation of the component runtime system. The micro kernel with *6SKB* image size is much smaller than the component runtime support and consists of the modular structure depicted in Figure 5-12. Specific device drivers such as for the keyboard and file system arc to be provided on top of the kernel.

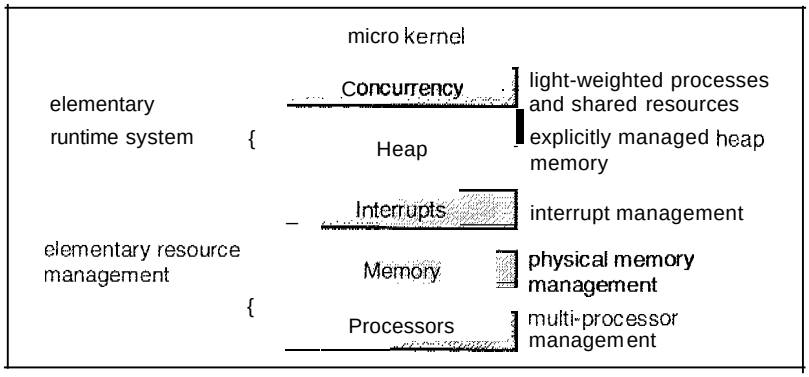


Figure 5-12. Modular kernel structure

The kernel design is inspired by AOS [Mul021, though we have implemented it for more generic runtime support (not fixed to a classical programming model) and in a more compact form. Parts of the low-level code for the processor initialisation, memory segmentation, interrupt setup, keyboard and file system driver were adopted from AOS, however with careful revision. A list of the most important technical differences can be found in Section 5.4.

5.3.1 Multi-ProcessorManagement

The first (lowest) kernel module is Processors, which supports and manages the existing (single or multiple) processors on the machine. It includes elementary spin locks for the mutual exclusion of processors in very short

critical sections. Alike ADS [Mu102], the locks can only be acquired in the linear order of the module-import hierarchy, to exclude deadlocks.

5.3.2 **Generic Memory Management**

The second module Memory implements generic management of raw main memory resources. Contiguous memory regions can be acquired for individual purposes, such as for one or multiple memory heaps or for driver-specific memory-mapped I/O. The main intention of this generic memory model is that the kernel could also support other programming languages in parallel with our component language. As a result, the specific other languages could have a runtime system which uses its own managed heap or particular stack space independently of the component runtime system. In contrast to ADS, which only support up to 2GB main memory, the new kernel is able to use the complete memory.

On an IA32 machine, memory segmentation is always provided as a feature that can not be disabled. Therefore, the *Memory* module sets up a minimum segmentation configuration, which serves at the same time to detect NIL-traps by the hardware. For this purpose, a NIL-pointer is represented as memory location that is beyond the available memory space of the defined segments. Naturally, the compiler of the kernel is also prepared to insert explicit NIL-checks in the code, if the system would be used on a machine that does not require or offer memory segmentation.

However, virtual memory management is optional and no longer used in our system. It can be identified as a redundant infrastructural element that only promotes design decisions which are not accurate anymore for our programming model:

1. Concurrency is only based on light-weighted processes, which share one address space. As the processes are written in a memory-safe program-

ming language and only interact in a well structured way, mutual isolation of processes by means of separate address spaces has become superfluous and impedimental for flexible process interactions.

2. Virtual memory often tempts system designers to realise procedural stack overflow checks by the machine-inbuilt paging mechanism. However, stacks then become unnecessarily heavy-weighted with a granularity fixed to page sizes. Stack management then also involves extra complexity due to the separate allocation in a different page-aligned memory space.
3. NIL-checks can be just as well implemented by the existing memory segmentation mechanism or by explicit instrumented checks.
4. Unnecessary complexity of distinguishing between virtual and real addresses (e.g. in drivers), as well as the effort of translation can be avoided, which is more considerable in a 64-bit address system.
5. Our runtime system is implemented in a memory-economic way, such that the real memory on current machines is sufficient to even execute very large program instances. In fact, the size of real memory on current 32-bit machines is often in the same order of magnitude than the virtual memory space, such that we could not profit from swapping. In addition, swapping usually involves unexpected and undesired latencies in the memory access time.

5.3.3 Interrupt Management

As a third elementary machine resource, software and hardware interrupts are supported by the kernel module called Interrupts. The protected processor mode is therein activated, such that an interrupt runs on a higher privilege level and on a separate kernel stack which is reserved for

each processor. The latter is actually the decisive reason for using the protection mechanism: an interrupt must not be executed on the stack of an application process, since the process' stack may be arbitrarily small.

5.3.4 Heap Implementation

Heap memory is the first infrastructure of the elementary runtime system, based on explicit free memory management. The heap permits direct allocations and deallocation of heap blocks that can have arbitrary size. The implementation is kept most generic and uniform: in contrast to other operating systems, the heap does neither distinguish artificially between block types (such as record, array, or type descriptor blocks) nor does it require particular meta-data for the heap blocks. The memory representation of heap blocks is thus as simple as illustrated in Figure 5-13.

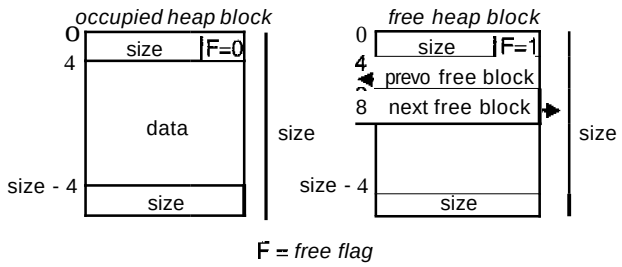


Figure 5-13. Heap blocks

Free memory is managed by segregated free lists, each storing free blocks of a defined size range. One free list is reserved as a collective list for all huge free blocks. To reduce external fragmentation, consecutive free blocks are immediately merged on deallocation. Allocation and deallocation are very efficiently implemented and only involve constant time, unless a huge block of the collective free list has to be allocated. This is possible because the allocation can directly extract a heap block from the free list which has a size range larger than or equal to the requested one. If that list should be empty, only a constant number of free lists for larger block sizes have to be consulted. Internal

fragmentation is thereby avoided, by splitting the unused part of a free block and inserting it in the corresponding free list. On deallocation, the neighbour blocks can be directly determined (due to size information at the beginning and end of a block) and merged if they are free. Sorting the blocks in a free list is hence not required.

The heap is designed to be generic and not to prescribe a specific model of memory management. For instance, the heap can be explicitly managed like in our language, where hierarchical compositions permit direct memory deallocations in a safe way. It is however also conceivable that the heap is used for another language, which requires automatic garbage collection. The collector would then have to run as a client of the heap, explicitly deallocating non-reachable heap blocks.

5.3.5 Concurrency Implementation

The elementary runtime support for concurrency is based on particularly light-weighted processes. The concurrency infrastructure is deliberately kept generic, not prescribing any specific process layout or synchronisation model (e.g. monitors). These issues have to be implemented by the specific runtime system. The basic concurrency infrastructure only provides the basic logic for process switches and process schedulers. The mechanism of preemption is left open to be either performed by timer interrupts or on instrumented software code (Section 5.2.8).

The kernel supports different stack implementations. Depending on the specific runtime system, a stack can be represented as either a single block of fixed size or a linear list of blocks (as used in our runtime system). For the first case, stack overflows can be determined by software checks at the entrance of a procedure[43]. For the second

43 The kernel reserves the ESI register for storing the stack boundary. The register can still be used for temporary computations, as the compiler automatically saves and restores the register to and from the stack.

case, checks can be used to dynamically extend the stack if necessary.

5.4 Related Work

The presented runtime system facilitates concurrency with a scalability and efficiency that has, to our knowledge, not yet been achieved in existing systems; see the next chapter for the experimental evaluations. Reasons for that is the particularly light-weighted support of concurrency. Of course, we also used many important ideas from existing systems.

Kernel design. The design and implementation of the kernel was primarily influenced by AOS [MuI02] and Native Oberon [WG89]. The code for device drivers was thereby adopted from AOS. However, the kernel offers many new technical features, such as the (1) the light-weighted processes with fine-granular stacks, (2) the software-based preemption for low-cost context switches, (3) both shared and exclusive locks for monitor-like resources, (4) a sophisticated process scheduler, (5) a uniform heap model with explicit allocations and deallocations (6), support of the complete 4GB main memory space and (7), it does not institutionalise garbage collection and (8), does not use virtual memory management. The micro kernel has a smaller size than AOS (40% smaller considering only the kernel) and is organised in only five small and meaningful modules.

Fine-granular stacks. The idea of representing a stack as a dynamic list of memory blocks is already known from existing systems [VC+03, HL+05]. In some of these systems, the stack size can however not be arbitrarily small but has to be at least of a page size (4KB) [Singularity]. Other works report on *stack sharing*, where process stacks are saved to extra memory on context switches [WD94]. A different known approach is to divide the stack into sub-parts that are used by other threads [MSB05]. In our

language and system, the stack sizes of processes can be adequately determined by the compiler, such that dynamic stack extensions only occur for relatively seldom system calls.

Communication support. Attempts have been made towards static checking of the communication protocol [RR02]. However, such an analysis typically involves model checking, which is impractical for general programs because of the highly exponential complexity[44] and conservative results. As the communication protocol in fact describes dynamic interactions of an execution process, we find it appropriate to also check the protocol dynamically.

Similar to our runtime system, the Singularity as [HL+05] incorporates a communication-oriented programming model [FA+06]. The kernel, drivers and applications are designed as separate object spaces that have to be isolated and can only interact by message exchange or via shared objects in a special exchange heap. Static compiler analysis thereby has to verify that no pointers illegally cross object spaces and that only one process may access the same object at the same time in the exchange heap. Where analysis is not possible (due to costs or accuracy), the isolation of the object spaces has to be trusted in the Singularity OS. In our approach, all normal application components ought to be programmed in the component language, such that encapsulation of the components can be inherently guaranteed.

Virtual memory. At present, most popular operating systems (like Windows and Linux) still rely on virtual memory management for process isolation. In more modern operation systems, like AOS [MuI02] and Singularity [HL+05], processes however do no longer need to be isolated by separated virtual address spaces. As these

---

44 In general, the computational complexity is exponential to the state space, defined as all possible combinations of data values in variables and the execution branches in the program code.

systems are programmed in memory-safe languages (except small parts of the kernel and driver code), all programs can safely run in the same virtual address space. However, unlike our kernel, AOS and Singularity OS utilise virtual memory management for the purpose of **NIL** checks.

Heap management. Whereas our new programming model permits safe heap management with hierarchically defined deallocations, other systems depend on automatic garbage collection in order to be memory-safe. To reduce the disruption times of garbage collection, rather complicated and expensive real-time collectors [Baker78, AEL88, Baker92, N093, CBOO, BCR03] exist. Region models [BSB+03] can be used to explicitly allocate objects in hierarchical object spaces (cf. Section 2.1.5.2). However, the use of regions is rather complicated and only optional, such that most objects are still easier allocated in a global heap, which has to be garbage-collected unless memory safety is sacrificed. The Singularity OS [HL+05] thereby takes a particularly interesting approach. As a result of the concept of isolated object-spaces, each space can be managed by its individual runtime system. Therefore, garbage collection becomes customizable at the granularity of the object spaces, meaning that the collection of one object space does no longer influence the management of others. Conceptually, our new component model also supports such customised runtime systems on the same platform, since terminal components can be implemented in their own programming language and could be supported by their individual runtime systems. However, we rather believe that it is time to use more structured programming languages with a clearly defined and closed lifecycle for components (or objects), such that the automatic garbage collection can be inherently abandoned.

Chapter 6

Technical Advances

Building a customised runtime system for a specific programming model has already often proved to result in a much more efficient solution than just using a standard system. This is particularly true for a programming language like ours, where the concepts remarkably deviate from the existing mainstream and are only insufficiently supported on conventional systems. With the new runtime system, a runtime infrastructure has been created for our component language that directly addresses the high needs of concurrency. This chapter summarises the main technical advances that have been made.

Many of the technical results have to be confirmed by means of experimental evaluations. Of course, a comparison with conventional systems is quite difficult, as standard benchmark suites only focus on conventional languages and do not agree with our substantially different programming paradigm. In particular, our focus is clearly on the support of concurrency, whilst standard benchmarks primarily engage sequential programming. Therefore, we assembled an own test suite, consisting of the following concurrent programs.

1. City. A city simulation with N houses, each featuring an internal process that consumes K units of electricity from a power plant and K units of water from a river. The power plant concurrently

produces electricity from water and is able to store up to C units of electrical energy in reserve.

2. ProducerConsumer. A producer-consumer scenario with N producers and M consumers, each exchanging K messages over a common bounded buffer with capacity C.
3. Eratosthenes. The computation of N prime numbers by the Sieve of Eratosthenes. The algorithm uses a pipeline of concurrent sieves, each filtering out multiples of a certain prime number.
4. News. A news-broadcast simulation with N customers and M reporters interacting with a common broadcasting agency. Each reporter publishes K different news messages, which are read in parallel by all customers.
5. Library. A scenario with N customers and M libraries, each storing K books. A customer uses all libraries as follows: first, they list through the entire book catalogue, then try to borrow a specific book and if available, read it and eventually return the book to the library.
6. TokenRing. A simulation of N players in a circle passing a token K times around.
7. Mandelbrot. A parallel computation of the Mandelbrot fractal, by splitting the plane into N parts. The plane consists of C points, while the number of iterations is bound by K.

The test programs can be parameterised in the number of components (N, M, K or C), allowing us to configure the scalability of the test case. The test series was implemented in our component language and in an analogous form in object-oriented languages C#, Java, and Active Oberon. Naturally, the versions for the different languages are modelled as similarly as possible, using threads in place of the intrinsic component processes. We implemented the object-oriented programs in two different ways: (1) by using communication-based interactions and (2) using

classical methods for object interactions. Evidently, the test programs of the latter implementation involve a fewer number of threads than in our language, which associates a separate service process for each communication. To realise the communication-based versions for C# and Java, we used the specific language dialects Active C# [GG04] and JCSP [Welch04], respectively. Alike our language, Active C# also runs a separate server-side process for each communication, while an object in JCSP serves all its communications by a single object-intrinsic process. All program sources are available, as described in Appendix D.

As runtime platforms, C# programs were executed with .NET CLR[45] under Windows[46], the Java version was tested with JVM[47] under Windows and, the Active Oberon program ran under *ADS*. Three different test machines served for the evaluation of the concurrency scalability:

1. *Server computer*. PC with 6 Intel Pentium 700MHz Xeon processors and with 4GB main memory.
2. *Desktop computer*. PC with 2 logical (hyper-threading) Intel Pentium 4, 2.4 GHz processors and with 1 GB main memory.
3. *Notebook computer*. PC with 1 Intel Pentium Mobile, 2.4GHz processor and with 256MB main memory.

---

45 .NET Framework version 1.1

46 Windows XP Professional version 2002 for the desktop and notebook computer. On the server computer, Windows Server 2003 R2 Enterprise Edition was used to support all six processors.

47 J2SE version 1.5.0

6.1 Degree of Concurrency

Our component runtime system was primarily designed to support a particularly high number of processes that is clearly beyond the capabilities of existing systems. In fact, this goal has been achieved, as we can show by measuring the maximum supported number of processes on our runtime system and on other systems. For this purpose, we limited ourselves to test programs that allow an increasing number of processes with a proportional growing execution performance. Due to the dissimilarities of the programming models, the test programs also need different configurations in the different languages to reach the same number of processes.

Tables 6-1 to 6-3 summarise the maximum number of processes for the three machines. With regard to the number of processes, our system outperforms traditional systems by an order of two to three magnitudes. The component system indeed permits millions of processes, where the number of processes scales linearly with the available size of physical main memory. As the test programs employ processes and components with different granularities, the degree of concurrency varies among the different test programs. For instance, the City program only requires rather simple interactions with a small amount of memory per component, whereas the TokenRing program occupies somewhat more memory space for interface connections. The evaluation clearly shows that classical systems only allow a very small number of threads. This deficiency can be traced back to the heavy-weighted stack design in those systems, as well as in their high memory demands for preemption backups. For C# and AOS, the degree of concurrency is even artificially limited by a constant on all machines, regardless of the available memory space.

Program	Component System	CN	ActiveCN	Java	JCSP	ADS
City	5010,000	1896	1,894	9,999	9,999	15,700
Ubrary	3,060,000	1,896	1,892	9,999	9,999	15,700
TokenRing	4,120,000	1,896	1,896	9,999	9,999	15,700

Table 6-1. Maximum number of processes, server machine (4GB main memory)

Program	Component System	CN	Active CN	Java	JCSP	ADS
City	1 300000	1 956	1954	7,126	7126	15700
Ubrary	810,000	1,958	1,947	7,126	7,126	15,700
TokenRino	1,090,000	1,956	1,954	7,126	7,126	15,700

Table 6-2. Maximum number of processes, desktop machine (1GB main memory)

Program	Component System	CN	Active CN	Java	JCSP	ADS
City	321,000	1,954	1,954	7,233	7,233	15,700
Library	200,000	1,954	1,947	7,233	7,233	15,700
TokenRino	272,000	1,954	1,954	7,233	7,233	15,700

Table 6-3. Maximum number of processes, notebook machine (256MB main memory)

Notably, the supported number of processes on our system is even significantly higher than on other systems, which also implement dynamic stack management. Such systems only support about 100,000 threads [VC+03]. A recent extension of AOS [Keller06] permits more threads but only if the threads (active objects) do not invoke procedures or methods and do not interact by monitor waiting. Clearly, this is impractical as the threads can no longer perform any non-trivial tasks. Our system does not impose such restrictions, as the dynamic stack management is supported for every process. Hence, the processes within components can always interact by message communication and can invoke component-internal procedures.

6.2 Execution Performance

The new runtime system also surpasses conventional systems with regard to execution performance of concurrent programs. To demonstrate this, we have measured the execution times of the test programs with specific configurations. Tables 6-4 to 6-6 denote the average runtimes in seconds for ten subsequent executions of each test example. All values are rounded to three significant figures and the best result for each test case is highlighted. For each measurement, the last column of the table denotes the speedup of our component system compared to the fastest other system. According to these measurements, the component system outperforms the other communication-based programs of Active C# and JCSP with a faster execution, by a median speedup factor between 11 and 15. In general, the component-oriented programs also run significantly faster than the analogous method-based programs. The average speedup of our runtime system is between 18 and 40, compared to the fastest other system. The median of the speedup to the best other system is 2.6 for the server, 2.0 for the desktop and 1.9 for the notebook computer. Naturally, this performance advantage is mainly due to the underlying concurrency model, which offers low-cost context switches, fast software-controlled preemption and shared monitor locks. Only for some simple programs with few concurrent processes do some method-based versions run faster on certain computer machines. This is however only the case because our language uses a higher number of processes for the same programs, i.e. extra service processes are involved for the client-individual communications. Therefore, the ProducerConsumer program (with N=M=l) requires four processes in our language and only two threads (or active objects) in the method-based languages. Moreover, the component language has not been particularly optimised for sequential or mathematical programming (such as for the Mandelbrot program) but rather concentrates on high-level concepts, such as dynamic

collections, interface connections or message communIcation.

Program	*Componen Sysfe*	C#	*Active* C#	*Java*	*JCSP*	*AOS*	*speedup of the new system*
City (N=1000, K=10, C=100)		0656	- (out of memory)	437	15.0	4.09	2.73
City (N=10,000, K=10, C=100)		- (out of memory)	- (out of memory)	- (out of memory)	- (out of memory)	629	225
ProducerConsumer (N=M=1,C=10, K=1,000,000)	7.9	4.34	-(intemal deadlock)	33.0	125	27.9	0.55
ProducerConsumer (N=M=2 C=10 K=1,000',000)'	15.9	18 8	(internal deadlock)	134	258	59.7	1.18
ProducerConsurner (N=M=5,C=10, K=1,000,000)	40	181	- (internal deadlock)	372	641	153	3.83
Eratosthenes (N=10,000)	1 '7	6.76	10.4	4.63	17.5	5.83	2.62
Eratosthenes (N=100,000)	115	- (out of memory)	- (out of memory)	(out of memory)	- (out of memory)	353	237
News (N=1000, M=10 K=10)		3.49	- (out of memory)	3.88	8,68	3,73	1.76
News (N=10,OOO, M=10, K=10)	23.9	- (out of memory)	- (out of memory)	- (out of memory)	- (out of memory)	432	1.81
Library (N=1000, M=10, K=10)	139	0.741	- (out of memory)	1.47	12.5	0.585	0.42
Library (N.I 0,000, M.I 0, KIO)		- (out of memory)	,(out of memory)	- (out of memory)	- (out of memory)	45,7	3.24
TokenRing (N=1000, K=1000)	2	21.8	83.5	21.9	32.0	17.7	8.51
lokenRing (N=1000, K=10.oo0)	20.7	211	941	207	314	176	8.50
TokenRing (N=10,000, K1000)	36.9	- (out of memory)	- (out of memory)	- (out of memory)	- (out of memory)	221	5.99
Mandelbrot (N=100, K=5,000, C=3,200)	0.875	0.425	0.486	0.391	0.434	.600	0.45

Table 6-4. Execution times in seconds, server machine (6 processors, 700MHz each)

Remarks:

- The reason for programs running out of memory is that they do not support the required numher of threads.
- Active C# has an internal error in its runtime system, which causes the ProdueerConsumer program to gct stuck after ahout 40,000 exchanged elements.

Program	Component System	C#	ActiveC#	Java	JCSP	AOS	speedup oft/Jenew system
City (N-I000, K=10, C=100)		559.0	- (out of memory)	195.0	10.0	2.4	20.5
City (N=10,000, K=10, C=100)		- (out of memory)	- (out of memory)	- (out of memory)	- (out of memory)	423,0	353
ProducerConsumer (N=M=1,C=10, K=1,000,000			- (intemal deadlock)	17.1	60.7	5.25	0.85
ProducerConsumer (N=M=2,C=10, K=1,000,000			- (internal deadlock)	51.4	1210	10.7	128
ProducerConsumer (N=M=5,C=10, K=1,000,000)		127.0	- (internal deadiock)	140.0	298.0	26,9	1.46
Eratosthenes (N=10,000)		8.23	5.69	6.04	20.9	2.34	2.69
Eratosthenes (N=100,000)		- (out of memory)	- (Ollt of memory)	- (out of memory)	- (out of memory)	145	1.96
News (N_1000, M=10, K=10)		3.36	- (out of memory)	2.00	418	0635	1.01
News (N=10.000, M=10, K=10)		- (oul of memory)	- (out of memory)	- (out of memory)	- (out of memory)	6.34	0.41
Library (N=1000, M=10, K=10)		.545	- (out of memory)	0.572	5.65	0.474	0.78
Library (N_10,000, M=10, K=10)		- (out of memory)	- (out of memory)	- (out of memory)	- (out of memory)		4.89
TokenRing (N=1000, K=1000)		10.2	39.4	11.0	14.4	3.36	265
TokenRing (N=1000, K=10,000)		973	4300	108.0	141,0	33,4	2,57
TokenRing (N=10.000, K=1000)		- (Olit of memory)	- (out of memory)	- (out of memory)	- (out of memory)	33.7	2.65
Mandelbrot (N=100, K=5,000, C=3,2oo)		0,467			0.391	1.50	0.34

Table 6-5. Execution times in seconds,
desktop machine (2 logical processors, 2.4GHz)

Program	Component System	C#	ActiveC#	Java	JCSP	ADS	speedup ottlJenew system
City (N=1000, K=10, C=100)		65,6	- (out of memory)	20,2	6.47	2.80	28.5
City (N=10,000, K=10, C=100)		- (out 01 memory)	- (out 01 memory)	- (out 01 memory)	- (out o(memory)	542	543
ProducerConsumer (N=M=1,C=10, K=l,Ooo,ooO)	2.59	48.8	- (internal deadlock)	3.28		2.33	090
ProducerConsumer (N=M=2,C=10, K=1,000,000	7,16	96,3	- (internal deadloek)	7.43	125	4.68	065
ProducerConsumer (N=M=5,C=10, K=1,000,000)	14.9	224	- (internal deadlock)	558	168	12.0	081
Lratosthenes N=10,OOO)	0:504	35.9	5.55	8.43			4.13
Eratosthenes (N=100,OOO)	60.7	- (out of memory)	• (out of memory)	- (out of memory)			255
News (N=1000, M=10, K=10)	0,641	0.921	- (out of memory)	1.73			0,75
News (N=10,000, M=10, K=10)	21.5	- (out of memory)	-(outol memory)	- (Ollt 01 memory)	- (out of memory)	4.86	0,23
Library (N=1000, M=10, K=10)	0.524	0,597	- (out of memory)	0.434		0.141	0,27
Library (N=10,Ooo, M=10, K=10		- (out of memory)	- (Ollt of memory)	- (out of memory)	- (out of memory)		10,6
lokenRing (N=1000, K=1000)		58.7	74,1	8,74	13,0	2,62	2,11
lokenRing (N=1000, K=10,000)		605	763	90.5	127	26.5	1.93
TokenRing (N=10,000, K=1000)		- (out of memory)	-(olltof memory)	- (Ollt of memory)	-(outof memory)	26.7	2,01
Mandeibrot (N=100, K=5,000, C=3,200)	1.78	0,623	0,623	0,557		,900	0.30

Table 6-6. Execution times in seconds, notebook machine (I processor,2.4GHz)

Remark:

- .ICSP seems to have a performance problem for the ProducerConsumer program on the notebook machine.

Of course, the runtime system offers a higher performance with an increasing number of processors. For instance, the Mandelbrot problem takes 5.40 seconds with a single processor on the scrvcr computer and has a runtime of 0.875 seconds in the case of six processors, resulting in a speedup of factor 6. Howcver, the remaining programs are not faster (and also not substantially slower) with more processors. This is because they involve frequent synchronisation due to the communications and monitor protection,

which is extremely slow on today's multi-processors computer machine.

6.3 Absence of Garbage Collection

A particular advantage of our new runtime system is the absence of an automatic garbage collector, to ensure memory safety in our programming language. Due to the hierarchical lifetime dependencies, not even reference counting has to be engaged. In fact, inner components can be automatically deleted when the surrounding component is disposed of. This is safe in our language, because inner components are always completely encapsulated and can not be directly accessed from outside. A single component may only be deleted explicitly in the program, if no required interfaces are connected to the corresponding component (cf. end of Section 3.4).

Without a garbage collector, our runtime system does not only have a substantially simpler implementation, but also prohibits unexpected system disruptions. Unlike Other runtime systems for object-oriented or pointer-based languages, our system does no longer have to be stopped by a global garbage collector. To illustrate the difference, we have performed a measurement series of 500 subsequent executions of the TokenRing program (N=K=lOO). While AOS (Figure 6-1) suffers from continuous peaks as a result of the garbage collector disruptions, our runtime system (Figure 6-2) exhibits a nearly constant execution time among all iterations. Moreover, AOS stops with a trap after about 900 iterations, as the garbage collector is sometimes not fast enough to reclaim the stacks of the terminated processes. This is because the process stacks have to be cleaned up by *finalisers*, which do not have any guaranteed execution time in the system.

The only time t1uctuations that occur in the component system are due to the non-predicable process synchronisation and caching effects. In fact, such synchronisations

happen very frequently in our model and for the implementation of any lock construct, tlle machine only supports very primitive atomic *test-and-set* instructions that do not give any time guarantees. In other words, our runtime system is not yet a real-time system because of the concurrency support of the hardware. However, the concept of pointer-freedom in the language and the absence of garbage collection in the system are already a substantial step in this direction. Supposing that we had an appropriate machine with real-time concurrency or that we would integrate the concept of component struchlres in a sequential programming model, the realisation of such a system would be conceivable.

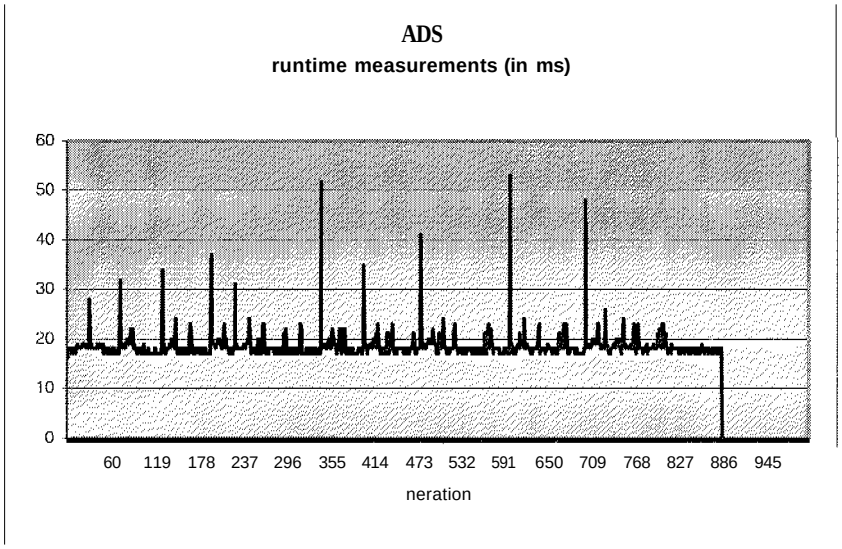


Figure 6-1. Series of runtime measurements with the TokenRing program (N=K=100) under AOS

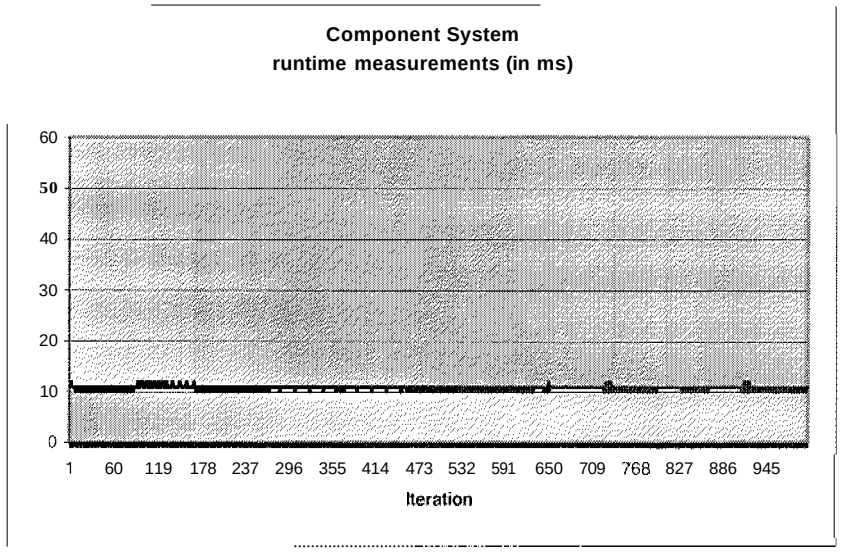


Figure 6-2. Series of runtime measurements with the TokcnRing program (N=K= I(0) under the component system

Chapter 7

Case Study

Concepts and systems are best evaluated experimentally by applying them to a real problem. For this purpose, we have performed a case study with the new programming language. As object-orientation was originally invented for simulation programming [DN66] and our component model should be more general than objects, we found it apt to also situate the study in the field of simulation programming. As a good example of a simulation problem, we focused on traffic simulation. It allows us to apply our language to a natural and practical problem and enables comparison to a canonical solution. In summary, the goals of the traffic simulation study were to analyse the following issues:

- *Modelling*
  It should be determined whether and how our programming language conveys a more natural description of the simulation program. We also investigate on the expressiveness of the pointer-free language, in particular the hierarchical composition.

- *Performance*
  The execution speed of our concurrent simulation program should be compared to a classical

sequential discrete event simulation system, which is developed in an ordinary object-oriented programming language.

7.1 Project Overview

In this project, we developed a traffic simulation package in our language. The package covers all features a traffic simulation should have, such as a detailed simulation of car movements on a network of roads, route planning, utility-based decision of departure times, as well as learning processes from traffic jams [Nagel05]. Thereby, we intentionally modelled the simulation in the spirit of the new programming language, without being influenced by the implementation of traditional simulations. The result is eventually compared to a traditional traffic simulation package [Nage105, Part IT], developed in an ordinary programming language (which is in this case C++). The two simulations differ significantly in their program models: the component-oriented simulation emphasises concurrent and autonomously driving vehicles, whereas the classical traditional simulation schedules the movements of all vehicles in a sequential and globally directed way. As a common feature, both simulation packages support roads based on cellular automata and on queues. It can be determined for each simulation, which road implementation is used.

7.2 New Traffic Simulation

The traffic simulation package that we have developed in the component language differs substantially from classical traffic simulation models. More specifically, the component-based simulation engages the following new concepts:

- *Autonomous cars.* All vehicles constitute autonomous components that drive concurrently. There

is no explicit global instance that directs all car movements on the road network.

- *Virtual time.* All vehicles run with a virtual time that corresponds to the time of the simulated world. The vehicles are exactly coordinated with regard to a virtual time, such that vehicles advance on the road in a time-synchronous way.
- *Individual planning.* The drivers of the vehicles plan their trip and route individually by only using their own experience of previous journeys. No global knowledge by a central authority is presumed for the cars.
- *Car transports.* Vehicle drivers can use car transportation services offered on certain routes. For this purpose, cars can be packed in a train or in a ferry and can so be transported. This is modelled in our language as a hierarchical composition, where car components are indeed contained within the transportation vehicle.

7.2.1 Virtual Time

For the purpose of simulation programming, the component language features an inbuilt concept of virtual time that can be used to directly represent the simulated time of the natural world within the program. Therefore, each component features an intrinsic discrete virtual time that is by default independent of the other components. The use of the virtual time is thereby defined as follows. All computing operations happen within zero virtual time, describing instantaneous actions or reactions in the simulated world. By use of the PASSIVATE-statement, a process may declare that the virtual time proceeds by a certain amount of time. The system automatically ensures that the execution after the PASSIVATE-statement only continues when all other processes within the same component wait at least for this virtual time.

PASSIVATE(duration)	The process suspends until a certain virtual time period has elapsed.

In addition, a process may also wait for an undefined virtual time, as long as a certain condition is not satisfied. This is done by using the AWAIT-statement, which suspends the execution until a certain Boolean condition is established for the first virtual time.

AWAIT(condition)	The process waits until a certain condition is fulfilled, now or in future of the virtual time.

It should be noted that with these semantics, the implementations of classical time-less components (which do not execute any PASSIVATE-statements) remain unchanged.

A process can read the inbuilt variable TIME of a component, to determine the current virtual time inside the component.

TIME	Read-only access to the current virtual time

As in nature, a component may contain immovable sub-components, which have the same virtual time like the container (no relativistic time dilatations due to different speeds or accelerations). A component may identify such sub-components by annotating the attribute SYNCHRONOUS to the variable. The system then ensures that the virtual time of the sub-components is exactly time-synchronous to the outer instance.

VARIABLE x: T {SYNCHRONOUS}	Sub-component x has the same virtual time like its surrounding component.

It is noteworthy that the SYNCHRONOUS attribute does not influence the perception of the virtual time inside the sub-component. This means that the implementation of a component does not have to be prepared for being used in a time-synchronous way. Therefore, the same component template can be used to create both time-synchronous and time-independent component instances.

7.2.2 **Main Structure**

The main structure of the simulation program is depicted in Figure 7-1. The Car components concurrently interact with the RoadNetwork component, which is made up by road links. Each car features its personal planning instance, that records the detailed travel times of the car and computes the best route and departure times at the level of the locally observed times. In order to determine a feasible route, the planner component (and therefore also the Car) needs information from the RoadMap interface.

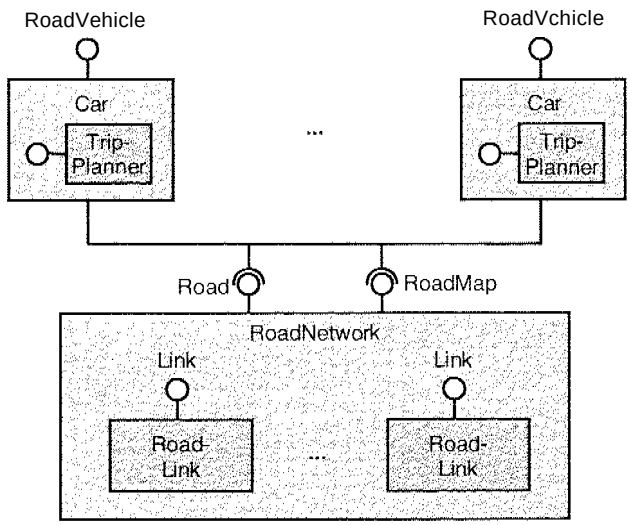


Figure 7-1. Main structure of the traffic simulation

7.2.3 **Road Network**

The RoadNetwork component contains a collection of link components, which are dynamically constructed according to an XML road network file (road links are therein encoded with integer identifiers). In this simulation package [NageI05J, road nodes do not contain any logic and therefore, do not need to be represented as components. The implementation of the Road interface directly performs the transitions over road junctions. The concrete template of the road links is deliberately left open, such that different implementations of the road links can be used. It is

important that the sub-components of the road links are synchronous with the virtual time of the road network itself. This ensures that all road parts run with equally fast virtual time, such that cars can drive from one road link to another without virtual time differences. Otherwise, a car could enter a road link at a time which already lies in the virtual past of the link.

```
COMPONENT RoadNetwork OFFERS Road, RoadMap;
  VARIABLE
    link[id: INTEGER]: ANY(Link) {SYNCHRONOUS};
  BEGIN
    (* construct network *)
END RoadNetwork;
```

The Road interface allows cars to drive an arbitrary route on the road. At each junction, a car has to decide which way should be taken next. The carId parameter is only needed here to allow global snapshots of the road network for the user output.

```
INTERFACE Road;
  IN Prepare(carId: INTEGER)
  {
    IN Start
    { IN Drive(linkId: INTEGER) OUT Departed OUT Arrived}
    IN Stop
  }
END Road;
```

The implementation of the Road regulates the driving activities of the cars on the road network and the branching at junctions. It is noteworthy, that a car only exits a link on a junction, when it can enter the desired next link. During the moments of waiting, virtual time may naturally elapse (using the AWAIT-statement).

```
IMPLEMENTATION Road;
VARIABLE carId, from, to: INTEGER;
BEGIN {SHARED}
 ?Prepare(carId);
 AWAIT(INPUT(ANY)); (* await any message *)
 WHILE ?Start DO
  ?Start(time); from := NoLink;
  AWAIT(INPUT(ANY));
  WHILE ?Drive DO
   ?Drive(to);
   Link(link[to])!Enter(carId);
   AWAIT(INPUT(Link(link[to]), Entered));
   Link(link[to])?Entered;
   !Departed;
   IF from # NoLink THEN Link(link[from])!Exit END;
   AWAIT(INPUT(Link(link[to]), EndReached));
   Link(link[to])?EndReached;
   from := to;
   !Arrived;
   AWAIT(INPUT(ANY))
  END;
  IF from # NoLink THEN Link(link[from])!Exit END;
  ?Stop;
  AWAIT(INPUT(ANY))
 END
END Road;
```

Depending on the road topology and decisions of the cars, the traffic simulation may run into "natural" deadlocks. For example, it is conceivable that, given a circular road sub-network, all links are fully congested and the exiting car of each link tries to enter another jammed link in a circular way, as illustrated in Figure 7-2. Similar to traditional simulations, we resolve such a problem by means of a timeout. If cars wait longer than a defined duration for entrance to a certain road, the jam is unblocked by allowing the cars to enter in spite of the congestion.

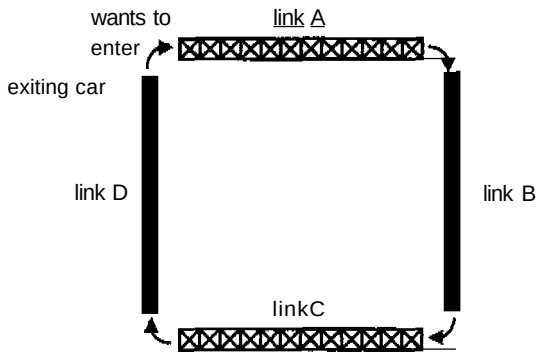


Figure 7-2. Deadlock situation: the exiting car of link D wants to enter link A, while the exiting car of A tries to enter B and so on.

7.2.4 Road Links

The RoadLink component constitutes the smallest building block of a road network, representing a single unidirectional roadway. As already mentioned, our simulation supports two implementations of road links, namely one based on cellular automata and another based on queues. The latter implementation is typically employed in practice for large-scaled simulations, as it offers higher performance [Nagel05, Chapter 18] (but is less accurate for congestion effects on the road link). The Link interface allows cars to enter the link, if there is sufficient space, at the beginning of the road. Subsequently, the car remains on the road until the end is reached and the car has left the road.

```
INTERFACE Link;
  IN Enter(carId: INTEGER) OUT Entered
  OUT EndReached IN Exit
END Link;
```

7.2.5 Cellular Automata

One implementation of the road links models them as cellular automata. For this purpose, the links are divided into small cells, each able to contain at most one car. The cars continuously adapt their speed to the free road space ahead of their current location.

```
COMPONENT RoadLinkCA OFFERS Link;
 VARIABLE
  cell[position: INTEGER]: BOOLEAN;
  exit: INTEGER;

 IMPLEMENTATION Link;
 VARIABLE carId, pos, speed, k: INTEGER;
 BEGIN {EXCLUSIVE}
  ?Enter(carId);
  AWAIT(cell[O] = NoCar);
  cell[O] := carId;
  !Entered;
  pos := 0; speed := 0;
  WHILE pos < exit DO
    IF cell[pos + 1] # NoCar THEN
      speed:= 0;
      AWAIT(cell[pos + 1] = NoCar)
    END;
    IF speed < maxSpeed THEN INC(speed) END;
    PASSIVATE(1); (* wait for one virtual second *)
    k := 1;
    WHILE (pas + k < exit) AND (k < speed)
      AND (cell[pos + k + 1] = NoCar) DO INC(k) END;
    speed := k;
    cell[pos + speed] := carId;
    cell[pos] := NaCar;
    INC(pos, speed)
  END;
  !EndReached;
  AWAIT(INPUT(Exit));
  cell[exit] := NoCar;
  ?Exit
 END Link;
END RaadLinkCA;
```

7.2.6 Queue-Based Links

The other implementation of the road links that is offered by our simulation uses a queue. The cars that enter the corresponding road link are put into a queue and are removed from the other side when they exit. With this implementation, a car can only exit after its front car has left the road. However, a car must not exit earlier than a certain minimum time which is required to drive over the clear road with free speed. The road also has a maximum capacity, limiting the number of cars that can be on the

road at the same time. Moreover, at most one car may enter the road at the same virtual time.

```
COMPONENT RoadLinkQB OFFERS Link;
  VARIABLE
    freeTravelTime, maxCars, cars: INTEGER;
    lastEntered, lastExited: INTEGER;
    canEnter: BOOLEAN;

  IMPLEMENTATION Link;
  VARIABLE carId, current: INTEGER;
  BEGIN {EXCLUSIVE}
    ?Enter(carId);
    AWAIT(canEnter); canEnter:= FALSE;
    INC(lastEntered); current := lastEntered;
    INC(cars);
    AWAIT(cars < maxCars);
    !Entered;
    PASSIVATE(1);
    canEnter := TRUE;
    PASSIVATE(freeTravelTime);
    AWAIT(lastExited = current - 1);
    !EndReached;
    AWAIT(INPUT(Exit));
    lastExited := current; DEC(cars);
    ?Exit
  END Link;
END RoadLinkQB;
```

7.2.7 Cars

A Car component encapsulates the more complex logic of an individual vehicle driver. A separation between a vehicle and driver component is here economised for the sake of simplicity. Each car offers the RoadVehicle interface that roughly defines the interest of an individual driver, i.e. the destinations they want to head for during their day-activities (as specified by the activities file). The car driver plans the detailed route and the departure time on its own.

```
INTERFACE VehicleActivitySetup;
  IN Initialize(carId, startLinkId: INTEGER)
  { IN Destination(linkId, desiredArrivalTime: INTEGER) }
  IN End
END VehicleActivitySetup;
```

Naturally, cars may be also instructed to drive according to predefined route plan. In this case, the following interface of the car is used:

```
INTERFACE VehiclePlanSetup;
  IN Initialize(carId: INTEGER)
  {
    IN BeginRoute(startLinkId, endLinkId, departTime: INTEGER)
    { IN Link(linkId: INTEGER) } IN EndRoute
  }
  IN End
END VehiclePlanSetup;
```

After initialisation, the car can be started by the Vehicle interface.

```
INTERFACE Vehicle;
  { IN Start OUT Ready IN Depart OUT Arrived}
END Vehicle;
```

A car only begins its journey after waiting for the right virtual departure time of the current day. However, if the car has arrived too late from a previous journey and the desired departure for the next trip has already passed, it starts immediately. The car trip continues as long as the car has not yet arrived at its final destination. During the trip, the car measures the departure and arrival time for each road link.

```
COMPONENT Car
  OFFERS Vehicle, VehicleActivitySetup, VehiclePlanSetup
  REQUIRES Road, RoadMap;

  VARIABLE
    carId, nofDestinations: INTEGER;
    bestDepartTime[destinationNo: INTEGER]: INTEGER;
    routeLength[destinationNo: INTEGER]: INTEGER;
    linkOnRoute[destinationNo, position: INTEGER]: INTEGER;

  IMPLEMENTATION Vehicle;
  VARIABLE linkId, departTime, arrivalTime, i, k: INTEGER;
  BEGIN {EXCLUSIVE}
    RoadIPrepare(carId);
    WHILE ?Start DO
```

```
      ?Start; PlanDay; !Ready; ?Depart;
      FOR i := 1 TO nofDestinations DO
       departTime := bestDepartTime[i];
       IF departTime > TIME MOD Day THEN
        PASSIVATE(departTime - TIME MOD Day)
       END;
       Road!Start;
       FOR k := 1 TO routeLength[i] DO
        linkid := linkOnRoute[i, k];
        Road!Drive(linkId);
        AWAIT(INPUT(Road, Departed));
        Road?Departed; departTime := TIME MOD Day;
        AWAIT(INPUT(Road, Arrived));
        Road?Arrived; arrivalTime := TIME MOD Day;
        RecordTravelTime(linkId, departTime, arrivalTime);
       END;
       Road!Stop
      END;
      PASSIVATE(Day - TIME MOD Day);
       (* wait for end of the day *)
      !Arrived;
      AWAIT(INPUT(ANY))
     END
   END Vehicle;

   (* ... *)
  END Car;
```

7.2.8 Route Planning

In this simulation, cars do their individual trip and route planning. For this purpose, each Car features a personal TripPlanner component that integrates the logic for a shortest path computation. The trip planner accepts feedback via the PlanAdjustment interface, stating how long it previously took to travel on a certain road. With this information, the component adapts the route plans to the locally observed congestion.

```
INTERFACE TripPlan;
 IN Compute(startLinkId, endLinkId, desiredArrTime: INTEGER)
 OUT Plan(bestDepartTime: INTEGER)
 { OUT LinkOnRoute(linkId: INTEGER; transport: BOOLEAN) }
 OUT End
END TripPlan;
```

```
INTERFACE PlanAdjustment;
  IN TravelTime(linkId, departTime, travelTime: INTEGER}
END PlanAdjustment;

COMPONENT TripPlanner
  OFFERS TripPlan, PlanAdjustment
  REQUIRES RoadMap;
  (* ... *)
END TripPlanner;
```

With the trip planner, each car determines its own subjectively optimal departure time, using a utility-based approach. For this purpose, the trip planner also employs an inner route planner component, which calculates the shortest paths on the road network. In contrast to the classical simulation, we do not try all possible departure times (in the granularity of time frames), as this would take too long if computed separately for each car. Instead, we use another heuristic for determining a reasonable departure time: first, an arbitrary departure time is selected and the corresponding expected arrival time is computed; then, the departure time is adjusted considering whether the time difference results in the car being too early or too late. This value finally serves as the departure time for the next iteration step of this heuristic process.

```
bestUtility := MIN(REAL}; depTime := desArrTime;
FOR i := 1 TO PlanStepsPerRoute DO
  RoutePlan(routePlanner}!CalcRoute(depLink, arrLink, depTime};
  IF RoutePlan(routePlanner}?NoRoute THEN
    RoutePlan(routePlanner}?NoRoute
  ELSE
    RoutePlan(routePlanner}?BestRoute(arrTime};
    utility := CalculateUtility(depTime, arrTime, desArrTime};
    IF utility> bestUtility THEN
      bestDepartTime := depTime; bestUtility := utility;
      RoutePlan(routePlanner}!GiveRoute; k := 0;
      WHILE RoutePlan(routePlanner}?Link DO
        INC(k}; RoutePlan(routePlanner}?Link(linkOnRoute[k]}
      END;
      routeLength := k;
      RoutePlan(routePlanner}?EndOfRoute
    ELSE
      RoutePlan(routePlanner}!DiscardRoute
    END;
```

```
    INC(depTime, desArrTime - arrTime);
    IF depTime < a THEN depTime := a END
  END
```

7.2.9 Car Transports

Our simulation has been extended to also support car transportation on defined links[48]. For this purpose, cars can be loaded into a transportation vehicle, which could be for example a train or a ferry. As a result, the cars are transported as part of the train or ferry. The transportation can be directly modelled by using the hierarchical composition offered by the language. The transportation process works as illustrated in Figure 7-3, involving the following steps:

1. A car has to signal its interest in transportation by a sending the PutOnTransport message via its required interface VehicleEvents. The message is handled by the car-individual service process of the implementation block that runs outside the car.

   ```
   VehicleEvents!PutOnTransport(carId, linkId)
   ```

2. The service process initiates the car to become inactive and disconnects its interfaces. This is automatically done when the car component should be migrated to another component. The car is designed in such a way that it can stop its intrinsic process when it should be transported.
3. The car component is moved by the outer service process into the transportation vehicle. More specifically, it is transmitted as part of the Enter message to the offered CarTransport interface of the transportation vehicle. As discussed in Section 3.5, components are removed from the sender and plugged in the receiver if they are sent as a parameter of a message.

---

48 The links with a car transport service have to be denoted with the attribute transport="true" in the XML road network file.

```
IMPLEMENTATION VehicleEvents;
VARIABLE carId, linkId: INTEGER;
BEGIN {EXCLUSIVE}
 IF ?PutOnTransport THEN
  ?PutOnTransport(carId, linkId);
  CarTransport(transport[linkId])!Enter(carId, car[carId])
  (* car[carId] is inactivated, disconn. and transmitted *)
 END
 (* ... *)
END VehicleEvents;
```

4. The car is placed inside the train or ferry, by the service process which runs inside the TransportationVehicle and receives the message containing the car component.

```
COMPONENT TransportationVehicle
 OFFERS CarTransport (* ... *);

 VARIABLE car[carId: INTEGER]: Car;

 IMPLEMENTATION CarTransport;
 VARIABLE carId: INTEGER; x: Car;
 BEGIN {EXCLUSIVE}
  IF ?Enter THEN
   ?Enter(carId, x); (* ... *) MOVE(x, car[carId])
  END
 END CarTransport;
 (* ... *)
END TransportationVehicle;
```

5. The car can now be transported as part of the transportation vehicle.
6. When the transportation vehicle arrives, it initiates the unloading of the cars via its required VehicleEvents interface.

```
VehicleEvents!RemoveFromTransport(linkid)
```

7. This is handled by the service process outside the TransportationVehicle component and moves the cars out of the transportation vehicle.

```
IMPLEMENTATION VehicleEvents;
VARIABLE carId, linkId: INTEGER; x: Car;
BEGIN {EXCLUSIVE}
 (* .. *)
 IF ?RemoveFromTransport THEN
  ?RemoveFromTransport(linkId);
  CarTransport(transport[linkId])!Exit;
  WHILE CarTransport(transport[linkId])?ExitingCar DO
   CarTransport(transport[linkId])?ExitingCar(carId, x);
   MOVE(x, car[carId]);
   (* connect and reactivate cars *)
  END
  CarTransport(transport[linkId])?NoMoreCars
 ELSE (* ... *)
 END
END VehicleEvents;
```

8. The interfaces of the car are reconnected and the Car components are reactivated for autonomous driving on the road.

```
CONNECT(Road(car[carId)), road);
CONNECT(RoadMap(car[carId]), road);
CONNECT(VehicleEvents(car[carId)), VehicleEvents);
Vehicle(car[carId])!Depart;
```

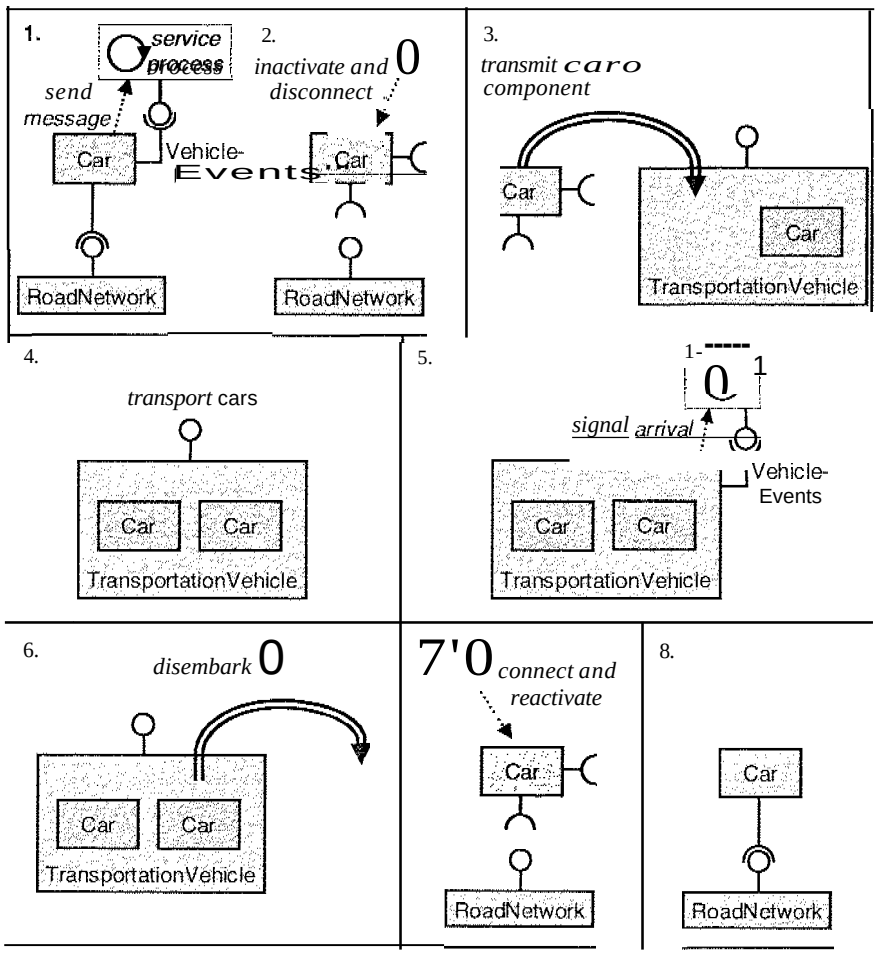


Figure 7-3. Car transportation process

The implementation of the transport.at.ion system is straightforward. Cars can be loadcd up to a defined capacity in the transportation vehicle. Subsequently, the t.ransport.at.ion system depart.s and is under way for a defined virtual time. At. the arrival, cars have t.o be unloaded from the transport.at.ion vehicle.

```
INTERFACE CarTransport;
   IN Enter(carId: INTEGER; car: Car)
|
   IN Exit
   {OUT ExitingCar(carId: INTEGER; car: Car) }
   OUT NoMoreCars
END CarTransport;
```

```
COMPONENT CarTransportSystem
 OFFERS CarTransport, TransportConfig
 REQUIRES VehicleEvents;

 CONSTANT Departing =1; UnderWay =2; Arrived =3;
 VARIABLE
   linkId, capacity, travelTime, state, nofCars: INTEGER;
   car[carId: INTEGER]: Car;

 IMPLEMENTATION CarTransport;
 VARIABLE carId: INTEGER; x: Car;
 BEGIN {EXCLUSIVE}
  IF ?Enter THEN
    ?Enter(carId, x);
    AWAIT((state = Departing) AND (nofCars < capacity));
    INC(nofCars); MOVE(x, car[carId])
  ELSE
    ?Exit;
    AWAIT(state = Arrived);
    FOREACH carId OF car DO
      !ExitingCar(carId, car[carId]); DEC(nofCars)
    END;
    !NoMoreCars
  END
 END CarTransport;

BEGIN state := 0
ACTIVITY {EXCLUSIVE}
 AWAIT(state =Departing);
 REPEAT
  AWAIT((nofCars> 0) OR TERMINATEDO);
  IF nofCars > 0 THEN
    state := UnderWay;
    PASSIVATE(travelTime); state := Arrived;
    VehicleEvents!RemoveFromTransport(linkId);
    AWAIT(nofCars = 0); state := Departing
  END
 UNTIL TERMINATEDO
END CarTransportSystem;
```

7.2.10 Entire Simulation

The entire traffic simulation package forms one component, which contains one RoadNetwork component, as well as a dynamic number of Car components, as specified by the configuration files. The cars all have to drive time-synchronously on the road network. The TrafficSimulation

component also requires the Filesystem and SystemTime interface for reading and writing files, and for measuring the runtime performance of the simulation.

```
COMPONENT TrafficSimulation
 REQUIRES Filesystem, SystemTime;

 VARIABLE
  car[id: INTEGER]: Car {SYNCHRONOUS};
  road: RoadNetwork {SYNCHRONOUS};
  networkReader: RoadNetworkReader;
 BEGIN
  NEW(networkReader);
  CONNECT(FileSystem(networkReader), FileSystem);
  NEW(road); CONNECT(RoadData(road), networkReader);
  BuildTransports; (* create transportation systems *)
  CreateCars; (* read configuration file to create the cars *)
  FOR iteration := 1 TO NofIterations DO
   WRITE("Start planning "); WRITELINE;
   FOREACH carId OF car DO
    RoadVehicle(car[carId])!Start
   END;
   FOREACH carId OF car DO
    RoadVehicle(car[carId])?Ready
   END;
   WRITE("Planning end"); WRITELINE;
   WRITE("Start simulation"); WRITELINE;
   FOREACH carId OF car DO
    RoadVehicle(car[carId])!Depart
   END;
   FOREACH carId OF car DO
    AWAIT(INPUT(RoadVehicle(car[carId]), Arrived);
    RoadVehicle(car[carId])?Arrived
   END;
   WRITE("Simulation end"); WRITELINE
 END
END TrafficSimulation;
```

7.3 Classical Traffic **Simulation**

To have a reference for comparison, we also developed a classical traffic simulation package. For this purpose, we followed the instructions of [Nagel05, Part II] and used C++ as the programming language. The simulation is

generally structured as follows: the road network consists of a set of *links* and *nodes,* where links represent unidirectional roadways and nodes the junctions between a set of links. Again, it can be chosen whether links are implemented as cellular automata or as queues. In both cases, the road links store direct pointers to the cars that drive on them. A node does not have any specific logic but just maintains pointers to links. The simulation model is sequential, Le. a global program loop iterates over the virtual time scale and computes the road state from one time step to the next one. Each road link maintains a separate event list for the cars that are scheduled to depart at a defined time from the corresponding link as well as for the cars which wait for free entry space on the link. According to the model of [NageI05, Part II], the movement of all cars is directed by a global program loop that traverses the links at each time step and performs the necessary transitions over the road junctions.

Apart from the core simulation, the package also consists of other units: a *route planner,* which computes the shortest paths by road for the defined trips of all vehicles; as well as a *trip planner,* which determines the adequate trips and departure times for all vehicles, given their desired day activities. All three units run separately from each other and are iteratively executed.

In this process, the result of the traffic simulation serves as feedback for the two planner units, exchanging data via files: (1) the core simulation reads pre-computed road network configuration and route plans from files and in turn, writes the departure and arrival times of each vehicle to an events file; (2) the trip planner then reads the desired day-activities and the events from a file and generates a trip file; and finally, (3) the route planner internalises both the trips and the events, in order to calculate the route plans, which are again imported by the core simulation unit.

7.4 Evaluation

7.4.1 Modelling

First of all, we have demonstrated that the language offers a sufficient expressiveness for solving a non-trivial problem in a smooth and elegant way. **In** particular, the pointers can be seamlessly replaced by the relations of the components, without loss of any useful flexibility. In particular, the composition scenario of car transports can be directly represented by the hierarchical compositions of the components, whereas a classical programming language does not allow such hierarchical structures with pointers.

We found that the component language also enables a more natural description of the car behaviour and traffic effects. The driving scenario of each car is individually described with regard to its own virtual time, reflecting the time of the simulated reality. In contrast to a classical traffic simulation, all cars drive concurrently without a global program loop that explicitly directs the movements of all cars. As the virtual times of the car activities are automatically kept synchronous by the system, we can abandon the intertwined description of all effects for the same virtual time, as it is required in sequential simulations.

Compared to the classical simulation, different technical artefacts have been eliminated in our model: no explicit park and wait queues are needed and no complicated global update-orders of the road network have to be considered (such as whether one has to move cars first on the links and then over the junctions or vice versa, selection policies on links to guarantee fairness in the sequential update logic). In addition, our implementation performs individual trip planning for each car with only local knowledge. Travel times are individually remembered by the affected car and do not need to be collected in a global events list (generally stored to a file). This probably conforms better to the real

world and also avoids expensive global collection and analysis of all car data in each simulation step.

The general component abstractions also add more flexibility in the deployment of the simulation on multi-processor and possibly also on distributed systems. Though the latter is beyond the scope of this work, the concepts of communication and the absence of pointers and garbage collection would be suited for an automatic program distribution on different machines. The communication-based interactions inherently fit with the network model and complicated distributed garbage collection is superfluous. Instead, each component has a clearly defined scope within the language, such that without change of the program, the components and their inner processes could be automatically distributed and migrated by the runtime systems between different processors and machines for optimisations (with regard to scalability or performance) behind the scenes.

7.4.2 Performance

We measured the speed of our new simulation and the classical C++ solution. Though the designs and the paradigms of both simulations are fundamentally different, we tried to evaluate the performance in a way that is as objective as possible. For this reason, we measured the core simulation phase, without planning and without file input and output. The queue-based implementation of the road links was used in the simulations. In a first step, the performance of the simulations was measured on a computer with a single Intel Xeon 700MHz processor and 4GB main memory. Then, the measurements were performed with six processors of this type. The C++ simulation was run under Windows Server 2003 R2 Enterprise Edition and was compiled with highest speed optimisation. As input data, we simulated the traffic flow of Greater Zurich, with a road

network of more than 28,000 links[49] and 260,000 cars with detailed routes[50] that were collected by the Swiss authorities [UVEK]. The input data was available in XML files with a size of more than 230MB. To evaluate the performance, we also measured the simulation with different numbers of the cars, taking a corresponding subset of the traffic data.

Table 7-1 summarise the average execution times in minutes of the core simulation phase using a single processor. All the times are in minutes and are rounded to two significant figures. As can be seen, our version offers a substantially higher performance than the classical implementation in C++. For example, our simulation is faster by a factor of 3.1 for 260,000 cars. In the case of fewer cars, the performance difference is significantly larger. The measurement series indicate that the component-based simulation scales linearly with the number of cars. In fact, the runtime also depends on the congestion of the road. Conversely, the runtime of the classical simulation depends to a large extent on the length of the road and on the difference between the maximum arrival time and the minimum departure time. This is because of the explicit scheduler loop in the classical simulation program, which iterates over the virtual time and all road links.

In addition, we also compared our simulation to an analogous concurrent implementation, written in C#. We deliberately made an effort on designing the simulation similar to our version, such that cars for example also feature their own threads for driving logic. However to retain the spirit of classical object-orientation, the C#

---

49 Data source: Digitales Strassennetz mit samtlichen Streckenangeboten und Widerstandsfunktionen, Bundesamt flir Raumentwicklung, Bern, July, 2007.

50 Data source: Verkehrsangebot, Bundesamt fUr Raumentwicklung, Bern, July, 2007.

version engages method calls instead of communication and also uses no separate service processes like in the component-oriented program. The virtual time has to be modelled explicitly in the C# program, synchronised by a global virtual timer object. Clearly, the measurements show that the threads of the C# language are not designed to support fine-granular concurrency in an efficient way. The C# simulation with 1,000 cars is already at the limit of the system, requiring a runtime that is by two to three orders of magnitudes longer than our component-based simulation.

Zurich traffic simulation (in minutes)	Component system	C++ (sequential)	C# (concurrent)
1 000 cars	0.03	140	19
10,000 cars	0.5	140	*out of memory*
100,000 cars	12	190	*out of memory*
260,000 cars	67	210	*out of memory*

Table 7-1. Zurich simulation, runtimes in minutes, one processor

In a further step, the simulation of the Zurich network was executed with activation of six Intel 700MHz Xeon processors on the same machine. According to the results given in Table 7-2, this unfortunately does not result in faster runtimes. The reason for this is that the simulation mostly consists of synchronisations with regard to the virtual time and the driving behaviour, while cars only perform very little independent logic (less than 1%). As the synchronisation costs become much more expensive as more processors are used (see Section 5.2.9), this kind of simulation with very simple car logic can not gain from multi-processor support. However, this does not mean that our traffic simulation can not gain from parallelism in general. On the contrary, we will show in the following paragraph how we can achieve a substantial speedup with multiple processors, if we equip the cars with more complex logic.

Zurich traffic simulation (in minutes)	Component system	C++ (sequential)	C# (concurrent)
1,000 cars	0.04	140	33
10,000 cars	0.6	140	*out of memory*
100,000 cars	13	190	*out of memory*
260,000 cars	76	210	*out of memory*

Table 7-2. Zurich simulation, runtimes in minutes, six processors

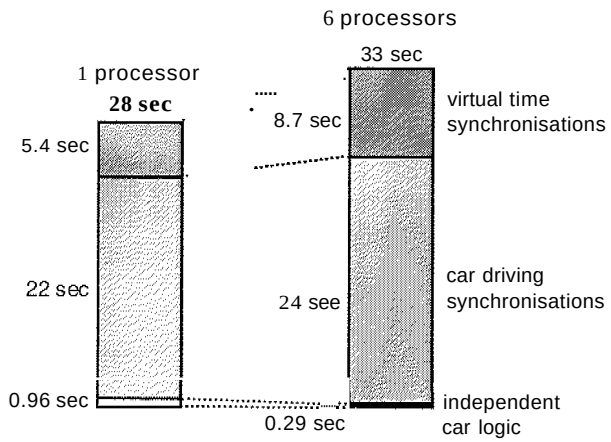


Figure 7-4. Cost factors of the Zurich simuJation, 10,000 cars

In order to demonstrate how our component-oriented simulation can benefit from multi-processors, we use more intelligent cars with inbuilt logic for trip and route planning (Section 7.2.8). Table 7-3 summarizes the results of measurements for the exemplary corridor network[51] and 10,000 cars[52] on the six-processor machine. According to this, the planning logic runs faster by a factor of 2.6 compared to the execution on a single processor. In total, the simulation achieves a higher performance by a factor of 2.] if we use six processors instead of one.

[51] Corridor network, http://www.matsim.org/files/test-net/nctwork

[52] Cars specified by two day activities each, activities]argc.txt, part of the digital material (Appendix D).

corridor traffic simulation (in seconds)	1 processor	6 processors	speedup
planning logic	66	25	2.6
driving logic	12	13	0.92
total time	78	38	2.1

Table 7-3. Multi-processor speedup with self-planning cars

7.5 Related Work

Traffic Simulation. Fully-fledged traffic simulation packages [Matsim] exist that are used in practice to compute traffic scenarios. Of course, it was not the objective of this case study to develop a simulation system that provides the same amount of functionality and details like these professional systems. We rather limited ourselves to the fundamental features of a traffic simulation (as described in [Nagel05, Part 11]), in order to evaluate our programming language. Nevertheless, our simulation offers sufficient functionality and has been used to simulate a realistic traffic scenario. To improve the performance on multiple processors, classical simulations usually employ parallelism in a technical way: the road network is explicitly partitioned into sub-parts, which are processed in parallel in each time step [Nage105, Chapter 25]. With this approach, car movements still remain controlled by a central program instance that iterates over the time and explicitly schedules the parallel updates of the road network parts. For high-performance parallelism, the partitioning and the update logic for parallel road parts need to be defined explicitly. In our simulation, the driving logic is however contained within the concurrent car activities and does not involve traversal of road cells, such that road partitioning does not necessarily lead to a better performance.

Virtual time. The concept of a virtual time has already been extensively discussed for distributed simulation or concurrency control. The most notable contributions came from Lamport [Lam78] and Chandy and Misra [CM81]. In these models, the virtual time is controlled by message

passing where the virtual time of the sender determines the virtual reception time of message in the receiver process (time-stamped messages). Though each process has its own virtual time, the virtual clocks of interacting processes can not run with different speeds, since a message can only be received by a process when it has reached (at least) the virtual time that the other process had at the time of sending. The notion of virtual time that we introduced in our language (see Section 7.2.1) is much more flexible, because components do not necessarily need to be time-synchronised if they exchange messages. Instead, our components can also have independent times and can perform arbitrary message communication within zero virtual time.

We can also demonstrate that our notion of virtual time is more powerful than the other models with time-stamped messages. If we do not allow the possibility of rollbacks in the other models, the virtual clock of a process can only proceed, if the process has received the next event message from each potential sender. Otherwise, a process with a lower virtual time could send a message to a process with a higher virtual time, such that the message would arrive in the virtual past of the receiver. With this conservative mechanism of time progression, we can not implement the traffic simulation (without awkward tricks): the virtual time in a RoadLink component can only advance if all cars signal their next event to the component (i.e. entering or exiting the roadway). However, a car can only send such an event if it has exited from a previous road link. This however leads to cyclic wait dependencies, as illustrated in Figure 7-5. An impractical fix for this dilemma would be to continuously send "no event" messages between components. In each virtual step, cars would have to notify the road link when they do nothing and road links must also inform the cars when nothing happens. In our program, the RoadLink components constitute time-synchronous subcomponents of the surrounding RoadNetwork components, wherein the driving activities of the cars run as service

processes. **The** system **automatically maintains the virtual synchronism between** these **components.**

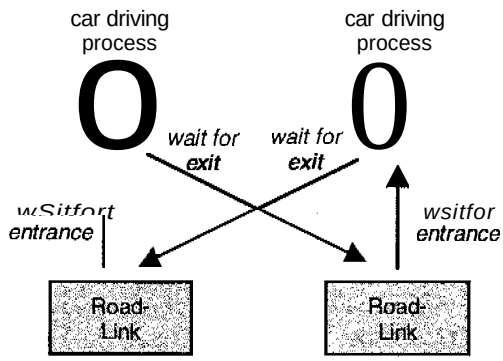


Figure 7-5. Cyclic event dependencies

Chapter 8

Conclusions

This dissertation has presented a new programming model that goes beyond existing paradigms with regard to the offered level of abstraction. With the use of a substantially new component notion, we have developed a programming language, which endorses accurate program structuring and promotes the use of natural concurrency, while retaining high dynamicity and safety. By means of an innovative runtime system, we have also demonstrated that this programming language can be efficiently implemented on current computer machines.

8.1 Conceptual Results

We can conclude that the following major conceptual advances have been made:

- *Hierarchically controlled structures*
  For the first time, a programming language has been invented which permits flexible structuring without explicit use of pointers (or references). In this language, both static and dynamic networks of components can be constructed with arbitrary shape in a clearly hierarchically controlled way. As our case study and various smaller examples show, the new relational concepts are capable of replacing pointers in terms of expressiveness and flexibility, by eliminating at the same time the

structural weaknesses of pointers (cf. Section 2.1.1).

- *Hierarchical encapsulation*
  As another innovation, general hierarchical encapsulation is supported and guaranteed in our programming language. Components are able to contain arbitrary static and dynamic structures of subcomponents. The ordinary object notion is thereby upgraded to a general component, which can indeed encapsulate an implementation of arbitrary complexity. As a result, a sustainable solution to the well-known and often encountered object-oriented encapsulation problems (cf. Section 4.1) has been found.

- *Safe concurrency*
  As our case study shows, the notion of autonomous components, which only interact by message communication and not by method calls, also conveys more natural modelling than in mainstream object-oriented languages. Components always have their own fully encapsulated processes. In contrast to languages that support method calls, a component process in our language can not synchronously execute the code of another component. Therefore, the executions of the components are fully disentangled and race-free concurrency can be guaranteed (see Section 4.6).

- *Symmetric polymorphism*
  We have been able to demonstrate that no artificial complicated concepts are needed to build extendible and reusable software systems. The clear separation between interfaces and implementation, combined with symmetric interface polymorphism (no preferred interfaces), allows us to achieve this objective in a very simple but powerful way. The fundamental relation of hierarchical composition already provides sufficient means for flexible and safe reuse of component functionality, avoiding

the technical problems and accidental complexity of inheritance or other code reuse mechanisms.

- *Interoperability*
  Our component language provides a general programming model, which is also capable of interoperating with other programming languages together in a well-organised and safe way. By design, we permit that terminal components can be implemented in any conceivable language. While the component language is designed for general programming at a high level of abstraction, low-level or other particular functionality (i.e. for device drivers or machine code) can be added in terminal components which are implemented in other languages. This allows us to keep our language simple and retain a high level of generality and conceptual consistency without the need of incorporating any unsuited or special-purpose constructs. We ourselves were able to successfully employ this flexibility, by implementing the kernel component of the system in a machine-close language.

8.2 Technical Results

With regard to the language implementation, we achieved the following major technical innovations:

- *High scalability*
  With an innovative design, our runtime system offers a particularly high scalability in the number of supported processes. **In** particular, programs with millions of parallel processes can be executed in an effective way. This has been realised by the integration of micro stacks and software-based preemption. The stacks are no longer based on the page granularity, while the preemption mechanism eliminates conservative register backups. In con-

trast, conventional systems only allow a very small constant amount of processes.

- *Efficient concurrency*
  With our runtime system, we have demonstrated that it is possible to also outperform conventional systems in the execution speed of concurrent programs. This performance improvement also results from the introduction of new concepts such as the fast context switches enabled by software-based preemption, as well as the omission of system artefacts such as garbage collection or virtual memory management, that are no longer needed for our programming model. By way of the traffic simulation, we have demonstrated that the system even supports programs with highly synchronised processes so efficiently, that it outperforms analogous classical sequential programming models.
- *Abolition of Garbage Collection*
  With our programming model, we have demonstrated that automatic garbage collection is no longer needed for programming with flexible dynamic structures and with guaranteed memory safety. In the field of imperative and object-oriented programming, garbage collection is widely known to cause technical problems which have not yet been satisfactorily solved for decades (see Section 2.1.5.2). As garbage collection has now become superfluous for our programming language, we have built a safe and reliable system that is free of any unexpected disruptions of the program execution.

8.3 Open Problems

During this dissertation, we also encountered problems that we had to leave open:

- *Deadlocks*
  Concurrency obviously involves a higher intrinsic (natural) complexity than a merely sequential programming model. Though artificial problems like data races and primitive interaction deadlocks have been eliminated in our language (see Section 4.6), more complex deadlocks resulting from cyclic component interactions remain possible (e.g. blocking on road junctions, Section 7.2.3). We did not want to enforce too restrictive technical rules, to alleviate such deadlocks (cf. Appendix C).
- *Insufficient hardware support*
  Current computer machines considerably lack in the adequate infrastructural support of efficient fine-granular concurrency. Synchronisation of threads is extremely slow on current machines due to the required maintenance of cache coherency. Therefore, many concurrent programs can not gain from parallelism with multiple processors, as they involve frequent synchronisations due to message communication and monitor locks. We nevertheless managed to design an efficient runtime system which offers high speedups with multiple processors for largely independent processes, without however penalizing frequently synchronised processes. This eventually allows us to use concurrency, whenever it improves the program model, and not only, when it improves the performance.

8.4 Further Directions

Below, we mention a few ideas that could be pursued in future projects but however went beyond the scope of this dissertation.

- *Other applications*
  As the object-oriented paradigm was initially invented for simulation programming and eventually turned out to also be suited for many other purposes, this may just as well be the same for our programming model. Therefore, it remains to be determined for which other application areas the language is also particularly suited. Possible areas are graphical user interfaces (due to the compositional relations), internet service applications (due to the client-individual service communications) or perhaps time-critical concurrent programming (due to the absence of garbage collection). The latter however requires a machine that offers real-time capabilities for concurrency, such as processor synchronisation within guaranteed time boundaries.
- *Other runtime platforms*
  Due to the high-level abstractions, our component language is also predestined for implementation on other runtime platforms. Without changes to the programming model, the language can not only run on a machine consisting of a single- or a multi-processor but could just as well be executed on a network of computers. The latter is possible, as the concepts of message communications and component networks should directly agree with the distributed computing model. In contrast to other languages, complicated distributed garbage collection is no longer necessary due to the hierarchical component structures. Of course, one could also investigate a runtime system that provides support of persistence or real-time execution for the components. In particular, built-in persistence could be considerably simplified as no garbage collector is required.

- *Influence on existing fields*
  This work consists of various conceptual and technical innovations, where some of them could be also integrated in other languages or systems, to improve their structure and safety. The concepts of pointer-free structuring with hierarchical composition and network construction (by interface connections) would for instance significantly enhance a classical object-oriented programming language. Though, this would not attain the same level of flexibly as in our language, which supports arbitrarily complex statefull interactions. Using ordinary method calls, the context of such long-lasting interactions usually needs to be stored at the client side by means of explicit pointers or other pointer-like constructs (see Section 4.1). Moreover, the concepts of autonomous components and message communication seem to be indispensable for the design of a natural object-oriented language. However, without hierarchical encapsulation and clearly specified structures, safe concurrency can not be achieved in a simple and comprehensible way. As for system construction, the kernel may serve as a generic basis for a well-structured compact operating system, where specific memory management or preemption mechanisms can be added on top of the kernel, acting only as clients of the heap or the concurrency module.

In this dissertation, we aimed to design a new programming language that noticeably improves the conceptual basis for more natural and more structured programming. We believe that we have found such a language and have demonstrated its suitability by means of the simulation case study and other concurrent programs. Therefore, we remain curious as to whether a programming language like ours could prevail within industrial or academic practice.

Appendix A

Language Report

A.I Notation

A.I.t Syntax

The syntax of the programming language is formally defined in the *Extended Backus Naur Formalism (EBNF)* [Wirth77b]. In Section A.13, the complete syntax specification of the language can be found.

A.1.2 Semantics

For each programming concept, the semantics is informally defined in prose text. Issues that are not specified have been intentionally left so, either because a corresponding rule can be inferred from the other specifications or because the issue should not matter for the understanding and use of a concept.

A.2 Program

A *program* has a dual nature with a *static* and *runtime* character. Statically, it is a collection of *templates*, which describe components with their interfaces in a textual format. At runtime, a program consists of components, which have been created as instances from these templates. The static description of a program has the following syntax:

```
Program = { Component | Interface }.
```

A.3 Components

A *component* constitutes a closed program unit at runtime, which encapsulates an inner state and behaviour. Components are only allowed to have external dependencies over explicitly defined interfaces. Each component can *offer* an arbitrary number of own interfaces and can *require* an arbitrary number of foreign interfaces that belong to other external components.

Three elementary relations govern structures among components:

- *Hierarchical composition*
  Each component can be hierarchically composed, by containing an arbitrary number of inner components *(sub-components).*
- *Interface connections*
  An arbitrary network of components can be built by connecting the required interfaces of a component to offered interfaces of other components.
- *Communication-based interactions*
  Components interact via interfaces by communications.

A component is created at runtime from a statically defined *component template,* which specifies the offered and required interfaces, as well as the concrete implementation. The same template can be used to create multiple components of the same specification.

```
Component          = COMPONENT Identifier
                       [ OFFERS InterfaceDeelList ]
                         [ REQUIRES InterfaceDeclList] ";"
                       ComponentBody
                     END Identifier ";".
InterfaceDeclList  = InterfaceDecl { "," InterfaceDecl }.
InterfaceDecl      = Identifier [ "[" NofInterfaces "]" ].
```

Each component template has an identifier that is specified both at the beginning and at the end of the declaration. The body of the declaration represents a new program scope, which describes the internal implementation of the creatable components. Offered or required interfaces are declared by the identifiers of their corresponding interface specifications in the OFFERS- and REQUIRES-list respectively. The declaration order within the list is irrelevant, Le. all interfaces of the same list have equal importance. The identifier of an interface specification can occur no more than once in the same list.

Examples:

```
COMPONENT StandardHouse
  OFFERS Residence, ParkingSpace
  REQUIRES Electricity, Water  (* ... *)
END StandardHouse;

COMPONENT River
  OFFERS Water;
END River;

COMPONENT HydroelectricPowerPlant
  OFFERS Electricity
  REQUIRES Water;  (* ... *)
END HydroelectricPowerPlant;
```

By default, a declared identifier of the REQUIRES-list means, that exactly one interface with this specification is required from an exterior component. A component is also able to require multiple interfaces with the same specification from outside. In this case, the number of required instances has to be specified after the corresponding declaration by using the following syntax:

```
NofInterfaces = Integer [ ".." ( Integer 1"*" ) ].
```

The number of interfaces of the same specification may be either static or defined in a dynamic range between a minimum and maximum value. The maximum number of interfaces may be unbound using the star symbol.

Examples:

```
COMPONENT AcademicAssistant
  OFFERS Student, Employee
  REQUIRES University, Supervisor [2..*];
     (* at least two Supervisors required *)
END AcademicAssistant;

COMPONENT CompanyBuilding
  OFFERS OfficeSpace, ParkingSpace
  REQUIRES Electricity [1 .. 3], Water [2];
    (* one to three Electricity and two Water lines required *)
END CompanyBuilding;

COMPONENT Programmer
  REQUIRES Computer, Network [0.. 1];
    (* Network optionally required *)
END Programmer;
```

A.4 Interfaces

An *inteiface* represents an external facet of a component and establishes a possible interaction point between the component and its exterior environment. As possible interactions, an interface enables *communications* between the *server* component, which offers the interface, and the *client* components, which have a required interface connected to the corresponding interface. The processes inside the surrounding component of the server may act as further clients of the offered interface.

The server component always maintains a *separate* communication with each client component individually. Thereby, a communication generally involves bidirectional message transmissions, specified by a formal *protocol.* The transmission of a message implies that one component sends the message and the other side explicitly receives it.

Each interface requires an *inteiface specification,* stating the communication protocol for that interface. The same interface specification can be used to declare offered or required interfaces for multiple components.

```
Interface = INTERFACE Identifier ";"
              Protocol
            END Identifier ";".
```

Each interface specification has an identifier that is denoted both at the beginning and at the end of the specification. A new program scope is induced by the specification, containing the protocol description.

Examples:

```
INTERFACE Student;
  (* ... *)
END Student;

INTERFACE Electricity;
  (* ... *)
END Electricity;
```

A.4.1 Protocol

The *protocol* of an interface specification defines all feasible sequences of message transmissions, which are allowed during the communication between a server and a cHent. The communication protocol is formalised in a regular EBNF-expression [Wirth77b], using message declarations instead of terminal symbols. A message declaration consists of a transmission direction, a message identifier and an optional parameter list. Messages with direction IN go from the client to the server, whereas messages with direction OUT are sent from the server to the client. Before a client finishes a communication, the predefmed message IN FINISH without parameters is automatically sent to the server and optionally received by the server.

```
Protocol        = [ ProtocolExpression ].
ProtocolExpr    = ProtocolTerm { "|" ProtocolTerm}.
ProtocolTerm    = ProtocolFactor { ProtocolFactor }.
ProtocolFactor  = MessageDecl
                | "[" ProtocolExpr "]"
                | "{" ProtocolExpr "}"
                | "(" ProtocolExpr ")".
MessageDecl     = ( IN IOUT) Identifier [ "(" ParameterList ")" ].
```

Examples:

```
INTERFACE SignalStream;
 { OUT Signal} OUT Finished
END SignalStream;

INTERFACE HotelService;
 {
   IN CheckIn
    (
      OUT AssignedRoom
      { IN EnterRoom IN ExitRoom}
      IN CheckOut
      OUT Bill
      [ IN DirectPayment]

      OUT FullyBooked

  }
END HotelService;
```

A.4.1.2 Message Parameters

A *parameter* in a message declaration specifies a component or data value which is transmitted at runtime within a corresponding message. Each parameter has an identifier that must be unique among all parameters in the same parameter list.

```
ParameterList     = ParamSection { ";" ParamSection }.
ParamSection      = IdentifierList ":" Type.
IdentifierList    = Identifier { "," Identifier }.
```

Example:

```
INTERFACE Library;
 {
   IN RequestBook(isbn: ISBN)
   ( OUT BorrowedBook(b: Book) |OUT Unavailable)

   IN ReturnBook(b: Book)
 }
END Library;
```

A.5 **Component Implementations**

The body of a component template describes the implementation of the components, which are created from the template. All constructs in the body are fully encapsulated and are neither visible nor accessible from outside.

```
ComponentBody  = { Declaration}
                 { Implementation}
                 [ BEGIN StatementSeq ]      // initialisation
                 [ACTIVITY StatementSeq ]    // main activity
                 [ FINALLY StatementSeq].    // finalisation
Declaration    = Component | Interface | ConstantList |
                 VariableList | Procedure.
```

A.5.1 Lifecycle

The component's lifetime is exclusively controlled by the surrounding component and consists of three phases:

1. *Initialisation.* The initialisation phase starts immediately with the creation of the component. The initialisation process defined in the BEGIN-block runs independently of the creator.

2. *Main activity.* After initialisation and before finalisation, the component accepts communications via its offered interfaces, by automatically starting an individual service process for each individual client communication. During this phase, the ACTIVITY-block also runs as an intrinsic process of the component.

3. *Finalisation.* Before the deletion of a component, the finalisation phase of the component has to be passed through. A customised finalisation process can be specified in the FINALLY-block of the component template.

A component is only deleted by its surrounding component, either explicitly (cf. Section A.7.7) or implicitly, when the outer component is also deleted. All communications of a deleted component are automatically closed.

A.5.2 Declarations[53]

Every identifier occurring in a program must be introduced by a declaration, unless it is a reserved keyword.

A declaration always belongs to a *scope,* which is either delimited by the directly surrounding program block (component template, interface specification, interface implementation, or procedure declaration) or the global scope, if not contained in a program block. Within the same scope, all declarations must have different identifiers.

An identifier can be used to refer to the program element of the corresponding declaration. This is however only possible in those program parts, which have visibility of the declaration. The following rules govern the visibility:

1. A declaration is always visible inside its own scope. Declarations in the global scope are visible to all programs in the system.
2. If a declaration is visible inside a scope, it is also visible within an inner (nested) scope, unless the identifier is reused for another declaration in the inner scope. The visibility of variables and procedures is restricted to those scopes, which are directly located in the same component.

A.5.3 Interface Implementations

An *interface implementation* specifies the service processes for an offered interface of the containing component. For each client of an offered interface, a separate service process is automatically incarnated, concurrently running inside the server component and communicating with the corresponding client side.

---

53 Parts of this section are adopted from the Oberon language report [Wirth90].

```
Implementation    = IMPLEMENTATION Identifier ";"
                      { Declaration}
                      [ BEGIN StatementSeq ]
                    END Identifier ";".
```

An interface implementation identifies the offered interface at the beginning and end of the declaration block. Interface implementations may feature own local declarations together with a statement sequence, describing the service process.

Example:

```
COMPONENT Hotel OFFERS HotelService;
 VARIABLE room[number: INTEGER]: HotelRoom;

 IMPLEMENTATION HotelService;
  VARIABLE freeRoomNo: INTEGER;
 BEGIN (* ... *)
 END HotelService;
END Hotel;
```

A.S.4 Message **Communication**

Messages are exchanged between a process of a client component and the service process of a server component by means of send- and receive-statements (Sections A.7.5 and A.7.6). In addition, receive guards (Sections A.8.1 and A.8.2) can be used to decide at runtime, which message can be received.

Examples:

```
HotelService!CheckIn;
 (* send the CheckIn message to HotelService interface *)
IF HotelService?AssignedRoom THEN
 (* test whether the next arriving message is AssignedRoom *)
 HotelService?FreeRoom(number)
  (* receive the AssignedRoom message from HotelService *)
ELSE
 Hotel?FullyBooked (* receive the FullyBooked message *)
END
```

For all client-side communication statements and receive guards, the interface of communication has to be explicitly designated, whereas it is omitted for all server-side communication commands. Server-side communication com-

mands can be only used within an interface implementation block. The interface designator at the client side refers to either a required external interface of the local component or an offered interface of a sub-component.

Examples:

```
HotelService!CheckOut (* client-side communication *)

1MPLEMENTATION HotelService;
BEGIN
 ?CheckIn (* server-side communication *)
END HotelService;
```

A message declared with transmission direction IN has to be sent by the client side and received by the server side, whereas the opposite holds for messages with direction OUT. All messages have to be transmitted according to the protocol of the corresponding interface specification.

A communication automatically starts, when the client executes the first communication statement or guard with regard to the corresponding interface. Thereby, the client waits until the interface is connected to a server. It finishes either when the last message according to the protocol has been sent or, when the server component is moved or deleted and the predefmed FINISH message is automatically sent to the server side (see Section *AA.I).* A client is also allowed to restart a communication via the same interface after the previous communication has been finished.

Inside a component, each process maintains a separate communication with an offered interface of a sub-component. They however share the same communication with regard to a required interface of the local component.

A.5.5 Constants

The declaration and use of constants IS the same as m Oberon [Wirth90].

```
ConstantList     = CONSTANT Constant { Constant }.
Constant         = Identifier "=" ConstantExpr ";".
ConstantExpr     = Expression.
```

A.5.6 Variables

A *variable* defines a separate container, in which components or data values can be stored. Variables are declared with an identifier and a signature of the therein storable components. Variables enable the construction of *hierarchical compositions:* a component contains direct sub-components within its variables which are declared in the scope of its template. For this purpose, a component can be created at runtime within a variable (see Section A.7.2) and its interfaces can be connected to matching interfaces of other components which are defined inside the same scope (see Section A.7.3).

A variable may be also declared in an interface implementation block (see Section A.5.3), such that each service process created from the block owns an individual variable. The content of a variable only exists as long as the surrounding scope is active, i.e. a sub-component is automatically deleted when the surrounding component is disposed of or the owning service process ends. The following syntax is used for declaring variables:

```
VariableList = VARIABLE VariableSection { VariableSection }.
VariableSection = IndexedIdentList ":" Type [ AttributeList] ";".
IndexedIdentList = IndexedIdent { "," IndexedIdent}.
IndexedIdent = Identifier [ "[" ParameterList "]" ].
```

A variable is either a *normal variable* or a *collection variable.*

A.5.6.1 Normal Variables

A *normal variable* is declared without parameters and represents a *storage location* for a single component of a defined signature or a data value of a defined type. The variable name directly identifies the contained component or value.

Examples:

```
VARIABLE
  Paul, Fred: AcademicAssitant;
```

```
livingRoom, bedroom: ANY(Room | Electricity);
counter: INTEGER;
name: TEXT;
```

Initially, the storage location of a normal variable with a component signature is *empty* until a component has been assigned to it. Variables with a data type are initialised with the following default values: FALSE for BOOLEAN, 0 for INTEGER, 0.0 for REAL, 0X for CHARACTER and "" for TEXT.

A.5.6.2 Collection Variables

A *collection variable* is declared with a parameter list and stores a dynamic collection of components of a defined signature or data values of a defined type. The parameter list specifies a set of *indexes* for the identification of the elements in the collection. Each index refers to a separate *storage location* of a component in the collection. An index is represented as an expression list of data values which are compatible with the declared parameter list. Parameters must have different identifiers within the same parameter list.

Examples:

```
VARIABLE
  book[isbn: INTEGER]: Book;
  person[firstname, surname: TEXT]: ANY(Person);
```

Initially, a collection variable has no stored elements and all possible indexes identify *empty* storage locations.

A.5.7 Procedures

A *procedure* forms a sequential execution part of an internal process in a component. Procedures only serve as internal implementation pieces of a component and can not be directly invoked from outside. Apart from this, the declaration and use of procedures is equal to Oberon [Wirth90].

```
Procedure      = PROCEDURE Identifier [ "(" [ ProcParamList] ")"
                    [ ":" Type] ] ";"
                  { Declaration}
                  [ BEGIN StatementSeq ]
                  END Identifier ";",
ProcParamList = ProcParSection { ";" ProcParSection }.
ProcParSection= [VARIABLE] ParamSection.
```

A.6 Types

A type specifies static properties of data values or the components. There are two kinds of types, namely *data types* and *component signatures.*

```
Type     = Identifier | ANY [ "(" AnyInterfaceList ")" ].
```

A.6.t Data Types

A *data type* determines a set of data values and thereon applicable operators [Wirth90]. Alike Oberon, the *data types* INTEGER, REAL, BOOLEAN, CHARACTER are predefined. In addition, the data type TEXT is introduced, comprising all character sequences.

A.6.2 Component Signatures

A *component signature* specifies static properties about a component that is used in a program. It may be a *concrete component signature,* an *abstract component signature* or the *generic component signature.*

A.6.2.t Concrete Component Signatures

A *concrete component signature* is defined by the template of the component.

Examples:

```
AcademicAssistant     StandardHouse     GraphicalSquare
```

A.6.2.2 Abstract Component Signatures

An *abstract component signature* does not detennine a specific component template but only postulates a set of offered and required interfaces of the component. The component can be of *any* template that fulfils the following requirements:

1. The component template *offers at least* the interfaces which are postulated as offered by the signature. These interfaces are always guaranteed to be provided by the component.

2. The component template *requires at most* the interfaces which are postulated as required by the signature. The range specifying the number of interfaces with the same name has to match exactly. These interfaces have to be provided by the exterior runtime environment of the component, in order to be able to interact with the component's offered interfaces.

An abstract signature is described by the ANY-construct, which states a list of postulated offered interfaces followed by a list postulated required interfaces. The two lists are separated by a vertical bar.

```
AnyInterfaceList  = [ InterfaceDeclList ] [ "|" InterfaceDeclList].
```

Examples:

```
ANY(Student, Employee |University, Supervisor)
  (* offers at least Student and Employee,
     requires not more than University and Supervisor *)
ANY(Rectangle, Graphic)  (* no required interfaces *)
ANY(OfficeSpace |Electricity [1..3], Water [2])
ANY( |Computer, Network [0..1])  (* no offered interfaces *)
```

A.6.2.3 The Generic Component Signature

The *generic component signature* represents any component. It is denoted by the keyword ANY without following round brackets. As no offered interfaces are postulated, no required interfaces have to be guaranteed either.

A.6.3 Type Compatibility

The notion of type compatibility serves to statically ensure a consistent use of components and data values in a program.

An expression or parameter of type X is *type-compatible* with an expression or parameter of type Y, if and only if,

- X and Y are identical types or,
- Y is an abstract component signature and X is a component signature, such that all offered interfaces of Y are declared to be also offered by X and all required interfaces of X are declared to be also required by Y or,
- Y is the generic component signature and X is a component signature.

A list X of expressions or parameters is *type-compatible* with a list Y of parameters or expressions, if and only if,

- X and Y have the same number of elements and,
- for each valid position i in X, the type of the $_{jth}$ element of X is type-compatible with the type of the $_{jth}$ element in Y.

A.7 Statements[54]

A *statement sequence* describes the sequential execution of statements, separated by semicolons. Statement sequences may be attributed with a directive for concurrency synchronisation (see Section A.7.1).

```
StatementSeq      = [ AttributeList ] StatementList.
StatementList     = Statement { ";" Statement }.
```

A *statement* denotes an action of the component runtime behaviour. Statements are either elementary or composed of other statements. Elementary statements are assignments, communication statements (send and receive), the NEW-, CONNECT-, DISCONNECT-, DELETE-, MOVE-, AWAIT-statement, as well as procedure calls, the RETURN-statement and the *empty statement.* Composed statements, like the IF-, WHILE-, REPEAT-, FOR- and FOREACH-statements, as well as statement blocks, allow the structured description of sequential, alternative, and repetitive execution. The empty statement, which is not written, does not have any effect but only serves to relax the use of semicolon separators in a statement sequence.

```
Statement = [ Assignment | New | Connect | Disconnect
            | Send | Receive | Delete | Move | Await
            | ProcedureCali | Return | If | While
            | Repeat | For | Foreach | StatementBlock].
```

The definition of assignment statements, procedure calls, and of the RETURN-, IF-, WHILE-, REPEAT- and FOR-statements is the same as in Oberon [Wirth90]. Assignment statements can be only applied to variables with a data type.

---

54 This section is partially based on the language reports of Oberon [Wirth90] and Zonnon [GZ05].

A.7.1 Concurrency Synchronisation

The attribute EXCLUSIVE or SHARED can be annotated to a statement sequence, establishing a component lock during the execution of that sequence. An exclusive statement sequence is only executable if no other shared or exclusive block runs at the same time inside the same component. Conversely, shared statement regions can be executed in parallel and are only mutually barred against exclusive regions of the component.

Examples:

```
BEGIN {EXCLUSIVE}
  (* statement list *)
END

BEGIN {SHARED}
  (* statement list *)
END
```

All potential parallel accesses to shared variables, which are directly declared in the component scope, as well as to required interfaces of the local component have to be appropriately synchronised. More specifically, this requires an exclusive lock for all modifications of the content of a shared variable, as well as for communication with the local required interfaces. All other statements, which only involve reading of data values, interface tests, type guards and communications with regard to shared variables, need at least a shared lock.

The initialisation and finalisation process of a component always runs exclusively and hence does not need explicit synchronisation. Nested use of exclusive or shared statement locks within a statement sequence is not allowed. If a procedure contains a synchronisation attribute, it can not be (directly or indirectly) called by another exclusive or shared region. A procedure may modify and read a shared variable without lock attribute, if there is no other possibility than to (directly or indirectly) call the procedure from an exclusive or shared region, respectively.

A.7.2 The NEW-Statement

The NEW-*statement* creates a new component from a defined component template and assigns it to a storage location, specified by the designator of the first argument. The possible previous content of the storage location is thereby deleted. If the designated storage location is declared with a concrete component signature, the new component is implicitly created from the declared component template. Otherwise, the component template ought to be explicitly specified by the identifier in the second argument. The component template of the second argument must satisfy the signature of the storage location.

```
New = NEW "(" Designator [ "," Identifier] ")".
```

Examples:

```
VARIABLE house: Mansion;
NEW(house)
(* create a new component of the Mansion component template *)

VARIABLE student: ANY(Student | University, Supervisor);
NEW(student, AcademicAssistant)
(* create a new component of the AcademicAssistant template*)
```

The NEW-statement may be also applied to allocate a new TEXT value of a certain length. The first argument thereby designates the corresponding storage location, whose previous content is also disposed of. The further arguments specify the length for TEXT value.

A.7.3 The CONNECT-Statement

The *CONNECT-statement* connects a required interface of a component to an offered interface of another component. The required and the offered interface thereby have to be of the same interface specification. The first argument of the statement denotes the required interface, whereas the offered interface is defined by the second argument.

```
Connect = CONNECT "(" Designator "," Designator ")".
```

Multiple required interfaces can be connected to the same offered interface. The CONNECT-statement must not be applied to a required interface that is already connected.

A component can also connect its external offered or required interfaces. An offered external interface of a component is regarded as a required interface for connections in the internal component implementation. Similarly, a required external interface represents an offered interface for connections inside the component.

There are three kinds of interface connections:

A.7.3.1 Connect Sub-Components

A component can connect the required interface of a sub-component to an offered interface of another sub-component. The first argument designates the required interface, while the second argument denotes the offered interface or, as a short cut, the component which offers this interface. In the latter case, the offered interface can be automatically derived from the first argument.

Examples:

```
VARIABLE house: StandardHouse; river: River;
BEGIN
  NEW(house); NEW(river);
  CONNECT(Water(house), river);

VARIABLE company: CompanyBuilding; powerPlant: PowerPlant;
BEGIN
  NEW(house); NEW(powerPlant);
  CONNECT(Electricity[2](house), powerPlant);
  (* CompanyBuilding requires multiple Electricity interfaces *)

VARIABLE motor: ANY(Motor | Gear); gearbox: ANY(Gear);
BEGIN
  NEW(motor); NEW(gearbox);
  CONNECT(Gear(motor), gearbox)
```

The required interface of a component, which is stored in a component-local variable, can not be connected to an interface of a component, which is contained in the scope of an implementation block or procedure declaration. This is

because the lifetime of these scopes is generally shorter than that of the component.

A.7.3.2 Redirect Offered Interfaces

A component can also connect an offered external interface of its own to an offered interface of a sub-component. The first argument designates the offered external interface, while the second argument denotes the offered interface of the sub-component or, as a short cut, the sub-component directly.

Example:

```
COMPONENT House OFFERS ParkingPlace;
  VARIABLE garage: Garage;
BEGIN
  NEW(garage);
  CONNECT(ParkingPlace, garage);
END House;
```

A.7.3.3 Redirect Required Interfaces

In addition, a component can connect a required interface of a sub-component to either a required external interface or alternatively, an interface implementation block of its own. The first argument designates the required interface of the sub-component and the second argument the name of the external required interface or the implementation block.

Example:

```
COMPONENT House REQUIRES Electricity, Water;
  VARIABLE bathRoom: ANY(Room | Electricity, Water);
BEGIN
  NEW(bathRoom);
  CONNECT(Electricity(bathRoom), Electricity);
  CONNECT(Water(bathRoom), Water)
END House;
```

A.7.4 The DISCONNECT-Statement

The *DISCONNECT-statement* disconnects the required interface of a component from its connected interface. It may also disconnect the offered interface of the locally surrounding component that is redirected to another inter-

face. Before disconnecting, the statement waits until the interface has no open communications.

```
Disconnect = DISCONNECT "(" Designator ")".
```

Example:

```
VARIABLE house: StandardHouse; river: River;
BEGIN
  CONNECT(Water(house), river);
  DISCONNECT(Water(house))
```

A.7.S **The Send-Statement**

The *send-statement* sends a message via an interface to the other communication partner. This may (but does not need to) happen asynchronously, i.e. without awaiting the acceptance of the message by the other side. The arguments of the expression list specify components or data values that are transmitted as content of the sent message.

```
Send = [ Designator] "!" Identifier [ "(" ExpressionList ")" ].
ExpressionList = Expression { "," Expression }.
```

An argument is either the variable designator of a component, which is delivered by *moving*, or the expression of a data type, which is sent by *copying.* In the first case, the component is removed from the designated variable. A component can only be sent within a message, if it does not have any connected interfaces or non-terminated communications. The argument list of a send-statement has to be type-compatible with the parameter list of the message declaration.

Examples:

```
HotelService(hotel)!CheckIn
  (* client-side send of a CheckIn message *)
PhoneBook!RequestPhoneNumber("John", "Smith")
  (* client-side send with two parameters *)
!LentBook(book)
  (* server-side send, the book is moved to the other side *)
```

A.7.6 The Receive-Statement

The *receive-statement* awaits the anival of a specific message from the other communication side and accepts the message on arrival. The possible contained components of the received message are eventually assigned to the storage locations, which are specified as arguments. Thereby, the previous content of the storage locations is automatically deleted.

```
Receive = [ Designator] "?" Identifier [ "(" DesignatorList ")" ].
DesignatorList = Designator { "." Designator}.
```

The execution of the receive-statement blocks as long as the message is not received. The statement does not release a monitor lock, in order to also allow the implementation of non-interfered communication processes. Instead, the AWAIT-statement has to be used if a monitor lock should be released while waiting for a message. The reception of a wrong message with the receive-statement violates the protocol and is forbidden. The parameter list of the message declaration must be type-compatible with the list of storage location designators, which are denoted as the arguments of the receive-statement.

Examples:

```
?CheckIn (* server-side receive of a CheckIn message *)
?RequestPhoneNumber(firstname, surname)
  (* server-side receive with two parameters *)
Library?LentBook(b) (* client-side receive *)
```

A.7.7 The DELETE-Statement

The *DELETE-statement* disposes of the component in a designated storage location. It is used to discard a component before the automatic deletion at the termination of the surrounding component or service process. The storage location must not be empty and the component must not have any connections to its offered interfaces. The required interfaces of the deleted component are automatically disconnected.

```
Delete = DELETE "(" Designator ")".
```

Example:

```
VARIABLE book[isbn: INTEGER]: Book;
BEGIN
  NEW(book[12345]);
  DELETE(book[12345])
```

A.7.8 The MOVE-Statement

The *MOVE-statement* removes a component from a storage location and installs it in another. The source location is designated first and must be type-compatible with the designator of the target location that follows as second argument. The possible previous content of the target storage location is thereby ended.

```
Move = MOVE "(" Designator "," Designator ")".
```

Examples:

```
VARIABLE source: ScientificBook; target: ANY(book);
BEGIN
  NEW(source);
  MOVE(source, target)
```

A component which is moved must not have any connected interfaces. All client communications with the moved component are automatically finished.

A.7.9 The AWAIT-Statement

The *AWAIT-statement* blocks the execution as long as a defined expression of a Boolean type evaluates to FALSE. An AWAIT-statement can only be used inside a statement sequence with an EXCLUSIVE or SHARED lock. It temporarily releases the lock while waiting, in order to allow the fulfilment of the condition by another exclusive region in the component.

```
Await = AWAIT "(" Expression ")".
```

Example:

```
AWAIT(length < Maximum)
```

A.7.10 The FOREACH-Statement

The *FOREACH-statement* iterates over all elements of a collection variable. Each iteration step assigns the index of a present element to the specified designator list. The iteration is sequential but there is no specific order of traversal.

```
Foreach = FOREACH DesignatorList OF Designator DO
            StatementSeq
          END.
```

The collection variable is denoted by the designator following the OF symbol. The designator list determines a list of storage locations, to which the values of the index are assigned in each iteration step. The declared parameter list of the collection variable must be type-compatible with the list of storage location designators. The values of the index must not be modified inside the FOREACH-statement.

Example:

```
VARIABLE person[firstname, surname: TEXT]: Person; x, y:
TEXT;
BEGIN
  FOREACH x, y OF person DO
    person[x, y]!Question
  END
```

A.7.11 Statement Blocks

A *statement block* groups a logically coherent statement sequence that may feature its own synchronisation attribute.

```
StatementBlock     = BEGIN StatementSeq END.
```

A.8 Expressions

The defmition of expressions is the same as in Oberon [Wirth90], except for some minor amendments. The logical operator AND is used in symmetry to the operator OR. Moreover, either one of the attributes EXCLUSIVE or SHARED can be annotated to an expression, in order to establish an exclusive or shared component lock during the evaluation of the expression.

```
Expression = [ AttributeList ]
               ( SimpleExpr
                  [ ("=" 1"#" 1"<" 1"<=" 1">" 1">=") ] SimpleExpr
             | Designator
                    ( OFFERS I REQUIRES) InterfaceDeclList
             | Designator IS Type ).
SimpleExpr =["+" | "-" ] Term { ( "+" |"-" IOR) Term }.
Term       =Factor { ("*" I"/" I DIV I MOD I AND) Factor }.
Factor     =Operand | "~" Factor | "(" Expression ")".
```

The OFFERS relation tests whether a designated component offers a set of interfaces, denoted by the identifier list of the corresponding interface specification. Analogously, the REQUIRES relation results TRUE if the designated component requires a specified set of interfaces. If multiple interfaces of the same specification are required, the result is only TRUE if the range for the number of interfaces is equal to the one in the component template. The IS relation describes a *type test*, determining whether the actual template of the component fulfils a specific signature.

Examples:

```
student OFFERS Student, Employee
student REQUIRES Supervisor [2..*]
student IS AcademicAssistant
```

Apart from the usual Oberon operands, receive guards and the EXISTS-test are available:

```
Operand    =Number |ConstChar |Text |Designator I
             ReceiveTest IInputTest I ExistsTest I FunctionCal1.
```

A.S.1 The Receive-Test

The *receive-test* results a Boolean value which is TRUE, if any or a specific message has arrived over a designated interface, by first awaiting a message input from the interface. It blocks the execution until the arrival of a message but does not receive the message or assign the parameters.

```
ReceiveTest      =[Designator] "?" MessagePattern.
MessagePattern   = Identifier |ANY |FINISH.
```

The message pattern is ANY, if an arbitrary message is awaited, or it is the identifier of a specific message. The predefined FINISH-message (see *A.5A)* signals the client-side termination of the communication.

Examples:

```
IF Library?BorrowBook (* client-side receive test *) THEN
 (* ... *)
END

IF ?BookRequest (* server-side receive test *) THEN (* ... *) END
```

A.S.2 The INPUT-Test

The *INPUT-test* results a Boolean value which is TRUE, if any or a specific message has arrived from a designated interface. The INPUT-test does neither block the execution nor accept the message.

```
InputFunction =INPUT "(" [ Designator "," ] MessagePattern ")".
```

Examples:

```
IF INPUT(Library, BorrowBook) (* client-side input test *) THEN
 (* ... *)
END

IF INPUT(BookRequest) (* server-side receive test *) THEN
 (* ... *)
END
```

A.8.3 The EXISTS·Test

The *EXISTS-test* results a Boolean value which is TRUE, if a component is contained in the designated storage location.

```
ExistsTest = EXISTS "(" Designator ")",
```

Example:

```
(* VARIABLE building[street: TEXT; number: INTEGER]: House;*)
EXISTS(building["Street A", 45])
```

A.9 Designators

A *designator* refers to a constant, variable, component, or interface. It is represented by an identifier and may be combined with a selector, if the designated element is part of another construct.

```
Designator::: Identifier
            | Identifier "[" ExpressionList "] // indexed selection
            | Designator "(" Designator ")" // interface selection
            | Designator "(" Type ")". // type guard
```

A.9.1 Variable Designators

A *variable designator* is denoted by the identifier of the corresponding variable. The designator has the declared type of the variable.

Examples:

```
book (* refers to VARIABLE book: Book *)
building (* VARIABLE building[postalAddress: TEXT]: House; *)
```

A.9.2 Storage Location Designators

A *storage location designator* refers to either a normal variable, a location in a collection variable, or a procedure parameter. It has the same type like the variable or procedure parameter.

In the case of a normal variable, the storage location designator is the variable designator itself.

For a collection variable, the storage location designator is V[E], where V is the variable designator and E is an expression list, which defines the index of the location in the collection. E must be type-compatible with the declared parameter list of the collection variable.

Examples:

```
x          building["12 Market Street"]
```

A.9.3 Component Designators

A *component designator* is denoted by the designator of the storage location, wherein the component is stored. It has the same signature like the storage location, which must not be empty when the designator is evaluated.

Examples:

```
book       bUilding["Market Street", 4]
```

Alternatively, a component designator may also be a *type guard* X(D), which asserts at runtime that the component designated by X satisfies the signature D.

Example:

```
VARIABLE student: ANY(Student | University, Supervisor [2..*]);
student(AcademicAssistant)
student(ANY(Student, Employee | University, Supervisor [2..*]))
```

A.9.4 Data Value Designators

A *data value designator* is denoted by the designator of the storage location, wherein the data value is stored. It has the same type like the storage location, which must not be empty when the designator is evaluated.

Example:

```
matrix[3, 4]
  (* VARIABLE matrix[row, column: INTEGER]: REAL; *)
```

For a data value A of the type TEXT, A[k] designates the k^{th} character in the character sequence of A, where k is an INTEGER expression.

A.9.4 Interface Designators

An *inteiface designator* denotes an offered or required interface of a sub-component or the local component itself. The place of use (context) defines whether it refers to an offered or a required interface.

For sub-components, an interface designator is typically denoted as J(X), where X is the designator of the sub-component and J the identifier of the interface specification. Only in the case of multiple required interfaces of the same specification has the interface to be designated by J[i](X), where i defmes the index in the form of an INTEGER expression. The index must be between 1 and the specified maximum number of required interfaces. The interface J must be specified as offered or required by the statically declared signature of X.

As a short cut, an offered interface of a component may be also referred to as the component designator, if the component only offers one interface.

Examples:

```
(*VARIABLE building: ANY(Residence | Electricity [1 .. 2], Water);*)
Residence(building)
building (* short-cut *)
Electricity[1 ](building)
```

For external offered or required interfaces of the local component, an interface is typically designated by the identifier of the corresponding interface specification. Only if the component requires multiple required interfaces of the same specification has the designator to be denoted as J[i], where J is the identifier of the interface specification and i the index expression.

Examples:

```
COMPONENT CompanyBuilding
  OFFERS OfficeSpace REQUIRES Electricity [2..*], Water;
BEGIN
  Water  (* required Water interface of CompanyBuilding *)
  Electricity[2]        (* second required Electricity interface *)
END CompanyBuilding;
```

A.tO Virtual Time

An inbuilt concept of virtual time is featured for simulation programming. By default, each component is equipped with an individual intrinsic virtual time that corresponds to the simulated physical time. All computing operations happen within zero virtual time, while the virtual time only proceeds by means of the PASSIVATE- and AWAIT-statements.

A.IO.I The PASSIVATE-Statement

The PASSIVATE-statement defines that the process is to be suspended for a certain amount of virtual time. The execution only continues when all other processes within the same component wait for a virtual time, that is equal or larger than the end time of the PASSIVATE-statement.

Example:

```
PASSIVATE(100)
 (* the process suspends for the duration of 100 units of
    the virtual time *)
```

If used inside an EXCLUSIVE or SHARED statement sequence, the corresponding lock is temporarily released when the process is waiting.

A.IO.2 The AWAIT-Statement

With the AWAIT-statement, a process waits for an undefmed virtual time, as long as a certain Boolean condition is not satisfied. The execution continues when the condition is established for the first virtual time.

Examples:

```
AWAIT(hasFreeRoom)
  (* awaits the condition, now or in virtual future *)
AWAIT(INPUT(Message))
  (* awaits the message, while virtual time may elapse *)
```

The AWAIT-statement can only occur inside an EXCLUSIVE or SHARED statement sequence. Thereby, the monitor lock is temporarily released while the process is waiting.

A.I0.3 The TIME-Variable

The predefined read-only variable TIME of the type INTEGER reflects the current virtual time inside a component.

Example:

```
TIME (* read-only access to the current virtual time *)
```

A.I0.4 Hierarchical Time-Synchronisation

A component may contain sub-components, which have the same virtual time like the container. By annotating the attribute SYNCHRONOUS to the corresponding variables, a component may identify such sub-components. The virtual time of the sub-components is then exactly time-synchronous to the outer instance.

Example:

```
COMPONENT TrafficSimulation
  VARIABLE car[id: INTEGER]: Car {SYNCHRONOUS}
  (* all subRcomponents car[..] have the same virtual time like
     the surrounding TrafficSimulation *)
END TrafficSimulation;
```

A.11 Language Symbols

The definition of lexical symbols and comments is equal to Oberon [Wirth90], except the introduction of attributes. An attribute can be associated to a syntactical construct, in order to specify additional semantics for the construct.

```
AttributeList:::: "{" IdentifierList "}".
```

A.12 Predefined Features[55]

A.12.1 Predefined Procedures

A set of procedures are already predefined. Some of them are generic, i.e. apply to several types of arguments.

Proper procedures:

Procedure	Arguments	Effect
ASSERT(x)	BOOLEAN	terminate the process with an error, if x is FALSE
ASSERT(x, n)	x: BOOLEAN n: INTEGER	terminate the process with an error, if x is FALSE; n is a parameter whose interpretation is lett to the system
HALT(n)	n: INTEGER	terminate the process with an error; n is a parameter whose interpretation is ieft to the system
INC(v)	INTEGER storage location	v := v + 1
INC(v, x)	v: INTEGER storage location x: INTEGER	v := v + x
DEC(v)	INTEGER storage location	v := v - 1
DEC(v, x)	v: INTEGER storage location x: INTEGER	v := v - x
PASSIVATE(n)	n: INTEGER	suspend the process for a defined non-negative virtual time, see A.I0.1
WRITE(x)	INTEGER, REAL, CHARACTER or TEXT	write value to the system output stream
WRITEHEX(x)	INTEGER	write in hexadecimal fonmat to the system output stream
WRITELINE		write a line break to the system output stream

Function procedures:

Procedure	Arguments	Result type	Result
COUNT{J)	required interface designator but without index	INTEGER	number of connected required interfaces but at least the specified minimum number
LENGTH(x)	TEXT	INTEGER	number of characters in x

55 Parts of this section are adopted from the Oberon language report [Wirth90].

TERMINATEDO		BOOLEAN	all internal service processes 01 the component are terminated and the com-Donent should be finalised
SQRT(x)	REAL	REAL	square root of x
SIN(x)	REAL	REAL	sine of x
COS(x)	REAL	REAL	cosine of x
TAN(x)	REAL	REAL	tangent 01 x
ARCSIN(x)	REAL	REAL	arcsine of x (1 <- x <= 1)
ARCCOS(x)	REAL	REAL	arccosine of x (1 <= X <= 1)
ARCTAN(x)	REAL	REAL	arctanllent 01 x
RANDOM(x, y)	x. y: INTEGER	INTEGER	random number uniformly distributed between x and v (includino them)
MIN(T)	T = INTEGER or REAL	T	minimum integer or real number
MAX(T)	T = INTEGER or REAL	T	maximum integer or real number

Type conversions:

Procedure	Arguments	Result type	Result
CHARACTER(x)	INTEGER	CHARACTER	character with code x
INTEGER(x)	CHARACTER or REAL	INTEGER	code of character x or, largest integer not greater than real x; note that INTEGER(i/j) = i DIV j
REAL(x)	INTEGER	REAL	(approXimated) real number represen-tation 01 x
TEXT(x)	CHARACTER	TEXT	text consisting of the one character x

A.12.2 Predefined Constant

The following constant is predefined:

```
PI = 3.14159265...  (* with machine-dependent precision *)
```

A.12.3 Predefined Variable

The following variable is predefmed:

```
TIME    (* current virtual time, see A.1 0.3 *)
```

A.12.4 Predefined Attributes

The following attributes are defined for the following syntactical constructs:

Attribute	Syntactical Constructs	Semantics
EXCLUSIVE	statement sequence expression	exclusive component lock during the execution of the construct
SHARED	statement sequence expression	shared component lock during the execution of the construct
SYNCHRONOUS	variable of component signature	sub-component in the variable is time-synchronous with the outer component

A.12.S Special Characters

The language employs the following special character or character pairs:

```
I {}  ()  [ ]  "(quotation mark)  ,(comma)  ; (semicolon)
: (colon)  . (dot)  .. (ellipsis)
~  +  -  *  /  =  #  <  >  <=  >=
```

A.12.6 Reserved Keywords

The keywords listed below are reserved and can not be used as identifiers:

ACTIVITY, AND, ANY, ARCCOS, ARCSIN, ARCTAN, ASSERT, AWAIT, BEGIN, BOOLEAN, BY, CHARACTER, COS, COMPONENT, CONNECT, CONSTANT, COUNT, DEC, DELETE, DISCONNECT, DIV, DO, ELSE, ELSIF, EXISTS, END, EXCLUSIVE, FALSE, FINALLY, FINISH, FOR, FOREACH, HALT, INC, INTEGER, IF, IMPLEMENTATION, IN, INPUT, INTERFACE, IS, LENGTH, MAX, MIN, MOD, MOVE, NEW, OF, OFFERS, OR, OUT, PASSIVATE, PI, PROCEDURE, RANDOM, REAL, REPEAT, REQUIRES, RETURN, SHARED, SIN, SQRT, SYNCHRONOUS, TAN, TERMINATED, TEXT, THEN, TIME, TO, TRUE, UNTIL, VARIABLE, WHILE.

A.I3 Syntax Summary

```
Program          = { Component | Interface }.

Component        = COMPONENT Identifier
                     [ OFFERS InterfaceDeclList ]
                     [ REQUIRES InterfaceDeclList ] ";"
                       ComponentBody
                     END Identifier ";".
InterfaceDeclList = InterfaceDecl {"," InterfaceDecl}.
InterfaceDecl    = Identifier [ "[" NofInterfaces "]" ].
NofInterfaces    = Integer ["" ( Integer |"·")].

Interface        = INTERFACE Identifier ";"
                       Protocol
                     END Identifier ";".
Protocol         = [ ProtocolExpr ].
ProtocolExpr     = ProtocolTerm { "|" ProtocolTerm }.
ProtocolTerm     = ProtocolFactor { ProtocolFactor }.
ProtocolFactor   = MessageDecl
                 | "[" ProtocolExpr "]"
                 | "{" ProtocolExpr "}"
                 | "(" ProtocolExpr ")".
MessageDecl      = ( IN |OUT} Identifier [ "(" ParameterList ")" ].
ParameterList    = ParamSection {";" ParamSection}.
ParamSection     = IdentifierList ":" Type.

ComponentBody    = { Declaration}
                     { Implementation}
                     [ BEGIN StatementSeq]
                     [ ACTIVITY StatementSeq ]
                     [ FINALLY StatementSeq].
Declaration      = Component | Interface | ConstantList |
```

```
                    VariableList I Procedure.
ConstantList        = CONSTANT Constant {Constant}.
Constant            = Identifier "=" ConstantExpr ";",
VariableList        = VARIABLE VariableSection { VariableSection}.
VariableSection     = IndexedIdentList ":" Type [ AttributeList )";".
IndexedIdentList    = IndexedIdent { "," IndexedIdent }.
IndexedIdent        = Identifier [ "[" ParameterList "]" ).
Procedure           = PROCEDURE Identifier
                      [ "(" [ ProcParamList )")" [ ":" Type ))";"
                        { Declaration}
                        [BEGIN StatementSeq )
                      END Identifier ";".
ProcParamList       = ProcParSection { ";" ProcParSection }.
ProcParSection      = [VARIABLE) ParamSection.

Implementation      = IMPLEMENTATION Identifier ";"
                        { Declaration}
                        [BEGIN StatementSeq)
                      END Identifier ";".

Type                = Identifier | ANY [ "(" AnyInterfaceList ")" ).
AnyInterfaceList    = [ InterfaceDeclList) [ "|" InterfaceDeclList).

StatementSeq        = [ AttributeList ) StatementList.
StatementList       = Statement { ";" Statement}.
Statement           = [ Assignment | New | Connect | Disconnect
                      I Send | Receive I Delete | Move
                      I Await | ProcedureCall | Return
                      IIf | While | Repeat | For I Foreach
                      | StatementBlock).
Assignment          = Designator ":=" Expression.
New                 = NEW "(" Designator [","  Identifier )")".
Connect             = CONNECT "(" Designator "," Designator ")".
Disconnect          = DISCONNECT "(" Designator ")".
Send                = [ Designator )"!" Identifier
                      [ "(" ExpressionList ")" ).
Receive             = [ Designator )"?" Identifier
                      [ "(" DesignatorList ")" ).
Delete              = DELETE "(" Designator ")".
Move                = MOVE "(" Designator "," Designator ")".
Await               = AWAIT "(" Expression ")".
ProcedureCali       = Identifier [ "(" ExpressionList ")" ).
Return              = RETURN [ Expression ).
If                  = IF Expression THEN StatementSeq
                      { ELSIF Expression THEN StatementSeq)
                      [ ELSE StatementSeq )
                      END.
While               = WHILE Expression DO
                        StatementSeq
                      END.
Repeat              = REPEAT
                        StatementSeq
                      UNTIL Expression.
For                 = FOR Designator ":=" Expression TO Expression
                      [ BY ConstantExpr ) DO
                        StatementSeq
```

```
                    END.
Foreach             = FOREACH DesignatorList OF Designator DO
                      StatementSeq
                    END.
StatementBlock      = BEGIN StatementSeq END.

ExpressionList      = Expression {"," Expression}.
ConstantExpr        = Expression.  // statically evaluable
Expression          = [ AttributeList ]
                      ( SimpleExpr
                        [ ("=" |"#" |"<" | "<=" |">" |">=") ] SimpleExpr
                      | Designator
                        (OFFERS | REQUIRES) InterfaceDeciList
                      | Designator IS Type ).
SimpleExpr          = [ "+" |"-" ]Term { ( "+" |"-" | OR) Term }.
Term                = Factor { ( "*" | "/" |DIV |MOD |AND) Factor }.
Factor              = Operand |"-" Factor | "(" Expression ")".
Operand             = Number | ConstChar |Text |Designator |
                      ReceiveTest |InputTest |ExistsTest |
                      FunctionCall.
ReceiveTest         = [ Designator ]"?" MessagePattern.
InputTest           = INPUT "(" [ Designator"," ] MessagePattern ")".
MessagePattern      = Identifier |ANY |FINISH.
ExistsTest          = EXISTS "{" Designator ")".
FunctionCall        = Identifier "(" [ ExpressionList ] ")".

DesignatorList      = Designator {"," Designator}.
Designator          = Identifier
                    | Identifier "[" ExpressionList "]"
                    | Designator "(" Designator ")"
                    | Designator "(" Type ")".

AttributeList       = "{" IdentifierList "}".
IdentifierList      = Identifier {"," Identifier}.
Identifier          = Letter { Letter |Digit}.
Letter              = "A" .. "Z" | "a" .. "z".
Digit               = "0" .. "9".
Number              = Integer |Real.
Integer             = Digit { Digit} |Digit { HexDigit} "H".
HexDigit            = Digit |"A" .. "F".
Real                = Digit { Digit} "." { Digit} [ ScaleFactor ].
ScaleFactor         = "E" [ "+" |"-" ]Digit { Digit}.
ConstChar           = "'" Character "'" |Digit { HexDigit } "X".
Text                = '"' { Character} '"'.
Character           = any character except quotation mark.
```

AppendixB

User Commands

The following user commands are supported by the runtime system to manage and use components in the system scope. The symbols in italics denote arguments that are defined by the user.

NEW(name, *template)*	create a new component with a unique name from a component template
CON*NECT(interlace(from)*, *to)* *CONNECT(interlace[/](from, to)*	connect a required interface of a component to the matching offered interface of another component; an index is specified if the component requires multiple instances of that interface
DISCONNECT(interlace(name)) DISCONNECT(*interlace[i)(name))*	disconnect the required interface of a component; an index is specified if the component requires multiple instances of that interface
DELETE(name)	delete a component
interlace(name)! message *interlace(name)!* message(*arg*1, ...)	send a message with possible arguments to an offered interface of a component
interlace(name)? message	try to receive a message (with its arguments) from an offered interface of a component

Example:

```
NEW(I, PublicLibrary); NEW(c, LibraryCustomer)
CONNECT(Library[1)(c), I)
Customer(c)! InterestedIn(123456)

DELETE(c); DELETE(I)
```

The predefined component with name SYSTEM represents the system itself and offers the following interfaces:

```
INTERFACE FileSystem;
 ( IN New(name: TEXT) |IN Open(name: TEXT) )
 (
   OUT Done
  {
    IN SetPosition(position: INTEGER)
   |IN GetPosition OUT Position(pos: INTEGER)
   |IN GetLength OUT Length(len: INTEGER)
   |( IN ReadByte | IN ReadLine )
     ( OUT Byte(x: CHARACTER) |OUT Line(x: TEXT) |OUT EOF )
   |IN Write(x: CHARACTER) |IN WriteText(x: TEXT)
   |IN Update
  }
  IN Close
 |
  OUT Failed
 )
END FileSystem;

INTERFACE SystemTime;
 IN GetSystemTime OUT SystemTime(ticks: INTEGER)
END SystemTime;

INTERFACE GraphicView;
 {IN Clear
 |IN GetSize OUT Size(width, height, bgColor: INTEGER)
 |IN Pixel(x, y, color: INTEGER)
 |IN Font(x, y: INTEGER; char: CHARACTER; color: INTEGER)
 |IN Fill(x, y, w, h, color: INTEGER)
 |IN SetLayer(level: INTEGER) |IN DrawLayers}
END GraphicView;
```

In addition, the following user commands allow inspecting the system state:

SHOW COMPONENTS	list all components in the system scope
SHOW RESOURCES	show an overview of the occupied and free system resources
SHOW VOLUMES	list all loaded file system volumes
SHOW FILES	list all files from all loaded volumes
MOUNT IDEX#y	load an Oberon file system volume from a disk partition
UNMOUNT IDEX#y	unload a file system volume

For evaluation purposes, the system supports different kinds of schedulers. The default is the smart scheduler which is described in Section 5.2.9.

SET MODE SMART	selective scheduling of independent processes on different processors
SET MODE SERIAL	scheduling all processes on the same processor
SET MODE PARALLEL	scheduling processes by using all processors

Appendix C

Deadlock Exclusion

This appendix shows how deadlocks can be abandoned in the component language by following three simple programming rules. *However, we regard these rules as too restrictive for general scenarios, such that we do not enforce them in the programming language.* With the subsequent approach, we merely exclude deadlocks that could be provoked by the monitor locks of components. Of course, processes may also wait perpetually if they are blocked by an AWAIT-condition that will be never satisfied in future. However, processes do then not really hold a monitor lock, such that we do not consider such situations in this deadlock exclusion scheme.

C.I Rules

The specific rules for deadlock exclusion are:

- Rules 1: Components can only be connected to acyclic networks.
- Rules 2: A process can only have one open client-side communication at the same time.
- Rules 3: During a client-side communication, a process must not acquire a monitor lock.

The first rule can be easily governed by requiring sub-components to only be acyclically connected according to the linear declaration order of the variables. A required interface of a sub-component can only be connected to an offered interface of another component, if the second component is declared in a variable after that of the first com-

ponent. Interface redirection from and to external interfaces of the super-component can be unrestrictedly performed. For components in the same collection, the relation of order between the index values is decisive. (If the index consists of a list of multiple values, the first values are more significant.) Rule 1 can be statically verified, unless sub-components of the same collection are connected. In the latter, a runtime check is needed.

Example:

```
VARIABLE
  house[postalAddress: TEXT]: 8tandardHouse;
  powerPlant: HydroelectricPowerPlant;
  river[number: INTEGER]: River;
BEGIN
  CONNECT(Electricity(house["12 Market Street"]), powerPlant);
  CONNECT(Water(house["12 Market Street"])), river[1]);
  CONNECT(Water(powerPlant), river[1])
```

The second rule is more restrictive and requires that a process, which communicates as a client via an interface, has to complete the EBNF-communication protocol, before it can initiate another client-side communication via the same or a different interface. However, a process may act as a communication server and as a communication client at the same time. If a communication protocol is finished, processes are also allowed to restart a communication via the same interface at a later time. A process is therefore required to only interact with the same interface between a potential first and definitive final message in a statement block. This may be dynamically checked.

Example:

```
INTERFACE Library;
  IN BorrowBook(isbn: INTEGER)
  (
    OUT Result(b: ANY(Book))
    IN ReturnBook(b: ANY(Book))
  |
    OUT Unavailable
  )
END Library;
```

```
COMPONENT Customer OFFERS Person REQUIRES Library;
 IMPLEMENTATION Person;
 VARIABLE isbn: INTEGER; b: ANY(Book);
 BEGIN
  ?InterestedIn(isbn);
  (* begin of client-side communication *)
  Library!BorrowBook(isbn);
  IF Library?Result THEN
   Library?Result(b); Library!ReturnBook(b)
  ELSE Library?Unavailable
  END
  (* end of client-side communication *)
 END Person;
END Customer;
```

The third rule prohibits an exclusive or shared monitor lock between a potential first and definitive final message of a client-side communication. As mentioned in Section 4.6.2, monitor locks may not be nested within a sequential execution.

Example:

```
IMPLEMENTATION Person;
VARIABLE isbn: INTEGER; b: ANY(Book);
BEGIN (* EXCLUSIVE/SHARED attribute is here permitted *)
 ?Interestedln(isbn);
 Library!BorrowBook(isbn);
 (* an EXCLUSIVE/SHARED block is here not allowed *)
 IF Library?Result THEN
  Library?Result(b); Library!ReturnBook(b)
 ELSE Library?Unavailable
 END
END Person;
```

C.2 Correctness Proof

First, it should be noted that the abovementioned linear order for acyclic component connections, implies a global ordering of components. Each component has an associated local number, which is its position in the linearly ordered sub-components of its directly surrounding component. As a consequence, a component may also have a global number $(x_1, X2, .., x_{n-1}, xn)$, where X_n is the local number of

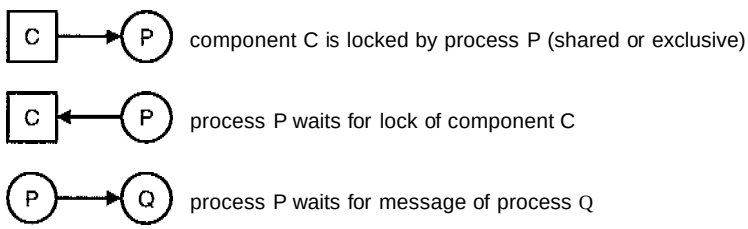
the component, x_{n-1} the local number of its direct super-component, x_{n-2} the local number of its next hierarchically containing super-component, and so on. Global numbers are ordered as follows: $(x_1, \ldots, x_n) < (y_1, \ldots, y_n)$, if and only if, $x_i <= y_i$ for $i = 1..n-1$ and $x_n < y_n$.

For a proof by contradiction, we assume that there would be a deadlock in a program. The deadlock situation always entails a cyclic lock dependency among components or processes. The lock dependencies among components and processes may be represented in a directed graph, by using the following notation:

Nodes;

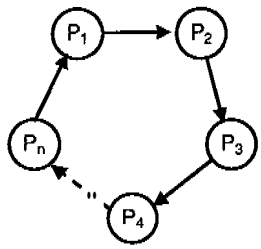
D component 0 process

Edges;

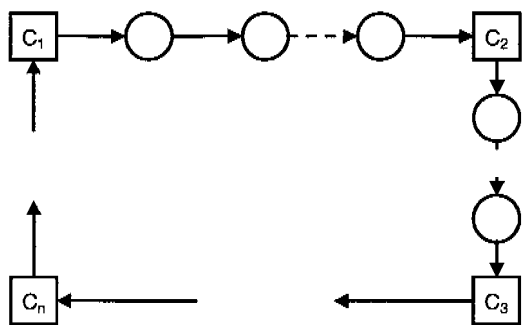


It can be seen that a deadlock always involves a cycle in the directed graph. If there would be deadlock without such a cycle, there would be a node involved in the deadlock (a component or a process) without an outgoing edge (as the graph is finite). This node can either run (in the case of a process) or the corresponding lock on a component may be acquired (in the case of a component).

It is prevented by the EBNF-communication protocol that two processes mutually wait for a message. It is also impossible that only processes cyclically wait for messages, as illustrated below. Since each process can only serve at most one other process at the same time, it can be said without loss of generality that Pl is a client of P2, P2 a client of P_3, and so on, until P_n is a client of Pl. However, the processes would then belong to components that are cyclically connected, which contradicts Rule 1.



Therefore, a deadlock scenario must also involve components and can be represented as the following graph:



In a deadlock scenario, two subsequent components in the cycle are indirectly dependent by a series of process nodes, which are connected with equally directed edges:

As a process can only perform one client interaction at the same time (Rule 2) and may serve at most one other process (due to client-individual communication), there are only two possible relations between processes:

1. PI is the server of P_2, P_2 the server of P_3 and so on.
2. P_j is the client of P_2, P_2 the client of P_3 and so on.

The first possibility can be excluded, because P_n would try to acquire a lock during a client interaction (Rule 3). Using Rule 1, the global number of A is less than that of B, since a process in a component can only interact as a client with components of a higher global number. As a consequence, the components may not be cyclically dependent and a deadlock can not occur.

AppendixD

Digital Material

Together with this thesis, the following material is provided on a compact disk medium.

- The binary code of the runtime system and the compiler (induding installation instructions)
- The source code of the runtime system and the compiler
- The complete language report
- The documentation of the memory model used in the runtime system
- All the programs used in the benchmarks
- The traffic simulation programs in source code with test files (Official data has to be requested at the Swiss Federal Department of the Environment, Transport, Energy and Communications [UVEK].)

Bibliography

[ABOl] C. Artho and A. Biere. Applying Static Analysis to Large-Scale, Multi-Threaded Java Programs. Proc. of the Australian Conf. on Software Engineering (ASWEC), August 2001.

[AC04] J. Aldrich and C. Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In Proc. of the European Conference on Object-Oriented Programming (ECOOP), June 2004.

[ACN02] J. Aldrich, C. Chambers, D. Notkin. ArchJava: Connecting Software Architecture to Implementation. In Proc. of the lntl. Conference on Software Engineering (ICSE), May 2002.

[AEL88] A. W. Appel, J. R. Ellis, and K. Li. Real-Time Concurrent Collection on Stack Multiprocessors. In Proc. of the Conf. on Programming Language Design and Implementation (PLDI), June 1988. ACM SIGPLAN Notices, 23(7):11-20, July 1988.

[Agha86] G. Agha. Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, 1986.

[A1l97] R. Allen. A Formal Approach to Software Architecture. PhD thesis, School of Computer Science, Carnegie Mellon University, 1997.

[Alm97] P. S. Almeida. Balloon Types: Controlling Sharing of State in Data Types. In Proc. of the European Conference on Object-Oriented Programming (ECOOP), June 1997.

[AOS06] The Active Oberon Language and the Bluebottle System. Ern Zurich, http://www.bluebottle.ethz.ch. 2004.

[Baker78] Jr. H. G. Baker. List Processing in Real-Time on a Serial Computer. Communications of the ACM, 21(4): 280-294, April 1978.

[Baker92] Jr. H. G. Baker. The Treadmill: Real-Time Garbage Collection without Motion Sickness. ACM SIGPLAN Notices 27(3):66-70, March 1992.

[Barnes80] J. Barnes. An Overview of Ada. Software - Practice and Experience, 10: 851-887, 1980.

[Barnes95] J. Barnes. Programming in Ada95. Addison-Wesley, 1995.

[BC90] G. Bracha and W. Cook. Mixin-Based Inheritance. In Proc. of the Inti. Conf. on Object-Oriented Systems, Languages, and Applications (OOPSLA), October 1990.

[BCF04] N. Benton, L. Cardelli and C. Fournet. Modern Concurrency Abstractions for C#. ACM Transactions on Programming Languages and Systems (TOPLAS) 26(5): 269-804, September 2004.

[BCR03] D. F. Bacon, P. Cheng, and V. T. Rajan. A Real-Time Garbage Collector with Low Overhead and Consistent Utilization. In Proc. of the Symp. on Principles of Programming Languages (POPL), January 2003.

[BE+94] P. Bims, M. Englehart, M. Jackson, S. Vestal. Domain-Specific Software Architectures for Guidance, Navigation and Control. International Journal of Software Engineering and Knowledge Engineering, 6(2), January 1994.

[BH72] P. Brinch Hansen. Structured Multiprogramming. Communications of the ACM, 15(7): 574-578, July 1972.

[BH73] P. Erinch Hansen. Operating System Principles, Section 7.2 Class Concept, Prentice-Hall, Englewood Cliffs, NJ, 226-232, July 1973. Reprinted in [BH02], Shared Classes, 265-271.

[BH75] P. Brinch Hansen. The Programming Language Concurrent Pascal. IEEE Transactions on Software Engineering, 1(2):199-207, 1975.

[BH02] P. Brinch Hansen, Ed.. The Origin of Concurrent Programming: From Semaphores to Remote Procedure Calls. Springer Verlag, 2002.

[Bläser06] L. Bläser. A Component Language for Structured Parallel Programming. In Proc. of the Joint Modular Languages Conference (JMLC), September 2006. Lecture Notes in Computer Science (LNCS), Vol. 4228, pp. 230-250, Springer Verlag, 2006.

[BLR02] C. Boyapati, R. Lee, and M. Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In Proc. of the Inti. Conf. on Object Oriented Programming Systems, Languages, and Applications (OOPSLA), November 2002.

[BROl] C. Boyapati and M. Rinard. A Parameterized Type System for Race-Free Java Programs. In Proc. of the Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), October 2001.

[BSB+03] C. Boyapati, A. Salcianu, Jr. W. Beebee, and M. Rinard. Ownership Types for Safe Region-Based Memory Management in Real-Time Java. In Proc. of the Conf. on Programming Language Design and Implementation (PLDI), June 2003.

[CBOO] P. Cheng and G. E. Blelloch. A Parallel, Real-Time Garbage Collector. In Proc. of the Conf. on Programming Language Design and Implementation (PLDI), June 2001. ACM SIGPLAN Notices, 36(5): 125-136, May 2001.

[COM06] Microsoft COM. http://www.microsoft.com/com. 2006.

[CPN98] D. G. Clarke, J. M. Potter, and 1. Noble. Ownership Types for Flexible Alias Protection. In Proc. of the Inti. Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), October 1998.

[CM81] K. M. Chandy and J. Misra. Asynchronous Distributed Simulation via a Sequence of Parallel Computations. Communications of the ACM, 24(11):198-206, April 1981.

[CPHP87] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: A Declarative Language for Programming Synchronous Systems, In Proc. of the Symp. on Principles of Programming Languages (POPL), January 1987.

[CR93] A. Colmerauer and P. RousseL The Birth of Prolog. In Second History of Programming Languages, ACM SIGPLAN Notices, 37-52, March 1993.

[CS06] C# Language Specification. ISOIJEC Standard 23270, 2003.

[Dij65] E. W. Dijkstra. Cooperating Sequential Processes. Technological University, Eindhoven, Netherlands, September 1965. Reprinted in [BH02], 65-138.

[DN66] O. - 1. Dahl and K. Nygaard. SIMULA - An ALOGL-based Simulation Language. Communications of the ACM, 9(9):671-678, 1966.

[DMN68] O.-J. Dahl, B. Myhrhaug, K. Nygaard. SIMULA 67 Common Base Language. Norwegian Computing Center 1968.

[EA03] D. Engler and K. Ashcraft. Racer X: Effective, Static Detection of Race Conditions and Deadlocks. In Proc. of the Symposium on Operating System Principles (SOSP), October 2003.

[FA+06] M. Fähndrich, M. Aiken, C. Hawblitzel, et al. Language Support for Fast and Reliable Message-Based Communication in Singularity OS. In Proc. of EuroSys 2006, April 2006.

[Fried07] F. Friedrich. WinAOS. http://winaos.de. Last accessed 2007.

[GA094] D. Garlan, R. Allen, and J. Ockerbloom. Exploiting Style in Architectural Design Environments. In Pmc. of the Symposium on the Foundations of Software Engineering (FSE), December 1994.

[GG04] R. Giintensperger and J. Gutknecht. Active C#. In Proc. of the Inti. Workshop on .NET Technologies, May 2004.

[GH+95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[GL+03] D. Gay, P. Levis, R. von Behren et al. The nesC Language: A Holistic Approach to Networked Embedded Systems. In Proc. of the Conf. on Programming Language Design and Implementation, June 2003. ACM SIGPLAN Notices, 38(5): 1-11, May 2003.

[GMW97] D. Garlan, R. Monroe, and D.Wile. Acme: An Architecture Description Interchange Language. In Proc. of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON), November 1997.

[Gut97] J. Gutknecht. Do the Fish Really Need Remote Control? A Proposal for Self-Active Objects in Oberon. In Proc. of the Joint Modular Language Conference (JMLC), March 1997. Lecture Notes in Computer Science (LNCS), Vol. 1204, Springer Verlag, 1997.

[GJS+OO] J. Gosling, B. Joy, G. Steele, and G. Bracha. The Java Language Specification, Second Edition, Addison Wesley, 2000.

[Gough02] J. Gough. Compiling for the .NET Common Language Runtime (CLR). Prentice Hall, 2002.

[GZ05] J. Gutknecht and E. Zueff. Zonnon Language Report. ETH Zurich, http://www.zonnon.ethz.ch. 2005.

[HL+05] G. Hunt, J. Larus, M. Abadi et al. An Overview of the Singularity Project. Technical Report MSR-TR-2005-135, Microsoft Research, October 2005.

[Hoare71] C. A. R. Hoare. Towards a Theory of Parallel Programming. In Operating Systems Techniques, Proc. of a Seminar at Queen's University, Belfast, Northern Ireland, August-September 1971. C. A. R. Hoare and R. H. Perrott, Eds. Academic Press, New York (1972),61-71. Reprinted in [BH02], 231-244.

[Hoare73] C. A. R. Hoare. Hints on Programming Language Design. Stanford Artificial Intelligence Laboratory Memo AIM-224 or STAN-CS-73-403, Stanford University, Stanford, California, December 1973.

[Hoare74] C. A. R. Hoare. Monitors: An Operating System Structuring Concept. Communications of the ACM, 17(10):549-557, October 1974.

[Hoare78] C. A. R. Hoare. Communicating Sequential Processes. Communications of the ACM, 21(8):666-677, August 1978.

[Hogg91] J. Hogg. Islands: Aliasing Protection in Object-Oriented Languages. In Proc. of the IntI. Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), October 1991.

[IA32] Intel Corporation. IA32 Intel® Architecture Software Developer's Manual, Volumes 1-4, 2004.

[IW+88] D. Ingalls, S. Wallace et al. Fabrik: A Visual Programming Environment. In Proc. of the IntI. Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), September 1988.

[PJ02] S. Peyton Jones (00.). Haskell 98 Language and Libraries: The Revised Report, http://www.haskell.org, February 2002.

[JB98] JavaBeans Component Architecture Specification, http://java.sun.com/products/javabeans, 1998.

[JL03] R. Jones and R. Lins. Garbage Collection: Algorithms for Automatic Dynamic Memory Management. Addison-Wesley, 2003.

[Keller06] R. Keller. Improved Stack Management in the Active Oberon Kernel. Master Thesis. ETH Ziirich, 2006.

[LabView] Lab View User Manual. National Instruments, Austin, TX.

[Lam78] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM, 21(7):558-565, July 1978.

[LK+95] D. C. Luckham, J. J. Kenney, L. M. Augustin et al. Specification and Analysis of System-Architecture Using Rapide. IEEE Transactions on Software Engineering. 21(4): 336-355, April 1995.

[LS05] Y. D. Liu and S. F. Smith. Interaction-Based Programming with Classages. In Proc. of the Inti. Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), October 2005. ACM SIGPLAN Notices, 40(10): 191-209, October 2005.

[Matsim] MATSIM. Multi-Agent Traffic Simulation. http://www.matsim.org. Last accessed March 2007.

[Meyer97] B. Meyer. Object Oriented Software Construction. Second Edition, Prentice Hall, 1997.

[MD+95] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In Proc. of the European Software Engineering Conference, September 1995.

[MK96] J. Magee and J. Kramer. Dynamic Structure in Software Architectures. In Proc. of the Symposium on Foundations of Software Engineering (FSE), October 1996.

[MMN93] O. L. Madsen, B. M011er-Pedersen, K. Nygaard. Object-Oriented Programming in the Beta Programming Language. Addison Wesley, 1993.

[MT+97] R. Milner, M. Tofte, R. W. Harper, and D. MacQueen. The Definition of Standard ML (Revised). MIT Press, 1997.

[MSA+85] J.R. McGraw, S.K. Skedzielewski, S.J. Allan et al. SISAL: Streams and Iteration in a Single Assignment Language: Reference Manual Version 1.2. Manual M-146, Lawrence Livermore National Laboratory, Livermore, CA, 1985.

[McIl68] M. D. McIlroy. Mass produced software components. In Proc. of NATO Software Engineering Conference, P. Naur and B. Randell (eds.), 1:138--150, October 1968.

[Med96] N. Medvidovic. ADLs and Dynamic Architecture Changes. In Joint Proc. of the SIGSOFT'96 Workshops, October 1996.

[Meyer97] B. Meyer. Object-Oriented Software Construction, Second Edition. Prentice Hall, 1997.

[MHOO] R. Monson-Haefel. Enterprise Java Beans, Second Edition. a'Reily, 2000.

[MP01] P. Millier and A. Poetzsch-Heffter. A Type System for Alias and Dependency Control. Technical Report 279, FernuniversiHit Hagen, 2001.

[MQR95] M. Moriconi, X. Qian, and R. A. Riemenschneider. Correct Architecture Refinement. IEEE Transactions on Software Engineering, 21(4):356-372, April 1995.

[MRT99] N. Medvidovic, D. S. Rosenblum, and R. N. Taylor. A Language Environment for Architecture-Based Software Development and Evolution. In Proc. of the Inti. Conference on Software Engineering (ICSE), May 1999.

[MSB05] B. Middha, M. Simpson, and R. Barua. MTSS: Multi Task Stack Sharing for Embedded Systems. In Proc. of the Inti. Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES), September 2005.

[Mu102] P. J. Muller. The Active Object System - Design and Multiprocessor Implementation. PhD Thesis, Diss. ETH No. 14755, ETH Zurich, 2002.

[NET06] The Microsoft .NET Framework. http://msdn.microsoft.comlnetframework,2006.

[Nage105] K. Nagel. Multi-Agent Transportation Simulation. http://www.vsp.tu-berlin.de/publications.
January 2005.

[Naur60] P. Naur (ed.), Revised Report on the Algorithmic Language ALGOL 60, Communications of the ACM, Vol. 3:299-316, 1960. Communications of the ACM, Vol. 6:1-17, 1963.

[N093] S. Nettles and J. O'Toole. Real-Time Replication Garbage Collection. In Proc. of the Conf. on Programming Language Design and Implementation (PLDI), June 1993. ACM SIGPLAN Notices, 28(6): 217-226, June 1993.

[NVP98] J. Noble, J. Vitek, and J. Potter. Flexible Alias Protection. In Proc. of the European Conference on Object-Oriented Programming (ECOOP), July 1998.

[Occ88] Imnos Ltd. Occam 2 Reference Manual. Prentice-Hall, 1988.

[OMG98] Object Management Group. The Common Object Request Broker: Architecture and Specification, Revision 2.3.492, December 1998.

[OMG04] Object Management Group. The Unified Modeling Language, UML 2.0 Superstructure Specification, http://www.uml.org, October 2004.

[OMT98] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-Based Runtime Software Evolution. In Proc. of the International Conference on Software Engineering, April 1998.

[Parnas72] D. L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. Communications of the ACM, 15(12): 1053-1057, 1972.

[ProI95] Prolog. ISO/IEC Standard 13211-1, 1995.

[RichOO] J. Richter. Garbage Collection: Automatic Memory Management in the Microsoft .NET Framework. MSDN Magazine, November 2000.

[RR02] S. K. Rajamani and J. Rehof. Conformance Checking for Models of Asynchronous Message Passing Software. In Proc. of Conf. on Computer Aided Verification (CAV), July 2002. Lecture Notes in Computer Science (LNCS), VoL 2404, Springer Verlag, 2002.

[SD+95] M. Shaw, R. DeLine, D. V. Klein et al. Abstractions for Software Architecture and Tools to Support Them. IEEE Transactions on Software Engineering, 21(4): 314-335, April 1995.

[SD+03] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: Composable Units of Behaviour. In Proc. of the European Conference on Object-Oriented Programming (ECOOP), July 2003.

[Strous98] B. Stroustrop. *C++* Programming Language, Third Edition, Addison Wesley, 1998.

[ST98] Programming Language Smalltalk, ANSJJINCITS Standard No. 319-1998, 1998.

[Szy98] C. Szyperski. Component Software, Beyond Object-Oriented Programming. Addison-Wesley, 1998.

[Reali04] P. Reali, Active Oberon Language Report, Em Zurich, http://www.bluebottle.ethz.chllanguagereport. 2004.

[US87] D. Ungar and R. B. Smith. Self: The Power of Simplicity. In Proc. of the IntI. Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), October 1987. ACM SIGPLAN Notices 22(12): 227 - 242, December 1987.

[UVEK] Schweizerische Eidgenossenschaft. Daten aus dem nationalen Verkehrsmodell. Bundesamt fUr Raumentwicklung (ARE) im Eidgenossischen Departement fiir Umwelt, Verkehr, Energie und Kommunikation. 3003 Bern.

[VC+03] R. von Behren, 1. Condit, F. Zhou et al. Capriccio: Scalable Threads for Internet Services. In Proc. of the Symp. on Architectural Support for Programming Languages and Operating System Principles (SOSP), October 2003.

[VHOO] IEEE Standard VHDL Language Reference Manual. IEEE Standard 1076, 2000.

[VG03] C. von Praun and T. R. Gross. Static Conflict Analysis for Multi-Threaded Object-Oriented Programs. In Proc. of the Conf. on Programming Language Design and Implementation (PLDI), June 2003.

[VP04] C. von Praun. Detecting Synchronization Defects in Multi-Threaded Object-Oriented Programs. PhD Thesis, Diss. ETH No. 15524, ETH Zurich, 2004.

[Wad92] P. Wadler. The Essence of Functional Programming. In Proc. of the Symp. on Principles of Programming Languages (POPL), January 1992.

[Welch04] P. H. Welch. The JCSP Home Page. University of Kent, http://www.cs.ukc.ac.ukiprojects/ofa/jcsp. 2004.

[WD94] K.-F. Wong and Benoit Dagevill. Supporting Thousands of Threads Using a Hybrid Stack Sharing Scheme. In Proc. of the ACM Symposium on Applied Computing, March 1994.

[WG89] N. Wirth and J. Gutknecht. The Oberon System. Software – Practice and Experience, 19(9):857-894, September 1989.

[WiI88] A. Williams. Deadling with the Unknown - or – Type Safety in a Dynamically Extensible Class Library. Draft, Microsoft Research, http://research.rnicorosoft.corn/comapps/docslUnknown.doc, 1988.

[Wi190] A. Williams. On Inheritance: What It Means and How to Use It. Draft, Microsoft Research, http://research.microsoft.corn/comapps/dicsllnherit.doc, 1990.

[Wirth70] N. Wirth. The Programming Language Pascal, Technical Report 1, Fachgruppe Computer-Wissenschaften, ETH, 1970. Acta Informatica, 1:35-63, 1971.

[Wirth77a] N. Wirth. Modu1a: A language for modular multiprogramming. Software Practice and Experience, 7(1):3-35, January 1977.

[Wirth77b] N. Wirth. What Can We Do About the Unnecessary Diversity of Notation for Syntactic Definitions? Communications of the ACM, 20(11):822-823, November 1977.

[Wirth82] N. Wirth. Programming in Modula-2. Springer Verlag, 1982.

[Wirth90] N. Wirth. The Oberon Language Report. http://www.oberon.ethz.ch, 1990.

[Wirth88] N. Wirth. The Programming Language Oberon. Software - Practice and Experience, 18(7):671-690, July 1988.

Curriculum Vitae

Luc Blaser

3 October 1979	Born in Luxembourg
1991 -1999	Gymnasium Kantonsschule Solothum, Eidg. Maturität Typus B (Latin)
Since 1999	Self-Employed Software Engineer, LBC Informatik, Zurich
1999 - 2004	Dip!. Informatik-Ing. ETH mit Auszeichnung, MSc. in Computer Science, ETH Zurich
2004-2007	Research and Teaching Assistant, Department of Computer Science, ETH Zurich