Universidade de Aveiro Departamento de Electrónica, Telecomunicações e Informática,
2007

Miguel Augusto Mendes Oliveira e Silva

Methodologies and Mechanisms for Concurrent Object-Oriented Programming Languages

Draft translation from portuguese (June, 2026)

Universidade de Aveiro Departamento de Electrónica, Telecomunicações e Informática,
2007

Miguel Augusto Mendes Oliveira e Silva

Methodologies and Mechanisms for Concurrent Object-Oriented Programming Languages

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Doutor em Engenharia Informática, realizada sob a orientação científica de José Alberto Rafael, Professor do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro

Dedico este trabalho à Paula, à Ana Miguel e ao João José.

o júri / the jury

presidente / president

José Joaquim Cristino Teixeira Dias
Professor Catedrático da Universidade de Aveiro (por delegação da Reitora da Universidade de Aveiro)

vogais / examiners committee

José Alberto dos Santos Rafael
Professor Associado da Universidade de Aveiro (orientador)

Pedro João Valente Dias Guerreiro
Professor Associado da Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa

Pedro Manuel Rangel Santos Henriques
Professor Associado da Escola de Engenharia da Universidade do Minho

António Manuel de Brito Ferrari de Almeida
Professor Catedrático da Universidade de Aveiro

António Rui Oliveira e Silva Borges
Professor Associado da Universidade de Aveiro

agradecimentos / acknowledgements

Os meus mais profundos agradecimentos vão, em primeiro lugar, para a minha família, por estarem sempre do meu lado. Aos meus (muitos) amigos sem os quais a vida seria uma chatice. Aos meus colegas pela ajuda que nunca me negligenciaram. Ao meu orientador pela paciência e compreensão que sempre mostrou ter pelos meus atrasos crónicos (e vergonhosos) no processo de escrita desta tese. Ao Tomás pela ajuda na revisão da tese e pelo apoio que sempre me deu. Por fim, um agradecimento muito especial ao João Rodrigues, sem o qual esta tese nunca teria chegado onde chegou. A paciência, espírito critico e interesse que sempre mostrou pelo meu trabalho foram uma ajuda insubstituível.

Resumo

Esta tese faz uma aproximação sistemática à integração de mecanismos de programação concorrente em linguagens orientadas por objectos com suporte à programação por contrato e sistema de tipos estático. Nessa integração deu-se prioridade à expressividade, segurança, abstracção e realizabilidade dos mecanismos propostos. É sustentado que essa integração deve possuir ambos os modelos de comunicação entre processadores – por mensagens e partilha de objectos – e que a sincronização seja automática e abstracta. Todos os aspectos de sincronização de objectos – intra-objecto, condicional e inter-objecto – são contemplados e integrados de uma forma segura e sinérgica com mecanismos de linguagens sequenciais orientadas por objectos. É proposta e parcialmente desenvolvida uma linguagem protótipo – denominada MP-Eiffel – onde estes mecanismos e abstracções estão a ser validados experimentalmente.

Abstract

This thesis makes a systematic approach to the integration of concurrent programming mechanisms in Design by Contract and static type system based object-oriented languages. In this integration priority was given to the expressiveness, safety, abstraction and feasibility of the proposed language mechanisms. We argue that this integration should provide both models of inter-processor communication – message passing and shared objects – and that synchronization should be automatic and abstract. All aspects of object synchronization – intra-object, conditional, and inter-object – were considered and integrated in a safe and synergic way with sequential object-oriented language mechanisms. We propose and partially develop a prototype language – named MP-Eiffel – in which these mechanisms and language abstractions are being validated.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Studied for more than 40 years in computer science, concurrent programming has, for various reasons, been largely ignored and very little used in practice since then. The main reason for this is probably due to the exponential evolution - unparalleled in any other area of engineering - of electronics and computer engineering well portrayed in Moore's well-known prediction [Moore 65] of that each year the number of transistors per circuit would double integrated[1]. So the performance of computers, and by extension of the programs that run on them, has increased at a high rate, relegating to the background (with the exception of operating systems) the possibilities of increasing performance opened up by concurrent programming.

Recently, central processing units have evolved towards parallel architectures (especially *SMP*: Symmetric MultiProcessing and *NUMA*: Non-Uniform Memory Access), which will inevitably increase interest in languages and concurrent programming methodologies.

On the other hand, object-oriented programming has established itself as one of the most important methodologies in program construction. The relative advantages we can associate with it are its suitability and flexibility in modeling different types of problems; its properties of modularity, reusability and extensibility; and, finally, its suitability for contract programming and therefore, the construction of programs with correctness and robustness.

This dissertation studies the problem of integrating mechanisms and concurrent programming abstractions into object-oriented languages. The systematic approach followed focused on four aspects:

- **Expressiveness**: concurrency mechanisms must clearly and simply cover all the desired programming abstractions;
- **Safety**: safety in the use of concurrency mechanisms must be guaranteed, as far as possible, before program execution time;
- **Abstraction**: the semantics of these mechanisms must be limited to their essential properties, avoiding excessive coupling with any practical realization;
- **Feasibility**: the mechanisms must be treatable by the compilation system.

[1]Prediction that has been verified with great approximation in practice over the last 40 years

Of the results obtained in this work, we highlight the abstract and automatic synchronization of concurrent objects, as well as the static safety and the expressiveness in integrating the majority of concurrent programming requirements in object-oriented languages with support for programming by contract.

1.1 Organization

This thesis is organized as follows.

Chapter 2 presents and discusses the problem of evaluating the quality of programming languages. In this sense, metrics and quality criteria are presented which will not only serve as a basis for choosing mechanisms, but will also serve as guides on the path to follow (or not) during the language construction process.

Chapter 3 gives a detailed presentation of sequential object-oriented languages and programming. Special emphasis is placed on the mechanisms and properties considered essential in these languages. It will be these mechanisms and properties that will dictate the restrictions and constraints to be taken into account when integrating concurrent mechanisms, since it is intended that this integration does not in any way jeopardize the qualities of object-oriented programming.

The chapter 4 analyzes the characteristics of concurrent programming, identifying the abstractions to be considered when integrating it into sequential languages.

The chapter 5 studies in detail various approaches to integrating concurrent mechanisms into object-oriented languages, taking into account the various aspects dealt with in the previous chapters: the language quality criteria in chapter 2; the essential mechanisms and properties of object-oriented languages in chapter 3; and finally the concurrent abstractions to be taken into account chapter 4. An attempt is made to identify not only the approaches that make sense to follow, but also those that should not be followed, and reasons are given, which are hopefully clear, to justify these conclusions.

Chapter 6 proposes a concurrent object-oriented language, called MP-Eiffel, in which the mechanisms discussed in the previous chapter are implemented. This language is used as a case study of concurrent object-oriented programming. It should be noted that the current implementation of the compilation system for this language is not yet complete, so full static safety is not yet guaranteed.

Some aspects considered important, related to the implementation of the MP-Eiffel compilation system, are presented in the appendix.

The conclusions of this work are presented in chapter 7, which also lists the contributions made.

At the end of this thesis (appendix E.4) there is a glossary with the definition of many of the terms and expressions used in this work.

Chapter 2

Programming Languages: Quality Criteria

Programming aims to find computable solutions to solve problems. While there are usually many computable solutions to the same problems, they differ in that they have different qualities. These qualities generally depend not only on the process of program construction — methodology — used, but also on the language (or languages) used to implement it.

In this chapter we are interested in defining criteria for quality in the evaluation and construction of programming languages that can improve the various quality factors of programs, especially those that are most important in the context of the problem to be solved. To this end, the most important program quality factors will be briefly described, after which the quality criteria of languages will be presented. Justifications will be given for the criteria presented, showing in what way they can improve program quality factors.

2.1 Evaluating program quality

Program quality factors can be divided into two groups [Meyer 88a, Ghezzi 91]: external factors and internal factors. External factors express the qualities visible to external users of programs. These include, for example, reliability, ease of use and performance. Internal factors refer to qualities that are only visible to programmers, such as modularity and readability.

It goes without saying that when it comes to the final product, only its external qualities will matter. It doesn't matter if a package of military defense software is modular and easy to understand if an error in the entry triggers a missile. Despite this observation, the key to achieving good external qualities lies precisely in the quality of the internal factors [Meyer 88a, page 4].

2.1.1 Correction

Correctness is the ability of software to perform its functions exactly as defined in its specifications.

This is by far the most important of all the quality factors. The first objective of any software product is to solve the problem it was made for. If that doesn't happen, everything else doesn't matter.

As follows from the definition, the correctness of a software product depends heavily on a sufficiently precise specification of the behavior it is intended to have. This rarely happens, and there is often only an informal specification using natural language, which encourages ambiguities and inaccuracies.

Two other problems related to the specification of programs arise from either incomplete specification (under-specification), or over-specification (over-specification) of the problem. On the one hand, an under-specified problem, even if rigorously, can give rise — taking into account the definition given — to a formally correct program that doesn't solve the problem. Over-specification, on the other hand, can exclude valid (and possibly better) solutions to the problem, for as well as negatively affecting other quality factors such as extensibility.

The art of specifying software products therefore involves avoiding under-specification without falling into the temptation of over-specification.

When building programs, from a methodological point of view, it is preferable to start from incomplete specifications - since these can be completed without the risk of an excessive impact on the remaining parts of the program - rather than from excessive specifications.

2.1.2 Robustness

Robustness is the ability of software systems to function even in abnormal situations.

The concept of robustness seems to be a little less clear from than that of correctness. What sense does it make to say that a program is robust if it works in unforeseen situations that are not part of its specification?

If these situations are part of the program's specification, then the problem would become one of correctness and not robustness. Therefore, the role of robustness is to somehow guarantee that, in the event of an anomalous situation, the program ends gracefully (without generating catastrophic events), or somehow recovers to a normal state of operation (i.e. within the program's specification).

In an ideal world, where it would be possible to develop programs that are guaranteed to be correct, there would be no place for robustness. However, programming exists in the real world where the formal demonstration of program correctness is restricted to a small number of low-complexity problems. On the other hand, program experimentation (testing at runtime) proves to have even more limitations in this respect. Quoting Dijkstra [Dijkstra 72, page 6]:

> Program testing can be used to show the presence of errors, but never to show their absence.[1]

[1] In this aspect we can draw a parallel with the physical sciences and Karl Popper's falsifiability criterion: a theory is scientific if it is falsifiable. In other words – just as in programming – a scientific theory must be testable in order to check whether it is false (the "truth" is approximated in this way by exclusion of parts).

Thus, in practice, a program is subject to programming errors and to failures that are sometimes difficult to predict and of low probability (properties that when combined can drastically reduce the quality of the software); for example, running out of free computer memory or disk space. Passing all these exceptional situations on to the normal specification of a program – foreseeing, for example, the occurrence of a lack of disk space whenever something is written in it – thus converting the robustness problem into a correctness problem, could make the problem specification much more complex, degrading other quality factors such as reliability and productivity. All these reasons justify the importance of this quality factor.

2.1.3 Reliability

> Reliability is the ability of an software system to be correct and robust.

This factor combines the two previous ones, expressing in general the degree of trust that can be had in an software product.

2.1.4 Extensibility

> Extensibility expresses the ease with which software products adapt to changing specifications.

This is another very important factor. It is very rare for an software product not to suffer during its development or after its release or marketing, various changes in its specifications, so its adaptability to these changes will be a very desirable property.

2.1.5 Reusability

> Reusability is the ability of software products to be used in part, or in their entirety, for new applications.

In addition to the obvious advantages of building programs by reusing existing components as much as possible, this factor also positively influences other factors such as the very important case of correctness[2].

2.1.6 Efficiency

> Efficiency expresses the ability to optimally use the resources of the hardware (CPU, memory, etc.).

This factor is generally associated with the speed or performance of software. Although this is generally the most important measure of efficiency, there are others that may also be important, such as memory usage.

[2]The correctness of an software product is all the more guaranteed the more it has been used in the past

2.1.7 Verifiability

> Verifiability is the ability to easily draw up procedures and test data to detect errors and faults.

There is hardly any minimally complex software product that has not had errors or flaws in its design. As such, in order to maximize its correctness as much as possible, it is important that it be developed by facilitating the development of test procedures for detecting errors. Developing software while ignoring or minimizing the possibility of errors would seriously compromise its correctness.

2.1.8 Productivity

> Productivity expresses the performance with which products of software are developed.

The most important measure of productivity is the development time of the software, although the concept of productivity can have a broader meaning, such as the use of human and logistical resources (an aspect completely outside the scope of this work).

2.1.9 Other external factors

Other external quality factors can be defined:

Compatibility: ease with which *software* products are combined with each other;

Ease of use: ease with which programs are used;

Portability: ease with which programs are transported to different execution contexts.

These factors, however, do not have the same importance for this work as those defined above.

Of course, in many situations there will have to be trade-offs between some of these factors. For example, maximizing performance (if taken to the extreme) can lead to poor portability, or even subtle correction problems.

2.1.10 Legibility

This internal factor is particularly important.

> Readability expresses how easy it is to grasp and understand the structure and code of products from software.

Programs should be constructed so that they are easy to read and understand. The readability of programs – much more than the ease of writing them [Hoare 73, page 3] – is an essential criterion for improving their correctness. However, since software is generally complex, this is a difficult quality to guarantee. Readability is approached using appropriate programming methodologies, and programming languages can make a decisive contribution to this.

2.1.11 Modularity

Another essential internal quality factor, which is even decisive for improving many of the external quality factors, is what is known as modularity.

A precise definition of modularity is not easy. Intuitively is a particular form of separation of interests, in which the problem is divided into individualized units (modules) and coherent, with value and meaning in their own right.

Meyer [Meyer 88a] proposes five criteria for evaluating modularity in program development methods:

Modular decomposition: It helps to decompose the problem into sub-problems, in such a way that solving each of these sub-problems can be done separately.

Modular composition: It favors the production of *software* units that can be freely combined with each other to generate new programs, even for problems very different from those for which were developed.

Modular understanding: Whether it facilitates the production of *software* units that are easily understandable by human observers (readable).

Modular continuity: If a small variation in the specifications of the problem results in changes in one or a few modules of the system obtained by this method.

Modular protection: A method satisfies this criterion if the effect of a situation that occurs during the execution of a module remains confined to that module, or spreads to a few neighboring modules.

2.2 Language quality criteria

Programming languages are the most important of the tools for the development of *software*, and many of the quality factors (of which the most important of them all: correctness) depend on them to a large extent. Whether or not it is easier to design and develop quality software depends primarily on the qualities of the programming languages used.

Despite this recognized importance, objective and systematic approaches to the problem of quality of languages are relatively rare, even in the presentation of languages in particular. The countless and often sterile discussions about the best languages would be much more productive if there was a concern to clarify different quality criteria.

The importance of these quality criteria is not restricted to the evaluation of existing languages, but is also essential in the design of new languages, since they allow us to guide this creative process towards improving the qualities desired.

The criteria presented here were essentially based on Hoare's classic article on this subject [Hoare 73] and on Meyer's work on language Eiffel [Meyer 92]. and in some references on type systems [Pierce 02, Bruce 02]. However, some of the criteria are the responsibility of the author, such as synergy.

Hoare [Hoare 73] considers that, in order to be useful help tools, programming languages should assist the programmer in the three most difficult aspects of programming: program design, documentation and debugging.

2.2.1 Expressiveness

In program design, the first essential challenge posed to a programming language is how easily the language expresses the mechanisms and abstractions relevant to the programming method (or methods) that the language is intended to support.

> The language should clearly and simply express all the abstractions and programming mechanisms it intends to support.

The expressiveness applied to an entire programming methodology – for example object programming – will measure the fullness with which that methodology is realized by the language.

2.2.2 Abstraction

Since the appearance of the first programming languages - directly and intimately linked to the system that supports program execution - the trend has been towards a progressive distancing from this *hardware*, and an increase in the abstraction with which solutions are expressed in languages (thus reducing the distance between methodologies and programming languages and the domain of the problems they are intended to program).

There seems to be every advantage in clearly separating the way programs are expressed and built from the way they are realized and implemented in the systems that support their execution. In other words, programs should be explicit about the behavior that is expected of them, and not necessarily how that behavior is translated into the low level languages used by computer processing units.

Of course, this aspect leaves it open as to which abstraction(s) will be suitable for expressing solutions to problems. These abstractions will depend to a large extent on the intended programming methodology[3]

> The semantics of the language should be expressed relatively to the important aspects of its mechanisms, and not to the details of possible realizations.

2.2.3 Comprehensibility

Program documentation is one of the aspects that tends to be least considered in programming languages – apart from support for the use of comments – leading to the respective programs being difficult to understand, debug and modify. Hoare argues that documentation should be seen as an integral part not only of the program development process, but also of the program itself.

> The language should encourage and facilitate the writing of readable and self-documenting programs.

[3]Actually four major methodologies can be identified: structured procedural programming, object-based programming, functional programming and logic programming.

Although ease of writing and ease of reading a program are not two antagonistic objectives (quite the opposite), it is important to reinforce the fact that the latter is much more important than the former. If such a choice ever had to be made, in general it would be far preferable to have more laborious program writing if such a choice resulted in easier program comprehension.

2.2.4 Security

The last aspect mentioned by Hoare – the debugging of programs – will probably be the one that most forces to make radical choices when building languages.

In program development, debugging tends to be the most time-consuming, difficult and least motivating phase for programmers. However, what comes out of it directly affects the most important quality criterion of all: correctness; which is why all the help that the language can provide in this regard becomes extremely important. Such help can be found in basically two areas: error detection and error localization.

The subset of errors that programming languages are most obliged to detect, are those related to incorrect use of their own mechanisms and their respective abstractions.

In this sense, Hoare proposes the safety criterion.

> A language is said to be safe if its mechanisms and abstractions do not produce meaningless results.

Pierce [Pierce 02, page 6] presents another interesting definition of security:

> A language is said to be safe if it protects its own abstractions.

Thus, a mechanism is secure if its use in a program is only accepted if there is a guarantee from that no nonsensical results will arise from its use.

Security can be guaranteed before the programs are executed (at compile time or statically), or tested while they are executed (at runtime or dynamically). Obviously, as far as this criterion is concerned, the first option is far preferable, since (discounting possible implementation errors of the compilation systems of the languages) it is the only one that guarantees the non-existence of certain errors - such as the important case of errors of types - during the execution time of the programs.

The most important language design option in terms of ensuring language safety has to do with the so-called type system of the language[4].

2.2.5 Synergy

One aspect of language quality that is not often mentioned (but is certainly easily recognized), is not only the degree of integration and cohesion of the language's various mechanisms with each other, but also when this fact results in added value with the appearance of new functionalities, emerging from the joint use of these mechanisms. In other words, this property assesses the possibility of the functionalities of the set of certain mechanisms being more than the individual sum of the functionalities of the mechanisms involved. We will call this criterion synergy.

[4]Described in section 3.1

Whenever possible, the mechanisms and abstractions of the languages should be constructed in such a way that, when used together, they generate new functionalities as long as these are consistent with their individual semantics.

An example of synergy is the recursion of routines in imperative languages. The functionality of recursion emerges because of the way the mechanisms for invoking routines and storing (on a stack) the values of arguments and variables local to the routine are implemented. Of course – as this example clearly demonstrates – the synergistic effects of mechanisms are rarely casual, but rather the result of careful design of these language mechanisms.

2.2.6 Orthogonality

Taking this perspective of analyzing the properties resulting from the joint use of mechanisms further, we find that when the whole has a value (in terms of functionalities) lower than the sum of the parts (each of the mechanisms seen in isolation), we are certainly in the presence of security problems; when this value is higher than the sum of the parts, we have synergistic qualities; and when it is the same, we are in the presence of independent or orthogonal mechanisms.

So, since safety must always be guaranteed, we have only two options when it comes to the joint operation of mechanisms: or they must be synergistic or orthogonal.

Language mechanisms and abstractions are orthogonal, if they work independently.

A notable example of orthogonality is the design of the structured procedural instructions (some of which can be seen in figure 3.1). Thus, within a conditional or repetitive instruction any other instruction can be used, enhancing in a simple way, the development of any (computable) algorithm.

A particular case where orthogonality can be very important is the situation - as happens in the prototype language developed as part of this work - in where you want to extend an existing language with new mechanisms for new functionalities. In this situation, it is desirable for the new mechanisms to be as orthogonal as possible to the base language, so that not only is the "new" language more consistent and understandable, but also existing modules can be reused as much as possible.

2.2.7 Other criteria

The quality criteria already presented will be the most important when evaluating languages. However, there are other criteria that should also be taken into account.

Feasibility: A programming language mechanism is feasable if there is at least one implementation, computable in the build system, that allows generation of the appropriate executable code in the system supporting program execution.

The feasibility of a language is a criterion to be taken into account especially in the language design phase.

Program efficiency: The language must enable the respective compilation (or, if applicable, interpretation) system to generate efficient programs[5].

Despite the dizzying - one might even say incomparable! — increase in the processing (and storage) capacity of the hardware systems that support program execution, efficiency will always be an objective that should not be overlooked in software engineering and particularly in the design and realization of languages. No matter how fast the execution system of a program is, it will be put to better use the more efficient the programs are.

There is another aspect of efficiency that applies to programming languages: compilation efficiency. Currently, and as long as the language is workable, this aspect is not very important, since even poorly optimized compilation systems tend to have a relatively low (and generally affordable) real execution time.

Language extensibility: Extensibility of programming languages expresses the ease with which new mechanisms can be added to them.

Programming languages, while by no means as volatile as their programs, tend to be modified over the course of their lifetime, mainly by adding new mechanisms. Obviously, extensibility in languages depends essentially on the simplicity of the base language, but the structure and semantics of the mechanisms to be added to it, is also decisive. In any case, the orthogonality of pre-existing mechanisms and those to be added will be the way to maximize this criterion.

Meyer [Meyer 92, Appendix B] presents two more criteria to take into account.

Uniqueness: Programming languages should provide one good way to express each operation of interest; should avoid providing two.

Consistency: Programming languages should be based on a small set of fundamental ideas and complete, and then realize them consistently to the last consequences.

[5]This quality factor is defined on the page 5

Chapter 3

Sequential Object-Oriented Programming and Languages

This chapter aims to achieve three objectives:

- to present sequential object-oriented programming;
- to list the properties and language mechanisms that support it;
- to analyze the interdependencies and possible interferences between these mechanisms.

Since object-based programming cannot be dissociated from the programming paradigm that preceded it[1] – the structured procedural programming – we will first give a presentation of this paradigm. We'll see that there are some properties of structured procedural programming that are maintained in object programming, and that should be taken into account with regard to possible synergies and interferences between mechanisms.

Different languages tend to use different terminologies for the same concepts and mechanisms, so this chapter will continue to establish the terms and definitions used in this thesis (the most important ones and those that lend themselves to confusion have also been included in the glossary).

3.1 Type systems

As briefly mentioned in the previous chapter, one of the most important language construction options to maximize security is based on the type system.

In languages, "types" describe the shape and properties of elements in a program that can be associated with values (in the case of pure object-oriented languages (page 19) these values are reduced to objects). The type system, in turn, not only associates – explicitly or implicitly – the types with all the relevant software elements, but also checks (to the best of its ability) that these are used correctly.

[1] Both are imperative.

In this paper we will refer to **entities with type**, as the syntactic elements of a language that are associated with a "type" (i.e. in object-oriented languages, these entities can contain objects or references to objects). Depending on the language, there may be different entities with a type, such as: local variables, attributes of classes, functions, formal arguments of routines, etc...

Type systems can be static[2], dynamic or mixed – depending on whether type checking is done, respectively, at compile time, at run time or both.

Type systems serve different purposes [Bruce 02, page 7] [Pierce 02, pages 4–8]:

- **Security**: a type system prevents the occurrence, at compile time or at run time, of a important set of incorrect uses of entities with type, such as the application of non-existent operations. This improves language security and the correctness of programs.

- **Abstraction**: the use of types to annotate the entities that manipulate values makes it possible to separate the use and implementation of values, which substantially improves the modularity of the software.

- **Documentation**: types, when expressed explicitly, also serve to make the intentions of the programmer clear, which can greatly improve the understandability of the language and the software.

- **Optimization**: type checking can provide the compilation system or the language interpreter with useful information for generating more efficient code.

Static type systems, compared to the dynamic type systems, improve all these aspects. Security is substantially improved since type errors are detected earlier, at compile time. The abstraction and documentation associated with types, being statically defined, makes the purpose of each type much clearer without having to analyze its dynamic behavior. Finally, the information provided by static type systems to the compiler opens up the possibility of substantially improving the efficiency of programs, not only by avoiding testing types at runtime, but also by using aggressive optimization techniques (such as replacing an invocation of a routine with its code).

However, static systems can also have some disadvantages. The most important of these are:

- **Tractability**: for the type system to be able to do its job at compile time, it is necessary for it to be realizable, i.e. for its complexity not to increase exponentially with the size of the programs. Therefore, it doesn't seem to be generally possible to have static type systems that guarantee the total correctness of the software. Generally, type systems are limited to checking that the values conform to the type of the elements of software that manipulate them[3]. For this reason, these systems tend to be conservative, being able to reject programs that, at runtime, would never have unsafe behavior.

[2] static

[3] This feature is important when choosing and comparing different approaches to subtype polymorphism as will be seen below (page 22)

- **Flexibility**: the imposition that entities in a program can only contain values that respect their type – and if the type system is limited and unimpressive – can be a substantial obstacle to reuse and software productivity.

The biggest problem with static type systems is the need for them to depend heavily on the form (syntactic) of the values, rather than their complete essential behavior (semantic)[4]

In order to substantially reduce the flexibility problems of static systems, we will look at two essential ways of making these systems more expressive: subtype polymorphism (section 3.8) and parametric polymorphism (section 3.10).

In this work, the choice of languages with static type systems was a basic choice, and it has been shown to be an essential choice for the results obtained. However, it is important not to lose sight that static type systems are not a guarantee of correctness, but only an approximation in this direction.

3.2 Structured procedural programming

Procedural programming starts from the basic idea of expressing solutions to problems as sequences of actions (commands) to be executed. In a correct program, as the actions are executed, the state of the system tends towards the solution of the problem (this solution can be explicitly expressed in program variables, or implicitly recorded in the command execution path that the program follows).

With this method, the programming problem "reduces" – in addition to an adequate (and sufficient) specification of variables for explicit storage of information of the program – to the "up-down" decomposition of the algorithm of the initial procedure, into a sequence of simpler actions[5] – which may themselves be new procedures, capable of a new decomposition – involving when necessary instructions for assigning values to variables, conditional instructions[6] and instructions repetitive[7]. This decomposition process is applied hierarchically to each action resulting from the previous decomposition, until the resulting algorithm is completely expressed as a function of pre-existing actions [Wirth 71, Wirth 74].

An important feature of this approach - shared by object-oriented programming - is its imperative nature. The expression of an algorithm is made up of a sequence of commands that can explicitly modify the state of the system (i.e. the execution of commands can have side effects on the program as a result of modifying the value of variables).

Another essential aspect of this approach is the use of so-called algorithmic abstraction. This type of abstraction consists of encapsulating algorithms within procedures (actions) or functions (calculation of values)[8], thus separating the use – generally simple

[4]We'll see (page 23) that the language EIFFEL has a type system that allows, albeit in a limited way, the semantics of the types to be part of them.

[5]Decomposition by "concatenation" according to Dijkstra [Dijkstra 72, page 19].

[6]Decomposition by "selection" according to Dijkstra [Dijkstra 72, page 19].

[7]These algorithmic elements are sufficient to express any computable algorithm [Bohm 66].

[8]There are languages, such as C that do not distinguish procedures from functions in a syntactically explicit way, although – even in this case – it can be considered that functions of the type **void** correspond to

```
if CONDITION then
  COMMANDS
end
```

```
while CONDITION do
  COMMANDS
end
```

```
repeat
  COMMANDS
until CONDITION
```

Figure 3.1: Structured conditional and repetitive instructions.

and easily understandable – from the implementation of that algorithm. In this way, the reuse of algorithms and the comprehensibility of programs can be substantially improved.

Understanding programs will be made easier the closer their static structure is to their dynamic (i.e. run-time) behavior [Dijkstra 68c]. One approach would be to make language instructions have only one entry point and one exit point [Dijkstra 72, pages 16–23] [Wirth 74]. In this way they can easily be isolated and interpreted as being a single action in a sequential computation. This property of structured procedural programming is very important as it facilitates the analysis and understanding of "up-down" algorithms. Thus, the properties (which can be expressed by axioms about the state of the program) of each instruction are defined from "outside-in", and not the other way around. This is the case with conditional and repetitive instructions structured, whose behavior is imposed by the externally visible structure of the instructions themselves (figure 3.1).

The `COMMANDS` commands – whatever they are – will only be executed if they are selected by the conditional or repetitive instructions in which they are inserted. We'll call language statements that fulfill this property pure structured statements.

This property makes it easy to associate any sequential action A with two assertions[9] – P and R – attesting to their correctness [Hoare 69]:

$$\{P\}\ A\ \{R\}$$

This formula, known as Hoare Triple, can be expressed as follows: if the precondition P is true at the beginning of the execution of the action A, then the postcondition R will be true at its end[10]

This axiomatic approach to program correctness – due mainly to Floyd [Floyd 67] and Hoare [Hoare 69] – will be one of the most important contributions of structured procedural programming (having been adapted and extended in object-oriented programming with contract programming).

An almost immediate consequence of this approach to algorithm construction is the inappropriateness of using goto instructions. In general, the use of "gotos" makes it more difficult to relate the dynamic behavior of a program to its static textual structure. This instruction can hide essential algorithmic structures such as repetitive structures

procedures.

[9] Predicates

[10] Hoare presents this formula with the curly braces surrounding the action instead of the assertions: $P\ \{A\}\ R$. These two formalisms differ only in the detail that in Hoare's original notation the post-condition only applies if the action ends (partial correction) while the notation used presupposes and imposes the termination (in finite time) of the action [Gries 81, page 109]. For the purposes of this work, however, this difference does not seem to be at all relevant.

```
     i = 1;             // (1)
l1:  printf("%d\n",i); // (2)
     i++;               // (3)
     if (i <= 10)      // (4)
       goto l1;         // (5)
```

Figure 3.2: Example of an algorithm with gotos in C.

or conditionals very far from their actual occurrence, which can make the algorithm very difficult to understand[11] (for this reason it is customary to refer to the use of "gotos" in programs as "spaghetti" code). In other words, the construction of algorithms with "gotos", unlike pure structured instructions, can force you to understand the algorithm from "inside-out".

Figure 3.2 exemplifies the implementation of a repetitive algorithm using a "goto" instruction. Thus it is only at `(5)` that the programmer can realize that he is dealing with a repetitive algorithm started at `(2)`. Although the "goto" instructions can be used in a disciplined way (Knuth being the great advocate of this regulated use [Knuth 74]), such an option means that there is no guarantee at compilation time that the algorithmic structure is simple, and the guarantee of the exclusive use of pure structured instructions is lost.

3.2.1 Limitations

Structured procedural programming begins to show its limitations as the complexity of the problem to be solved increases. In fact, for problems with a certain amount of complexity, it doesn't make much sense to assign importance to a single top procedure. You can easily define several top procedures – probably with quite different "top-down" decompositions – for the same problem to be solved, which may depend, for example, on the type of interaction between the user and the program (graphical interface, text console, etc.). Making the algorithmic decomposition dependent on this choice is clearly a mistake and over-specification.

Another more critical problem lies in the fact that this approach has poor modularity. In general, procedures and functions are not self-sufficient, needing to be associated with appropriate data structures. For example, a function that indicates whether any date (defined by day, month and year) is valid is closely linked to the data structure that represents dates (which could be made up of three integer values, by a structure with three integer fields, or any other representation). Any change to the data structure will most likely involve changing the procedures and functions that depend on it.

Of the five modularity criteria presented (page 7), three are directly called into question with this approach:

- *Modular composition*: each module must be linked to the data types it uses (which, in turn, may have a high degree of cohesion with other modules).

[11] As in all rules, there are however some exceptions. In languages without exception mechanisms, the use of "bounces" can be justified to deal with exceptional situations so as not to "pollute" the normal code and to simplify programs.

- *Modular understanding*: understanding each module also involves understanding the data types associated with it, when it does not also involve understanding other functions (modules).
- *Modular protection*, there is a great deal of cohesion with types of data external to the module.

These shortcomings of modularity in the methodology of structured procedural programming are addressed and resolved in object-oriented programming.

3.3 Object-oriented programming

The name "object-oriented" has been used and abused ever since it was given the same status of quality that once belonged to structured (procedural) programming. In reality, different schools of programming - generally closely linked to different languages - have a different perception of what this type of programming is. In fact, the author of this thesis is not completely immune to this problem either, as he advocates an approach to object-oriented programming in particular based on many of the principles that underlie the method and language EIFFEL. Despite this possible limitation, we will try to present not only the properties that are almost consensually attributed to object-oriented languages, but also other properties and mechanisms considered important.

Although the concepts of programming by objects (methodology) should be separated from the programming languages in which the programs are expressed, this section will mix these two worlds a little. This choice (which is certainly debatable) is justified by the author by the fact that this work essentially focuses on programming languages, and in particular from the perspective that these can make a decisive contribution to software correctness. The reinforcement of the correctness of programs depends heavily on the programming methodology followed, so it can be considered that both worlds come together for the same purpose. This approach is once again influenced by the language EIFFEL which is presented by its author as not only a language but also a programming method.

First, we'll look at the six mechanisms and essential properties that we believe are the minimum to define both languages and object programming itself. We will then discuss other mechanisms frequently used in object-based languages, many of them desirable for the positive impact they can have on the quality of both programs and languages; others undesirable for the opposite reason.

3.4 Object: data structure + methods

A first approach to object-oriented programming is the result of two (complementary) findings drawn from the analysis of procedural programming and from which the same result is drawn. The first observation is that many methods (functions and procedures) tend to be closely and strongly linked to certain data structures. A change in the data structure often implies a partial or even total modification of the methods that depend on it directly. On the other hand, if we look at the problem from the side, data

structures in themselves are passive entities whose behavior (semantics) is, to a large extent, externally imposed on them precisely by the methods that directly manipulate them. For example, a data structure with three integer fields, can be used to represent a date (day, month and year) as well as a clock (hours, minutes and seconds) or any other "thing" involving three integer values. However, the behavior in each of these possibilities will be quite different (and incompatible with each other). It won't make much sense to assign the value `15` to a month, nor `2006` to the seconds of a clock.

Therefore, there seems to be an advantage, both from the perspective of methods and from that of data structures, in combining both into a single entity. This entity, in programming by objects, is called **object**.

The elements of the data structures that define and allow to store the state of the object are usually referred to as **attributes** (their behavior within each object, is similar to that of variables in procedural languages). These attributes can be variables or constants. We'll also indistinctly call the set of methods (which we'll also call routines) and attributes applicable to objects **services** (features in the terminology used in the language Eiffel). Thus, an object is made up of a set of services, which can be attributes or methods. When justified, you can also divide methods into functions and procedures. Functions are algorithmic abstractions for observing or querying the state of the object. Procedures are algorithmic abstractions of commands that are applied to the object in order to modify its state. Functions that have no side effects on the observable state of the object, nor on the observable state of any other object in the program, are called "pure". Another very useful classification of object services is the separation between commands (commands) and queries (queries). The commands of an object will be its procedures, while the queries will be its attributes and functions (which should preferably be pure).

Unlike routines and data structures – which need and depend on each other – objects are self-sufficient for building programs. It is therefore possible to define object-oriented programming languages in which the entire program is exclusively built using objects. These languages are called pure object-oriented languages.

3.5 Objects and classes

There are basically two linguistic approaches to the construction and instantiation of objects. In the first, the behavior of objects is defined separately in syntactic entities called classes[12], and each object is created as an instance of a class. In this approach classes are also the basis for defining the types of objects. The second approach is based on prototypes [Borning 86, Lieberman 86, Ungar 91]. An object is created directly from a description of the set of methods and attributes desired, or by cloning and adapting another object (prototype).

The vast majority of object-oriented languages follow the first approach: Simula [Dahl 68], Smalltalk [Goldberg 89], Eiffel [Meyer 92], C++ [Stroustrup 97], Java [Gosling 05], CLOS [Bobrow 88], BETA[13] [Madsen 93]. There is, however, a (small)

[12] Although classes are syntactic entities, there are languages, such as Smalltalk that allow them to be modified at runtime

[13] This language also allows objects to be created without classes.

group of languages based on prototypes: SELF [Ungar 87], CECIL [Chambers 04].

This work focuses only on class-based object-oriented languages.

3.6 Information hiding

Information hiding (due to David Parnas [Parnas 72b, Parnas 72a]) in objects is the possibility for them to hide a subset of their services from their external users.

Although this possibility does not exist in the language considered to be the source of object-oriented programming – the SIMULA language –, and in the SMALLTALK language (where the designation "object-oriented" first appeared) information hiding is predefined by the language (attributes are always private and methods public); there is little doubt today about the essential importance of information hiding for object-oriented programming.

Information hiding meets three of the modularity criteria presented previously (page 7):

- *Modular understanding*: an object can be understood (and used) only by taking into account the subset of methods (we'll discuss the problem of public attributes later) considered essential.
- *Modular continuity*: the methods and attributes that are not visible from the outside can be freely removed or modified without running the risk of directly affecting the object's clients.
- *Modular protection*: there is the possibility that the objects can be the only ones responsible for controlling the correctness of their internal state, preventing the occurrence of incorrect uses (for example, setting the day `32` in a `DATA` object).

From the discussion, particularly with regard to modular protection, we can conclude that a object should not have attributes that can be directly modified by clients (public in the terminology of the C++ and JAVA languages). In this situation, not only can the object no longer control its own correctness, but it also directly links its interface to a particular choice of representation of its state (overspecification).

Encapsulating information has a direct effect on the following quality factors: correctness, extensibility, reusability, verifiability and understandability.

3.7 Inheritance

Another mechanism considered essential to programming by objects (based on classes) is the so-called "inheritance". This mechanism makes it possible to build new classes from pre-existing ones, reusing and eventually redefining methods and attributes.

When a class inherits from another (ascending class or super-class), it automatically has all of its methods and attributes, and can redefine some of these if necessary or just convenient. In this way, inheritance promotes a style of programming by difference – making it possible to build new classes at the expense of pre-existing ones – thus minimizing code redundancy and increasing the possibilities of reuse.

When a class `A` inherits from another class `B`, `A` is said to be a **subclass** or descendant class of `B`. Meyer [Meyer 97, page 464] generalizes the definition by making a descendant of a class either the class itself or any of its direct or indirect heirs.

3.7.1 Information hiding

An important aspect – and one for which you can find different approaches in different languages – has to do with the interference between inheritance and information hiding.

On the one hand, the question arises as to whether or not there should be information hiding in relation to the subclass. Some languages (for example: C++ and JAVA with private services) allow this behavior. Others (EIFFEL) do not.

On the other hand, there is also the problem of to what extent subclasses can redefine information hiding existing in the ascending class (or ascending classes, in the case of multiple inheritance). Here too, the approach differs depending on the languages considered. In C++ and JAVA a subclass can only maintain or further restrict information hiding in the parent class. In EIFFEL, on the other hand, there is complete orthogonality between the two mechanisms. Meyer [Meyer 97, page 57] supports this option, using the so-called "Open-Closed" principle of modularity.

> A module must be both open and closed.

This principle argues that a module should be open to being modified and adapted to new situations and needs, and closed so that it can be used safely by clients. The "trick" to reconciling this apparent paradox is based precisely on the inheritance mechanism (a module should be open to being appropriately modified in subclasses) and the orthogonality of this (also) with regard to information hiding[14].

3.8 Subtype polymorphism and dynamic binding (simple)

An object of type `T` is said to be a **subtype** (according to in the language terminology EIFFEL) of a type `U` (`T <: U`) if an object of type `T` can be used in all contexts where objects of type `U` are expected.

This possibility of associating an entity of type `U` with an object of a subtype is called **inclusion or subtype polymorphism** [Cardelli 85].

In order to be able to associate a `target` entity of type `T` with a `obj` object of any subtype `U`, it is necessary that the invocation of any service through `target` selects the appropriate service of `obj` in `U`. If the same `target` entity can be associated at program runtime with objects of different types, then this selection will have to be made dynamically, depending on the type of the object with which `target` is associated. In object-oriented languages, this choice is made by the object itself through a mechanism called by **simple dynamic binding** (in the literature there are various names for the same mechanism, such as: dynamic binding, or simple dispatch).

[14] However, this freedom can raise some problems, such as the guarantee of substitutability, as will be seen below.

This essential feature of object-oriented languages, in which the service to be executed is selected by the object itself, justifies the use for invoking object services, of the alternative (but equivalent) name for sending messages used above all in languages of the SMALLTALK family.

The mechanisms of subtype polymorphism and dynamic routing make it possible to tremendously increase the flexibility of the static type system[15], without committing to it[16].

Although it may make some sense to talk about subtypes in dynamic type systems – since in these you can usually try to pass one object for another, with substitutability being checked dynamically message by message, and not for the full type of the object – it is in static type systems that this relationship is most important, and where the most difficult challenge is also posed of how to express the subtype relationship in a safe way.

3.8.1 Dynamic choice of routines *versus* dynamic choice of objects

It's interesting to compare this object-oriented approach – in which it's the object itself that dynamically determines the service to be executed – with the procedural (and also functional) approach, in which it's the routine that dynamically determines (with a multiple selection instruction) which type of object it's being applied to. Although the two approaches are dual, the choice between the two is generally not indifferent. Data structures tend to be much more stable than routines, so adding new services to classes tends to have less effect on the modularity of the program than adding new data types to functions (being dual approaches, we are comparing the also dual extensions in both approaches). On the other hand – thanks to inheritance – classes don't need to implement (or even often even know about) all their services (programming by difference shows its power here). The procedural approach, on the other hand, doesn't have an inheritance mechanism applicable to routines similar to that of object-oriented languages, so it forces all those routines to know the data types to which they are applied.

We can see that the two approaches have a very different impact on the continuity modularity criterion (page 7).

The so-called conventional implementations of data types abstractos[17] – existing, for example, in the `packages` of the language ADA [Ada95 95], in the modules of MODULA-2 [Wirth 85] and in the clusters of the language CLU [Liskov 77] – also follow the procedural approach presented, burdening the routines with the internal choice of which data type (the representation of the abstract data type) is being applied to.

3.8.2 Nominal and structural subtype relationships

As far as subtype relationships are concerned, inheritance is not the only way to express them. In fact, there are two distinct ways of expressing this relationship in pro-

[15]Later on in the section 3.10 another polymorphism mechanism will be presented – called parametric – which further increases the guarantees of correctness at compile time of the type system.

[16]The problems related to inheritance will be analyzed later on (page 23).

[17]The abstract data types are presented later (page 25).

gramming languages: either explicitly (nominally), or implicitly (structurally). In the former – which is by far the most common in object-oriented languages (EIFFEL, JAVA, C++) – the subtype relationship is expressed explicitly through an appropriate language mechanism, usually the inheritance mechanisms[18] (subclass). In the second form (existing for example in EMERALD [NC 87]) – more often in programming languages more oriented towards functional programming – the subtype relationship is implicit and guaranteed whenever the subtype shares (at least) the same structure (names and signatures) as the supertype (this property is called: structural equivalence).

Both approaches have advantages and disadvantages. The structural approach has the advantage that it can be easily extended with supertypes, without affecting the respective subtypes in the slightest. In this way it is easier to redefine the program's subtype graph, without having to mess with the existing types. Another advantage of this approach is the possibility (well documented in the literature [Cardelli 85, Pierce 02, Bruce 02]) of implementing safe and tractable static type systems, where the (structural) safety of subtypes is guaranteed. However, this approach has two major drawbacks. The first is the fact that the subtype relationship between types is (by definition) implicit and casual, and does not result from a choice explicitly made by the programmer. Thus, a subtype can easily cease to be a subtype, or *vice-versa*, just by changing the form of its services. The second, and undoubtedly the most important, has to do with the meaning and usefulness of types in program design. Using the definition presented above (page 13), types describe the form and properties of entities that can be associated with values in a program. However, a structural equivalence approach to subtypes drastically restricts the possibility of associating and imposing properties on the subtype relationship, beyond the obvious ones that only have to do with the formal structure of the types (names and signatures of the services).

With structural equivalence, it is perfectly possible for a type corresponding to a `STACK` to be substitutable for another corresponding to a `QUEUE`, if they both share the same structure (which is often the case), although – of course – these types are not at all substitutable, since they have different and incompatible behavior. Approaching subtype polymorphism using inheritance has the advantage in this respect of guaranteeing that only objects that are explicit descendants of a given type are replaceable. In the case of the language EIFFEL, this advantage is even greater as the semantic properties of classes are obligatorily inherited in descendant classes (section 3.12).

3.8.3 Security

Despite these advantages, the direct relationship of subtypes with inheritance can lead to static security problems in the type system. This is what can happen when is allowed to change the external visibility of services (so that, for example, a public service in the parent class becomes private in the descendant class); or when is allowed to redefine covariant[19] of entities with a type that can be the target of value assignments (left-values in the terminology of the language C) [Bruce 02].

This problem – although it can pose serious security problems – is outside the scope

[18] In JAVA, in addition to classes, this relationship can also be expressed by interfaces.

[19] i.e. in the same direction as the inheritance relationship.

of this work. The EIFFEL language has this problem, and there are several proposals to solve it, either by forcing the type system to perform a global analysis of the programs (system validation [Meyer 97, page 633]); prohibiting the existence of polymorphic catcalls[20] (i.e. prohibiting the use of subtype polymorphism on covariant services); or even separating the inheritance and subtype mechanisms[21] [Cardelli 88, Cook 90, Bruce 93].

Another alternative that we think could be valid is to add to the language a mechanism for multiple dynamic routing oriented by objects[22].

3.8.4 Subclass *versus* subtype

In this work we will not only assume the explicit subtype relation, but we will also consider that a subclass relation (inheritance) implies a subtype relation[23].

How acceptable is this approach? Several references are well known in the literature which are very critical of this link [Cook 90][Bruce 02, pages 24-26], essentially as a result of the security problems already mentioned.

Inheritance is – by definition – a reuse mechanism. In other words, a class inheriting from another (or others) should be absolutely equivalent to another class that directly implements the services of that parent class. On the other hand – since the class is reusing the services of the parent class – in the general case it will have all the possibilities of fulfilling the same contract (i.e. of respecting the same ADT[24]) as that parent class. Exceptionally – for the reasons already given – this may not happen, but full compliance is the rule. In other words: as a rule, a subclass relation has all the conditions to be considered a subtype relation.

Why then impose a separation between the two mechanisms, when – for most cases – this will force the duplicate use of both in parent-child class relationships?

It may be defensible to have a separate mechanism for these exceptional cases (such as the recent proposal for non-compliant inheritance for the EIFFEL language), but it would be a huge mistake to make all subclass relationships not also be subtype relationships (in JAVA, although they only have simple inheritance, subclass relationships also imply a subtype relationship).

3.9 Objects and abstract data types

The characteristics considered essential in object-oriented languages are the previous five: objects, classes, information hiding, inheritance and subtypes. However, there is still a lack of theoretical support that allows objects to be described in an appropriate way, and that not only includes all these mechanisms but also gives them coherence, consistency and meaning. This is the role of *abstract data types*.

[20] Change Availability of Type calls

[21] Option which seems to us to be going in the wrong direction.

[22] Which, in our opinion, excludes the multi-method approach of the CLOS language.

[23] This is the choice made in the language EIFFEL, although recently the inclusion of a subclass mechanism that does not imply subtype [ECMA-367 05, page 16] is being considered.

[24] Abstract Data Type (page 25)

Liskov and Zilles [Liskov 74] originally defined an Abstract Data Type (ADT) as being:

> A class of abstract objects that are completely characterized by the operations on those objects.

However, this definition is not completely satisfactory. If an ADT is seen as being defined only by the names and signatures of the operations that apply to it, then – as with the structural approximation to the aforementioned subtype relation (page 22) – one can easily have the same ADT for different and incompatible abstractions (irreplaceable) [Guttag 77]. For example, an ADT for a "stack" ($STACK$) can be formally defined as follows (adapted from [Meyer 97, page 139]).

TYPES
$STACK[T]$

FUNCTIONS
$new : \rightarrow STACK[T]$
$put : T \times STACK[T] \rightarrow STACK[T]$
$remove : STACK[T] \nrightarrow STACK[T]$
$top : STACK[T] \nrightarrow T$
$empty : STACK[T] \rightarrow BOOLEAN$

This same structure can be applied without modification (apart, of course, from the type name) to "files" ($QUEUE$), although, of course, in no case are objects that implement these ADTs substitutable for each other[25].

A more appropriate and complete definition of ADT – where the (axiomatically defined) semantic of ADT is explicitly included – is presented by Guttag [Guttag 77] and Meyer [Meyer 88b, Meyer 97].

Abstract Data Type (ADT)

A class of abstract objects that are completely characterized by the operations that exist on those objects and by their semantics.

ADTs provide solid formal support for describing objects and their classes.

Class

A class is a possibly partial implementation of an abstract data type [Meyer 97, page 142].

[25] As we saw above (page 22) this is one of the criticisms that can be made of type systems that define substitutability only by structural equivalence.

ADTs also provide support for information hiding, allowing a suitable choice to be made as to which services of each class should or should not be public [Meyer 97, page 144].

The semantics of ADTs must be expressed axiomatically by associating three types of assertions with the class: invariants, preconditions and postconditions. Invariants are axioms that must always be verified in any interaction with the instances of the class (i.e. when any of its services is used externally). The preconditions and postconditions are defined for each service of the class, and are applied, respectively, when that service is invoked and when its execution ends.

Thus, for any service `S` belonging to a class with the invariant `INV`, the following correctness condition applies [Meyer 97, pages 368–370]:

$$\{INV \textit{ and } PRE_S\}\ ROUTINE - BODY_S\ \{INV \textit{ and } POST_S\}$$

In other words, the execution of any service is correct (in relation to the assertions expressed) if, immediately before the start of its execution, the invariant of the class to which it belongs and the precondition of that service are true; and if the same happens to the invariant and the postcondition immediately after that execution. We can thus see that the axiomatic support for the correctness of services is based on the application of Hoare's suit (page 16) to the services of the class.

Although the semantics of ADTs should always apply to the classes that implement them, whatever the object-based language used, it is extremely desirable that the language itself supports the expression of this semantics, since this approach has a very strong impact on the correctness of programs (in addition to positively affecting their robustness, readability and verifiability). Unfortunately, few languages – among which stands out the Eiffel language – offer this support. In section 3.12 we will present the methodology of programming by contract which is based precisely on this support.

It's important to note that although, with the exception of the Eiffel language, none of the best-known object-oriented languages have basic mechanisms for expressing these assertions in classes, this doesn't mean that they should be seen as implementations (possibly partial) of ADTs. Although this perspective is not yet assumed explicitly by everyone in object programming, in the author's opinion this will be more or less inevitable, given the overwhelming advantages that result from it.

This work will (explicitly) assume this view of object programming, with being one of the aspects where the greatest care has been taken to safely integrate concurrency mechanisms into object-oriented languages.

Meyer's definition for object-oriented programming [Meyer 97, page 147] will therefore be considered:

Object-Oriented Programming

Object-oriented programming is the construction of software systems as structured collections of implementations, possibly partial, of abstract data types.

One last note. Instead of using an axiomatic definition, the semantics of ADTs

can be expressed in a way operational[26]. However, this approach has several problems [Guttag 77]. Not only does it easily generate over-specifications, but it also makes it difficult to understand ADTs, reducing their usefulness. Another very important aspect to bear in mind is the interest in making semantics part of ADTs, and – as far as possible – of their implementations (an aspect dealt with in section 3.12).

The mechanisms considered essential (and minimal) to exist in object-oriented languages are those presented in these last six sections. We will now present other mechanisms that are frequent options in many object-oriented languages. Most of these are well integrated into object programming and make an important contribution to improving its quality.

3.10 Parametrization of types: parametric polymorphism

A very useful mechanism is the possibility of specifying classes according to generic types (without the over-specification of having to choose just one type in the implementation of those classes). For example, the ADT of a stack does not depend in any way on the type of elements that can make it up. Therefore, it makes sense to build the `STACK` class parameterized with regard to the type of the elements, so that you can create different types of stacks, such as a stack of integer numbers or of dates, without having to create a "new" `STACK` class for each of these types of elements. What's more, it's also desirable to be able to know, for each stack, which particular type is shared[27] by all its elements, so that these can be used to take advantage of their ADTs. This mechanism is called parametric polymorphism [Cardelli 85] (the first definition and classification, albeit incomplete, of the various types of polymorphism, including parametric polymorphism, is due to Strachey in 1967 [Strachey 00]).

This mechanism is relevant in languages with static type systems. In languages with dynamic type systems, there is much more flexibility in the mixing and substitutability of objects, so that class parameterization can be done easily without the "opposition" of the type system (the price to pay for this flexibility is a much lower level of language security).

The quality factors of programs positively affected by this mechanism are reusability, extensibility and correctness (the latter from the security with which this type of polymorphism can be implemented in languages with static type systems).

3.10.1 Relationship with subtype polymorphism

In pure object-oriented languages, generally all objects are subtypes of a single type (in Smalltalk will be the object `OBJECT` and in Eiffel the class `ANY`). In these cases, you can simulate the parametric polymorphism by using subtype polymorphism,

[26] As will be seen in later chapters, the same dilemma arises when choosing the semantics of concurrent mechanisms in programming languages, especially with regard to the synchronization of concurrent objects. Unsurprisingly, the axiomatic approach is much simpler and safer.

[27] Subtype polymorphism is applicable so that objects can be of different types as long as they are descendants of the type of the stack element specified.

simply by using as the class parameter for this common supertype (or any other that is convenient). This way, the class can be reused for objects of any other descendant type. However, this option is not desirable, as you lose the static type information of these parameters, which can jeopardize the correctness of your programs.

So, although you can relate the two types of polymorphism, in static safe type systems, both are important and generally serve different purposes [Meyer 86].

3.10.2 Parametric restricted polymorphism

Some parametric polymorphism mechanisms allow, when desired, restrictions to be placed on the parameters of types. This type of polymorphism is called parametric-restricted polymorphism (*bounded*[28]). [Cardelli 85]. For example, if you want to build a class to implement ordered lists (the condition that the elements of the list are always ordered could be one of the invariants of this class), parameterized with respect to the type of its elements, it becomes necessary to guarantee that this list can only be instantiated with elements that establish an order relationship between them. If there is a third type – `COMPARABLE` – with the ADT order relation (operations `greater-than` and `lower-than`), then you can build the list class by restricting the type of its elements to be descendants of that type `COMPARABLE`, thus statically guaranteeing that the class will only be parameterized with elements that define an order relation between them.

In constrained parametric polymorphism you can generalize the constraint condition imposed on the types of the parameters by making it expressed by a function of types, instead of a predefined constant type. This type of polymorphism is called parametric polymorphism F-restricted (*F-bounded*) [Canning 89].

3.11 Multiple inheritance

Simple inheritance allows the construction of a class at the expense of another pre-existing one and, if it also implements the subtype relationship, defines the rules for polymorphic substitution of entities with type of the program. Multiple inheritance generalizes this mechanism, allowing the construction of classes at the expense of more than one ascending class.

This mechanism is by no means consensual in the object programming community. Its bad reputation is partly justified by the approximation made to it by one of the most popular object-oriented languages: C++ (so that it was not included in Java[29]). The fact that the first object-based languages – Simula67 and SmallTalk – didn't have multiple inheritance also contributed to it being viewed with considerable suspicion from the outset.

A recurring argument (in this as in many other mechanisms) is based on the possibility of being able to simulate multiple inheritance with simple inheritance using, for example, the technique of "twin objects" [Moessenboeck 93, Templ 93]. This approach, however, not only omits the problem of repeated inheritance (which occurs whenever

[28] At Eiffel the term constrained is used

[29] Where, however, a mechanism – interfaces – was added to allow relationships of subtypes similar to multiple inheritance

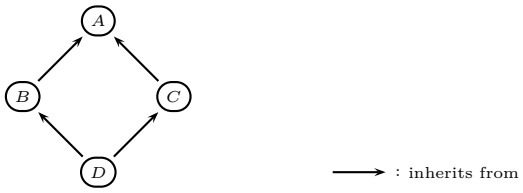

Figure 3.3: Repeated inheritance.

the static inheritance relationships between classes cannot be expressed by a tree), but is also an over-specification of this mechanism (expressing it as a function of a possible implementation).

An interesting difference – continuing to assume that inheritance establishes subtype relationships – between simple inheritance and multiple inheritance, is the possibility of a class being a subtype of two (or more) classes that are not related to each other by a subtype relationship (a property that is always verified in simple inheritance).

Another interesting property is the fact that static inheritance relationships between classes can be represented by a (directed) graph, and not necessarily a tree-type data structure.

3.11.1 Repeated inheritance

One of the problems - called repeated inheritance - raised by this mechanism occurs whenever a class, directly or indirectly, inherits more than once from the same class.

Figure 3.3 exemplifies this situation: The class `D` inherits "twice" from the class `A`. Should the attributes of `A` all be duplicated in `D`; shared or a judicious mix of both cases? In C++ there are only the first two possibilities and there is full sharing or separation when respectively in `B` and `C` the class `A` is inherited, or not, virtually. This approach is clearly wrong as it forces to make this very important decision for `D` in the `B` and `C` classes (and not in the `D` class itself).

On the other hand, there is also the problem of sharing or not the remaining services of `A` inherited repeatedly in `D` through `B` and `C`. Again in C++ the approach taken is rather inelegant and problematic. The use of a service of class `A` that wants to use the attributes of `A` inherited by `B` will have to explicitly indicate this base class `B` in the invocation of this service at `D`.

3.11.2 Name clashing

Another security problem raised by multiple inheritance consists of the situation in which a class inherits from two (or more) classes a service with the same signature or just the same name. In this situation, which of the services, if any, should be selected for execution? In C++ the situation is aggravated by the fact that this language allows

the overloading of services (section 3.14) which can generate ambiguities, sometimes difficult to detect and correct.

In EIFFEL all these problems are solved in an extremely elegant way. This language does not allow the possibility of a class having two (or more) services with the same name[30]. Whenever a class inherits a service with the same name from two or more classes, it is obliged to change the name of at least one of these services by so that one name corresponds to only one service in the class. This renaming mechanism resides in the class where the problem arises, and allows for an elegant solution to the sharing or replication of services in repeated inheritance. Using the example from figure 3.3, services from `A` will be shared if they are inherited in `D` with the same name and if they have not been redefined in `B` and `C`, or replicated in the opposite case[31].

3.11.3 Equivalent classes

A very important consequence of this approximation made in EIFFEL, is that it guarantees, for any class, the existence of an absolutely equivalent class built without inheritance[32]. It is even possible, if the ascendant classes are not needed anywhere in the program for possible uses of subtype polymorphism, to replace any class with the equivalent class.

A relatively frequent misunderstanding of the inheritance mechanism (as for example in [Ryant 97]) in object programming, is to consider that an instance of a class, implemented by inheriting from ascendant classes, somehow contains an object from each of those classes[33]. Inheritance is not a mechanism for including objects, but for sharing the code of classes (subclass relationship), and for substituting objects (subtype relationship).

3.12 Support for Design-by-Contract programming

Design-by-Contract programming [Meyer 97, page 331] makes it possible to complete the practical implementation of the ADTs provided by the classes, making it possible to express the respective semantics – invariants of the class, preconditions and postconditions of the public services – by total assertions or partially[34] executable.

This not only makes it possible to verify the correctness of each class and each of its uses (giving a new meaning to the exception mechanism, as you'll see in section 3.13), but also distributes responsibilities explicitly and clearly between the classes and their clients (as opposed to the methodology of defensive programming [Meyer 97, page 344][Liskov 86]). Thus, the class will be responsible for guaranteeing the respective invariant in its object stable times [Meyer 97, page 364]; i.e. when objects can be

[30] Service Overloading

[31] In the more complex case where services have the same name but have been redefined in intermediate classes, the EIFFEL language still allows in certain cases to combine these services in a single, but we won't go into that situation here.

[32] This operation is called flat form in EIFFEL [Meyer 97, page 541].

[33] The C++ approach to multiple inheritance will be one of those responsible for this confusion.

[34] The use of comments in these assertions is encouraged, whenever it is not possible or convenient to their formal expression [Meyer 97, page 399].

	Obligations	Rights
Client	Satisfy the precondition of each required service.	Ensure that both the class invariant and the post-condition of the required service are met when the service ends its execution.
Class	Ensure that the class invariant is verified at stable times. Ensure that, at the end of the execution of each of your services, the respective post-condition is met.	Whenever one of your services is requested, demand verification of the corresponding precondition.

Table 3.1: Design by Contract (Adapted from [Meyer 97, page 342]).

externally used , as well as guaranteeing the post-conditions of its services, being the responsibility of its clients to guarantee the preconditions of these services (table 3.1).

3.12.1 Class Assertions

The assertions that implement the semantics of ADTs – invariants, preconditions and postconditions – will be referred to as class assertions.

3.12.2 Other assertions

Although not as important as the class assertions, other types of assertions can be defined that can be used within the algorithms (preferably structured) that implement the services of each class. This is the case with generic assertions (the `check` instruction in EIFFEL, the *macro* `assert` of the standard library of the C language and the `assert` instruction of the JAVA language) – applicable at any point in an algorithm – and with assertions that can be associated with repetitive instructions: invariants and variants of cycles (existing in EIFFEL).

Whatever the assertion involved, the responsibility for having it checked always lies with the program involved upstream of it[35].

3.12.3 Assertions and class interface

An essential aspect for a language to support contract programming is the need for class assertions to be part of the class interface (i.e. the ADT). Both the clients and the inheritors of a class must be obliged to comply with the class contract. If this is not allowed, then the description of classes as implementations of ADTs and subtype polymorphism falls apart. So, in the case of the inheritance mechanism, all the invariants of the ascendant classes of a class must be inherited (the invariant of the new class must respect all of them), as well as the preconditions and postconditions of each inherited service. Viewing the inheritance mechanism as a means of subcontracting [Meyer 97, page 576] (i.e. descendant classes have to at least respect the contracts of

[35]The same is true if the assertion is concurrent (section 5.14), although it may happen that part of the upstream program has not yet been executed when the assertion is first checked.

ascendant classes) whenever subtype polymorphism is involved, then the preconditions can be weakened, and the postconditions and invariants can be strengthened.

A very important aspect that must be taken into consideration when using assertions is the need for them to be – as much as possible – applicative and not imperative [Meyer 97, page 351]. So care must be taken not to use functions with side effects on the observable state of the program in class assertions [Meyer 97, page 400].

3.13 Exception mechanisms

The most important quality factor to take into account in a program is its correctness. However, we must also take into account the possibility of unwanted events occurring during execution, such as failures in the support system for program execution (for example: lack of memory, disk space or hardware), or in the program itself due to errors in its design. For the program to be robust, these situations must be taken into account and there must be the possibility of dealing with them in a predictable and, if possible, disciplined way. This is the function of the exception mechanism in programming languages.

If a fault occurs in the program, an exception is generated (implicitly by the program's execution system, or explicitly by the program itself) interrupting the normal execution of that program. This exception is propagated through the program's service execution stack, until it is "catched" by specific code to that effect, or until the end of the stack, at which point the program ends its execution. At that point, it is indicating the point in the program where the exception was initially generated and, if possible, also showing the contents of the program's execution stack at that time (since – in the majority of cases – the error is due to the program running before the point where the exception was generated).

If there is a need to guarantee the robustness of the program, making it fault-tolerant, the exception mechanism can serve to meet this need without the need to "contaminate" the program's normal algorithm with specific code to this situation.

A serious security problem that exists in most languages with exception mechanisms (as happens in Ada, C++ and Java) is the possibility of "cheating" the program by catching an exception and letting the program continue its normal execution without solving the problem that caused the exception. The problem here lies in the lack of a specification about what code that handles exceptions can and cannot do. Thus, this code is allowed to "catch" an exception, write an error message, and normally terminate the execution of the service where the exception was caught without propagating this exception to the rest of the program (moreover, this situation is sometimes presented as example in books presenting the exception mechanism of these languages). This situation negatively interferes with the simple relationship that should exist between objects and ADTs. An exception managed in this way can cause objects to be, wittingly or unwittingly, used outside their stable times, i.e. for which the ADT axioms may not make sense.

So what should be allowed in code that catches and deals with exceptions? Meyer [Meyer 97, page 417] argues that – in the execution of a service – only one of two actions is acceptable:

1. Try to correct the cause of the exception and re-run the service (retrying);
2. Clean up the environment (reset to an object stable time state), and report the failure (propagating the exception) to the service client (failure).

In this way, it is no longer possible to allow the program to continue its normal execution unless the cause of the exception is corrected. The exception mechanism in EIFFEL is based on this behavior, and is therefore called a disciplined exception mechanism.

Another essential aspect of the exception mechanism in EIFFEL is its relationship with assertions. So whenever an assertion is not checked an exception is generated, thus giving total coherence and simplicity to the implementation of ADTs at EIFFEL[36]. We thus have a synergistic use of all these different mechanisms, simplifying and giving consistency to the language (this elegant integration will surely be one of the strongest reasons why this language captivates many of the programmers who are exposed to it).

It's important to note that the exception mechanism is used to deal with failures in the support system for program execution and errors in programs. It is not used for normal, predictable situations that should be part of the program specification. The use of this mechanism for these situations is nothing more than the covert adoption of a "goto" instruction with all the associated complexity problems.

3.14 Polymorphism *ad-hoc*: service overloading

Some languages with a static type system (C++, JAVA) allow a class to have several services with the same name, as long as their signatures are statically different. The service to be executed is decided at compile time depending on the respective signatures. This type of polymorphism is called *ad-hoc*[37] [Cardelli 85].

This mechanism, while appearing to be useful in some particular cases, generates complicated problems of ambiguity and language safety. The ambiguity results from the fact that the name of a service in a class may no longer be sufficient for to locate it. The situation gets even more complicated if the structure of ascendant classes of that class is complex.

An unavoidable unsafe interference occurs with the subtype polymorphism mechanism. This situation is exemplified with the following program:

```
class A                                   p(a: A) is ... end;
  ...                                     p(b: B) is ... end; -- invalid Eiffel!
end;                                    end;

class B
inherit A                               ...
  ...
end;                                      local
                                            a: A;
class C                                     b: B;
feature                                     c: C;
```

[36]The complete list of situations that generate exceptions in EIFFEL can be found at [Meyer 97, page 413].

[37]Cardelli identifies another form of polymorphism *ad-hoc*, which will not be discussed here, associated with type coercion.

```
do                                                    end;
  a := b;
  c.p(a); -- (1)
  c.p(b); -- (2)
```

Thus, although the invocations in `(1)` and `(2)` are the same at runtime (both pass an object of type `B`) they are treated differently by the program (thus not being object-oriented, but by the static type of the entity that manipulates them).

If the language has multiple inheritance, we will have another source of potentially unsafe interference from this mechanism. Thus, whether the programmer explicitly wishes it, or just for the sake of distraction, is now allowed to inherit services with the same name as long as they have different signatures (for example, the two `p` services of the `C` class from the previous example could come, in a way not intended, from two different ancestor classes).

For all these reasons, and even taking into account the few situations where this mechanism seems to have some utility, it seems very questionable to us to adopt it.

3.15 Memory management

There are languages that delegate to the programmer the responsibility of managing the memory used by the program (C++) and others that assume this responsibility by automating this management (Eiffel, Java).

On the one hand, "manual" memory management makes it possible to fine-tune this process, ensuring that the execution system is not using up resources (in particular execution cycles of the central processing unit) at inappropriate times. On the other hand, this management is extremely sensitive to errors and omissions on the part of programmers, generating catastrophic consequences for the program in execution (when, for example, the same portions of memory are being used "simultaneously" for different purposes), or progressive losses of free memory in the execution system. Languages such as C++ that allow complex manipulations (such as the use of arithmetic operations) with memory pointers make these problems even worse, and can make it difficult to understand programs and detect and correct subtle memory management errors in the program.

Automatic memory management using garbage collectors avoids these serious problems with program correctness and the security of the language itself, while at the same time simplifying the programmer's work. The objections that can arise in specific applications, for example, of real-time runtime guarantees, can be mitigated if it is possible to adjust the parameters of the garbage collector at runtime (allowing, for example, it to be temporarily disabled).

3.16 Class services

Languages such as C++ and Java allow to define class services that are shared by all direct or indirect instances of the class in which they are declared (**static** type services). These types of services can be invoked regardless of whether an entity with a type is associated with an object, which can be useful in certain situations.

For example, if you want to build a class `DATA`, made up of the services day, month and year, and if you want to guarantee that its instances always represent a valid date - that is, this condition will be one of the invariants of the class - then it is useful to have a class service that validates dates, thus allowing this validation for possible clients, without it being necessary to implement this service outside the class.

Within the scope of this work it is necessary to include an analysis of this mechanism because it – by its very definition – directly interferes with some concurrency mechanisms.

3.17 Once execution routines

The EIFFEL language introduces another type of services: once execution routines.

Originally these services guaranteed a single execution for all (direct or indirect) instances of the class in which they were defined, and were therefore very useful for initialization (procedures) and sharing of objects (functions). More recent evolutions of the language [ECMA-367 05] allow the definition of single execution contexts differentiated using textual keys, and are also considering the possibility of in future allowing other contexts such as the same processor (as an alternative to it applying to the whole program), only for the object, for the class, and for free keys[38].

3.17.1 Comparing with class services

The comparison between single execution services and class services is interesting. Both are a mechanism for sharing services beyond of the object itself. However, the scope of that sharing, and the semantics in their execution, are significantly different.

While in the case of class services the scope of the sharing always applies to all instances of descendant classes[39] of the class where these services are declared; while in the case of single execution services this scope can be adapted to various situations (for the class, for the object, for all processors, or just for one).

In execution semantics, class services are executed whenever required, while single-execution services are only executed once, and in the remaining invocations either do nothing – if they are procedures – or – in the case of functions – simply return the value returned in the first invocation. They are therefore a very elegant form of both shared initialization of resources, and object sharing.

Class attributes are also a form of object sharing. However, they differ from single-execution functions in that they can have side effects. From this perspective, single-execution functions have a more functional approach, in contrast to the more imperative approach of class attributes.

Since single-execution functions are used to share objects, this means that the services of the objects thus shared behave - if the scope applies to all instances of the class - as if they were class services.

[38]The modification made by the author of the compiler SMALLEIFFEL (appendix D) implements all these variants.

[39]Including itself.

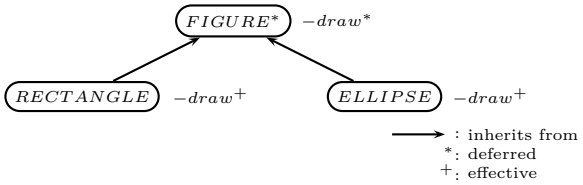

Figure 3.4: Abstract service example.

These different properties, as you might expect, will have quite different consequences in their integration with concurrent mechanisms, as will see in chapter 5.

3.18 Abstract services

A very useful mechanism in the design and construction of object-oriented programs, is the possibility of being able to declare in (non instantiable) classes only the interface of some of their services, relegating possible implementations to descendant classes (pure abstract services in C++ and **deferred** in EIFFEL).

The existence of this mechanism makes it possible to maximize the uses of subtype polymorphism. Its usefulness is well demonstrated by the example shown in figure 3.4.

The `FIGURE` class has no way of giving an implementation that makes sense for the `draw` service, so the possibility of defining services without an implementation solves this problem (as well as avoiding the instantiation of objects as direct instances of that class).

This means that abstract services allow the construction of classes without the need to (eventually) associate with an internal representation, since in many cases this would be an overspecification of the class's ADT.

3.19 Putting all together: interference between mechanisms

The number of different mechanisms that exist in object-oriented languages far exceeds those presented here. However, in this work, we have chosen to present those that are considered essential, and those that we think are more important (generally on the positive side, although here and there also on the negative side, as was the case with service overloading). Another criterion we took into account in this presentation, was to include mechanisms that by their very definition interfere with the inclusion of concurrency mechanisms in these languages (as will be seen in the chapter 5).

In this section we will complete the presentation of object-oriented languages by summarizing some of the possible interferences, unsafe or synergistic, in the joint use of these mechanisms.

As we analyzed in the previous chapter, understanding how the mechanisms of a language can interfere negatively or synergistically with each other is an absolutely

A:	Pure structured instructions	**J:**	Exception mechanism
B:	Information hiding	**K:**	Service overloading
C:	Simple inheritance	**L:**	Class services
D:	Subtype polymorphism and simple dynamic binding	**M:**	Once execution services
E:	Objects as Abstract Data Types	**N:**	Abstract services
F:	Parametric polymorphism	**O:**	Direct external modification of attributes
G:	Multiple inheritance	**P:**	Command and queries separation
H:	With contract programming	**−:**	Negative (unsafe) interference
I:	Without contract programming	**+:**	Positive (synergistic) interference

Table 3.2: Legend of mechanisms.

essential aspect for gauging the quality of the language as a whole. The quality of the language will therefore be greater the more it guarantees the absence of unsafe interference between mechanisms, and the more it benefits from meaningful synergistic interference between them.

Table 3.3 summarizes some of the most important unsafe interferences that can occur between some of the mechanisms of object-oriented languages, as well as possible solutions to these situations.

Similarly, table 3.4 presents some synergistic properties that are important for programming by objects.

Each letter presented in the first column of these tables corresponds to a particular mechanism, and these are presented in table 3.2.

Finally, the table 3.5 presents a summary of the characteristics of some of the most important object-oriented languages.

In this work, perhaps with the exception of the language EIFFEL[40], we chose to take a more focused approach to the properties and mechanisms of object-oriented programming (both individually and in their joint properties), rather than a detailed analysis of each of the object-oriented languages. Such a detailed presentation (beyond the table presented) would not, in our opinion, bring any added value to this work, and could even make it difficult to understand the work carried out.

[40] which served as the basis for the implementation of the mechanisms studied and proposed.

–	Description:	Languages:	Solutions:	Refs.:
A – J	pure structured statements can be interrupted by exceptions, and the program can continue without guaranteeing the post-condition that is implicitly associated with them	Ada95, C++, Java	adopt the disciplined exceptions mechanism	(page 32), (page 33)
B – C/D	descendant class may have a more restrictive information hiding	Eiffel	global program analysis, *CAT-Calls*	(page 21), (page 23)
B/E – F	ADT of type parameter can hide services required by parametric class	C++	parametric restricted polymorphism mechanism	(page 28), (page 28)
B/E – H	preconditions using non-exported services	Eiffel	statically prevent this situation in the compilation phase	[Meyer 97, page 357]
C – D	covariant redefinition of attribute types or service arguments	Eiffel	global program analysis; forbid polymorphic catcalls; multiple dynamic routing	(page 23)
C/D – I	the language does not require classes to respect the ADTs of ancestor classes	C++, Java, Ada95	global program analysis; forbid polymorphic catcalls; multiple dynamic routing	(page 23)
C/G – K	unintentional overloading of legacy services	C++, Java	eliminate K mechanism	(page 29)
D – K	ambiguity in the selection of services to be executed	C++	eliminate K mechanism	(page 33)
E – J	undisciplined exception mechanisms allow objects to be used outside their stable times	Ada95, C++, Java	enforce the disciplined exception mechanism	(page 32)
E – O	the class is no longer solely responsible for guaranteeing its invariant	C++, Java	eliminate O property	(page 20)
G – G	name clashing	C++	name change mechanism	(page 29)
G – G	in the presence of repeated inheritance which services of the inherited ancestor class several times should be duplicated, and which ones should be shared	C++	name-changing mechanism	(page 29)
H – H	use of functions with side effects on the observable state of the program in assertions	Eiffel	only allow the use of pure functions in assertions	(page 32)
I – J	there is no guarantee that correctness errors in the program will generate exceptions	C++, Java, Ada95		(page 33)
J – J	do not propagate exceptions whose cause has not been resolved	Ada95, C++, Java	adopt the disciplined exceptions mechanism	(page 32), (page 33)
J – J	use of exceptions for the program's normal algorithm	All	adopt the mechanism of disciplined exceptions, and restrict the use of assertions only to checking the correctness of the program	(page 33), [Meyer 97, page 346]

Table 3.3: Some unsafe interference between mechanisms.

+	Description:	Languages:	Refs.:
A + P	detection of pure functions	EIFFEL	(page 19)
B + E	ADTs define the desirable information hiding for each object	All	(page 26)
B + H	information hiding with class assertions implement the ADT of that class	EIFFEL	(page 30)
C/D + H	inheritance of contracts: subcontracting	EIFFEL	(page 31)
D + F	parametric restricted polymorphism	EIFFEL	(page 28)
E + N	abstract services allow you to build classes without implementation, or with a partial implementation, for your ADT	EIFFEL	(page 36)
H + J	since assertions are used to check the correctness of programs, exceptions are the appropriate response whenever there is a breach of contract	EIFFEL	(page 33)
H + P	assertions should only use query-type services (no side effects)	EIFFEL	(page 33)

Table 3.4: Some synergistic interference between mechanisms.

Language	Source	Characteristics	References
SIMULA	1967[a]	First language with object programming mechanisms. No information hiding. Abstract services. Simple inheritance (initially called concatenation). Subtype polymorphism. Dynamic binding (but not by default). Automatic memory management.	[Dahl 68]
SMALLTALK	1972	First time the term object-oriented is used. Pure object-oriented language. Dynamic type system. Classes can be manipulated as objects (meta-classes). Information hiding is predefined to hide all attributes and make all methods public. Simple inheritance. Dynamic binding. Class services. Automatic memory management.	[Goldberg 89]
C++	1983	Hybrid language designed as an extension of the C language with object-oriented mechanisms. Static type system. Information hiding. Multiple inheritance. Subtype polymorphism. Dynamic binding (but not by default). Parametric polymorphism. Method and operator overloading. Class services. Exception mechanism. Manual memory management.	[Stroustrup 85, Stroustrup 97]
EIFFEL	1986	Pure object-oriented language. Static type system. Information hiding of client-adjustable information. Subtype polymorphism. Dynamic binding. Multiple inheritance. Support for Design-by.Contract. Parametric polymorphism (restricted). Single execution services. Disciplined exception mechanism. Automatic memory management.	[Meyer 88b, Meyer 92, Meyer 97]
ADA95	1995[b]	Hybrid language. Incomplete approach to object-oriented programming (there is a syntactic separation between data - tagged record types - and functions/procedures). Static type system. Information hiding. Simple inheritance. Dynamic binding (but not by default). Parametric polymorphism. Exception mechanism. Support for concurrent programming.	[Ada95 95]
JAVA	1995	Static type system. Information hiding. Simple inheritance. Dynamic binding. Interfaces (with multiple inheritance from other interfaces). Method overloading. Exception mechanism. Automatic memory management. Support for concurrent programming.	[Gosling 96, Gosling 05]

[a] There was an earlier version from 1964, known as SIMULA 1.

[b] The first version of ADA is from 1979, but it wasn't until 1995 that the language moved closer to object orientation.

Table 3.5: Description of some object-oriented languages.

Chapter 4

Procedural Concurrent Programming

This chapter describes procedural concurrent programming, presenting its problems and challenges, as well as the most common solutions to them. Object-oriented approaches to concurrent programming have been deliberately excluded and will be covered in the next chapter.

4.1 Basic Concepts

A concurrent program differs from a sequential program in that it can be composed of more than one "subprogram" with autonomous execution.

In general, these "subprograms", despite having autonomous execution, cooperate with each other so that the program as a whole achieves one or more common objectives (which is why it makes sense to call it a program, and not a set of independent programs).

By convention, we will refer to the entities that execute the "subprograms" as "processors"[1], which are defined as follows[2]:

Processor

A processor is an autonomous processing unit capable of supporting the sequential execution of instructions.

We will also refer to these "subprograms" as programs of each processor.

It is important to make it clear that a concurrent program does not necessarily imply the simultaneous (in time) execution of processors. This execution can be, for example, cyclically alternated over time as happens in operating systems with preemptive scheduling of processes on computers with a single processing unit. In the

[1] Many authors (e.g., [Andrews 83]) use the term "process" for the same effect. However, we have chosen to use a different term, since that designation is often used for a particular implementation of processors in preemptive scheduling operating systems. In this way, we hope to avoid possible confusion with this particular implementation.

[2] This definition is similar to that used by Meyer [Meyer 97, page 964] for the concurrent extension SCOOP proposed for the language Eiffel.

particular case where execution is guaranteed to be simultaneous (as can happen, for example, in *SMP* architectures), it is usual to refer to concurrent programming as parallel programming.

4.1.1 Explicit approach to concurrency

We can define two possible approaches to building concurrent programs: one explicit and one implicit.

In the first, it is the programmer's responsibility to explicitly use appropriate concurrent abstractions for each processor's programs, making it visible to the processor which parts of the program are executed concurrently. In the second approach, the responsibility for partitioning a program into concurrent "subprograms" lies exclusively with the compilation and execution system. For this objective to be achievable, it is necessary to use appropriate programming languages that do not impose excessive sequential dependencies on programs, as is the case with declarative languages.

The adoption of this latter approach in imperative languages is much more complex, since these languages tend to impose a rigid sequencing on algorithms, making their parallelization difficult (in this respect, we can say that imperative languages are more susceptible to overspecifying the construction of algorithms).

Obviously, there is also the possibility of joint approaches to concurrent programming, simultaneously using explicit concurrency mechanisms and automatic parallelization algorithms at compile and runtime. However, in this work we will focus our attention only on explicit approaches to concurrency.

4.1.2 Concurrent programming systems

Approaches to concurrent programming can be based on *software* libraries (as is the case with the POSIX threads library for the C language [3] [Butenhof 97]); in concurrent languages (Concurrent Pascal [BH 75]); or a mixture of both (Java). We will refer to any of these approaches indiscriminately as a concurrent programming system.

4.1.3 Abstract processors

In concurrent programming systems, it is common to associate processors with specific program execution supports, such as processes in operating systems, or different threads within a single process.

This is the case, for example, with the Java language, whose concurrency mechanisms are statically linked to threads. However, the vast majority of concurrent program properties do not depend on specific execution supports for each processor, so this approach of rigidly associating each processor with a single execution support is, in many cases, clearly an over-specification (as already mentioned, this was one of the reasons for using the term "processor" instead of "process"). It is preferable to allow the possible definition of different execution supports for each processor, such as: processes, threads, a set of processes involving a group of networked computers, or using

[3] The compilation system must, however, be informed of this situation.

parallel and distributed programming support systems such as *PVM* [Geist 94] or *MPI* [Forum 94].

The system will be classified as having *heterogeneous processing* if it allows the association of different processing devices with processors, otherwise it will be designated as a *homogeneous processing* system.

Heterogeneous processing is a desirable property for concurrent systems as it reinforces the separation between programs and execution support devices, making them more easily adaptable to new execution contexts. It is important to note, however, that there are certain special cases of concurrent programming, such as real-time programming or embedded systems, where strong restrictions may be placed on heterogeneous associations of processors so that programs meet the objectives for which they were built.

4.1.4 Processor scheduling

When there are more processors than processing devices, or when there is competition between several processors for a shared resource, it becomes necessary to select which processors to execute. The strategy used for this selection is called processor scheduling.

In general, three factors are involved in this scheduling [Ruschitzka 77]:

- The decision mode;
- The priority function;
- The arbitration rule.

The decision mode characterizes the moments in time at which processor scheduling is decided (for example, in preemptive time-sharing operating systems, these moments occur at a constant frequency). The priority function consists of the processor ordering algorithm. Finally, the arbitration rule is the strategy used to choose between processors of equal priority.

The choice of scheduling can affect the safety of concurrent programs, since it can prevent some deadlock problems, or – when an extremely "unfair" algorithm is used – it can cause liveness problems[4] (see section 4.2.2) such as never choosing any processor for execution (starvation).

This work will not address the problems associated with processor scheduling. It will be assumed that the concurrency support system guarantees some fairness in access to execution for all existing processors.

4.1.5 Real-time programming

A very important area of concurrent programming that will not be addressed in this paper is real-time programming. In this type of program, it is essential to ensure not only the logical correctness (and robustness) of the programs, but also their temporal

[4] I could not find an acceptable translation for this term.

correctness. Temporal correctness occurs when it is guaranteed that the various components of the program terminate their execution within the time limits imposed in the program specification [5].

Much of the evolution of programming has been based on the abstraction of time in program execution (reducing it to merely an imposition of logical causality between the various actions of a program), so real-time programming requires, in a way, the reformulation of programs so that execution time becomes one of their essential aspects again.

Generally, approaches to this type of programming are based on the use of libraries and support systems specific to real-time execution (real-time operating systems). The use of specific language mechanisms for real-time programming is very rare, even though at first glance it seems that they could facilitate such programming (making it more abstract, and therefore simpler). This is one of the areas in which we hope to develop work in the future.

4.2 Correction of concurrent programs

Lamport [Lamport 83] defines two groups of essential properties to be verified in concurrent programs:

- safety;
- liveness.

4.2.1 Safety

Concurrent programs can create security problems (see definition of security in section 2.2.4) that are very complex and sometimes difficult to detect. These problems are always linked to incorrect synchronization between processors (processor synchronization is presented later in section 4.6).

This type of error is undoubtedly the most serious correction problem posed by concurrent programming, since they may depend on the relative execution time of each processor (which in general is not at all predictable and controllable), and in many cases are difficult to reproduce and detect.

Errors due to desynchronized competition[6] are the simplest of this type of problem. These errors occur whenever there is no adequate synchronization of a shared resource and there are several processors competing with each other for access to that resource. This situation can cause none of the processors to do what they are supposed to do correctly, leaving the shared resource in an inconsistent state. One possible solution to this problem is to protect access to that resource within a critical region, using semaphores, for example [Dijkstra 68a].

The severity of security problems in concurrent programs justifies the search for language mechanisms that guarantee the absence of such problems (axiomatic approach

[5] This does not mean that programs have to execute as efficiently as possible, but only sufficiently to guarantee the temporal specification.

[6] race conditions.

to synchronization). When, on the contrary, the responsibility for correct synchronization is passed on to programmers (operational approach to synchronization), as is the case in the vast majority of concurrent programming systems currently in use, there is always a risk of insecurity in programs.

4.2.2 Properties of *liveness*

Lamport [Lamport 83] presents these properties as those that describe what the program has to do. In other words, these properties, if verified, guarantee that programs achieve certain ends.

In concurrent programs, there are several situations that can prevent the verification of these properties.

Deadlocks

Deadlocks, which Dijkstra originally referred to as "deadly embraces between processors" [Dijkstra 68a], are situations in which processors wait indefinitely for resources reserved by others. For this situation to occur, four conditions must be met [Coffman 71]:

1. Mutual exclusion (exclusive access to resources);
2. Reservation and waiting (waiting for access to a resource while keeping at least one other resource reserved for oneself);
3. Circular waiting (waiting for a resource that is reserved by another processor,
4. Non-preemption (once a resource is reserved by a processor, only that processor can release it);
5. Circular waiting.

It is sufficient for one of these conditions not to be met to guarantee the absence of deadlocks.

There are three strategies for tackling this problem [Coffman 71]:

1. Static prevention;
2. Dynamic prevention;
3. Detection.

Static prevention guarantees the absence of deadlocks by ensuring that at compile time (statically) at least one of the four conditions is not met. For example, by allowing a processor to reserve a maximum of one resource at a time (preemption allowed), or by requiring processors to reserve all the resources they need at once (reservation and wait denied), or if an ordered reservation of resources is imposed (circular wait impossible); then deadlocks cannot occur. However, it should be noted that some care must be taken when using these prevention techniques, as they tend to be very detrimental to the overall performance of the program.

Another safe strategy is to use dynamic deadlock prevention techniques. If information about the current and possible future occupation If information about the current and future possible occupation of resources is available, then this knowledge can be used to avoid circular waits (such as the *banker's algorithm* [Dijkstra 68a, Habermann 69]).

The third possibility consists of having algorithms for detecting deadlocks and repair strategies (which can reuse the language's own exception mechanism).

Only the first two strategies are guaranteed to be safe, since they do not affect the normal execution of the processors, and should therefore be the main ones to consider when constructing safe languages.

Unlike mutual exclusion – which is a local problem with local solutions – deadlocks arise as a result of global interference in the program between processors. This characteristic makes this problem much more difficult to deal with.

Other problems

The occurrence of deadlocks is undoubtedly the most frequent problem in ensuring the liveness of concurrent programs. However, it is not the only one. There may also be livelock problems, which, like deadlocks, eternally prevent (unless resolved, of course) the progression of the program (or part of it), but with the difference that it is not due to passive blocking of the processors, but rather active blocking in which they are in a busy waiting process[7] for each other.

Another possible problem is the eternal suspension (starvation) of a processor (or several) simply because the processor scheduling system never selects it for execution.

4.3 Essential requirements

In this work, we are interested in studying mechanisms for object-oriented concurrent programming languages with abstract processors without real-time requirements.

A first step in this direction will be to clearly identify the essential requirements for procedural concurrent programming.

These requirements are divided into three groups [Andrews 83]:

- Concurrent execution of processors;
- Communication between processors;
- Synchronization between processors.

4.4 Concurrent execution of processors

Concurrent programming systems must have appropriate mechanisms to start, support, and terminate the execution of processors. This basic behavior can be obtained directly through specific programming language mechanisms, or indirectly by using appropriate *software* libraries. The first approach is the natural choice for concurrent

[7] busy waiting.

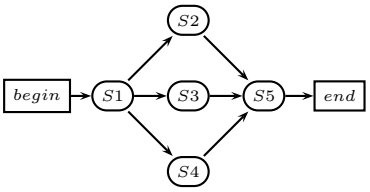

Figure 4.1: Example of structured concurrent execution instruction.

languages because it allows the compilation system to know the points in the program where new processors are created. This makes the subprograms associated with the processors explicit, improving the compilation system's knowledge of the concurrent program.

The second approach is justified when one wishes to introduce concurrency into sequential languages without modifying them[8]. In this situation, it is important to note that, although the base language may not be affected, the same is not true for the compilation system. In order to generate concurrent programs that work properly, the compilation system will generally always have to know whether or not the program being compiled is concurrent. In this work, we will focus exclusively on the first approach.

4.4.1 Structured concurrent execution instruction

One possibility for expressing processors is the use of concurrent execution instructions. Dijkstra [Dijkstra 68b, page 12] proposed one of these structured concurrent execution instructions as an extension to the language ALGOL 60[9]:

```
begin
   S1;
   parbegin
     S2;
     S3;
     S4;
   parend;
   S5
 end
```

Where `S1`, `S2`, `S3`, `S4` and `S5` are blocks of language instructions, the behavior of the program will be to execute `S1` followed by the execution of `S2`, `S3` and `S4` and finally, and only after these three blocks have finished their execution, `S5` will be executed. Figure 4.1 shows the execution graph of this program.

Although this instruction has the very important property of being purely structured (page 16), it limits the expressiveness of the language since it only allows the construction of concurrent programs in which groups of processors are always created and destroyed together. Another limitation of this instruction is that it only allows

[8]This is the case with POSIX-THREADS for the C language.

[9]A more appropriate name for this type of instruction would be **cobegin-coend** [Andrews 83, page 8].

the expression of a statically predefined number of processors (in the example given, three).

4.4.2 Processor fork and join instructions

A more generic alternative for creating new processors is based on the **fork** instruction [Conway 63, Dennis 66]. This instruction allows the creation of a new processor associated with the (concurrent) execution of a procedure. This instruction is complemented by the **join** instruction, which is used to make a processor wait until a concurrent procedure finishes its execution. The example presented above, implemented with these instructions, would look like this (assuming that `S2`, `S3`, and `S4` would be procedure calls):

```
begin
   S1;
   fork S2;
   fork S3;
   fork S4;
   join S2;
   join S3;
   join S4;
   S5
 end
```

With this group of instructions, unlike the previous instruction, it is now possible to express any concurrent execution graph of programs as well as create an unlimited number of processors at runtime.

4.4.3 Static association of processors to procedures

Another possibility is to statically associate processors to procedures. In this situation, the execution of these procedures will, by definition, be concurrent with the program that invokes them.

If there can be only one instance of each of these procedures, the number of processors will be statically imposed. If, alternatively, a type can be associated with these procedures, then we will have the possibility of multiple instances of each of these procedures, allowing for a dynamically variable number of processors.

4.5 Communication between processors

There are essentially two (abstract) models of communication between processors [Andrews 83]:

- Message passing (direct communication);
- Shared memory (indirect communication).

In the message passing communication model, processors communicate directly using any communication channel between them. There is thus a sending processor

(client) and a receiving processor (server), with the possibility of communication taking place, after the message is sent by the first processor, only when the receiving processor is available (and willing) to do so. This form of communication is well suited to processors that are not very dependent on each other (loosely coupled), as is the case in distributed systems or client-server topologies.

Communication by shared memory is an indirect mechanism, in which communication is carried out using a shared entity that can be modified and observed. This communication model is well suited to situations in which processors often need to share changeable information (strongly connected).

As mentioned in [Lauer 78], either of the two communication models can be simulated with the other, so it can be argued that, in principle, a concurrent programming system would only need one of them. However, this conversion almost always represents a loss not only of efficiency but also, and more importantly, of expressiveness, since both represent different communication abstractions. It is therefore defensible to adopt both models in concurrent languages.

4.5.1 Synchronous and asynchronous communication

A communication mechanism is defined as synchronous in relation to a processor if, from the point of view of that processor, the communication only ends when it is successfully completed. In this situation, the processor may be forced to wait (block) until communication is complete[10]. In cases where full processing of the communication is not required before the processor can proceed with the respective algorithm, the communication is said to be asynchronous. Communication can also be a combination of both cases, when part of the communication is synchronous and another part is asynchronous. This is the case, for example, when the sender is synchronous with the placement of the message in the receiver's pending message queue or with the start of execution of the receiver processor, but asynchronous with the actual desired processing.

In the message model, communication can be either synchronous or asynchronous relative to the sending processor. In the first case, the sender processor will wait until the receiver processor receives and completely executes the request made. In the second case, the sender processor can continue executing its program immediately after sending the message. Both approaches have advantages and disadvantages. Synchronous communication guarantees the postcondition of the service executed at the point in the sending processor's program immediately after the communication instruction, but, on the other hand, it serializes the execution of the programs associated with these processors, thus reducing their potential for concurrent execution. Asynchronous communication, in turn, enhances the concurrent execution of the two processors but makes it difficult to understand the combined effect of this execution. In contrast, asynchronous communication requires the temporary storage of messages sent to the receiving processor in a queue-type structure, a requirement that does not apply to synchronous communication, or to partially asynchronous communication in which the sending processor waits until the receiver begins to execute the sent message.

[10]This will not always be the case, such as when using non-blocking synchronization mechanisms.

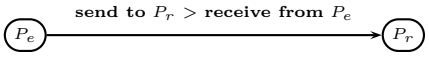

Figure 4.2: Direct identification.

In the memory sharing model, communication is, by definition, synchronous with respect to the data structure that represents the shared memory (since execution is performed by the same processor), and asynchronous with respect to other processors that may use the same shared structure.

4.5.2 Communication by messages

Message communication between two processors can be described as the performance of two operations, one on the processor sending the message and the other on the receiver. For communication to take place, it is necessary for both processors to be synchronized so that the reception operation takes place after the transmission operation.

This type of communication can be presented in the following abstract form [Andrews 83, page 25]:

```
SENDER: send EXPRESSION to RECEIVER

RECEIVER: receive VARIABLE from SENDER
```

Thus, the sending processor sends the message `EXPRESSION` to the processor identified by `SENDER`, which, in turn, receives the message from `RECEIVER` and stores it in `VARIABLE`. The set of identifiers `SENDER` and `RECEIVER` define a communication channel.

Identification of communication channels

The first requirement for this communication is a way to identify, in the concurrent programming system, the communication channels between processors. There are two possible alternatives for this purpose: either direct or indirect identification [Andrews 83].

In **direct identification**, identifiers are associated with each processor, and communication is carried out by directly expressing the processors involved (figure 4.2).

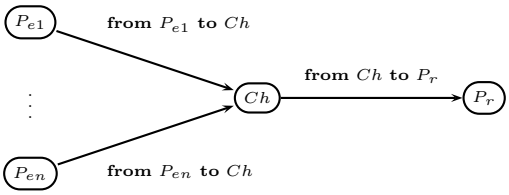

Figure 4.3: Indirect identification.

```
program SIMPLE_PRINT_PROGRAM

   process client
     var
       job: PRINTER_JOB
     begin
       loop
         job := fetch_new_job;
         send job to printer
       end
     end

   process printer
     var
       job: PRINTER_JOB
     begin
       loop
         receive job from client;
         print(job)
       end
     end

 end -- SIMPLE_PRINT_PROGRAM
```

This form of identification, however, has a serious problem that greatly limits its expressiveness. When using it, it is not possible to express the reception, in the same instruction, of messages originating from different sending processors.

The other possibility for identification, called **indirect identification** of processors, consists of associating identifiers with the communication channels themselves[11] (figure 4.3). As is clearly visible in the figure, with this form of identification of processors, it becomes possible to have multiple senders for one or even multiple receivers.

The identification of processors can also be classified as static or dynamic depending on whether, respectively, the identification of communication channels between processors is only possible at compile time or if it can also be done at runtime.

In static identification of communication channels, it is not possible to express communication channels that can only be known at runtime, and there is also the problem that they are associated with processors throughout the entire lifetime of the program (even if they are only used for a short period of time).

To perform dynamic channel identification, data types can be associated with processors, in direct identification, or with communication channels, in indirect identification. This makes it possible to dynamically create and destroy communication

[11] Sometimes referred to as mailboxes (Mailboxes).

channels.

Sequential process communication

Sequential process communication [Hoare 78] (CSP – Communicating Sequential Processes) is a concurrent programming notation based on synchronous communication and direct and static identification of communication channels.

Communication is done by input and output commands. The output command (emission) has the following form (`destination` is the name of a process):

```
destination!expression
```

The input command (reception) looks like this (`source` is also the name of a process):

```
source?target-variable
```

The combined effect of the two operations will, if the operation is successful, be equivalent to the following value assignment:

```
target-variable := expression
```

We thus have that this competition notation is based on an abstraction of remote value assignment (but restricted, as already mentioned, to the direct and static identification of the communication channel)[12].

Remote procedure call

Another possible notation for direct communication between processors consists of the abstraction of the procedure call instruction (instead of the value assignment instruction as in CSP notation). This notation is called remote procedure call (RPC – Remote Procedure Call).

The RPC notation allows for greater expressiveness in communication between processors since, unlike the CSP notation, it allows for the direct expression of bidirectional communication. The sending processor (local) can send information to the receiving processor (remote) through the procedure arguments and receive remote information through the procedure result (that is, if the procedure is a function).

In this notation, the name of the procedure designates the communication channel. Thus, in the case of direct identification, this name will also be the name of the processor (each remote procedure will be associated with a processor). In the case of indirect identification, there must be an alternative way to identify the receiving processor. This identification can be done, for example, by associating a set of procedures with the receiving processors (using, for example, the language type system, as is the case with the *rendezvous* mechanism of the Ada language [Ada95 95]).

[12]The CSP notation has other important aspects, such as conditional communication between processes, which we will not present since they were not important in the design and proposal of object-oriented concurrent mechanisms.

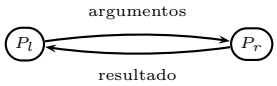

Figure 4.4: Two-way communication in RPC notation.

4.5.3 Communication by memory sharing

In communication by memory sharing, communication between processors is done using a shared data structure that can be modified by the sending processor and observed by the receiving processor. For this communication to be successful, all information to be shared must be written and read consistently as a whole (i.e., without the risk of the written information being read in an incomplete or inconsistent manner). To ensure temporal consistency in the shared information, it is also desirable that subsequent reads after a write consistently observe the result of that modification (in the next chapter in section 5.3.1, a correction criterion that guarantees these properties will be presented).

4.5.4 Identification of shared memory

Shared memory between processors can be identified either explicitly – by clearly labeling shared data structures – or implicitly – by using synchronization mechanisms that ensure the correct behavior of the data structures they synchronize –.

Although these two approaches seem at first glance to be just two complementary ways of looking at the same problem, they actually represent two very different approaches. In the first, synchronization is implicit (automatic), its correctness being guaranteed by the semantics of the shared data structures themselves (axiomatic approach). In the second approach, synchronization is explicit (programmed directly by the programmer), and the correctness of the use of shared structures is guaranteed by the correctness of the synchronization program (operational approach).

The great advantage of the first approach is the guarantee, at compile time, of the absence of synchronization errors in shared structures, that is, their security (section 2.2.4). The second approach, in turn, has the advantage of being much more flexible and adaptable to new forms of synchronization[13].

There are several languages that use explicit identification of shared data structures, such as Concurrent Pascal [BH 75] and Ada95 [Ada95 95] (protected types). However, implicit identification, despite its potential security issues, is undoubtedly the most frequently used (for example, the POSIX threads library in C [Butenhof 97]).

4.5.5 Relationship between both communication models

In either of the two communication models between processors, there is a sharing of information between them. In the case of the memory sharing model, the information

[13]The next chapter will present a proposal for abstract synchronization that, to a large extent, manages to have both advantages.

Figure 4.5: Communication by shared memory and messages.

is directly shared and usable by the processors. In the case of the message passing model, the information is packaged (possibly after a copy is taken) and sent together with the message. In other words, in this case, sharing is done by (possibly) replicating and sending the desired information. There is thus a duality between the two communication models [Lauer 78] (figure 4.5).

It is important to note that there can often be a mixture of communication models. This is the case when there is memory sharing in the information sent between processors in the message passing model (INF_1 and/or INF_2 in the example in Figure 4.5). In this situation, as is evident, the shared information behaves as in the memory sharing model, inheriting all its advantages and possible problems.

4.6 Synchronization between processors

We can define synchronization between processors as the control of all possible interactions between the respective programs in order not only to avoid the occurrence of undesirable interactions for the programs of the processors involved, but also to ensure the correct result of the desired interactions (as is the case with communication between processors). Thus, a set of processors will be synchronized if, at the points where interference (desired or undesired) between their respective activities may exist, that interference has controlled, predictable, and desired results.

The main application of synchronization mechanisms is undoubtedly (secure) communication between processors. In this situation, synchronization mechanisms must ensure a causal relationship between the event of "execution of an action" by one processor and the event of "detection of that action" by the others. In this situation, synchronization can be seen as the set of restrictions placed on the ordering of events from the various processors [Andrews 83, page 5].

4.6.1 Aspects of synchronization

We can define three distinct aspects of synchronization necessary in the construction of concurrent programs:

- internal;
- conditional;
- external.

Internal synchronization is related to the need for a shared data structure to protect its internal state against unsafe uses. Conditional synchronization results from the need

for access to a data structure to sometimes depend on its state. For example, accessing a list to remove an element from it only makes sense if the list is not empty. Finally, external synchronization results from the need to coordinate the concurrent use of multiple shared data structures in order to ensure that they are all accessed as if they were a single shared structure.

This separation between these three aspects of synchronization is common in the literature, although alternative terms are used. Holmes [Holmes 97] refers to these aspects as, respectively, exclusion, state, and transaction constraints[14]. Sometimes, internal synchronisation is also referred to as server synchronisation, and external synchronisation as client synchronisation [Puntigam 05]. In our opinion, the terminology used by Holmes, in particular that of exclusion, does not adequately represent the respective aspect of synchronization, since there are alternatives for internal synchronization that do not require mutual exclusion of concurrent processors. In any case, the terms are substantively analogous.

As expected, the communication model between processors is decisive in the way these various aspects of synchronization are conditioned.

4.6.2 Internal synchronization

This aspect of synchronization only arises, by definition, in the shared memory communication model. In the (pure) message passing model, there is no direct sharing of information, so that a block of information can only be used (directly) by a single processor at most.

If information is shared between several processors, then it becomes necessary to ensure that this information is not corrupted by any processor. To this end, there are several synchronization schemes – from the most conservative ones that impose mutual exclusion between the various processors, to the most liberal ones that allow the concurrent use of shared information – which guarantee, under certain conditions, the correctness of this sharing.

Although these synchronization schemes apply to procedural languages (and not necessarily to object-oriented languages), we chose to present them only in chapter 5 (section 5.10) in this thesis. This way, we believe we can make the In this way, we believe we can clarify the proposals presented in this thesis.

4.6.3 Conditional synchronization

As already mentioned, access to a shared resource often depends not only on the need to prevent errors due to unsynchronized competition, but also on the verification of a certain condition dependent on the state of the shared resource. For example, a document printing processor is required to wait conditionally until its input queue is not empty.

Conditional synchronization must be aggregated to both internal and external synchronization, applying to both models of communication between processors.

[14]In his doctoral thesis [Holmes 99] Holmes identifies two other aspects related to the concurrent system's response to message failures and message scheduling. However, these aspects are not important for the scope of our work.

Conditional synchronization strategies

Given the need to conditionally access a shared resource (or the conditional delivery of a message), there are basically three possible responses if the resource is not available [Lea 00, page 179]:

- report the failure immediately (balking);
- wait until the condition is verified (guarded suspension);
- wait until the condition is verified, but only for a certain period of time (time-outs).

In this work, we will focus on the most common case of conditional waiting until the condition is verified.

Message passing model

In the case of synchronous communication, this aspect of synchronization forces[15] the sending processor to block its execution until the synchronization condition is verified by the receiving processor.

In the case of asynchronous communications, the wait does not apply (by definition) to the sending processor but rather to the message queue of the receiving processor. An important aspect to consider in this case has to do with the restrictions imposed on the order of messages in the queue. While it is acceptable that messages originating from other processors can pass ahead of a message in conditional waiting (so that the waiting condition can be changed), the same cannot be said for messages originating from the same processor. If the order of these messages is allowed to be changed without the knowledge and consent of the sending processor, this may compromise the sending processor's program if it depends on the order of these messages (which can happen frequently).

These aspects of message queue management in the message passing communication model, to which are added those related to different priority issues (raised by real-time programs), will not, however, be addressed in this work. We will consider that the management of message queues is sequentially consistent (page 63), which implies that the order of messages originating from a given client processor is maintained in the server processor.

Memory sharing model

In this communication model, conditional synchronization blocks the processor until exclusive access to the shared data structure is guaranteed in a state where the wait condition is verified. Both internal synchronization schemes and

Both internal synchronization schemes and external synchronization schemes are directly affected by this aspect of synchronization, and for this reason there must be a strong link between them.

[15] Assuming, as mentioned above, the conditional waiting strategy.

As in the case of internal synchronization schemes (and also because of this), we chose to present conditional synchronization in more detail in chapter 5 (section 5.11).

4.6.4 External synchronization

The last aspect of synchronization refers to the need to act simultaneously on a set of shared data structures without interference from other processors. There are basically two approaches to this end. One is based on reserving all these data structures for exclusive use. This makes it possible to act atomically on all these data structures. The other possibility is to use transaction algorithms [Lea 00, page 249]. Transactions have the advantage of not requiring the exclusive reservation of the data structures involved, but they do require the voluntary participation of all the structures involved, as well as the possibility that the transaction may fail, requiring it to be repeated until it is successful.

In this work, we will adopt only the first possibility of reserving shared data structures.

4.6.5 Selection of data structures involved

External synchronization, by definition, usually involves several shared data structures. Thus, the mechanisms for expressing this type of synchronization (either explicitly or implicitly) need to identify which shared data structures are to be reserved.

The classic way to achieve this is based on a (structured) critical region statement, possibly conditional [BH 72].

```
region VAR-LIST do
   STATEMENT-LIST
 end
```

In the next chapter (section 5.15), we will see other possibilities for selecting data structures.

Chapter 5

Approaches to Concurrent Object-Oriented Programming

Having presented, in sufficient detail, sequential object-oriented programming and procedural concurrent programming, we will now study in depth and with some systematization various possibilities for integrating concurrent mechanisms into object-oriented languages.

As is evident, there are countless possibilities for integrating concurrent mechanisms into object-oriented languages, so it does not make much sense to present them all, much less without making an effort to compare their relative qualities. Thus, it is imperative, on the one hand, to clearly identify the quality criteria of languages that are to be guaranteed, and on the other hand, to delimit the characteristics of object-oriented languages that will serve as the basis for this integration.

In this work, as has been indicated, and sometimes justified, throughout the previous chapters, we chose to study concurrent mechanisms in object-oriented languages with the following characteristics:

- they are pure object-oriented languages (page 19);
- have static type systems (page 14);
- consider objects as instances of ADT (section 3.9);
- support contract programming mechanisms (section 3.12);

The quality criteria for evaluating and constructing languages considered most important were the following:

- expressiveness (section 2.2.1);
- abstraction (section 2.2.2);
- security (section 2.2.4);
- synergy (section 2.2.5);
- feasibility (page 10).

This chapter is organized as follows. After presenting some basic definitions, the approach to concurrency is made by first referring to the aspects of concurrent programming presented in the previous chapter. Next, we will discuss some of the mechanisms of object-oriented languages (all of which are presented in chapter 3) that can negatively interfere with concurrent programs. To solve these problems, we study the semantics that they should have in a concurrent context, trying to take advantage of this situation for the emergence of synergistic (safe) behaviors that make sense.

5.1 Basic definitions

For a better understanding of this chapter, it is important to define some concepts.

5.1.1 Concurrent objects

A *concurrent object* is an object whose services can be requested by more than one processor in overlapping time periods (concurrently), or in which the processor that directly invokes one of the services and the processor that executes them may be different. The first situation concerns shared objects model and the second concerns message passing.

All objects that are not concurrent are *sequential objects*. From the point of view of the language and the respective compilation system, sequential objects must be absolutely equivalent to sequential objects in sequential languages (so that the advantages associated with them, such as their security and efficiency, are not lost).

5.1.2 Concurrent conditions

A Boolean expression (condition) is said to be concurrent if it can depend, in the context in which it is tested, on another processor besides the one responsible for executing the test. A necessary condition for a condition to be concurrent is to depend, directly or indirectly, on queries to at least one concurrent object. However, this condition is not sufficient since it may happen, in the context in which the condition is tested by a processor, that any concurrent objects involved are reserved for exclusive use by that processor (therefore, their state can never be changed). Another situation in which conditions involving concurrent objects may not be concurrent occurs when the logical result of the expression does not depend on the concurrent objects involved (regardless of whether or not they are exclusively reserved for that processor). For example, the Boolean expression: $i \geq 0$ *and not buffer.empty*, involving the integer variable i and a concurrent object of type list referenced by *buffer* in the case where the value of i is negative is always evaluated to the false value, and is therefore not a concurrent condition.

5.1.3 Concurrent assertions

An assertion is said to be concurrent if the condition that defines it is concurrent.

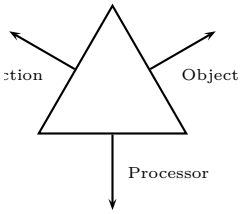

Figure 5.1: The three forces of computing [Meyer 97, page 964].

5.1.4 Reader and writer processors

When executing a service on an object, we will refer to a processor that is (or intends to be) executing a service that can modify the state of that object (or other objects or entities external to the program) as a **writer**. If, on the contrary, it is executing pure query services, then it will be designated as a **reader**.

5.2 Processors and objects

Meyer [Meyer 97, page 964] argues that there are three basic ingredients of computation: objects, processors, and actions (figure 5.1). Performing any computation involves using processors to apply actions to objects.

In the case of concurrent programs, we may have several processors executing actions on objects.

In pure object-oriented languages, all actions are located within objects (or at least encapsulated in their respective classes). In this situation, any shared memory will always be achieved within objects, so in the context of concurrent object-oriented languages, shared memory will now be referred to as shared objects.

5.2.1 Localization of concurrent objects

To ensure the safety and efficiency of concurrent programs, it is essential that the compilation system of the concurrent object-oriented language be able to identify all concurrent objects. These objects require the compilation system to associate them with appropriate synchronization code.

A safe way to identify these objects is to use the language's own static type system.

To this end, it is necessary to add appropriate type annotations that unambiguously associate concurrent type entities with concurrent objects.

The SCOOP language (Appendix A) achieves this goal using only the **separate** type annotation.

The approach followed for the prototype language developed in this work – named MP-Eiffel – is described in section 6.5.

5.3 Correctness of objects

Section 3.9 presented the essential theoretical support for understanding and correctness of (sequential) objects: an object is an instance of an implementation, possibly partial, of an abstract data type (ADT) [Meyer 97, page 142]. Thus, the correctness of a program depends essentially on the correctness of each of the ADTs it implements, regardless of the possible complex interactions that may occur between them. We therefore have that a necessary condition for an object to be correct is that its ADT is never compromised by its sequential or concurrent use.

In sequential languages, the requirement that objects can only be used in their stable times (page 31) guarantees the validity of the respective ADT, without compromising any of the important qualities of the respective sequential programs.

This same requirement can, of course, be applied to concurrent object-oriented programming. However, this implies that at most only a single processor can act within any given object. This is the situation that occurs, by definition, in communication mechanisms between processors based on message passing, but which, in the case of communication mechanisms by sharing objects, prevents the existence of intra-object concurrency (i.e., the possibility of multiple processors executing concurrently within an object).

We are interested in weakening this requirement without, however, losing the static guarantee that the ADTs associated with objects are not compromised in any way.

Concurrent Integrity of Objects

Intra-object concurrency cannot in any case compromise the implementation of the abstract data type of the respective class.

An immediate consequence of this criterion is the need to prohibit the existence of modifiable public attributes (page 20). To minimally ensure the semantic sanity of objects, these attributes would require the propagation of internal synchronism to all clients that could directly modify these attributes.

This criterion ensures that the correctness and integrity of each object considered individually is not compromised in concurrent systems. However, it is not sufficient to guarantee the correctness of the systems themselves as a whole. Each processor has a sequential program associated with it that imposes causal relationships between its actions. This causality cannot, under any circumstances, be compromised in concurrent programs, otherwise the sequential programs associated with the processors cease to make sense.

Thus, it is also necessary to ensure that the order of actions imposed by each processor's program is not compromised. It would not be acceptable for a reordering of the actions of a processor on an object to result in a reversal of the logical causality of those actions that is not equivalent to that imposed by the respective program.

Intra-Processor Sequentiality

Intra-object concurrency cannot in any case compromise the logical causality imposed by the programs of each processor.

That is, if a processor `P` requests an object to perform services: `s1` and `s2`, in that order, in no case may the possible effect resulting from the execution of `s2` in the system precede the effect of `s1`.

This criterion is similar to the so-called sequential consistency criterion defined by Lamport [Lamport 79].

Sequential Consistency

A concurrent execution of operations on a shared resource is sequentially consistent if it is equivalent to at least one sequential rearrangement of all operations on the resource, in which the order of execution of operations on each processor is maintained.

5.3.1 Linearizability

Sequential consistency only imposes the causality of instructions on each processor, and the relative order of instruction processing on different processors may vary arbitrarily. This freedom can cause problems in the practical verification of this criterion. In particular, this criterion does not have the property of being local [Herlihy 90b]. That is, the composition of sequentially consistent objects does not guarantee the sequential consistency of the program as a whole.

Thus, the correctness criterion considered appropriate for concurrent objects is not sequential consistency, but rather *linearizability* [Herlihy 87, Herlihy 90b].

Linearizability

An object is linearizable if a call to any of its services appears to have an instantaneous effect on that object at any time between the invocation and the return of the service.

Linearizability, unlike sequential consistency, has the property of being local. Another very interesting property of this criterion is that it does not require blocking (as is the case with monitors and reader-writer schemes). This creates the possibility of safely using non-blocking synchronization mechanisms, reducing or even eliminating the risk of deadlocks and starvation.

When verifying linearizability, each object will be considered together with any executable assertions (invariants, preconditions, and postconditions).

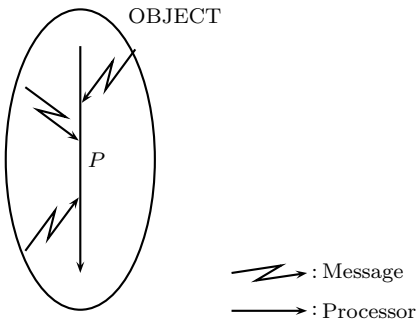

Figure 5.2: Active Objects.

5.4 Concurrent execution of processors

What possibilities, then, make sense for associating processors with their respective (sub-)programs in object-oriented languages?

In the previous chapter (section 4.4), several possibilities were presented for procedural languages.

The application of structured concurrent execution (section 4.4.1) would be one possibility, but given the limitations of expressiveness that it represents, we will not consider it.

5.4.1 Association of procedures to processors

The association of procedures to processors is a natural choice and well suited to procedural languages. The same approach in object-oriented languages in the case of procedures not belonging to any object is not acceptable (and is not even possible, by definition, in pure languages). We therefore have that such procedures (or rather: routines) must be part of some object.

5.4.2 Promoting processors to objects

One possibility is to make the processors also objects, usually referred to as active objects.

In these objects, one of the services contains the processor algorithm and necessarily also all the synchronization code necessary for communication to and from the outside (figure 5.2). The creation of one of these special objects implies the creation of the respective processor and the full execution of its subprogram (which is, as mentioned, associated with a single service of the object). This is the approach followed by the languages POOL [America 87b], EIFFEL// [Caromel 93] and also ADA [Ada95 95].

This possibility raises several problems. One of them[1] is to consider that this type of processor is a valid abstract data type, which is difficult to accept (it would be an

[1]Other problems with this approach are briefly discussed later: (page 69) and (section 5.16).

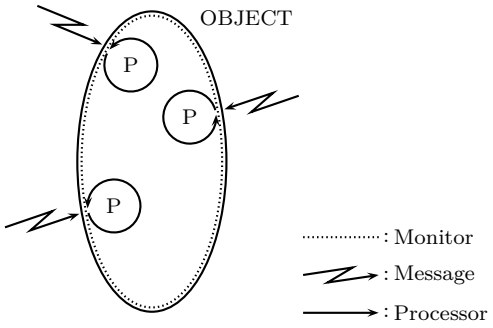

Figure 5.3: Actors.

abstract data type with only one operation). If this concept were applied to the special case of concurrency in a sequential program (a processor), it would become evident that we would be contradicting the basic definition of object-oriented programming (construction of software systems as organized collections of implementations of abstract data types).

5.4.3 Associating processors to objects

A better alternative is presented by the "actors" model [Agha 86, Agha 99] (figure 5.3). In this model, instead of considering processors as executing the algorithm of a single service in special objects, the actor objects are associated with a processor (not shared with other actors) capable of executing any of the object's services (we thus have a static association of processors not to a single procedure, but to a group of procedures belonging to the actor object).

As with the previous approach, a processor is created together with the creation of the respective actor object. After its creation, the processor becomes available to execute, at the request of clients, any of the public services of the object.

This approach is based exclusively on the model of communication between processors by message passing, and as such, is well suited to the distributed modular nature also oriented to messages (between objects) of object-oriented programming. However, it has the limitation of making processors and objects inseparable entities, making it impossible to implement communication mechanisms by shared objects.

5.4.4 Distributing objects by processors

The next logical step is to allow the same processor to handle (exclusively) multiple objects, instead of just one as in actors (despite this generalization, an object is always executed by the same processor).

This is what happens in Meyer's proposal (figure 5.4) to include concurrency in the Eiffel language [Meyer 97, page 951]: SCOOP[2] (see appendix A for a brief introduction to this language).

[2] *Simple Concurrent Object-Oriented Programming*

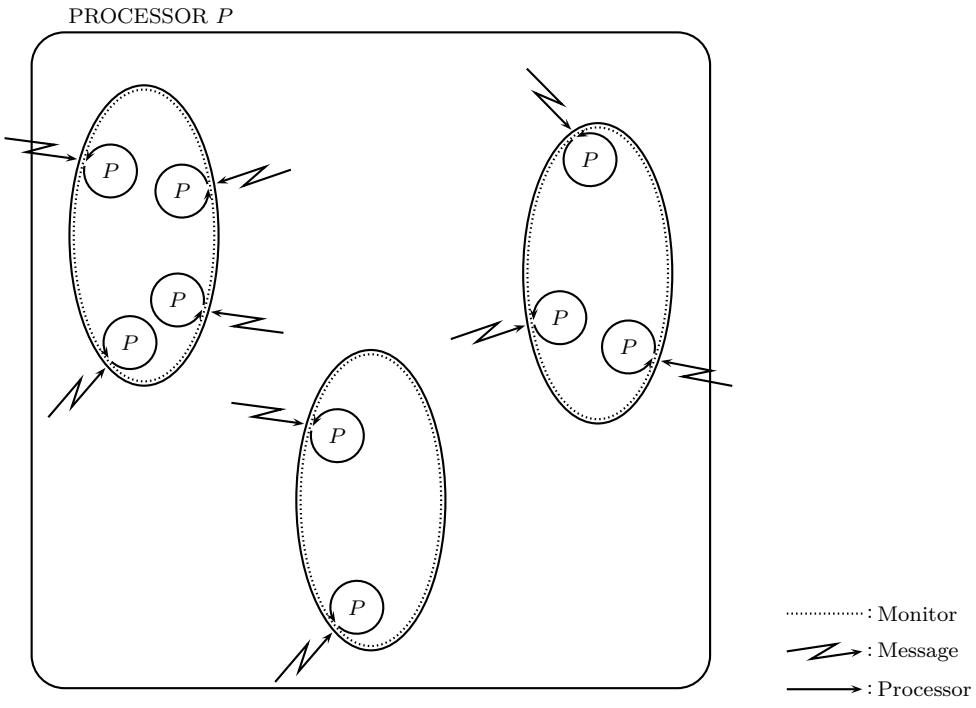

Figure 5.4: SCOOP.

However, as with the actor approach, this approach has the problem of restricting communication between processors to the message-passing communication model.

5.4.5 Orthogonal objects and processors

A fourth possible approach consists of making objects and processors completely independent entities. In other words, allowing different processors to perform actions on the same objects, that is, having mechanisms that express the communication model between processors by shared objects.

This is the approach taken in several very popular concurrent systems, such as Java and the protected types of Ada. However, if this possibility is not implemented properly, serious security problems can arise.

5.5 Communication between processors

The most important aspect of expressiveness in the integration of concurrency in object-oriented languages is the relationship between communication between objects and communication between processors.

Object-oriented languages use a uniform mechanism for communication between objects based on message passing (page 22). Therefore, it would seem natural to reuse it as a mechanism for communication between processors (it is in accordance with the criteria used in the design of languages of abstraction, security, synergy, uniqueness, and

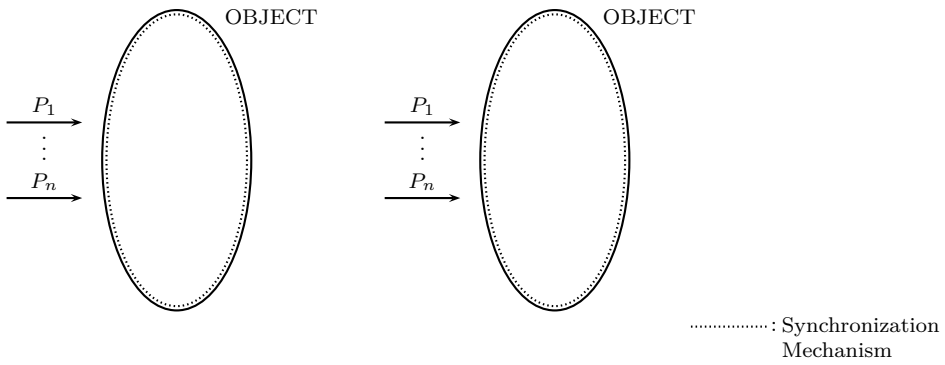

Figure 5.5: Orthogonal Objects and Processors.

consistency). However, since it is reasonably agreed that the execution of each processor should be similar to a sequential object-oriented execution of the respective program (in which the processor creates objects and establishes communication between them), the same does not necessarily have to happen with communication between processors.

At first glance, since objects communicate with each other through messages, the choice might seem obvious: the model of communication between processors by message passing. However, although both are models of communication by messages passing, they apply to different entities: objects and processors. Thus, the uniform model of communication between objects by messages used in object-oriented systems is, as will be seen, perfectly compatible with either of the two models (or both) of communication between processors: message passing and shared objects.

The model of communication by messages between processors, in a "pure" object-oriented language, would be the correct (and only) option if each object were at most executable by a single, and even, processor. This is what happens with languages such as Actors and SCOOP.

However, this choice radically limits the system's concurrency possibilities, since it prevents the existence of intra-object concurrency.

Instead of assigning each object in the program to a single processor, we can choose to share it among more than one processor, thus implementing the shared objects model of communication between processors. The question then is not to question the fact that objects communicate with each other by message passing (which always happens), but rather to decide which processors are responsible for fulfilling the request by executing the appropriate service for each object.

If a concurrent program is seen as a set of sequential programs communicating with each other (one for each processor), then when there is intensive resource sharing, the simplest and most intuitive solution will certainly be the shared objects communication model. On the other hand, in client-server architectures, or in distributed systems where communication between processors is low, then it will be simpler and more intuitive to use the direct communication model between processors by message passing.

This is one of many situations in which language design rules can be considered conflicting, making it necessary to choose the most important ones (although there will always be some subjectivity and many compromises in the choice made). Considering only the uniqueness rule (page 11) and also the fact that either of the two communication models can be implemented with the other [Lauer 78], choosing only one of them seems to be the correct option. However, as already mentioned, the two models represent two different ways of expressing communication between processors, for generally different concurrency needs, so without both options, the language will be less complete, expressive, and simple (thus going against the most important quality criterion of expressiveness (section 2.2.1)).

5.6 Communication by message passing

Let us first look at possible integrations of the model of communication between processors by message passing. It will be necessary to study how this communication model can be integrated into classes, since in "pure" object-oriented systems, processors can only exist and perform work within objects. As mentioned in chapter 4 (page 50), this communication model requires the identification of communication channels between processors.

There are two possible approaches to this identification: direct or indirect. The first, as described in section 4.5.2, is excessively restrictive on the receiver side, so we will not consider it. Instead, we will present an approach in which the receiver is directly identified by the sender without, however, the reverse direct identification occurring.

5.6.1 Direct identification of the destination processor

One possibility in this regard would be to associate each new processor with a unique enumerable value, for example of the integer type, corresponding to its temporal order of creation (example in PSEUDO-C in Figure 5.6). This option is, however, excessively insecure since it does not guarantee, except (possibly) at runtime, formally correct communication between the various processors (it would not be possible to guarantee that the information passed is what is expected by the receiving processor).

The ADA language – which is undoubtedly an interesting language – uses the type system for this task, including a specific type for processors, in this case designated by **task**.

With this strategy, it becomes possible to have mechanisms for direct communication between processors in a minimally secure way (although not completely secure, since there may be problems of desynchronized competition if the data structure passed to the processor is shared).

However, as is clearly visible in the example shown in Figure 5.7, there is a serious problem in deciding which messages are acceptable to the receiving processor. In ADA, this choice is made in the processor program using the **accept** statement (possibly within a **select** to allow multiple choices) applicable only to one of the **entry** type declarations made in the respective specification.

```
void proc_main(void)
 {
   // proc algorithm
 }

 int main(void)
 {
   int proc;
   Message msg;

   proc = new processor(&proc_main);

   send msg to proc;
 }
```

Figure 5.6: Example of explicit identification of processors with an integer value.

Since Ada is not a "pure" object-oriented language (the Ada95 version extended the previous Ada83 version with inheritance and polymorphism mechanisms, but in essence Ada95, like like C++, is a hybrid language), it could be argued that this **tasks** mechanism is not object-oriented.

In fact, a similar situation occurs with languages based on active objects (section 5.4.2). This option is not suitable for object-oriented languages, since the choice of messages to be accepted by the receiving processor has nothing to do with the ADT of the respective object. Worse than that, they are accepted and executed at unstable times of the object, so the notion of invariance of the object and the simplicity in understanding and using it is lost.

In an object-oriented language, communication with objects is done through the respective interface, so it is not surprising that direct identification of the destination processor is inappropriate.

5.6.2 Indirect identification

To perform unambiguous indirect identification of processors in the context of pure object-oriented languages, there are few alternatives other than making use of the objects themselves.

A simple approach is to associate each object, throughout its lifetime, with a single processor (which in principle should be the processor that created it). In the family of "actor" languages (section 5.4.3) and in SCOOP (section 5.4.4) this is the method chosen to identify processors. A message sent to an object belonging to another processor will be a direct communication between the respective processors. This option has the advantage over the previous one of being well suited to object-oriented systems, avoiding the very problematic situations of communications between processors at times when the invariant of the receiving processor object (i.e., the one that will have to process the message) may not be verified. In this case, the receiving processors will only respond when the respective object is in a stable time, which drastically reduces the complexity of these interactions.

This approach is similar to a remote procedure call (page 52) applied to public

```
-- a_processor.ads
package A_Processor is
  task type Processor is
     entry Start(A_Argument: in Positive);
     entry Another_Rendezvous;
     entry Finish;
  end Processor;
end A_Processor;


-- a_processor.adb
with Ada.Text_IO;
use Ada.Text_IO;
package body A_Processor is
  task body Processor is
     Done : Boolean;
  begin
     accept Start (A_Argument: in Positive) do
       Put_Line("Processor started with argument: " & Positive'Image(A_Argument));
     end Start;
     Done := false;
     while not Done loop
       select
         accept Another_Rendezvous do
           Put_Line("Rendezvous...");
         end Another_Rendezvous;
       or
         accept Finish do
           done := true;
         end Finish;
       end select;
     end loop;
  end Processor;
end A_Processor;


-- main.adb
with Ada.Text_IO;
use Ada.Text_IO;
with A_Processor;
procedure Main is
  proc: A_Processor.Processor;
begin
  proc.Start(10);
  proc.Another_Rendezvous;
  proc.Finish;
end Main;
```

Figure 5.7: Example of explicit identification of processors with the type system.

services of objects, with the advantage that the choice of services to be remotely invoked is properly contextualized by the ADT of objects (i.e., taking advantage of object-oriented methodology).

5.6.3 Synchronous and asynchronous communication

In chapter 4 (section 4.5.1) it was mentioned that, in this model, communication could be either synchronous or asynchronous. From the point of view of language expressiveness, both can be useful. Asynchronous communication increases program concurrency since it allows the sender processor to continue executing its algorithm independently of the receiver processor. On the other hand, synchronous communication guarantees the postcondition of the service executed remotely immediately after the message is sent, which can have important consequences in ensuring the correctness of the algorithm.

A very interesting synergy can be achieved if one takes into account the semantic difference between command-type and query-type services (page 19). In fact, invoking a command can be considered a communication directed solely from the client to the object, and therefore adapts perfectly to asynchronous communication (except with regard to precondition verification, as we will see later). On the other hand, invoking a query on an object is a bidirectional communication and therefore justifiably should be synchronous.

Caromel [Caromel 89, Caromel 93] proposes an alternative, called "wait-by-necessity" in which the wait is not performed immediately when invoking query services, but only when the respective result is needed. Meyer, in the extension SCOOP [Meyer 97, page 987], adopted the same idea. However, this wait-by-necessity mechanism can interfere negatively with other mechanisms of languages, in particular with mechanisms that support contract-based programming. The potentially most serious interference occurs with the verification of the precondition of the remotely invoked service (in the case, obviously, that this precondition exists). In fact, a failure in the precondition is the responsibility of the client (and not the object), so allowing the verification of this assertion to be asynchronous with the client processor program has extremely negative effects. First of all, the possibility of signaling, through an exception, at the appropriate point in the program of that processor the failure that is the responsibility of that same processor is lost. The result of this situation is the degradation of the robustness of the program, which may even make it impossible to implement adequate fault tolerance algorithms. For these reasons, it seems to us that, regardless of the type of asynchronous communication (whether by invoking a command or due to the wait-by-necessity mechanism), it is mandatory to to impose synchronous verification of the precondition[3].

In the case of applying waiting by necessity to query services, there is also the problem of verifying the postcondition of the service and the invariant of the object. This situation is much less serious than in the case of preconditions, since it can be accepted that any exception (to be propagated to the client) can be delivered at the wait point

[3]Only with regard to the sequential part of the precondition, since the concurrent part (if it exists) has another semantics as will be seen later (section 5.14).

(instead of at the invocation point). This is an acceptable semantics for the situation, although it may cause problems since client processor programs will eventually have to replicate the fault management code for multiple locations (all of which may await results from the initial invocation). The most important justification for adopting this wait-by-need mechanism is based on increasing the program's concurrency potential, since client processors can continue their agenda without waiting "unnecessarily"[4] for the other processor. However, this problem only arises if the language adopts only the model of communication between processors by message passing. If the language adopts both models (as in our proposal presented in the next chapter), then the program's concurrency potential can be maximized by the shared objects model. In this latter model, communication is synchronous, so it does not cause any of these negative interferences with the exception mechanism (section 5.18).

5.7 Communication by shared objects

The application of this communication model in procedural concurrent programming (section 4.5.3) is done using shared data structures. In object-oriented integration, obviously these data structures will have to be replaced by shared objects. However, it is very important to note that objects are not data structures (section 3.4). Since object-oriented programming is imperative, it is common for objects to have associated data structures. However, these are internal to the object, and the object's services may not apply exclusively to that internal structure (and may have side effects, not always reversible, on other objects or even on entities outside the program itself). These characteristics typical of objects (but non-existent in data structures) can affect the feasibility of secure implementations of shared objects (these problems will be addressed in the sections on intra-object synchronization intra-object 5.10).

An interesting aspect of the integration of this communication model is that it shares a very important characteristic with communication between objects in sequential languages: the processor that requires the execution of a service from an object is the same one that will then execute that service. In other words, although it is common to use the terminology of message passing between objects in object-oriented languages, in fact the communication model between processors by shared objects is, in this respect, more natural than the communication model between processors by message passing.

We will see that for many of the mechanisms of object-oriented languages, such as the exception mechanism (section 5.18), this communication model allows its behavior to be similar to that of sequential languages.

However, communication by shared objects, when compared to the model by message passing, generally hinders the synchronization of shared objects. This problem will be addressed in section 5.10.

[4]The quotation marks are justified because waiting may in fact be necessary.

5.8 Integration of both communication models

One can choose to adopt only one of the communication models – message passing (Actors, SCOOP) or shared objects (Java) – or choose both (Ada95). Programming languages serve as a means to solve computational problems. Thus, when considering the best choice, the first question we must answer is which of the three possibilities facilitates the work of programmers. Of course, the answer to this question may depend on the application domain required by each programmer.

To express algorithms in general-purpose languages, there is no doubt that both models are useful depending on the programs to be developed. In the case of the message-passing model, it is well suited to concurrent programs in which the processors are loosely connected to each other (for example, in distributed client-server systems). The shared objects model, on the other hand, is well suited to concurrent programs in which the processors are strongly connected, with frequent object sharing.

Of course, as already mentioned, it is always possible to convert programs expressed in one model to the other. However, this conversion will generally be done at the expense of less expressiveness and less efficiency. Thus, it is our opinion that it is desirable to have mechanisms for both models in languages oriented towards concurrent objects of general application.

5.8.1 Distinct interfaces?

In this situation, the question arises as to whether it is acceptable to use the same interface (i.e., the same perspective of the object's ADT) for both forms of communication.

At first glance, it seems that, in this situation, the interfaces do not necessarily have to be the same. The two forms of communication involve a very different commitment from the processors that may be involved. In communication by message passing, the direct collaboration of at least two processors is mandatory, so it seems excessive to require the receiving processor to respond to the invocation of any of its public services on its normal interface. In fact, this situation does not apply only to this case of concurrent communication. In the case of object creation, the vast majority of the object's public services cannot be used as a possible object initialization service. It seems to us that the most appropriate integration is to reuse the normal interface of objects for communication by shared objects, and to enable the definition of a separate interface (sharing the services of the object) for communication by message passing.

5.9 Synchronization between processors

Synchronization between processors – essential, among other things, for them to communicate with each other – is undoubtedly the requirement that has traditionally raised the most problems for the integration of concurrency in object-oriented languages [Holmes 98, Briot 98]. It is the author's opinion that a large part of these problems is due to the use of mechanisms with explicit synchronization (page 53), that is, using an

operational approach to synchronization, placing in the hands of the programmer the responsibility of correctly synchronizing concurrent objects.

We intend to follow the alternative approach of implicit (or automatic) synchronization in an explicit approach to concurrency (section 4.1.1). Of course, this approach to the problem requires not only mechanisms in the language that adequately express and abstract communication between processors[5] (the primary cause for the need for synchronization), but also to verify the feasibility of possible automatic implementations (i.e., to be performed by the compilation system) of appropriate and correct synchronization schemes.

5.9.1 Abstract synchronization

An automatic approach to the synchronization of concurrent objects may have the disadvantage of poor adaptability of the synchronization scheme to different situations and needs. In fact, if we statically restrict the synchronization of a shared concurrent object, for example, to mutual exclusion in the execution of its services, we may be excluding perfectly safe concurrent uses of the object, such as allowing multiple reader processors to observe its state.

On the other hand, if the synchronization of objects is the responsibility of the programmer, there is a potentially much more serious risk of building incorrectly synchronized objects.

In a secure (section 2.2.4) approach to a concurrent language, it is essential that the correctness of the synchronization mechanisms does not depend in any way on the programmer. In a safe and abstract (section 2.2.2) approach to a concurrent language, in addition to the previous requirement, it is essential that the programmer be able to choose any synchronization scheme as long as it is guaranteed to be safe and feasible by the compilation system. The various possible approaches to choosing synchronization schemes will be discussed in section 5.10.10.

5.9.2 Synchronization aspects

In the context of object-oriented concurrent programming, the synchronization aspects defined in section 4.6.1 are best described with the following terms:

- intra-object (internal);
- conditional;
- inter-object (external).

In the following sections, we will study the automatic feasibility of these various synchronization aspects, including the automatic integration of all these aspects into the same concurrent object.

[5] A proposal to this effect will be presented in chapter 6 within the scope of the advanced prototype language in this work.

5.10 Intra-object synchronization

This aspect of synchronization, as already mentioned (section 4.6.2), applies to the model of communication by shared objects.

In this section, we will present several synchronization schemes, identifying, for each one, the conditions of feasibility imposed on their automatic implementation by concurrent language compilation systems.

5.10.1 Concurrent object availability

In order to compare different intra-object synchronization schemes, it is useful to have some kind of objective metric that indicates the maximum concurrency potential of an object. This is the purpose of the *concurrent object availability* metric.

Considering that N_x is the maximum number of processors that share any property x (for example: reader or writer) intending to operate on an object, and that N_c is the maximum number of these processors that can safely act concurrently there ($N_c <= N_x$), the concurrent object availability (COA_x) relative to processors with property x is defined as:

$$COA_x[\%] = \frac{N_c}{N_x} \tag{5.1}$$

This factor measures the maximum percentage of processors with a given property that can safely operate concurrently within an object.

It should be noted that this value is not necessarily unique in each synchronization scheme, as it may depend on the concurrent state of the object (for example, the use of an object by processors with a given property may exclude its use by processors with other properties).

5.10.2 Total object coverage

A necessary requirement for any intra-object synchronization mechanism intra-object synchronization mechanism can be safely applied to objects is the need for all exported services of the object to be synchronized[6].

Full object coverage

It is a necessary condition for correctness in the synchronization of shared objects that all of their non-strictly private services be synchronized with some mechanism.

One of the strong objections [BH 99] to the basic concurrency mechanisms of the JAVA language lies precisely in the fact that there is no guarantee of full coverage in object synchronization, since if this condition is not met, problems may arise due to desynchronized competition.

[6] In JAVA [Lea 00, page 78] objects with this property are called completely synchronized or atomic objects.

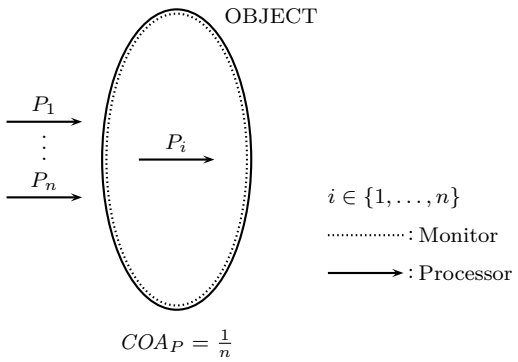

Figure 5.8: Monitors.

5.10.3 Monitors

A simple and sufficient approximation to ensure linearizability is to consider each object as a monitor [Hoare 74] (figure 5.8). It is interesting to note that Hoare [Hoare 74] and Brinch Hansen [BH 93] themselves recognized the importance of the class concept of the first object-oriented language – SIMULA – when they proposed monitors.

Monitors are the simplest of all intra-object synchronization schemes. The price to pay for this simplicity is the fact that monitors are only available to one processor at a time. For n processors, the value COA of a monitor is $\frac{1}{n}$, which is the smallest useful value possible. The concurrency mechanisms of the JAVA language were initially designed to be approximations of monitors [Gosling 96, page 399], but their intentions failed in some important respects [BH 99]. The current version of the language [Gosling 05], although it does not solve some of the basic problems with monitors, allows the use of other synchronization schemes besides monitors[7] [Lea 00].

Feasibility

Monitors place relatively few conditions on compilation systems. A basic requirement[8] is the need to identify all public services of the object. These services need to be protected with the monitor's synchronization code.

One possible algorithm for implementing this synchronization scheme is to create a new class that encapsulates the unsynchronized class, maintaining the same interface, and in which the monitor's synchronization code is implemented. This possibility has the advantage of avoiding the problem of oversynchronization (repeated or recursive synchronization) in the call of public services within the object itself. Section C.1.2 presents, as an example, a possible automatic implementation of the monitor synchronization scheme for a stack-type structure (`LIFO`[9]) which, in turn, is presented in section C.1.1. As is easy to verify, the automatic synchronization of the stack-type

[7] Maintaining, however, an explicit approach to synchronization.
[8] In addition, of course, to the identification of concurrent objects (section 5.2.1).
[9] Last In First Out.

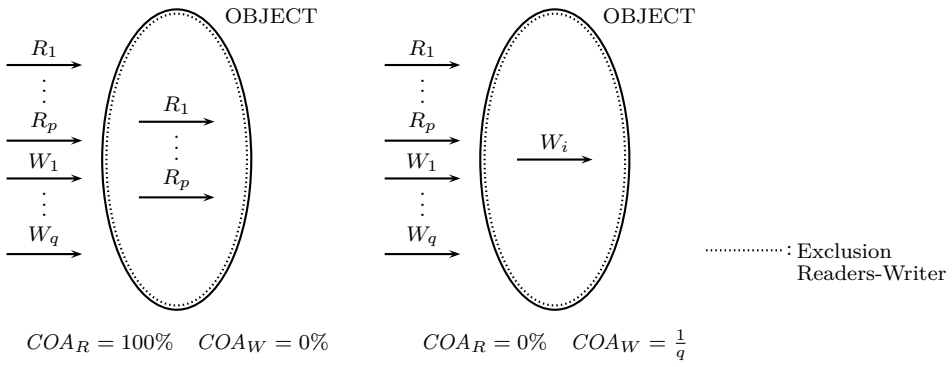

Figure 5.9: Exclusion between readers and writers.

class (for the class MONITOR_STACK) requires little semantic knowledge about the unsynchronized class on the part of the compilation system. Although the conditional synchronization algorithm (which will be described in section 5.11) takes advantage of the ability to distinguish impure commands and queries from pure queries, this is not a requirement of monitors but merely an optimization of this algorithm.

5.10.4 Exclusion between readers and writers

The imposition of mutual exclusion in the processing of object services can be considered an excessive restriction. Often, some of the processors are only trying to query (without side effects) the object to obtain certain information. In these cases, it is sufficient to ensure mutual exclusion when processing any service that may modify the state of the system (or the object itself or others), allowing concurrent processing of the remaining services (pure queries).

Therefore, an approach using the reader-writer synchronization scheme [Courtois 71] (one writer excludes all other processors, but multiple readers can concurrently access the object) is also a valid and safe option (Figure 5.9). This scheme has a higher average *COA* value than monitors, and is therefore less likely to block access to concurrent objects, which can reduce the risk of *liveness* problems such as *deadlocks*.

This synchronization scheme is used in the language Ada95 (protected types), and was also the initial approach taken in the language MP-Eiffel proposed by the author [OeS 04] (later modified to abstract synchronization [OeS 06a]).

Feasibility

Using this scheme improves the concurrent object availability, but the compilation system needs to extract more information from the classes to be synchronized. Unlike monitors, this scheme requires the ability to distinguish impure commands and queries from pure queries. In appendix B, section B.3 informally describes the algorithm followed in the MP-Eiffel language to solve this problem.

Although reader-writer exclusion synchronization has less contention than monitors,

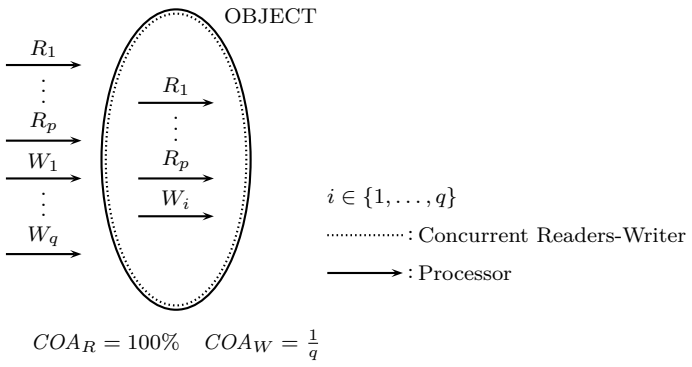

Figure 5.10: Concurrent Readers-Writers.

it nevertheless has a heavier implementation than a simple mutual exclusion mechanism, penalizing (albeit very slightly) the sequential efficiency of each processor in accessing services that modify the object. This aspect, which will also be seen in other choices of synchronization schemes, is similar to the optimization problems that exist in sequential languages. Thus, the ideal would probably be for the concurrent system not to mandatorily impose a particular implementation, but rather to ensure correct behavior, leaving the work of choosing how they are implemented to a system compilation optimization system. In other words, here too, the option for abstract synchronization proves to be correct.

Section C.1.3 presents a possible automatic implementation of this synchronization scheme for a stack.

5.10.5 Concurrent readers-writers

Lamport [Lamport 77] proposed a generalization to the previous synchronization scheme, which allows concurrent access between multiple "read" services and a "write" service. Mutual exclusion is only necessary for multiple writer processors (Figure 5.10). This way, reader processors never block a potential writer processor. In Lamport's proposal, query services must be repeated whenever they occur in competition with a writer processor.

When integrating this synchronization scheme into objects, it is necessary to anticipate the situation in which the invariant of the objects is not verified at the beginning or end of the execution of query services simply due to the concurrent execution of a writer. This situation must be properly handled, ensuring that breaks in the invariant, or any other assertion occurring before or after the execution of read services, and if there has been or is concurrent writing, result in the repetition (transparent in the behavior of the program) of the execution of these services. If, on the contrary, the failure in one of these assertions occurs without there being a concurrent execution of a writer, then an exception must be generated, as would be expected when using an incorrect object.

This synchronization scheme is very interesting because it imposes, in terms of im-

plementation, few more restrictions than the reader-writer exclusion scheme. It has less contention (a relatively higher *COA* or, in the worst case, equal) in the execution of writer processors, which reduces the risk of *deadlocks*. However, it can create *starvation* problems in reader processors when the execution of write services is excessively frequent [Lamport 77, Peterson 83].

A possible solution to this problem, applicable in certain cases, is proposed by Peterson [Peterson 83]. The basic idea is based on the duplication of shared data (which, in this case, would be the duplication of the state of the objects). In the particularly important case where there is only one writer processor, Peterson [Peterson 83] proposes an algorithm with no waiting for any processor (i.e., $COA = 100\%$).

Feasibility

This type of synchronization maintains the restrictions imposed on the previous scheme, extending them with the need for read operations to be repeated in case of failure (i.e., whenever there is a concurrent write).

This repetition (hidden from object clients) does not raise serious implementation problems, nor does it affect the expected behavior of objects because, by definition, pure query services do not alter the state of objects. However, as mentioned, it is necessary to anticipate the situation in which assertion failures occur in execution by a reader processor as a result of changes in the state of the object due to a writer processor. Thus, this synchronization scheme requires a language in which it is possible to transparently catch all exceptions generated during the execution of query services, allowing verification of whether the cause of the failure is due to interference with a concurrent writer processor – in which case the exception can be ignored and the execution of the service repeated – or if it is in fact a real failure in an assertion. This restriction is essential in order to implement this mechanism correctly, since only then is it possible to distinguish real failures from those resulting from desynchronized competitions (in this particular case, innocuous).

This problem of temporary invariant violation can be completely avoided in the particular case where there is only one writer processor. In this situation, there are algorithms, such as that of Peterson [Peterson 83], in which the reader processors always observe the shared object in a stable time.

Section C.1.4 presents a possible automatic implementation of this synchronization scheme for a stack.

5.10.6 Non-blocking synchronization

A group of synchronization schemes that has been gaining increasing interest is called non-blocking synchronization [Herlihy 91] (figure 5.11). This type of synchronization is characterized by ensuring that processors can execute operations on a shared data structure independently of the execution times of other processors, and that at least one of them will always be successful. An important special case is synchronization without waiting, in which it is guaranteed that all processors can perform the desired operation in finite time.

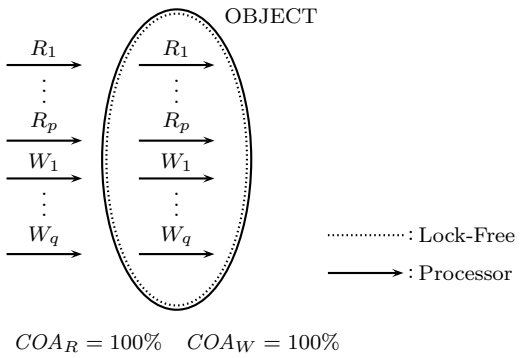

Figure 5.11: Non-blocking synchronization.

The advantages of this scheme are based on the absence of processor blocking[10] (making them immune to *deadlocks*) and their tolerance to failures of other processors. These characteristics make it particularly suitable for real-time systems [Anderson 97].

Currently, this type of synchronization is rarely used, although some change in this situation is foreseeable. One sign of this was the public release of a library of classes for JAVA that makes use of this synchronization (JSR 166: Concurrency Utilities [Sun Microsystems Java Specification Requests 04]).

The reasons why this type of synchronization is so rarely used are its complexity, the specificity of many of its algorithms, and especially the difficulty in ensuring safe implementations.

In this section, we are only interested in a preliminary approximation for future secure automatic implementations of these schemes. It is important to note that, apart from some experimentation with non-blocking algorithms in C, no experimentation has been done with these schemes in the proposed prototype language.

Basic Concepts

In general, non-blocking synchronization algorithms are based on the total or partial duplication of shared data structures (objects, in this case) and, when necessary, on the concentration at a single atomic instant of all modifications to that data structure required by each operation. This atomic modification of the object's state generally uses special hardware instructions, such as CAS (Compare-And-Swap) or LL/SC (Load-Linked, Store-Conditional) instructions. In these algorithms, as was the case with the synchronization scheme for concurrent readers and writers, it is necessary to anticipate the possibility of failures in updating the state of the object due to the action of other concurrent processors.

In such cases, it is necessary to repeat the entire process (until it is successful). In the special case of no-wait algorithms, as already mentioned, a maximum limit on the number of repetitions is guaranteed.

[10] Only for the aspect of intra-object synchronization.

Herlihy [Herlihy 90a, Herlihy 91] demonstrated that there are universal algorithms capable of implementing this synchronization in concurrent objects while respecting the linearizability criterion, and also presented universal methodologies (albeit not very efficient) [Herlihy 90a, Herlihy 93] for its implementation. The methodology presented, as mentioned by Herlihy, can be performed automatically by the compilation system.

Other possible schemes related to synchronization without blocking are based on software transactional memory systems [Herlihy 03]. These algorithms work similarly to transactions in database systems. Transactions are processed in three steps. First, the transaction is stated, then the required operations are executed, and finally, an attempt is made to submit the result of the transaction. If this submission fails, it is guaranteed that the transaction attempt did not modify the state of the object, and it can be retried. If successful, the result of the transaction will take effect (atomically) on the state of the object. This transaction process is repeated until it is successful. Harris and Fraser [Harris 03] propose a mechanism for the JAVA language (strongly based on Hoare's conditional critical regions) that takes advantage of the possibilities offered by software transactional memory systems for non-blocking algorithms (the proposal also includes a mechanism for conditional synchronization). If the requirements imposed on the compilation system, presented below, are observed, the proposal by Harris and Fraser can, in principle, be used to implement this synchronization scheme. For this to be possible, however, it is required that synchronization be applied to all public services of the object. As mentioned above, the possible future adoption of these synchronization schemes will require adequate testing beforehand.

Feasibility

Both Herlihy's generic algorithm [Herlihy 93] and the software memory transaction algorithms require the ability to retrieve copies of the state of objects and the possibility of repetitions in the execution of services. It is this last requirement that imposes the most restrictions on the static feasibility of these algorithms.

In fact, even taking into account that the execution of a service by a processor is applied to a separate stable copy of the object, not all services can be repeatedly executed without harmful side effects for other processors (or for the system as a whole). For example, a service that invokes a routine for writing to an external device (or, for that matter, to any external file), or that receives information from entities external to the program, cannot, of course, be repeated transparently. On the other hand, services that only modify attributes of the object are repeatable.

Repeatable services

A service will be repeatable if its effect on the state of the system – program and any external entities that interact with the service – as a result of its execution, is disposable as if the service had never been executed.

Thus, this synchronization scheme is statically achievable in a safe manner if the compilation system is able to correctly identify all repeatable services of each concurrent object (not allowing their choice if any of the services are not repeatable).

Once again, it should be noted that, unlike the synchronization schemes presented above, non-blocking synchronization is not yet integrated, and properly tested, in the prototype language that is being developed (this situation is expected to change in the future).

In appendix C.1, section C.2 presents, for illustrative purposes only, a first approach to implementing this synchronization scheme.

	Monitors	Exclusion Readers-Writers	Readers-Writers Concurrent	Non-blocking
Concurrent objects identification	**Yes**	**Yes**	**Yes**	**Yes**
Pure queries identification	**No**	**Yes**	**Yes**	**Yes**
Repeatable pure queries identification	**No**	**No**	**Yes**	**Yes**
Repeatable service identification	**No**	**No**	**No**	**Yes**

Table 5.1: Requirements imposed by simple synchronization schemes.

5.10.7 Mixed synchronization schemes

Table 5.1 summarizes the most important requirements placed on the compilation system for the four synchronization schemes presented. As can be easily seen, the schemes that have a higher average value of *COA* are also those that impose the most requirements on the compilation system.

However, there is no reason, theoretical or practical, to use a single uniform scheme for the synchronization of concurrent objects. One can also consider the possibility of using different synchronization schemes, simultaneously or alternately in time, on the same concurrent object. This opens up the possibility, among other things, of optimizing, in a way adapted to each object, its concurrent availability.

As with simple synchronization schemes, choosing a mixed scheme requires verification of all conditions for correctness, including, in particular, the need for full coverage of the object (section 5.10.2).

5.10.8 Mixed mutual exclusion synchronization schemes

One possible way to combine several synchronization schemes in an object is to enforce their mutual exclusion. That is, allow only one to be active at any given time. For example, an object may have a group of services that can be synchronized by non-blocking methods between them, and others that, not being repeatable, require mutual exclusion, reader-writer exclusion or concurrent reader-writer exclusion, with all other services of the object (figure 5.12). For these cases, it would be perfectly safe to use an asynchronous mutual exclusion mechanism for groups[11] [Joung 00], in which multiple processors could concurrently access services with synchronization without blocking, in mutual exclusion with processors attempting to access the object's other services. At runtime, the concurrent object would alternate (with or without imposing different priorities), depending on the needs, between the various synchronization sub-schemes.

[11]Interestingly, the author thought of and developed a class to implement this synchronization scheme (section D.11) before realizing that there was already a publication describing it.

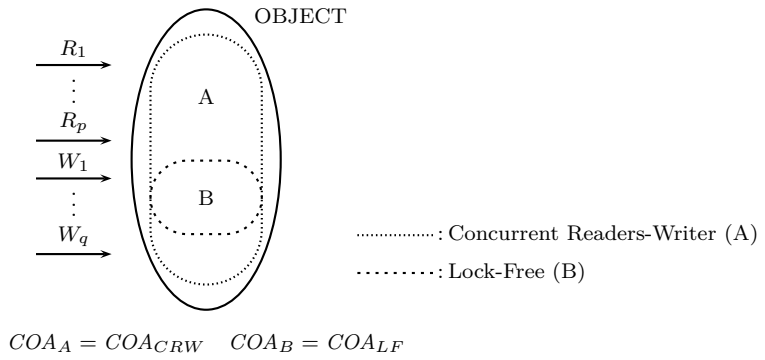

Figure 5.12: Example of a mixed synchronization scheme.

Another situation with a similar solution occurs when there is an interest in an object having different synchronization depending on the use and context in which it is used. For example, there may be a need to reserve the exclusive use of an object for a sequence of calls to its services[12]. If that object has non-blocking synchronization by default, and if this situation is not taken into account, it would not be possible to implement this type of exclusive use of the object, limiting the usability of non-blocking synchronization. A solution to this problem would be to implement both types of synchronization (non-blocking and reader-writer exclusion), again using the asynchronous mutual exclusion mechanism of groups to prevent the simultaneous use of both types of synchronization (which cannot, under any circumstances, be applied simultaneously to the same group of object services). This allows for the safe dynamic use of different types of synchronization on the same objects, taking full advantage of the least restrictive mechanisms in terms of intra-object competition.

Correction in the mixing of synchronisms by mutual exclusion

The use of any combination of mixed schemes in mutual exclusion is safe if the following conditions are observed:

a) There is full coverage of the object;

b) Each of the synchronization sub-schemes is safe with respect to the set of services of the object to which it applies (which will be a subset of all the object's services).

The demonstration of this correction criterion is immediate. Since the asynchronous mutual exclusion mechanism of groups, by definition, ensures that at most only one of the synchronization sub-schemes is active, and since it is also guaranteed that all

[12]This case is discussed in more detail in section 5.12.

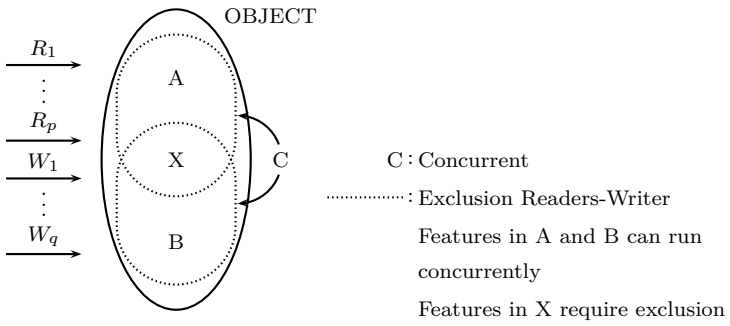

Figure 5.13: Double read-write exclusion.

services of the object are synchronized by at least one of the synchronization types (they may be subject to more than one, although, obviously, not simultaneously), it is easy to conclude that it is sufficient to ensure that each of the synchronizations is safe with respect to the subset of services of the object to which it applies.

5.10.9 Concurrent mixed synchronization schemes

By definition, the vast majority of concurrent combinations of synchronization schemes are not safe. Concurrent modification of an object's attributes almost always leads to desynchronized competition problems over those resources, which can result, in an unpredictable way, in meaningless values for those attributes, breaking the class invariant.

However, in certain very specific situations, it seems to make sense to allow disciplined concurrent access to the object, even without requiring non-blocking synchronization or reader-writer concurrency. For example, the concurrent use of two or more mutual exclusion zones or reader-writer zones (Figure 5.13) within an object – each protecting a distinct group of attributes – is generally not safe, since there is no guarantee that the invariant will hold in this situation – can, provided that certain restrictions are imposed on its use, be linearizable.

Using an analogy with a real example, if we had an object of type `CAR`, we could safely replace a tire while tuning the engine, even without being forced to use non-blocking synchronization (i.e., without requiring both operations to be repeatable).

The execution of an object service will be correct if the service condition criterion is verified (page 26).

Therefore, assuming only calls to object services that can modify its state (in general: commands), the execution shown in figure 5.14 is not correct, since processor P_1 cannot test the invariant safely in the interval $[t_3, t_4]$ between two calls to object services.

Linearizable verification of invariants

Analyzing figure 5.14, we can make some observations. From the point of view of processor P_1, it would be linearizable to anticipate the verification of the invariant at

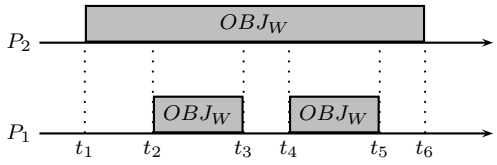

Figure 5.14: Wrong execution in an object with synchronization mixing in concurrency.

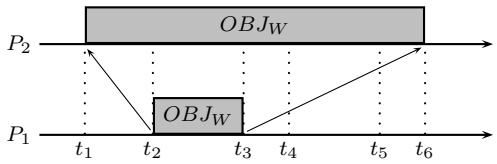

Figure 5.15: Correct execution in an object with synchronization mixing in concurrency.

time t_2 to time t_1, since, if only processor P_1 were executing the object, if the invariant is verified at t_1, it will also necessarily be verified at t_2. It would therefore be perfectly acceptable to reuse the invariant test performed by P_2 at t_1 for processor P_1 at t_2 (i.e., accept the result of the invariant test at t_1).

Similarly, it will be linearizable to delay and reuse the invariant test of P_1 at t_3 for P_2 at t_6, provided that no further calls to the object by P_1 are allowed in the meantime (figure 5.15). Furthermore, in order to generate exceptions correctly, processor P_1 must be blocked until time t_6, since only then can the class invariant be tested (which may fail if the program has errors, and this failure may result from the execution of either of the two processors).

The situation shown in Figure 5.16, despite involving two invocations by processor P_1 in competition with a single invocation by P_2, can be considered safe, since the invariant is not altered during the execution of pure queries of the object, so the invariant verified in t_1 can be reused in t_2, t_3 and t_4.

The case presented in Figure 5.17 is not correct since, when P_1 starts executing the

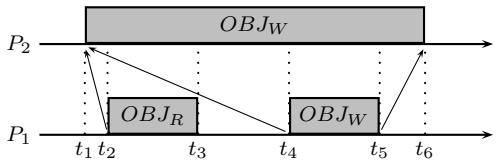

Figure 5.16: Correct execution in an object with concurrent synchronization mixing.

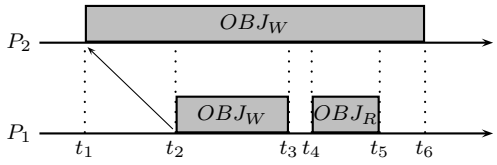

Figure 5.17: Wrong execution in an object with mixed synchronization in concurrency.

read service on the object at t_4, it is not possible to reuse or verify the invariant.

To complete the analysis of this type of synchronization, two situations still need to be addressed. The first occurs when the first concurrent execution on the object is performed in a read service. In this case, it is easy to see that the invariant verified at the beginning of that service can be directly reused for other services that are subsequently executed concurrently (since, by definition, read services do not modify the class invariant).

Finally, there is nothing to prevent the last write service to be performed in competition on the object from being the first to initiate that competition zone (as shown in the figures presented). What is required is that the input invariant be the one existing at the beginning of the execution of the first writer processor and that the output invariant be the one occurring at the end of the execution of the last writer processor.

Generalizing all these cases:

Concurrent invariant checking

In the concurrent execution of multiple processors on an object in the presence of mixed schemes of concurrent synchronization, it is linearizable to verify the invariant only when the first writer processor begins execution on the object, and when the last writer processor finishes, if in that time interval the following conditions are verified:

a) Each processor executes, at most, a single write service on the object;

b) Each processor executes zero or more read services provided that they necessarily precede the execution of any write service on the same processor.

Returning to the car example, with a concurrent synchronization scheme with multiple reader-writer exclusion zones respecting this criterion, we would have the possibility of simultaneously tuning the engine and changing tires by different employees (processors), but with the restriction that each employee can only perform one operation for each operation performed by all other employees. In other words, each employee can

only continue their work with the guarantee that the previous one has been done correctly (if there is a postcondition in the respective operation) without compromising the correctness of the car's state (expressed by the respective invariant). It is not difficult to see that all these considerations are equally applicable to the mixing by competition of other types of synchronization.

Correctness in the mixing of synchronisms with concurrency

It is safe to mix two or more synchronism mechanisms in concurrency provided that the following conditions are met:

a) Full coverage of the object;

b) Each synchronism mechanism protects a different group of attributes of the object;

c) The criterion for concurrent verification of invariants is satisfied.

Feasibility

An interesting feature of mixed synchronization schemes is that the requirements imposed by each sub-scheme do not necessarily apply to the entire object, but only to a subset of it.

For automatic verification of the feasibility of mixed synchronization schemes in competition, it is necessary for the compilation system to associate each service with the set of attributes that can be modified (directly or indirectly). Only services that never interfere with each other can be executed concurrently

To implement a synchronization algorithm for this scheme, it is sufficient to use a simple approximation based on a shared atomic counter.

Section C.3 shows a possible safe implementation (in C) of invariant checking for objects with this synchronization in the case where the processors are POSIX-THREADS.

In this implementation, all necessary synchronization is done in the invariant check.

5.10.10 Choosing synchronization schemes

Having presented the various safe synchronization schemes that can be automatically implemented by the compilation system, it is now necessary to address the problem of expressing their choice in concurrent programs.

Predefined selection in the language

This option is by far the most common. It is followed, for example, by the language ADA95, in which shared objects (protected types) are synchronized in a safe manner with the reader-writer exclusion synchronization mechanism [Ada95 95].

Another more flexible option would be to define in the language different annotations (in the type system) for different synchronization schemes, leaving it up to the programmer to choose the desired scheme for each object.

```
-- synchronization keywords:
--   monitor, exrw, crw, lockfree


-- class declaration definition:
shared monitor class SHARED_OBJECT
...
end


shared exrw class SHARED_OBJECT
...
end


shared crw class SHARED_OBJECT
...
end
```

```
-- mixed synchronization scheme:
shared class SHARED_OBJECT
feature lockfree
...
feature exrw
...
end


-- entity declaration definition:
class SOME_CLASS
...
feature
  a_procedure is
    local
      obj: shared crw OBJECT;
    do
      ...
    end
...
end
```

Figure 5.18: Example of direct choice of synchronization scheme.

Figure 5.18 shows an approximation (in PSEUDO-EIFFEL) in which, in addition to indicating the sharing of each object (**shared**), an annotation is included referring to the choice of the desired synchronization: **monitor** to indicate mutual exclusion; **exrw** for reader-writer exclusion; **crw** for concurrent reader-writers and **lockfree** for synchronization without blocking.

This approach is simple and makes the association between synchronization schemes and shared objects direct and obvious.

However, it goes against one of the objectives established in this work – the abstract synchronization of objects (section 5.9.1) – so it will not be an option to consider.

Automatic selection by the compilation system

One option for abstract synchronization is to delegate the choice of the most appropriate synchronization schemes entirely to the compilation system. For this choice, the compilation system can For this choice, the compilation system can make use of appropriate heuristics. For example, if the possibility is identified that the intra-object synchronization of a concurrent object could generate *deadlocks*, the compilation system may choose to use, if possible, a synchronization scheme without blocking, thus solving this problem.

However, this option may be inflexible, since it does not allow the programmer to have a say in this choice (especially given that there are no optimal heuristics for all

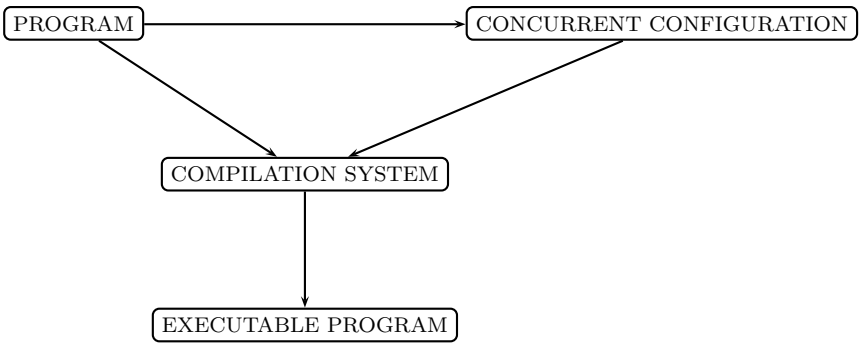

Figure 5.19: Schematic representation of shared synchronization selection.

possible applications of concurrent objects).

Shared selection

A third possibility is to share the choice between the compilation system and the programmer. This would be the ideal approach, provided that the compilation system does not allow the programmer to choose unsafe schemes, but at the same time gives freedom to choose any safe scheme. This combines the best of both worlds: the security of static synchronization choice and the flexibility of the programmer choosing the most appropriate synchronization for each object.

For this coexistence to be possible, it is desirable that any choices made by the programmer are not made directly within the program, but rather in a separate specification using, for example, an external configuration language and concurrent specification of the program.

Figure 5.19 schematically represents this approach. The concurrent configuration uses the program source code to unambiguously identify the concurrent objects for which a particular synchronization is to be chosen. In turn, the compilation system needs both the program (obviously) and the concurrent configuration to statically verify that the choices made are possible, and if so, generate the executable program.

In the prototype language that has been developed in this work, the intention is that the concurrent configuration be done using a concurrency control language. A presentation of this language can be found in chapter 6, section 6.7.

5.11 Conditional synchronization

In the context of pure object-oriented languages, and assuming a wait strategy (section 4.6.3), conditional synchronization is a mechanism, with possible blocking, for conditional exclusive access to objects.

The need for this synchronization may result exclusively from a condition internal to the object or, alternatively, from external conditions imposed by clients of that object. In the first case, conditional synchronization applies to intra-object synchronization, and in the second case, to inter-object synchronization. Both models of communication

between processors – message passing or shared objects – may also require conditional synchronization mechanisms.

Regardless of the model, if communication is synchronous (section 4.5.1), this synchronization will impose a block on the processor that requires the (conditional) execution of a class service. In the case of asynchronous communication, the wait occurs in the queue of messages to be processed associated with the object (or processor).

This section addresses only the problem of the automatic feasibility of this synchronism. The problem of choosing the language mechanisms that can express it will be addressed in section 5.14.

5.11.1 Synchronous communication

In the implementation of conditional synchronism for synchronous communication mechanisms between processors, an approximation similar to that used in monitors [Hoare 74] can be made. Monitors use so-called condition variables for this purpose. These variables, which are not associated with any value, are abstractions for processor queues and can be used to perform three operations[13]: wait (*wait*), signal (*signal*) and signal all (*broadcast*). The effect of these operations is as follows. The wait operation causes the processor that requests it to be placed in the wait queue associated with the condition variable (freeing the monitor for other processors); the signal operation causes one of the processors to be removed from the queue and given, as soon as possible, exclusive access to the monitor; finally, the broadcast operation does the same as the previous operation but for all processors in the wait queue. The POSIX-Threads library for the C language implements this type of variable.

This approach to conditional synchronization has, however, a serious problem: it is not statically safe, since it delegates to programmers the responsibility of declaring and using condition variables correctly. In addition to not being safe, it is also not sufficiently abstract, since the programmer is forced to construct the conditional synchronization code by explicitly linking it to the actual conditions associated with the state of the objects (operational approach). Hoare himself [Hoare 74, page 556] acknowledges that an alternative approach based on conditional wait instructions would be simpler and safer. On the other hand, this approach allows the implementation of very efficient synchronization algorithms, since the programmer can decide at which points in the program it is necessary to signal processors and, more importantly, can decide to which processors these signals will be addressed (using different condition variables).

In the Java language, the responsibility for managing the (basic) conditional synchronization mechanisms (called Wait, Notify, and NotifyAll) also belongs to the programmer. However, unlike the original monitors, it is not possible to declare multiple condition variables per object, and as such, to choose different groups of processors (which in Java are *threads*) in signaling (notification) operations. In Java there is a single condition variable per object, to which the wait and notification operations apply. Thus, a notification signal wakes up any processor present in the wait queue, regardless

[13] In the initial proposal by Hoare (and Brinch Hansen) [Hoare 74] there were only two operations: *wait* and *signal*.

of the wait condition associated with it. If there are several processors waiting for different synchronization conditions, there is a possibility that a notification will wake up the wrong processor (a situation that advises the alternative use of notifications for all [Lea 00, pages 191–192]).

However, none of these approaches comes close to the desired objectives: safe, abstract synchronization that is automatically achievable by the language compilation system.

A possible algorithm in this regard[14] would be to associate a single condition variable with each object (as in JAVA), implementing all conditional wait actions as wait operations on that variable (whether related to intra-object synchronization or inter-object synchronization), and placing signaling operations for all processors on that variable at the end of all public routines of the object[15].

When processors gain exclusive access to the object, they check whether the condition that caused them to wait (if any) is true, executing the routine if it is, or returning to wait on the condition variable if it is not. Obviously, this algorithm, despite meeting the intended objectives, is potentially very inefficient.

This algorithm can be improved if the compilation system has the ability to distinguish between commands and (pure) queries. In this situation, it is only necessary to signal all processors waiting at the end of the execution of commands (and any non-pure queries), since only these routines can change the waiting conditions.

This is the automatic implementation currently used in the prototype language being developed as part of this work (MP-EIFFEL) [OeS 06a]. The examples of automatic implementation of the various intra-object synchronization schemes presented in section C.1 also use this algorithm.

Possible more efficient implementations

In this problem of implementing conditional synchronization, the operational approach – in which programmers implement it directly – despite its (static) insecurity, is still the one that best manages to build very efficient algorithms.

In [OeS 06a] we propose two approaches (which still need to be implemented and experimentally validated) that can provide safe algorithms for this problem and that come much closer to the efficiency of algorithms made directly by programmers. One of the approaches makes use of concurrent assertions, and the other of the association between all the routines of the class and the attributes on which they depend or which they modify.

5.11.2 Asynchronous communication

In asynchronous communication between processors, there is no processor blocking as a consequence of this aspect of synchronism. The wait occurs instead in the mes-

[14] Similar to that presented by Hoare [Hoare 74, page 557] in the description of the implementation of conditional wait instructions.

[15] Once again, we can see the importance of enforcing the absence of publicly modifiable attributes, since, in this situation, the signaling of processors in the queue might have to be propagated to all clients of the class that could modify public attributes.

sage queue to be processed by the receiving processor. In this situation, the receiving processor will only remove the message from the queue if the waiting condition is met. Otherwise, it will move on to the next message (provided that, in order not to compromise sequential consistency (page 63), it did not originate from the same processor). At the end of the processing of each message received, and preferably even before moving on to the next message in the queue, the receiving processor will have to check if there is any previous message in conditional wait and, if the wait condition is true, execute it.

5.12 Inter-object synchronization

The (automatic) realization of this synchronization requires the use of algorithms for exclusive object reservation. These algorithms depend on the communication model to be used.

As in the case of conditional synchronization, in this section we will only address the problem of the automatic feasibility of this synchronization. Its integration into concurrent languages will be discussed in section 5.15.

5.12.1 Communication by message passing

In communication mechanisms (between processors) by message passing, it is necessary to be able to reserve remote objects to respond only to messages originating, directly or indirectly, from the processor where this synchronization is required.

With this communication model, it is necessary to anticipate situations in which a processor may have to respond to messages from processors other than the one that made the exclusive reservation of objects, because the latter processor has delegated that responsibility to them. For example, let's assume that we have three processors: *P1*, *P2* and *P3*, each of them managing messages sent, respectively, to the objects: `o1`, `o2` and `o3`. If part of the program in `o1` has the following remote call: `o2.do_something(o3)`, then if *P1* does not temporarily pass the reservation to *P2*, we will likely end up with the program being permanently blocked (*deadlock*). These problems of passing the baton in inter-object synchronization, in the context of the SCOOP proposal, are addressed in [Nienaltowski 06a].

5.12.2 Communication by shared objects

Automatic implementation in this communication model is done using a monitor-type mutual exclusion scheme (section 5.10.3). Since the cause for this synchronization is external to the object, even though its implementation may reside in the object itself (as we shall see), the intra-object synchronization mechanism (even if it is a monitor) cannot be used for this purpose as well. This means that concurrent objects may have two synchronization schemes associated with them: one to ensure intra-object safety and another to ensure inter-object reservation.

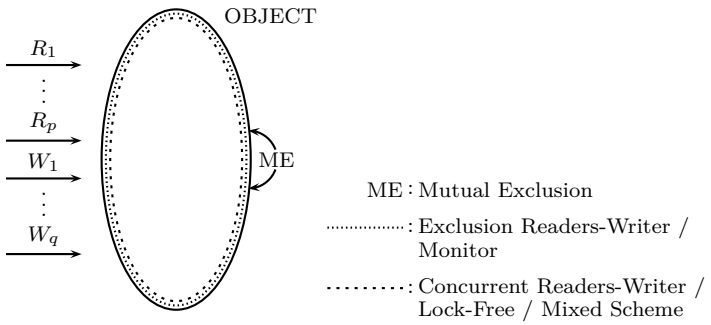

Figure 5.20: Mixed synchronization scheme for object reservation.

5.12.3 Integration with intra-object synchronization

This situation obviously raises the problem of the automatic feasibility of this integration of synchronization mechanisms, since one of them – the intra-object – may even be in a non-blocking synchronization scheme.

A simple and elegant solution to this problem is based on the mixed synchronization scheme by mutual exclusion (section 5.10.8). Figure 5.20 shows how this integration works. Intra-object synchronization (whatever scheme is used) belongs to one group, and inter-object synchronization belongs to another. Thus, it is not possible for unsafe interference to occur between the two. On the other hand, problems of *liveness* may arise, which will not be addressed in this paper.

An interesting aspect of the automatic implementation proposed for inter-object synchronization is the fact that synchronization, despite being required externally, resides in the object itself, which greatly facilitates its practical implementation.

It is important to note that an approach to this problem based on recursive mutual exclusion zones (*mutex*), as encouraged in JAVA, is not acceptable. Not only because it restricts intra-object synchronization to a monitor (which would be enough in itself to rule it out), but also because it is insecure by not clearly separating the two aspects of synchronization.

5.13 Other object-oriented mechanisms under concurrency

One of the most complex aspects when extending object-oriented languages with concurrent mechanisms is the possible interactions between these mechanisms and object-oriented mechanisms. Some of these interactions can be potentially insecure, so it is necessary to find solutions that avoid these problems. Others, on the contrary, open up the highly desirable possibility of defining synergistic behaviors when used in concurrency.

The following sections will study the security issues and the possibilities of synergies for some of the object-oriented mechanisms presented in chapter 3 in the context of concurrent languages.

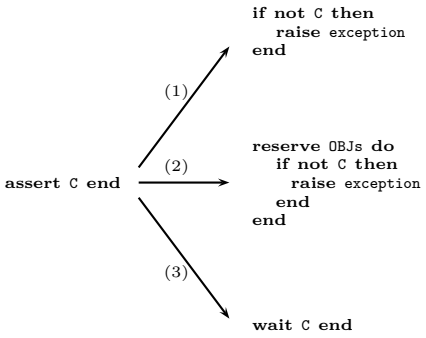

Figure 5.21: Possible behaviors in the presence of concurrent assertions.

5.14 Concurrent assertions

How should a program behave in the presence of concurrent assertions [16]?

Figure 5.21 shows the three possible responses. Since, by definition, a concurrent assertion depends on at least one processor other than the processor testing the assertion, unsynchronized sequential behavior unsynchronized behavior – behavior `(1)` in the figure – would clearly create an unsynchronized competition for the verification of condition `C`, and is therefore an unsafe and unacceptable option.

Another possibility – designated `(2)` in the figure – would be to unconditionally reserve all concurrent objects involved in the assertion, testing it later as if it were a sequential assertion. This behavior is also a potential source of desynchronized competition, although less critical than the previous one. Since the exclusive reservation of concurrent objects does not depend on the condition in the assertion, unless that condition is guaranteed by the invariants of those objects, that condition can be true or false depending only on when that reservation occurs (i.e., the relative speed of the processors involved). In this situation, the assertion would simply cease to be usable as a correctness test, losing its usefulness.

The last possibility consists of associating concurrent assertions with conditional waits: a concurrent assertion causes the processor testing it to wait until it is verified [OeS 06a]. The SCOOP proposal from the outset associated this behavior with separate preconditions [Meyer 97, page 993], but only very recently has it been proposed that this behavior be extended to other assertions [Nienaltowski 06b][17].

We can view assertions as being correctness conditions that apply to the program excerpts existing upstream of their location. It is always the responsibility of that code to ensure that these assertions are verified. In the case of preconditions, it will be the responsibility of the clients, while it is the responsibility of the class itself to ensure the invariant and postconditions. In a sequential program, there is only one processor, so if it is verified (usually by testing it at runtime) that an assertion is false, then we are unequivocally in the presence of an error in the program (all actions in the

[16] Section 5.1.3.

[17] Although the existence of concurrent assertions in SCOOP is limited to conditions using separate formal arguments.

program can only be executed by that processor). However, concurrent programs can have more than one processor, so if an assertion is concurrent, there is a possibility that its value may vary independently of the processor program that verifies it at runtime. We therefore have that the responsibility for ensuring this assertion does not necessarily belong to the processor that is verifying it, but possibly to other processors. It therefore remains a criterion of correctness, but not necessarily applicable to the processor that verifies the assertion, so again we conclude that the only safe behavior is to make these assertions conditional wait instructions.

We thus have a very interesting synergy between the mechanisms supporting contract execution and conditional synchronization.

Conditional wait, however, is not sufficient to guarantee the validity of some of the assertions, as is the case with concurrent preconditions. Preconditions serve to guarantee the verification of a condition at the beginning of the routine to which they are linked. In other words, for a precondition to make sense, it is necessary that between its verification and the execution of the body of the routine, the condition remains valid. Therefore, in addition to the possible conditional wait, it is also necessary to ensure that the concurrent objects associated with the concurrent precondition are reserved for exclusive use in that routine. In other words, in this situation, it is necessary to impose a conditional inter-object synchronism on these objects. The same applies to the class invariant at the beginning of the routine, but not to the postconditions or the invariant at the end of the routine[18].

5.15 Algorithmic selection by concurrent conditions

We can consider that the preconditions of a routine, as well as the invariant of the class, conditionally select the program expressed in the body of that routine, since it only makes sense to execute the body of the routine if these conditions are met. This is the underlying reason why it is necessary to guarantee the exclusive reservation of concurrent objects that may be associated with them. That is, inter-object synchronization.

It is very interesting to note that this axiomatic reasoning does not apply only to these assertions. In fact, the same is true for pure structured statements (page 16) that select algorithms by concurrent conditions, as is the case with conditional and repetitive statements. Figure 5.22 shows the axiomatic behavior that is expected in these two instructions[19].

Thus, these pure structured instructions will only be safe if, in this case too, the exclusive reservation (applicable throughout the instruction block) of any concurrent objects involved in the conditions expressed therein is imposed.

Another very interesting semantic aspect in all these synergistic effects is the fact that, unlike concurrent assertions, it makes no sense to associate a conditional wait action with these pure structured instructions. In fact, the concurrent conditions that may be involved in these instructions are not conditions of correctness (but rather of

[18] This reasoning also applies to other algorithmic assertions.

[19] We omit the repetitive instruction **repeat**... **until** because it trivially converts to a repetitive instruction of the type **while**.

```
if CONDITION then
  precondition CONDITION do
    COMMANDS
  end
end
```

```
while CONDITION do
  precondition CONDITION do
    COMMANDS
  end
end
```

Figure 5.22: Structured conditional and repetitive instructions.

algorithmic selection), so both possible values of the condition are essential for the correctness of the algorithm.

Thus, the behavior of exclusive reservation of objects should not be confused with that of conditional waiting required in concurrent assertions. This is despite the fact that in the case of preconditions, both behaviors are associated with them.

With this semantics associated with concurrent conditions [20] it is possible to simultaneously guarantee security and improve the expressiveness of the language, as well as optimize the concurrent object availability, since the exclusive reservation of concurrent objects will only be done when strictly necessary.

There are, of course, other ways to express inter-object synchronism. One of them consists of using the structured instruction presented in section 4.6.5. Another possibility is the one used in SCOOP (section A.6). However, none of these approaches (or any others), to be safe, avoids the need to ensure the exclusive reservation of objects in the use of concurrent preconditions and in selection and repetition statements that make use of concurrent conditions.

5.16 Inheritance (subclass relationship)

Interference between the inheritance mechanism (subclass) and the synchronization code of concurrent objects has been one of the most studied areas and one that has caused the most problems in the integration of concurrency mechanisms in object-oriented languages [America 87a, Briot 87, Kafura 89, Matsuoka 93]. The problems identified are basically related to the difficulty in reusing synchronization code, forcing it to be partially or even totally redefined. These problems have been designated as *inheritance anomalies* [Matsuoka 93], and there are numerous proposals to solve them [Matsuoka 93, McHale 94, Baquero 95, Holmes 99, Lu 01].

Despite the large number of publications referring directly to these inheritance anomalies, most do not define the term precisely. Holmes [Holmes 99, page 43], recognizing this difficulty, proposes a definition:

> Consider an object-oriented language with a particular inheritance mechanism and notations for providing concurrency and synchronization. If we use inheritance on a base class and find that introducing new methods requires redefining the methods of the base class or its synchronization, then we are faced with an inheritance anomaly problem.

[20] For which we have an article in development to be submitted for publication [OeS 06b].

In approaches that take an explicit approach to synchronization (page 53) it is natural for inheritance anomalies to arise. Since it is the direct responsibility of the programmer to construct a correct synchronization algorithm, that algorithm tends to be strongly linked to the class for which it is made, and may not adapt properly to the appearance of new services or redefinitions of existing services in subclasses. This strong link also makes it difficult to modify the synchronization scheme in subclasses.

On the other hand, an implicit approach to synchronization, as studied and proposed in this work, tends to be immune to these problems since the proper implementation of synchronization is done automatically by the compilation system. Abstract synchronization largely prevents this option from representing a loss of control and adjustment of the synchronization of concurrent objects.

5.17 Subtype polymorphism

Object-oriented approaches to concurrency that do not use the type system to identify the type entities associated with concurrent objects raise security issues. Since subtype relationships are imposed by the type system, under these conditions it is easy to pass sequential objects as if they were concurrent or vice versa, generally creating[21], serious problems of concurrent use of unsynchronized objects. We thus have another very strong reason (in addition to the one presented in section 5.2.1) for using the type system to separate concurrent objects from sequential ones.

5.17.1 Message passing communication model

In the message passing communication model (assuming an indirect processor identification mechanism as presented in section 5.6.2), there is no intra-object concurrency, and substitutability problems arise essentially when a concurrent object is associated with an entity of sequential type [Meyer 97, page 973]. In this situation, the program (and the compilation system) expects synchronous communication to and from the same processor and never potentially asynchronous remote communication. The reverse situation, associating a sequential object with an entity of concurrent type, may not be as critical, since communication with the same processor can be considered a special case (and therefore substitutable) of generic communication between one processor and another (for example, in SCOOP this situation is allowed).

5.17.2 Shared objects communication model

When we move to a model of communication between processors by shared objects, the situation is reversed (there is a duality between the two models). Here, the most unsafe situation is to pass a sequential (unsynchronized) object where a concurrent one is expected (i.e., in a context where there may be several processors trying to use the object). In this case, we would have problems of desynchronized competition in the use of the object with unpredictable consequences on the behavior of the program. The

[21] Depending on the communication model between processors and the implementation of synchronization in each language.

reverse situation of passing a concurrent object where a sequential one is expected may not be critical, since the use of a shared object by only one processor does not raise security issues.

5.17.3 Substitutability of intra-object synchronization schemes

An interesting aspect of the abstract intra-object synchronization approach proposed in this work is the total substitutability between concurrent objects (obviously related by subtype) with different intra-object synchronizations. As long as each concurrent object has a secure synchronization scheme associated with it, the object, from the point of view of its ADT, behaves externally in the same way regardless of the synchronization scheme used.

5.18 Exception mechanism

Exceptions essentially serve as a mechanism for internal signaling of failures in the operation of a program (section 3.13). They are an internal communication mechanism, just like routines, but with the difference that they abruptly interrupt the normal execution of programs and transfer execution to specific code to handle them.

In sequential languages, this communication always involves the same processor, and the objects involved can only be used by it. In a concurrent context, the situation can be quite different. On the one hand, exceptions may have to be delivered to a processor other than the one that was executing the code that triggered them. On the other hand, it may happen that a shared object is no longer available for concurrent uses due to an exception occurring in it. These situations concern, respectively, the communication model between processors by message passing and that of shared objects.

In this work, we are interested in studying in detail exception mechanisms closely linked to contract programming, that is, disciplined exception mechanisms (page 33).

There are several published works that analyze exception mechanisms in concurrency (for example: [Issarny 01, Xu 95, Mitchell 01]) but which omit the relationship, essential in the approach followed by object-oriented programming, with contract-based programming.

Recently [Arslan 06] an approach to this problem was proposed within the scope of SCOOP (model of communication between processors by message passing). However, the proposal made there has several problems, such as those we presented on page 71. In 2003, [OeS 04] presented a proposal for this integration within the scope of the prototype language MP-Eiffel. Although many of the aspects presented in that article remain, the current proposal differs in some respects (which we will clarify later).

A disciplined concurrent exception mechanism must take into account four aspects that we consider essential:

1. propagation of exceptions to the correct recipient;
2. concurrent object availability after the exceptions occur;
3. recovery of objects in unstable times;

4. termination of processors.

5.18.1 Propagation to the correct recipient

For the exception mechanism to make sense, it is essential to ensure that exceptions are handled, if the programmer so wishes, at the correct location, i.e., on the side of the party responsible for the failure. Programming by contract (section 3.12) distributes distinct responsibilities among the various parts of a program depending on the type of assertion involved (see table 3.1). Thus, a failure in a precondition is the responsibility of whoever invoked the service. Failures in the remaining assertions are the (internal) responsibility of the object to which the service belongs.

This requirement applies easily, by definition, to synchronous communication mechanisms between processors (between the processor that invokes a service and the processor that executes it), such as communication mechanisms by shared objects, or synchronous communication mechanisms by message passing. The problem becomes more complicated, as already explained on page 71, in the presence of asynchronous communication mechanisms by messages. In this situation, to maintain the contractual integrity of the exception mechanism, it is necessary to impose synchronous verification of the precondition (obviously, only its sequential part, if any). For the remaining assertions, it does not make sense to impose synchronous verification (it would make synchronous a communication that was intended to be asynchronous), but it is necessary to provide for the possibility that the object may not be able to resolve the cause that led to the occurrence of the exception (which was its responsibility), and, therefore, have to propagate the exception to whoever requested the execution of the service (indicating that it was not possible to fulfill its part of the contract). The semantics that seem to make the most sense to us consist of propagating the exception synchronously with the next attempt to use the object by the same processor, regardless of whether, in the meantime, the object has been recovered by other processors (section 5.18.3). This semantics differs from the proposal made in [OeS 04].

5.18.2 Concurrent object availability

The second important aspect (irrelevant in sequential languages) has to do with the concurrent object availability in which an exception was generated. It seems clear that in the case of execution in these objects being interrupted by an exception at an unstable time – and in which the object itself proved unable to resolve the problem and possibly also unable to restore its invariant – it cannot be allowed to be used later as if nothing had happened (a situation in which we would have a serious security problem, since the objects could be used without respecting their respective ADTs).

Obviously, the problem does not arise when a precondition fails. In this case, the object remains in a stable time and, as such, perfectly usable by any processor (including the one responsible for the precondition failure).

In the case of unresolved failures in other assertions, the object will have to be placed in a state of concurrent unavailability until it is eventually recovered. Any subsequent normal use of the object should result in the synchronous sending of an exception to the client (invariant failure).

5.18.3 Object recovery

The third aspect to consider refers to the sometimes necessary need for a mechanism to recover objects that are in a state of concurrent unavailability (this aspect can also be useful in sequential languages). In a concurrent context, it is important that this recovery can be performed by another processor that is not necessarily the one that triggered the sequence of actions that led to the failure, since that processor may no longer be running (for example, due to an inability to recover from the exception). This recovery will obviously have to go through the execution of some service of the class (in this case: procedure) but such an invocation cannot be done normally.

Meyer [Meyer 97, pages 417–418] maintains that when a routine fails, before the exception is propagated to the client, the invariant of the object *must* be restored. However, this requirement can hardly be guaranteed at runtime, since it could generate programs with infinite loops. Thus, in practice, it is possible for a routine to pass the exception to the client without ensuring that the object to which it belongs has its invariant intact. It is therefore useful to allow a processor, whenever it receives an invariant failure exception (and only in that case), to be able to handle exceptions in the code[22] directly invoke any of the procedures for creating the object (but, obviously, without resorting to the creation statement itself) before trying to use the object again[23].

This proposal is based on the following reasoning. Of all the services in a class, the only ones that are not required to check the invariant at the beginning of their execution are the procedures for creating objects [Meyer 97, page 370]. Furthermore, these procedures exist precisely to initialize objects to a state where the invariant is verified. Therefore, everything works together synergistically so that the creation procedures can also serve this very important purpose in concurrent programs (but which can also be useful in sequential programs).

It is important to note again that recovering an object, despite putting it in a stable time, does not prevent an exception from having to be propagated to the processor that triggered the actions that led to the object's failure. Only then will that processor be properly informed of the contract failure.

5.18.4 Exceptions and processor termination

The last problem we need to address is the complete relationship between exceptions and processors. In sequential languages, a program terminates by indicating a runtime failure when an exception reaches the top of the execution stack (i.e., when it reaches the routine where the program started). In concurrent languages, it also seems clear to us that, normally, a processor should terminate when an exception is propagated to its creation routine.

On the other hand, a concurrent program usually has several processors, each with an associated subprogram. It seems obvious to us that it would not be acceptable for the failure of one processor to result in the total failure of the program. It would be a

[22] In EIFFEL this will be in the **rescue** blocks.
[23] In EIFFEL with the **retry** instruction.

bit absurd, to make a simple analogy, for a failure in a juice machine at an airport to render it unavailable for any other use (such as traveling somewhere by plane).

In an object-oriented context, it is not the processors that are in charge: it is the objects. Thus, a concurrent program should only terminate completely if none of its processors are able to perform their task, or if there is a higher order for all of them to terminate (the latter case, more related to real-time programs, will not be addressed in this work).

In summary, exceptions along the way (when propagated from one side to the other) can leave objects unavailable (invariant failure), possibly recoverable later, and may even terminate the execution of processors.

5.19 Class services

Class services (section 3.16), especially attributes, directly interfere with concurrency mechanisms. If a class with this type of service has instances running on different processors (whether concurrent or not), then these services are shared by all these processors, requiring proper synchronization with an intra-class synchronization scheme (which includes all instances of the class). The interference of this mechanism can be even greater if class services are shared with descendant classes.

To deal with this problem, the language JAVA, in addition to a monitor per object, also has a monitor per class. It is the programmer's responsibility to use these synchronization schemes correctly.

Given the complexity of the interference that this mechanism seems to cause, and also because the EIFFEL language does not have this type of service, we chose not to include it in the prototype language.

5.20 Once execution services

Once execution services (section 3.17) can be adapted to concurrent languages. However, if these services are shared between multiple processors[24], the language compilation system must synchronize access to these services independently of the intra-object synchronization scheme (since these services may be shared by all objects that are instances of a class).

These services can be used in concurrent programs as another way for different processors to access references to concurrent objects.

5.21 Local-processor attributes

A mechanism that can be useful in concurrent programs is the possibility of declaring attributes local to processors[25]. The use of this type of attribute in shared objects would be completely safe, regardless of the intra-object synchronization scheme implemented.

[24] That is, if the service execution context includes the entire program.

[25] This mechanism has not yet been adopted in the prototype language developed, because we have not found a simple way to syntactically express it.

The basic idea behind this mechanism is very simple. Knowing that in the concurrent execution of objects, interference between processors is due to the fact that they act on a shared state of the object, why not allow objects to have specific states for each processor when it is important? In the POSIX-THREADS library [Butenhof 97], which adds concurrency to the procedural language C, there is what is known as thread local data, based on the same basic idea (although not adapted or applied to object-oriented languages).

If the definition of local attributes to processors is allowed in concurrent object-oriented languages, shared object services that only modify this special type of attribute will be, from the point of view of intra-object concurrency, equivalent to services that only observe the state of the object.

The implementation of caching[26] schemes in objects will be one of several interesting applications of this mechanism.

5.22 Summary of interference between mechanisms

Tables 5.2 and 5.3 summarize some of the negative and synergistic interference discussed in this chapter.

[26]For example, to temporarily store results of computationally heavy queries.

–	**Description:**	**Refs.:**
Modifiable public attributes – **ADT**	The existence of this type of attribute means that it is not only the object that is responsible for ensuring its invariance, forcing the propagation of internal synchronism to all clients that can modify it	(page 62)
Active objects – **ADT**	The choice of messages (services) to be accepted by the object may have nothing to do with its ADT, and may be made at unstable times for the object	(page 69)
Asynchronous communication **Wait by necessity** – **ADT** **Programming by contract**	Since the preconditions are assertions imposed on the clients of a service, in case of non-compliance, it is up to them to assume that responsibility. If the verification of this assertion is asynchronous, this important distribution of responsibilities is lost	(page 71)
Modifiable public attributes – **Conditional synchronization**	The existence of public attributes may require propagating the conditional synchronization code to all clients that can use them	(page 91)
Explicit synchronization – **Inheritance**	Designated as inheritance anomalies, these negative interferences derive from the impossibility – in this synchronization option – of reusing the inherited synchronization	(page 96)

Table 5.2: Some unsafe interferences between concurrent mechanisms.

+	**Description:**	**Refs.:**
Object creation procedure + **Creation of processors**	When justified, the creation of certain objects can also create new processors	(page 65)
Communication between objects + **Communication between processors**	Communication between objects can be reused as a mechanism for communication between processors, provided that each object belongs to a processor	(page 69)
Separation of commands and queries + **synchronous and asynchronous communication**	A command is typically a unidirectional sending of a message to an object, so it can easily be assigned asynchronous behavior. A query operation, on the other hand, is bidirectional, so it lends itself to synchronous behavior	(page 71)
Concurrent assertions + **Conditional synchronization**	In order to continue to make sense, an assertion that depends on a processor other than the one testing it must have conditional wait behavior	(page 94)
Algorithmic selection by concurrent conditions + **inter-object synchronization**	These instructions (which include conditional, repetitive, and preconditions) only make sense if the state of the objects involved in the condition becomes, from that moment on, dependent only on that processor	(page 95)
Object creation procedure + **Retrieve objects to a stable time**	Since object creation procedures are the only ones that, by definition, do not require the invariant to be verified at the the beginning of their execution, they can be reused in mechanisms for recovering objects in unstable times	(page 100)
Single execution services + **Sharing references to concurrent objects**	Single execution services can be reused to be a mechanism for sharing concurrent objects	(page 101)

Table 5.3: Some synergistic interference between concurrent mechanisms.

Chapter 6

The MP-Eiffel Language

6.1 Introduction

In the previous chapter, several approaches for integrating concurrent programming mechanisms into object-oriented languages were analyzed critically and in detail, and several choices were made in these approaches, with the appropriate justifications. This chapter presents a programming language – called MP-EIFFEL: Multi-Programming Eiffel – where these ideas are being applied and experimented with. Its main features are as follows:

- static safety[1];
- abstract processors;
- abstract synchronization of concurrent objects;
- automatic synchronization of concurrent objects;
- communication mechanisms between processors via messages and shared memory;
- static type system with concurrency annotations;
- concurrent exception mechanism (as described in section 5.18);
- concurrency control language for possible choice of processor realizations and intra-object synchronization schemes.

In the design of this language, it was decided to fully include the language EIFFEL [Meyer 92]. This choice resulted not only from the rigorous and careful approach that this language takes to object programming (undoubtedly the author's favorite), but also from the fact that it is practically the only language with appropriate mechanisms to support contract programming. Contract programming (section 3.12), is an essential tool with the aim of maximizing correctness in software, but also as a practical implementation of the ADTs (section 3.9) of each class. It is the author's opinion that

[1] Possible static safety problems of the EIFFEL language related to the covariance of types in the arguments of redefined routines are not addressed here, as they are outside the scope of this work.

object programming will always be an incomplete methodology if it does not consider contract programming.

This option also raised another interesting challenge: maximize the usefulness of existing modules in EIFFEL without limiting the concurrency potential of programs in MP-EIFFEL. In other words, we wanted it to be possible to use directly sequential classes (developed in EIFFEL) to create concurrent objects. This objective was achieved, with contributing to orthogonality and synergy in the integration of the concurrent mechanisms.

From a strictly syntactic point of view, MP-EIFFEL adds only three reserved words to EIFFEL: **shared**, **remote**, and **trigger**.

Like SCOOP, MP-EIFFEL takes an axiomatic approach axiomatic approach to defining concurrency mechanisms. Thus, the concurrent status of objects results directly from the semantics associated with each mechanism, and it is up to the compilation system to ensure the safe use of these mechanisms and their implementation. This approach differs from that followed in the JAVA language, where the programmer is called upon to assume – if not all – at least at least a significant part of the responsibility for ensuring correctness in the use of concurrent mechanisms. A clear example of this situation is the explicit use of the synchronization annotation **synchronized** in methods that require exclusive access to concurrent objects, or alternatively taking into account the complex memory model of the language [Lea 00, page 90].

One of the first difficulties in designing the language was selecting appropriate abstractions to implement the two models of communication between processors: message passing and shared objects. The first attempt in this direction was, naturally, to find orthogonal mechanisms for each one of the models. For the reasons presented in the previous chapter (section 5.2.1), it was decided from the outset to use annotations in the type system to identify concurrent objects.

Thus, in the case of the shared objects model, we decided to reuse the **shared** type annotation introduced by Brinch Hansen for monitors [BH 73, section 7.2].

An object of type **shared** will then be a concurrent object whose access, done in the same way as access to sequential objects, makes use of the communication model by shared objects.

In the case of the message passing model, the annotation used in SCOOP – **separate** – would be a possibility. However, in our opinion, this word does not adequately express the communication property that we wish to abstract. This property should be closer to the concept of remote service invocation underlying to this form of communication (section 5.6.2). The choice thus fell on the annotation **remote**.

At this point, several problems arose. First, although normal service invocation could also be used for this mechanism of communication between processors, this option did not seem correct to us, since the semantics of communication is very different (it can be asynchronous, section 5.6.3). Added to this problem was the convenience, as argued in section 5.8.1, of being able to define a different interface for the reception of messages originating from other processors. Finally, message communication between processors requires that remote objects be unambiguously associated with a single processor (receiver), so either we adopted an approach such as SCOOP in which there is a total separation between the objects of each processor – for which the use of shared

objects would seem a little forced – or an alternative semantics for remote objects had to be devised.

As will be seen later in this chapter, all these problems were solved – in our opinion, quite elegantly – by adding a new group of language abstractions (which are not type annotations) called triggers (in both senses that this word can have: a trigger mechanism or the action of triggering). This option also resulted in a very interesting synergistic effect, which This option also resulted in a very interesting synergistic effect, which was that remote entities could also be used in the shared objects communication model, but with the restriction that only pure query services could be used through them.

6.2 Shared objects communication

In MP-Eiffel, a normal invocation of a service applied to a concurrent entity (**shared** or **remote**) constitutes a communication between processors by shared objects (as justified in page 72).

6.2.1 Shared objects

Shared objects are concurrent objects that can – provided, of course, that the respective ADT is respected – be freely observed and modified by all processors that have access to them. This type of object does not belong to any particular processor (not even the processor responsible for its creation). In MP-Eiffel, these objects can only be referenced by entities with the type annotation **shared**. The rules for assigning values to entities with type ensure that a shared object can never be associated with an entity with a type that is not also shared (section 6.5).

Figure 6.1 shows an example of the application of shared objects. The shared objects (in this case, in principle, there will be only one) in this example are used to implement classes for logistical recording of internal program information.

Thus, we have a shared object of type `LOG_REGISTER` where all this information can be centrally recorded.

One of the important features of this language, which is easily seen in this small example, is the possibility of constructing classes without concurrency annotations (`LOG_REGISTER`) – that is, literally in Eiffel – without preventing future uses of concurrent instances of these classes (in this case, shared objects). In this way, concurrency annotations can be restricted only to where they are strictly necessary, enhancing the possibilities for class reuse and facilitating the understanding of programs. Note also that a single type annotation applied to a normal class is sufficient to have a concurrent object (without the burden of redundancies existing, for example, in SCOOP).

6.2.2 Remote objects

Like shared objects, remote objects are also concurrent objects, but they differ in two essential aspects: they belong to a (single) processor, and they can only be observed (without side effects) by the other processors that have access to it. The rules of the

```
class LOG_REGISTER

inherit
  LOG_USER;

creation
  make;

feature

  make(filename: STRING) is
    require
      not is_logging
    do
      ... -- open file handler
    end;

  start is
    do
      log(Current,"Starting logging...");
      is_logging := true
    end;

  stop is
    do
      log(Current,"Stopping logging...");
      is_logging := false
    end;

  is_logging: BOOLEAN;

feature

  log(source: LOG_USER;message: STRING)
    require
      source /= Void;
      message /= Void;
      is_logging
    do
      file.writeln_array_string(
        <<"[",current_date.to_string,
          "] ",source.id,": ",message>>);
    end;

  ...

end -- LOG_REGISTER
```

```
deferred class LOG_USER

feature

  id: STRING is
    do
      Result := class_name
    end;

end -- LOG_USER
```

```
class EXAMPLE_LOG

inherit
  LOG_USER

feature

  set_log_register(log_reg: shared LOG_REGISTER) is
    require
      log_reg /= Void
    do
      log_register := log_reg
    end;

  log_register: shared LOG_REGISTER;

  foo is
    do
      log_register.log(Current,"Hello world!");
    end;

end -- EXAMPLE_LOG
```

Figure 6.1: Shared objects example.

language's type system allow these objects to be referenced by other processors, besides their creator, but they statically prevent any attempt to modify these objects by remote processors (i.e., they only allow the invocation of pure queries).

Figure 6.2 shows an example of the application of these objects[2]. In the problem in question, there is a class – `EARTH` – where the state of some climate variables (in the example, the temperature value and the wind speed vector) can be accessed in real time (time is implicit in this example). On the other hand, there is also a class that abstracts an atmospheric station, which periodically collects this information from the `EARTH` class. Since the atmospheric station does not affect, nor can it affect the behavior of the `EARTH` class, and since there may be several stations collecting information, it makes perfect sense for these objects to have a remote reference to the instance of the `EARTH` class.

It is important to note that intra-object synchronization of remote objects is much less demanding than that of shared objects. A concurrent reader-writer synchronization scheme (section 5.10.5) gives full concurrent availability to these objects.

6.2.3 Synchronization

This language fully adopts the mechanisms and solutions described in the previous chapter for the various aspects of synchronization:

Intra-object synchronization: is abstract (section 5.9.1) and automatic (section 5.10), and the programmer can participate in choosing the synchronization scheme using a concurrency control language (section 6.7).

Inter-object synchronization: is performed when there is algorithmic selection by concurrent conditions, as described in section 5.15.

Conditional synchronization: is performed by concurrent assertions (section 5.14).

6.3 Communication by message passing: *Triggers*

The MP-Eiffel language implements communication by message passing between processors through a set of mechanisms called triggers, in which the identification of processors is indirect (section 5.6.2).

A trigger is a direct message between processors. For this communication to take place, it is necessary to have processors capable of receiving these messages (triggers, in the sense of triggers), and an appropriate instruction to send them (trigger, in the sense of triggering).

For a processor to receive messages from other processors, it must have associated objects whose classes explicitly declare some of their services as triggers. This association is made simply by creating these objects (which cannot be of type **shared** or **remote**) by the processor (these objects will then belong to the processor).

[2]The same problem can be better solved by using not only remote objects but also triggers, as will be seen later.

```
class ATMOSPHERIC_STATION

feature

  valid_longitude(long: REAL): BOOLEAN is
    do
      Result := long >= -180.0 and long <= 180.0
    end;

  valid_latitude(lat: REAL): BOOLEAN is
    do
      Result := lat >= -90.0 and lat <= 90.0
    end;

  valid_altitude(alt: REAL): BOOLEAN is
    do
      Result := alt >= 0.0
    end;

  longitude,latitude,altitude: REAL;

  set_position(long,lat,alt: REAL) is
    require
      valid_longitude(long);
      valid_latitude(lat);
      valid_altitude(alt);
    do
      longitude := long;
      latitude := lat;
      altitude := alt;
      position_defined := true
    end;

  position_defined: BOOLEAN;

  earth: remote EARTH;

  set_earth(the_earth: remote EARTH) is
    require
      the_earth /= Void
    do
      earth := the_earth
    end;

  earth_defined: BOOLEAN is
    do
      Result := earth /= Void
    end;

  start(sampling_period,num_iters: INTEGER) is
    require
      position_defined;
      earth_defined
    do
      from i := 1 until i > num_iters loop
        fetch_data;
        wait(sampling_period);
        i := i + 1
      end
    end;

end -- ATMOSPHERIC_STATION
```

```
class EARTH

feature

  temperature(long,lat,alt: REAL): REAL is
      -- real-time temperature value
    require
      valid_longitude(long);
      valid_latitude(lat);
      valid_altitude(alt);
    do
      ...
    end;

  wind_speed(long,lat,alt: REAL): VECTOR[REAL] is
    require
      valid_longitude(long);
      valid_latitude(lat);
      valid_altitude(alt);
    do
      ...
    end;

end -- EARTH
```

Figure 6.2: Example of using remote objects.

```
class C

trigger
  tick

feature

  tick is
    do
      ...
    end

end -- C
```

Figure 6.3: Example of a triggers declaration.

A trigger declaration is syntactically identical to the declaration of constructors in EIFFEL, with the difference that the reserved word used is **trigger** (and not **creation**). Figure 6.3 illustrates a declaration of triggers. Objects of class `C` (or descendants) can receive remote invocations to their service `tick`. The execution of these services is the responsibility of the processor that created the object.

Trigger sending is done through a trigger invocation instruction, which differs syntactically from a normal invocation of an object's services only in that this invocation is preceded by the reserved word **trigger**.

```
x: remote C;
...
trigger x.tick;
```

Since this communication model requires the unambiguous identification of the receiving processor, triggers only make sense if they are sent to remote objects (i.e., objects associated with entities of type remote). These are the only objects that can belong to processors other than the one sending the message and to which a processor is associated. Thus, the entity `x` in the given example must necessarily be remote.

Unlike the declaration of creation services, triggers are inherited in descendant classes, and even their names can be changed using the renaming mechanism of the EIFFEL [Meyer 92, page 81] language. Thus, subtype relationships are perfectly compatible with triggers, and there is no room (beyond the problem of covariance) for unsafe interference between the two mechanisms.

The language guarantees that no trigger is lost, and that they are normally served in order of arrival. This order can, however, be changed (by the message scheduling system of the receiving processor) as long as the sequential consistency of the messages is maintained (page 63).

In the future, it may be possible to set different priorities for different triggers through the concurrency control language, but these are adaptations of the mechanism

that have not yet been properly thought out.

Obviously, the receiving processors of triggers can only execute one of these messages at a time. This execution can only take place when the processor is available for execution, that is, when it is in a waiting state (section 6.4 describes the different phases of the lifetime of processors).

Figure 6.4 shows an implementation with triggers for the atmospheric stations problem presented above (page 109).

We can see that the introduction of triggers made it possible to easily separate the problem of periodic activation of atmospheric stations (done using a metronome) from the observation of the state of the object `earth`. This improves the quality of the solution and makes it easy to add new features (such as the stop service: `stop`).

6.3.1 Synchronous and asynchronous triggers

The execution of triggers can be either synchronous or asynchronous depending on the remotely requested service. Thus, as explained on page 71, the execution of query services will be synchronous, and the execution of commands will be asynchronous.

An interesting consequence of this difference in behavior results from the possibility, allowed in MP-Eiffel, of sending triggers to the processor itself. In the case of a query service, the result of the trigger instruction will not differ from the direct invocation of the service. In the case of command triggers, processing will only take place when the trigger is scheduled for execution in the processor's wait phase (which can be useful in some situations).

6.3.2 *Triggers* and information hiding

The interface of triggers is given – not by the **feature** clause where the service associated with it is declared and eventually implemented – but directly in the declaration clause of the triggers[3]. Thus, a service that is associated with a trigger has two distinct interfaces: one for normal clients of the respective object and another for triggers (see figure 6.5).

6.3.3 Formal arguments of *triggers*

Nothing prevents triggers from being linked to class routines that have formal arguments.

However, as can be easily seen, the types of these formal arguments will have to be subject to some restrictions: they are either of an expanded type or of a concurrent type (**shared** or **remote**).

It would not make sense to have a formal argument that is not expanded and not concurrent in a trigger, since this would imply that the remote processor that wanted to invoke this trigger would have to pass as an argument a reference to an object local to the receiving processor itself (which is impossible).

[3]In a manner perfectly similar and consistent with what happens with object constructors.

```
-- MP-Eiffel library class

remote class METRONOME

creation
  begin_ticking;

feature

  begin_ticking is -- new processor
    do ... end; -- ticks registered
                -- METRONOME_RECEIVER's

  stop_ticking is -- ends processor
    do ... end;

feature

  user_exists(user: remote METRONOME_RECEIVER):
      BOOLEAN is
    do ... end;

  start(user: remote METRONOME_RECEIVER;
        period: INTEGER) is
    do ... end;

  stop(user: remote METRONOME_RECEIVER) is
    require
      user_exists(user)
    do ... end;

end -- METRONOME
```

```
-- MP-Eiffel library class

deferred class METRONOME_RECEIVER

trigger
  tick

feature

  tick is
    deferred
    end;

end -- METRONOME_RECEIVER
```

```
class ATMOSPHERIC_STATION

inherit
  METRONOME_RECEIVER;

feature

  (...)
  -- valid_longitude, valid_latitude, valid_altitude
  -- longitude,latitude,altitude
  -- set_position, position_defined
  -- earth, set_earth, earth_defined

  metronome: remote METRONOME;

  set_metronome(the_metronome: remote METRONOME) is
    require
      the_metronome /= Void
    do
      metronome := the_metronome
    end;

  metronome_defined: BOOLEAN is
    do
      Result := metronome /= Void
    end;

  working: BOOLEAN;

  start(sampling_period: INTEGER) is
    require
      not working;
      position_defined;
      metronome_defined;
      earth_defined
    do
      working := true;
      trigger metronome.start(Current,
                              sampling_period)
    end;

  stop is
    require
      working;
    do
      working := false;
      trigger metronome.stop(Current)
    end;

feature {METRONOME}

  tick is
    do
      file.writeln_array_string(
        <<"Temperature at ",current_date.to_string,
          " is ",earth.temperature.to_string,".">>);
    end

end -- ATMOSPHERIC_STATION
```

Figure 6.4: Trigger example.

```
class C

trigger {X} -- only X descendants can trigger foo
  foo;

trigger     -- anyone can trigger bar
  bar;

feature {Y} -- only Y descendants can call foo

  foo is
    do
       ...
    end;

end -- C
```

Figure 6.5: Example of triggers declaration with information hiding.

If the argument is expanded[4] [Meyer 92, page 194], a full copy semantics of the object (pass-by-value) applies, so the problem does not arise.

6.4 Processors

In MP-Eiffel, processors are created implicitly whenever the object creation instruction is applied to a remote entity. This option is consistent with the semantics of remote objects since, by definition, they belong to another processor. Therefore, the creation of a remote object, being a command, will have to imply the prior creation of the processor that will execute it.

Normally, a processor exists from the moment it is created until the respective subprogram ends (selected creation service).

This does not happen, however, if the processors have triggers associated with them. In this situation, these processors will remain in a waiting state and can be awakened by the remote invocation of one of their triggers, or terminated when the program ends. Figure 6.6 shows the complete state diagram of a processor's lifetime.

6.5 Type system

The type system of MP-Eiffel is safe with regard to concurrency annotations[5]. The rules that guarantee this static safety are as follows. Let `x` be an entity with a type to which a value can be assigned (an attribute, a formal argument or a local

[4] Or to be more precise: completely expanded.

[5] There are still some holes inherited from the Eiffel language, as mentioned on page 23.

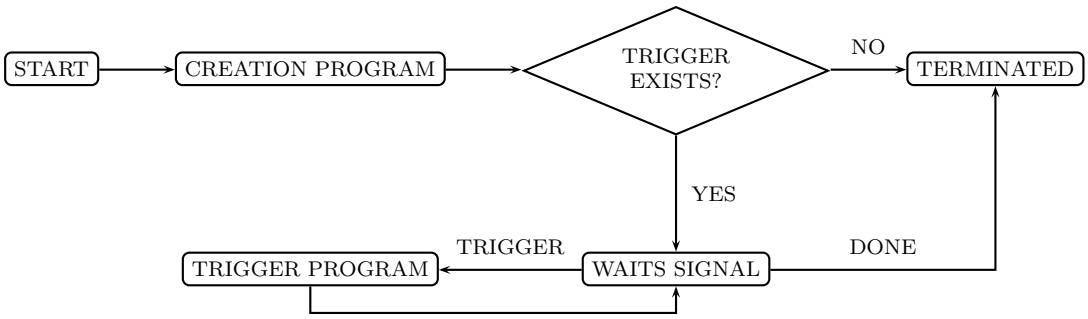

Figure 6.6: Processor lifetime.

variable), and `expr` an expression such that the type of `expr` conforms to the type of `x`. In MP-Eiffel, the expression `expr` can be assigned to `x`,

```
x := expr,
```

under one of the following conditions:

1. if `x` and `expr` are both shared; or both remote; or both without concurrency annotations;
2. if `x` is remote and `expr` has no concurrency annotations (objects that may be associated with `expr` become concurrent).
3. if `x` is expanded (provided that it does not contain, directly or indirectly, any attribute that is a reference).

As can be seen by comparing with the subtype restrictions mentioned in the previous chapter, section 5.17.2, there is an apparent contradiction with rule 2. In that section, it is mentioned (and rightly so) that it is unsafe to assign a sequential object to a concurrent entity, that is, exactly what rule 2 seems to propose. The problem is solved in MP-Eiffel by the compilation system. In fact, the object associated with `expr` cannot be sequential (it must be synchronized), and it is up to the compilation system to detect all such objects.

Unfortunately, this desirable behavior has not yet been implemented in the compiler of MP-Eiffel due to the complexity of the type system of the Eiffel language[6], an implementation is being made with another type annotation – **visible** – made specifically for this purpose. Thus, at this point, rule 2 is:

2. if `x` is remote and `expr` is visible.

6.6 Single-execution services

The MP-Eiffel language allows you to define five different execution contexts for single-execution services: program or processor, object or class, and free key. It is also

[6]In particular, the existence of "anchors" [Meyer 92, page 211] makes type checking by the compiler a little more complicated.

```
r_proc is
  once {processor}
    ...
  end

r_all is
  once
    ...
  end
```

```
r_proc_obj is
  once {processor,object}
    ...
  end

r_proc_all is
  once {object}
    ...
  end
```

Figure 6.7: Example of single execution services.

possible to combine these execution contexts, except for program–processor and object–class (which would be a contradiction in terms). By default, the execution context of these services is per processor and per class.

As mentioned in section 5.20, this type of service, if it includes the program as the execution context (i.e., sharing between all processors), require appropriate synchronization.

It is also necessary to prevent unsafe interference with other language mechanisms. In the case of MP-Eiffel, there may be unsafe interference with the type of entities that may be used in the invocation of these services (formal arguments and the result of functions).

The (static) security rule is simple. In the case of single-execution services where the execution context includes the entire program, only the types of formal arguments and, in the case of functions, the types of their respective results are allowed to be (completely) expanded or concurrent (shared or remote). This ensures that a non-concurrent reference can be visible to multiple processors. Execution contexts that do not include the entire program do not interfere in any way with the concurrency mechanisms (they work exactly as in sequential languages).

These services are also very useful in concurrent programs, as they provide another elegant way to access shared and remote objects. Figure 6.7 illustrates the declaration of some of these services.

6.7 Concurrency Control Language

One of the striking features of the MP-Eiffel language is that it relegates aspects related to particular implementations of concurrency mechanisms outside its programs, such as concurrency synchronization schemes or the assigning different priorities for access to shared resources.

To this end, a language is being designed to support the compilation system where these aspects can be defined and adapted to different execution contexts.

This language is called the Concurrency Control Language of MP-Eiffel (Concurrency Control Language): MP-Eiffel-CCL.

```
synchronize class X
  default: crw; -- concurrent readers-writer
  procedure_one,procedure_two,
  procedure_three: lockfree
end
```

```
synchronize local entity a
  at some_method in class X;
  default: exrw
end
```

Figure 6.8: Example of synchronization using MP-Eiffel-CCL.

In this language, for the choice of intra-object synchronization, there are four annotations reserved for each of the possible synchronization schemes: **monitor**, **exrw** (reader-writer exclusion), **crw** (concurrent reader-writer), **lockfree** (lock-free). These synchronization schemes can be specified either to classes as a whole or only to the entities through which objects are created. Figure 6.8 illustrates these two situations.

The specification of mixed synchronization schemes is done by declaring the services of the object to which one wants to associate specific types of synchronization. The compilation system of the MP-Eiffel language is responsible for verifying the validity and feasibility of the proposed specification.

Chapter 7

Conclusions

This work has taken a systematic approach to building concurrent object-oriented languages. In order to make the various choices that have been made in this process clear and objective, care has been taken to define quality criteria for languages (chapter 2). From the outset, the intention was to integrate concurrent programming into object-oriented languages (and not the other way around), which is why this type of programming was presented in detail in chapter 3. Recognizing that there are many variants of this type of language, to the extent that they may have important differences in the programming methods that apply to them, care was taken in that same chapter not only to identify some of these differences, but also to make clear the basic choices made for this work. Chapter 4 presents the requirements for concurrent programming. The chapter 5, where most of the contributions made were concentrated, takes a systematic, and largely objective approach to the integration of concurrency mechanisms in object-oriented languages (always giving priority to the object-oriented programming methodology). Finally, the chapter 6 presents a prototype language – MP-Eiffel – where all the choices and functionality described in chapter 5 have been integrated. With regard to the final result represented by this language, it is worth highlighting the static security, expressiveness and abstraction of the proposed mechanisms, as well as the high degree of synergistic integration achieved in many cases.

7.1 Contributions

The following contributions have been made to this work:

- Systematic and objective approach to integrating concurrent mechanisms into object-oriented languages (chapter 5);
- Abstract synchronization of concurrent objects (section 5.9.1);
- Automatic synchronization of concurrent objects (sections 5.10, 5.11 and 5.12);
- Mixed schemes for automatic intra-object synchronization (section 5.10.7);
- Solution for the automatic integration of intra-object synchronism and inter-object synchronism in concurrent objects (section 5.12.3);

- Safe behavior of concurrent assertions[1] (section 5.14);
- Proposal for synergistically and securely expressing inter-object synchronism (section 5.15);
- Disciplined mechanism for concurrent exceptions (section 5.18);
- Synergistic integration, in the language MP-Eiffel, of abstractions for both models of communication between processors (sections 6.2 and 6.3).

7.2 Future work

At the end of a piece of work like this we have the feeling that there is still a lot to be done, despite all that has achieved. Undoubtedly the most frustrating aspect (for those who assume they enjoy being an engineer) was the author's inability to have a complete and usable compilation system in a safe way for the prototype language proposed in the chapter 6. The completion of this system (which is not expected to take long) will be the main priority for future work, not least because the author is convinced that the interesting and powerful features of the language will make it a target of interest (if only to contribute to the emergence of other more expressive and secure concurrent languages).

The integration of real-time requirements into concurrent languages will be a second area that we hope to develop. The fact that there are very few linguistic approaches to this area of programming, and also the existence of a fast-growing research group in this field in the department to which I belong, make this challenge more interesting and with good prospects of being successful. On the other hand, the characteristics of the proposed approach seem to be an appropriate basis for integrating real-time mechanisms (we'll see if this is the case).

Finally, we intend to define the concurrency control language more precisely and implement it. This will fulfill of the objectives proposed with this approach (shared with the SCOOP approach) – abstract processors – and we can also facilitate the integration of real-time mechanisms.

[1] Although, in the author's opinion, this is just a generalization of Meyer's proposal for concurrent preconditions.

Appendix A

Introduction to the SCOOP language

The language SCOOP [Meyer 97, chapter 30] is a proposal to extend the EIFFEL language with concurrency mechanisms.

A.1 Explicit approach to concurrency

The type system is used for the explicit approach to concurrency taken in SCOOP. For this purpose, a type annotation has been added using the reserved word **separate**. In fact, this is the only reserved word added to the EIFFEL language, and this small syntactic difference is enough for the appearance of a very rich set of concurrency mechanisms.

A.2 Creating processors

To create a new processor, simply use the instruction to create a new object on, an entity declared as separate. This new processor will execute the creation service (if one is selected), and will then be available to execute object services in response to invocations from other processors.

A.3 Communication between processors

Communication between processors exclusively follows the message passing communication model. This communication is similar to the qualified invocation of object services, with the difference that the invocation applies to a separate entity.

```
x.f(y)
```

Thus, if the type of the entity `x` is **separate** and if a separate object is linked to that entity (SCOOP allows a separate entity to be linked to a non-separate object [Meyer 97, page 973]), the processor to which the current object (`Current`) belongs will be sending a message to the processor of the object linked to that entity.

The separate invocations rule [Meyer 97, page 985] means that only separate formal arguments can be used as the destination of separate invocations.

A.4 Abstract processors

Processors are not linked to a specific execution support. Therefore, using a concurrency control file [Meyer 97, page 971] it is possible to associate each existing processor in programs with an execution medium that is available.

A.5 Intra-object synchronization

In SCOOP, any object belongs to a single processor (although a processor can have many objects), and only the execution of the object's services on that processor is allowed. Thus, in its original proposal, intra-object concurrency is not allowed, with the object only being available to the processor that created it.

A.6 Inter-object synchronization

In its original proposal (semantics of separate invocations [Meyer 97, page 996]), all objects linked to separate formal arguments are reserved exclusively during the entire execution of the routine. This execution will, if necessary, be postponed (blocked) until this requirement is met.

A more recent proposal [Meyer 05, Nienaltowski 06a] imposes this semantics only on formal arguments that are linked[1] (the definition of linked entities can be found here [ECMA-367 05, page 75]). If the formal arguments are not linked, then there is no need to reserve any objects that may be referenced by these arguments.

A.7 Conditional synchronization

Conditional synchronization of objects is done using preconditions applied to separate formal arguments. In this situation, the reservation of objects depends not only on their availability but also on the verification of the separate precondition. Preconditions that involve separate formal arguments are thus called concurrent preconditions and their behavior is similar to a conditional wait.

[1] attached.

Appendix B

MP-Eiffel Language Implementation Considerations

This appendix discusses the solutions found to implement some of the compilation system's functionalities. Because the MP-Eiffel language has an axiomatic approach to concurrency, especially when it comes to automating the synchronization of concurrent objects, the implementation of the compilation system raised some problems that we think are sufficiently interesting to be presented here.

In any case, it should be noted that the solutions presented (and implemented) here are just one of several possible practical approaches, which essentially serve to demonstrate the feasibility of the proposed mechanisms, and to test the language prototype. Much work remains to be done, so that the compilation system can be considered usable for programming in MP-Eiffel.

B.1 Framework

Since this work is part of the study of concurrent mechanisms for object-oriented languages, which still require adequate practical experimentation, we decided to facilitate the implementation of the compilation system as much as possible by "minimizing" the time of its implementation, to the detriment of the compilation time and also – in certain cases – the execution time of the programs in MP-Eiffel.

One of the initial choices was to restrict the execution platform of the language compilation system to a single operating system – Linux – and a single concurrent execution support for processors – the POSIX-Threads library for the C language.

Another of the initial options was to simplify of the escolhendo compilation system as the target language which is closest to MP-Eiffel. Thus, the compilation system generates code in Eiffel. This code is then compiled using a public domain Eiffel compiler (SmallEiffel).

B.1.1 Thread-Safe SmallEiffel

The SmallEiffel compiler was born in 1995 as a project to implement an open-source version of a compiler from Eiffel[1]. In 2002, SmallEiffel was "abandoned" by its implementors in favor of a new line of compiler development, then called SmartEiffel (which was intended to implement the concurrency mechanisms proposed by Meyer in the SCOOP model).

As part of the work on this thesis, and since the code (C) generated by the compiler SmallEiffel was not safe to be compiled and run with the existing POSIX-Threads library on Linux, the author of this thesis in 2000 changed the compiler so that the code generated was safe. Likewise, a library was created at Eiffel to encapsulate the library POSIX-Threads. This "new" compiler has been designated by Thread-Safe SmallEiffel[2]. The appendix D contains a description of this library developed on top of SmallEiffel.

Following the appearance of SmartEiffel, and since it aims to implement the SCOOP model, decided not to adapt the safe version for this new line of development of the Eiffel compiler.

B.1.2 PCCTS

The construction of the MP-Eiffel compiler was based on a group of tools for generating lexical analyzers and syntax called PCCTS[3].

B.2 Detection of concurrent objects

One of the most difficult problems raised in the implementation of the MP-Eiffel compilation system consists of the location at compile time – without security flaws or an excess of false positives[4]. – of concurrent objects. The type system of MP-Eiffel was designed from the outset to not only make this problem possible, but also treatable. Only in this way is it possible for the compilation system to automatically and safely implement the synchronization of concurrent objects, without penalizing the implementation of the remaining sequential objects (which, in a normal program, will tend to be the vast majority).

As already mentioned in chapter 3, programs do not manipulate objects directly. These are created and used through entities with the type of that program, i.e. through: attributes, functions, local variables and arguments formal procedures or functions. Thus, in MP-Eiffel an object is concurrent if, and only if, it can be associated with a concurrent entity.

For an entity to be concurrent, it is a necessary condition that it is a reference, or contains directly or indirectly an attribute that is itself a reference. If an entity is completely expanded (as happens with some of the basic Eiffel types such as `INTEGER`,

[1] At the time there was no other free compiler

[2] It is in the public domain and is available at `http://www.ieeta.pt/~mos/thread-safe-se/index.html`

[3] Pardue Compiler Construction Tool Set.

[4] i.e. without annotating purely sequential objects as being concurrent

`REAL` and `BOOLEAN`), then – as the semantics of assigning value to these entities always implies copying the entire object – this new object will not be a competitor.

In addition to this condition, an entity will only be a competitor if one of the following situations occurs:

1. if the entity is shared (type **shared**);
2. if the entity is remote (type **remote**);
3. if it is a normal entity, and is passed as parameter of a remote formal argument of the procedure for creating a new processor;
4. if it is a normal entity, and is passed as parameter of a remote formal argument in the invocation of an trigger of another processor;
5. if it is a normal entity and is accessed, directly or indirectly, by passing a shared entity or remote.

The first two situations are, by definition, obvious: entities of the shared or remote type are competitors. The remaining situations are a little more complicated and are directly related to the semantics of remote objects. When a remote entity in the program of a processor is associated with an object, that object will (necessarily) be a normal object of another processor, and as such will probably be associated with normal entities in the program of that other processor. Because of this, those normal entities that exist in the program of the processor that owns that (concurrent) object, will also have to be concurrent entities (although, with the very important property of only being modifiable by a single processor, so their external behavior is semantically equivalent to sequential objects).

Since, by definition, a remote entity can only invoke services without side effects from the objects with which is associated, these entities cannot be directly defined by the programs of processors other than the processor to which they belong, i.e. in whose program they are declared. For example, the code shown in the figure B.1 – although at first glance it may seem correct – is not a valid program:

The error in this program lies in the invocation of a procedure via a remote entity.

In order for the program associated with the respective processor to be responsible for associating the remote entities, only has three possibilities:

- when the new processor is created;
- using triggers;
- via another remote object.

All these possibilities are in line with the semantics expected of remote entities. Passing the reference of a normal object when creating a new processor, consists of defining its initial execution state. Invoking an trigger, is formally equivalent to a remote invocation of a service, so nothing prevents this service from having collateral effects for the respective remote processor. Finally, using a pre-existing reference of a remote object to access references of other remote objects has no side effects on the processor that owns that remote object, so is a normal use of an object service.

```
-- assume this class to be part of
-- processor 1's program
class A_PROC1_CLASS

  -- ...

  proc2: remote A_PROC2_CLASS;

  abc is
    local
      obj: CLASS_X -- normal entity
    do
      create obj;
      proc2.def(obj); -- incorrect call!
    end

end -- A_PROC1_CLASS
```

```
-- assume this class to be part of
-- processor 2's program
class A_PROC2_CLASS

  -- ...

  remote_obj: remote CLASS_X;

  def(rem_obj: remote CLASS_X) is
    do
      remote_obj := rem_obj;
      -- ...
    end

end -- A_PROC2_CLASS
```

Figure B.1: Wrong program..

The detection of concurrent objects, and since there are only these three possibilities for associating remote entities with objects, is solved by propagating the concurrent property to all the normal entities that are used in these three situations.

The following program exemplifies the use of the procedure for creating a new processor to pass the reference of a remote object.

```
class A_PROCESSOR

creation
  make

feature{NONE}

  make(obj: remote CLASS_X) is
    do
      ...
    end;

end -- A_PROCESSOR
```

```
class SOMEWHERE

feature

  abc is
    local
      x: CLASS_X;
      proc: remote A_PROCESSOR;
    do
      ...
      create x;
      ...
      -- new processor with remote argument:
      create proc.make(x);
      ...
    end;

end -- SOMEWHERE
```

Thus, as the local variable `x` of the procedure `abc` of the class `SOMEWHERE` is passed as a parameter where a remote formal argument is expected, this entity becomes a competitor (all the objects it may be associated with will also be competitors).

This example – which in this aspect of detecting concurrent objects is no different from the use of triggers – is one of the simplest cases in checking whether normal entities are competitors since the entity is a local variable (its scope is restricted to the body of the procedure where is declared). The problem becomes more complicated if the normal entity is an attribute. In this case, that attribute can become a concurrent entity anywhere in the program of the respective class, or even outside it (if it is public). In the case where the attribute becomes a concurrent entity because it is assigned to a

remote entity somewhere in the program of the class itself – even though this problem is a little more complex than that of the local variable –, remains a decision local to the class. The second situation, on the other hand, is much more complex and means that the decision can no longer be made locally to the class (preventing a separate complete compilation for each class), forcing a global analysis of the program.

From the outset, we see no theoretical reason for not allowing this last situation (which is why it is allowed in the current definition of the language presented in chapter 6), however, its implementation is much more complex.

A solution to this problem, which we think is perfectly feasible, consists of, during the compilation phase, generating a (directed) graph with the relevant association relations between all the entities with a type in the program. So a normal entity is remote if and only if it can be assigned (as a parameter of a formal remote argument) to a remote entity.

However, it should be noted that currently the implementation of the MP-Eiffel compilation system does not take this situation into account, and it has been decided to simplify this problem (enormously) by introducing a new annotation to the type system, complementary to remote access, called visible (type **visible**). Thus, remote entities can only be dependent on visible entities.

The last situation results from the possibility of a concurrent object being able to give access to the value of its attributes and functions. The value of these attributes or functions is itself (in pure object-oriented languages) an object, so these objects, if they are not completely expanded, will naturally also have to be concurrent. This case, however, differs from the other in that it may require the sharing of the synchronism between the initial concurrent object (through which the reference of these other objects was obtained) and these objects.

Thus, in the currently implemented compilation system, an entity is concurrent if it is shared, remote, visible or, in none of these cases, if it is accessible through a concurrent entity.

B.2.1 Dependencies graph between entities

A program entity `x` is said to depend on another entity `y`, if there is a possibility that `y` will be assigned to `x`. In Eiffel this situation can only occur either through of the value assignment instructions (`x := ...y...` or `x ?= ...y...`), or if `x` is a formal argument of a routine, and `y` one of its current parameters.

Thus, a (normal) entity is a competitor if, and only if, it depends directly or indirectly on another competitor.

It is important to note that the size and complexity of this graph varies in proportion to the size of the program, more specifically of the respective number of entities with type, so its complexity does not grow exponentially with the program.

B.3 Detection of services without side effects

Another essential aspect of a safe implementation of remote objects (and also of the adoption of synchronization schemes with less containment), is the need for the

compilation system to detect – without fail – which services have no side effects on the visible state of the program[5]. The invocation of remote services will only be statically allowed in these cases.

From this declarative perspective, it doesn't make sense to allow the invocation of procedures on remote entities (already that these, by definition, are commands, and as such can change the state of the program). There are two possible exceptions to this rule, both of which will be studied in more depth at in the future. The first is the case of single execution services (mainly functions), since, even if they have side effects, these may not be considered as a result of the remote invocation, but only of the very semantics of these services. The result of the program is the same, regardless of the particular processor responsible for the first invocation of these services. The second case has to do with the possibility that there may be attributes local to each processor (section 5.21). Services that use this variety of attributes do not, at least in this respect, have side effects for the execution of the other processors and can therefore be considered pure in this respect. Apart from these two possible exceptions, there is the possibility of invoking attributes or functions.

The first case doesn't pose any major problems, since, again by definition, the (safe) observation of the state of attributes doesn't produce side effects.

In the case of functions, it is necessary for the compilation system to properly analyze the respective algorithm, as well as the algorithm of all the services used, whether from the object itself or from others.

The simplicity of the EIFFEL language, or not allowing the assignment of value to formal arguments of functions (which are read-only), and by only allowing the assignment of the value of attributes within the respective class, makes this problem tremendously easier. So the only elementary imperative instructions that are responsible for changing the state of objects are value assignment instructions. And even these are only important if they don't apply to local variables (since these, by themselves, don't affect the visible state of any object).

Since a function can invoke other functions (including itself), you can only be sure that a function is pure if its algorithm doesn't contain value assignments to attributes, and if it doesn't invoke any other function that isn't also pure.

In object-oriented languages, it is necessary to also take into account the possible existence of subtype polymorphism and dynamic routing (section 3.8). Thus, in qualified invocations to routines we take the conservative approach of checking that all routines that can be executed as a result of these mechanisms are also pure. Recursive routines (whether invoked directly in the routine or via other routines), do not pose any major problems since the compilation system keeps track of the routines for which it has already checked are pure.

B.3.1 Polymorphic invocations

With what has already been presented, it is possible to annotate all functions as having, or not having, side effects. However, one of the essential characteristics of

[5]The visible state of a program in a pure object-oriented language, is that given by the set of visible states of all its objects

```
class CLASS_X

creation
  make

...

end -- CLASS_X
```

```
class CLASS_X_PROCESSOR

inherit
  CLASS_X;
  PROCESSOR
    rename
      main as make
    end

end -- CLASS_X_PROCESSOR
```

Figure B.2: Processor realization.

object-oriented languages remains to be taken into account: polymorphism and dynamic routing. In fact, every time a service is invoked, you have to take into account which can, at runtime, be invoked different services (but with the same contract) from different classes. Therefore, all classes that are descendants of the type for which the service is invoked must be taken into account. All it takes is for one of the services of one of these classes not to be pure for the service where is invoked not to be pure either.

B.3.2 Service invocation paragraph

It is therefore necessary for the compilation system to create a (directed) graph, whose nodes will be all the services[6] of all the classes in the program, and whose links between the nodes are all the possible invocations (including, of course, all polymorphic invocations). This service invocation graph – as in the case of the dependency graph between entities – depends proportionally on the size of the program, so its complexity is manageable.

B.4 Processors

In this prototype of the MP-Eiffel language, we restricted the mapping of processors to threads within the same process on a single computer. Although other processor mappings can be made – processor mappings - such as processes on the same computer or on physically separate computers - could raise problems and very interesting and relevant experimental conditions, we decided to give priority to other aspects of the mechanisms. In the future, we hope to be able to extend the compilation system in this direction.

[6]Only the "lives". That is, those that can be used at runtime by the program

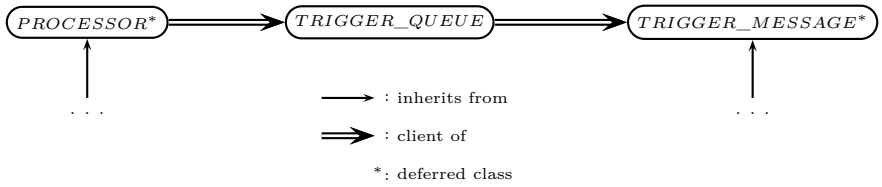

Figure B.3: Trigger implementation.

In the current prototype, the processors are implemented as descendant classes of a non-instantiatable class called `PROCESSOR`[7]. The compilation system, whenever there is the possibility of creating a remote entity (i.e. creating a new processor), generates a new descendant class either from the `PROCESSOR` class (the constructor used will be implemented as the redefinition of the processor program), or from the class associated with the entity.

The figure B.2 exemplifies this situation.

B.4.1 Program termination detection

A program in MP-Eiffel will be terminated when none of its processors are running (i.e. has either already terminated, or is in a state of waiting for triggers.

This behavior has been implemented in the class associated with the processors. The end of the program is detected by checking for the occurrence of two simultaneous conditions:

- if the number of waiting processors is equal to the number of existing processors;
- and if all the triggers message queues associated with each processor are empty.

Neither of the two conditions separately is sufficient to guarantee the total inactivity of all the processors in the program. It can happen that the number of processors is temporarily equal to the number of processors waiting, and there are still triggers to execute (since it is the processor itself that increments the counter of waiting processors, and in that interval it can send a new trigger). Also, of course, the fact that there are no triggers at a given instant does not invalidate the possibility that there are processors executing the respective programs.

B.5 Triggers

The implementation of this mechanism, as expected, proved to be much simpler than the mechanisms for communicating between processors by sharing memory.

In this implementation, the following aspects of this mechanism had to be taken into account:

[7]The source code can be consulted in the appendix E.1

- triggers have different semantics depending on whether they are associated with procedures or other services, and whether they behave asynchronously or synchronously, respectively. This affects not only the code to be associated with the program on the senders' side, but also that of the receivers, since exceptions behave quite differently in the two cases (see section 5.18).

- Any service can be associated with an trigger, so the mechanism must take into account the immense diversity between these services. In particular, it is advisable, when necessary, to find an efficient way of passing arguments to the triggers.

- The behavior of concurrent preconditions must be taken into account in this mechanism as well.

- A failure in a sequential precondition must be properly propagated to the processor issuing the respective trigger.

Another aspect that has also been taken into account, although it is relatively important, is the fact that a processor can only receive triggers if it creates at least one object that declares them (and also – although this aspect has not been considered – if the processor program makes the references of these objects available for remote access).

Figure B.3 shows the basic structure of the classes developed to generate the EIFFEL code to support the runtime implementation of this mechanism[8].

For each different trigger, the compilation system creates a new descendant class of the `TRIGGER_MESSAGE` class where all the important aspects for the subsequent execution of the trigger – namely, the connection to the service associated with the trigger, the passing of any arguments, and the identification of the issuing processor (without which it would not be possible to propagate possible exceptions) – are encapsulated. The class `TRIGGER_MESSAGE` (section E.2) contains the ADT common to all triggers and sufficient for their execution polymorphic in the class `PROCESSOR` (section E.1) class.

Each processor (instance of the `PROCESSOR` class) will be associated with an instance of the `TRIGGER_QUEUE` class, which implements a *FIFO* queue of triggers.

[8]The source code for these classes can be found in the appendix E

Appendix C

Implementation of synchronization schemes

C.1 Examples of realizing simple synchronization schemes

The code presented here is EIFFEL pure testable and has been compiled with the safe version THREAD-SAFE SMALLEIFFEL (appendix D).

C.1.1 Stack

```
-- Generic unbounded STACK class

deferred class STACK[E]

feature

  count: INTEGER is
      -- Number of elements
    deferred
    end;

  empty: BOOLEAN is
    do
      Result := count = 0
    end;

  top: E is
      -- STACK's last pushed element
    require
      not empty
    deferred
    ensure
      same_count: count = old count
    end;

  push(elem: like top) is
    deferred
    ensure
      one_more: count = old count + 1;
      element_placed_on_top: top = elem;
    end;

  pop is
    require
      not empty
    deferred
    ensure
      one_less: count = old count - 1
    end;

invariant

  count >= 0;
  empty = (count = 0)

end -- STACK
```

C.1.2 Stack: Monitor

```
class MONITOR_STACK[E]

  creation
    make

  feature {NONE}

    stack: STACK[E];
    mtx: MUTEX;
    cnd_var: CONDITION_VARIABLE;

  feature

    make(s: STACK[E]) is
      require
        s /= Void
      do
        stack := s;
        create mtx.make;
        create cnd_var.make
      end;

  feature

    count: INTEGER is
      do
        mtx.lock;
        Result := stack.count;
        mtx.unlock
      end;

    empty: BOOLEAN is
      do
        mtx.lock;
        Result := stack.empty;
        mtx.unlock
      end;

    top: E is
      do
        mtx.lock;
        from until not empty loop
          cnd_var.wait(mtx)
        end;
        Result := stack.top;
        mtx.unlock
      end;

    push(elem: like top) is
      do
        mtx.lock;
        stack.push(elem);
        mtx.unlock;
        cnd_var.broadcast
      end;

    pop is
      do
        mtx.lock;
        from until not empty loop
          cnd_var.wait(mtx)
        end;
        stack.pop;
        mtx.unlock;
        cnd_var.broadcast
      end;

end -- MONITOR_STACK
```

C.1.3 Stack: Readers-Writer Exclusion

```
class RW_EXCLUSION_STACK[E]

  creation
    make

  feature {NONE}

    stack: STACK[E];
    rwl: READ_WRITE_LOCK;
    mtx: MUTEX;
    cnd_var: CONDITION_VARIABLE;

  feature

    make(s: STACK[E]) is
      require
        s /= Void
      do
        stack := s;
        create rwl.make;
        create mtx.make;
        create cnd_var.make
      end;

  feature

    count: INTEGER is
      do
```

```
      rwl.read_lock;
      Result := stack.count;
      rwl.read_unlock
    end;

  empty: BOOLEAN is
    do
      rwl.read_lock;
      Result := stack.empty;
      rwl.read_unlock
    end;

  top: E is
    do
      rwl.read_lock;
      from until not empty loop
        rwl.read_unlock;
        mtx.lock;
        cnd_var.wait(mtx)
        mtx.unlock;
        rwl.read_lock;
      end;
      Result := stack.top;
      rwl.read_unlock
    end;

  push(elem: like top) is
    do
      rwl.write_lock;
      stack.push(elem);
      rwl.write_unlock;
      cnd_var.broadcast
    end;

  pop is
    do
      rwl.write_lock;
      from until not empty loop
        rwl.write_unlock;
        mtx.lock;
        cnd_var.wait(mtx)
        mtx.unlock;
        rwl.write_lock;
      end;
      stack.pop;
      rwl.write_unlock;
      cnd_var.broadcast
    end;

end -- RW_EXCLUSION_STACK
```

C.1.4 Stack: Concurrent Readers-Writer (Lamport)

```
class RW_CONCURRENT_LAMPORT_STACK[E]

  creation
    make

  feature {NONE}

    stack: STACK[E];
    mtx: MUTEX;
    writer_in,writer_out: INTEGER;
    cnd_var: CONDITION_VARIABLE;

  feature

    make(s: STACK[E]) is
      require
        s /= Void
      do
        stack := s;
        create mtx.make;
        create cnd_var.make
      end;

  feature

    count: INTEGER is
      local
        success: BOOLEAN;
        v: INTEGER
      do
        from until success loop
          v := writer_in;
          Result := stack.count;
          success := v = writer_out
        end;
      rescue
        if v /= writer_out then
          retry
        end
      end;

    empty: BOOLEAN is
      local
        success: BOOLEAN;
        v: INTEGER
      do
        from until success loop
          v := writer_in;
          Result := stack.empty;
          success := v = writer_out
        end;
      rescue
        if v /= writer_out then
          retry
```

```
    end
  end;

top: E is
  local
    success: BOOLEAN;
    v: INTEGER
  do
    from until success loop
      v := writer_in;
      from until not empty loop
        mtx.lock;
        cnd_var.wait(mtx)
        mtx.unlock;
      end;
      Result := stack.top;
      success := v = writer_out;
    end;
  rescue
    if v /= writer_out then
      retry
    end
  end;

push(elem: like top) is
  do
    mtx.lock;
    writer_in := writer_in + 1;
    stack.push(elem);
    writer_out := writer_out + 1;
    mtx.unlock;
    cnd_var.broadcast
  end;

pop is
  do
    mtx.lock;
    from until not empty loop
      cnd_var.wait(mtx)
    end;
    writer_in := writer_in + 1;
    stack.pop;
    writer_out := writer_out + 1;
    mtx.unlock;
    cnd_var.broadcast
  end;

end -- RW_CONCURRENT_LAMPORT_STACK
```

C.2 Example of non-blocking algorithms

Generic algorithms for this type of synchronization are based basically on three phases: a (stable) copy of the object's state is taken; the desired operation is applied to this copy; and finally, if the object has not been modified since the copy was made, the current state of the object is atomically replaced by this modified copy. The process is repeated until it is successful.

In the (desirable) case of separating object services into commands and queries, we can greatly simplify the algorithm. In fact, for the latter, the atomic substitution of the object's state is not necessary, it is enough for the operation to be successful to ensure that it is applied to a valid (stable) copy of the object.

Generic non-blocking algorithm for commands

```
1.  fail = true;
2.  do
    {
3.    obj_cpy.copy(obj);
4.    if (obj_cpy.copy_succeed(obj))
      {
5.      obj_cpy.command(...);
6.      fail = !obj.atomic_replace_on_linearizability(obj_cpy);
      }
    }
7.  while(fail);
```

Generic non-blocking algorithm for queries

```
1.  fail = true;
2.  do
    {
3.    obj_cpy.copy(obj);
4.    if (obj_cpy.copy_succeed(obj))
      {
5.      result = obj_cpy.query(...);
6.      fail = false;
      }
    }
7.  while(fail);
```

The previous algorithms, in pseudo-code type C++, exemplify possible approaches to automatic synchronization with a synchronization scheme. In both cases, a copy of `obj` is made to `obj_cpy` (`3.`), after which, if this has been successful (`4.`), the desired operation is invoked using the copy of the object (`5.`). In the case of commands, and if the linearizability is verified, the state of `obj` is replaced by that of `obj_cpy` (`6.`). If this substitution is not possible, the whole process is repeated (`7.`).

C.3 Checking the invariant in mixed synchronization schemes with concurrency

C.3.1 Implementation of the invariant check

```
#include <pthread.h>

typedef struct
{
  int counter;
  int done_start;
  int Result_start;
  int Result_end;
  pthread_mutex_t mtx;
  pthread_cond_t cnd;
} INVARIANT_SYNCH;
#define INVARIANT_SYNCH_INIT \
  {0,0,0,0,PTHREAD_MUTEX_INITIALIZER,PTHREAD_COND_INITIALIZER}

int command_test_invariant(int (*inv)(void *obj),void *obj,
                           INVARIANT_SYNCH *synch,int start_of_routine)
{
  int Result;

  pthread_mutex_lock(&synch->mtx);
  if (start_of_routine)
  {
    synch->counter++;
    if (!synch->done_start)
    {
        // Invariant checked only in the first routine
        // (except for creation command, instead of rechecking
        //  the invariant, we could reuse the last Result_end).
      synch->Result_start = (*inv)(obj);
```

```
      synch->done_start = 1;
    }
      // Invariant result reused for all concurrent routines
    Result = synch->Result_start;
  }
  else // end_of_routine
  {
    synch->counter--;
    if (synch->counter == 0)
    {
        // Invariant checked only in the last routine
      synch->done_start = 0;
      synch->Result_end = (*inv)(obj);
        // awake all waiting processors (barrier end)
      pthread_cond_broadcast(&synch->cnd);
    }
    else
    {
        // wait for the last routine
      while(synch->counter > 0)
        pthread_cond_wait(&synch->cnd,&synch->mtx);
    }
    Result = synch->Result_end;
  }
  pthread_mutex_unlock(&synch->mtx);

  return Result;
}

int query_test_invariant(int (*inv)(void *obj),void *obj,
                         INVARIANT_SYNCH *synch)
{
  int Result;

  pthread_mutex_lock(&synch->mtx);
    // fetch last invariant verification
  if (synch->done_start)
    Result = synch->Result_start;
  else
    Result = synch->Result_end;
  pthread_mutex_unlock(&synch->mtx);

  return Result;
}
```

C.3.2 Implementation of (pure) query services

```
1.    if (!query_test_invariant(...))
1.1.    raise_invariant_exception(...);
2.    if (!test_precondition(...))
2.1.    raise_precondition_exception(...);
3.    Result = execute_query_body(...);
4.    if (!test_postcondition(...))
4.1.    raise_postcondition_exception(...);
```

```
5.    if (!query_test_invariant(...))
5.1.    raise_invariant_exception(...);
```

C.3.3 Implementation of command services

```
1.    if (!command_test_invariant(... ,1))
1.1.    raise_invariant_exception(...);
2.    if (!test_precondition(...))
2.1.    raise_precondition_exception(...);
3.    execute_command_body(...);
4.    if (!test_postcondition(...))
4.1.    raise_postcondition_exception(...);
5.    if (!command_test_invariant(... ,0))
5.1.    raise_invariant_exception(...);
```

Appendix D

Thread-Safe SmallEiffel

When designing the MP-Eiffel compiler, it was decided to use a tool for generating parsers and scanners: PCCTS, and to implement all the code in Eiffel. For this to be possible, it was necessary for the Eiffel compiler used – SmallEiffel – to generate thread-safe code, so it was necessary to change the SmallEiffel compiler itself.

As part of this work, a thread-safe version of SmallEiffel (which has been placed in the public domain) was created, along with with a library of classes for handling threads.

This library is made up of the following classes:

- THREAD
- THREAD_CONTROL
- THREAD_ID
- MUTEX
- CONDITION_VARIABLE
- READ_WRITE_LOCK
- ONCE_MANAGER
- THREAD_BARRIER
- THREAD_PIPELINE
- THREAD_ATTRIBUTE
- GROUP_MUTEX

D.1 Classe THREAD

```
deferred class THREAD

inherit
  THREAD_CONTROL

feature {THREAD}

  main
      -- New thread starting point (main routine).
      -- Is not called directly, but in 'start*' routines
      -- The new thread start object will be 'Current'
      -- The thread terminates at the end of 'main'
    deferred
    end;

feature

  start
      -- start new thread
    require
      not is_expanded_type

  start_detached
      -- start new thread on detached (unjoinable) state

  start_with_name(n: STRING)
      -- start new thread named 'n'
    require
      not is_expanded_type;
      n /= Void

  start_detached_with_name(n: STRING)
      -- start new thread on detached (unjoinable) state,
      -- named 'n'
    require
      n /= Void

feature

  my_birth_id: THREAD_ID;

end -- THREAD
```

D.2 Classe THREAD_CONTROL

```
class THREAD_CONTROL

feature

  running: BOOLEAN

  detached: BOOLEAN

  is_same_thread(other: THREAD): BOOLEAN
      -- is the calling thread the same as the
      -- owner of 'other'?

  is_main_thread,is_root_thread: BOOLEAN
      -- are we in main (root) thread?

  thread_name_defined: BOOLEAN

  thread_name: STRING
    require
      thread_name_defined

  set_thread_name(n: STRING)
    require
      n /= Void

  detach
      -- detach current thread
    require
      running;
      not detached

  exit
      -- forces termination of current thread
    require
      running

  join(other: THREAD)
      -- The caller will block while 'other' thread is running
    require
      not other.detached;
      not other.running or else not is_same_thread(other)

  join_all_childs,join_all
      -- The caller will block while all direct child threads
      --  of the owner of current object are running
      -- Ignores detached direct childs.
      -- This feature is usable by the thread owning Current
      --  object (unlike 'join' feature).
      -- Returns immediately if there isn't any child.

feature

  thread_id: THREAD_ID

end -- THREAD_CONTROL
```

D.3 Classe THREAD_ID

```
class THREAD_ID

inherit
  THREAD_CONTROL

creation
  make

feature

  make
      -- fetches the id of the creation thread!

  same_as(other: like Current): BOOLEAN
    require
      other /= Void

end -- THREAD_ID
```

D.4 Classe MUTEX

```
class MUTEX

 creation
   make

 feature

   initialized: BOOLEAN

   make

   destroy
     -- destroys mutex

   lock

   try_lock: BOOLEAN
     -- on lock success returns true (false otherwise)

   unlock

end -- MUTEX
```

D.5 Classe CONDITION_VARIABLE

```
class CONDITION_VARIABLE

 creation
   make

 feature

   initialized: BOOLEAN

   make

   destroy
     -- destroys condition variable

   wait(m: MUTEX)
       -- m must be locked

   timedwait(m: MUTEX;timeout: INTEGER): BOOLEAN
     -- Returns false on timeout, and true if signaled
     --  timeout is the absolute time in seconds (relative
     --  to 00:00:00 GMT, January 1, 1970)
     -- Absolute time is used, instead of elapsed time,
     --  because of spurious wakenings (always possible
     --  with cond. variables).

   signal

   broadcast

end -- CONDITION_VARIABLE
```

D.6 Classe READ_WRITE_LOCK

```
class READ_WRITE_LOCK

 creation
   make,make_with_write_priority,make_with_read_priority

 feature

   make,make_with_write_priority

   make_with_read_priority

   destroy

   read_lock

   read_try_lock: BOOLEAN
     -- on lock success returns true (false otherwise)

   read_unlock

   write_lock

   write_try_lock: BOOLEAN
     -- on lock success returns true (false otherwise)

   write_unlock

   write_lock_priority: BOOLEAN

   read_lock_priority: BOOLEAN

end -- READ_WRITE_LOCK
```

D.7 Classe ONCE_MANAGER

```
expanded class ONCE_MANAGER

 feature

   refresh(key: STRING)
     require
       key /= Void

   refresh_some(key_list: ARRAY[STRING])
     require
       key_list /= Void

   refresh_all

end -- ONCE_MANAGER
```

D.8 Classe THREAD_BARRIER

```
class THREAD_BARRIER

 creation
   make,make_static

 feature

   make

   make_static(size: INTEGER)
     require
       size > 0

   terminated: BOOLEAN

   terminate

   release
       -- all waiting threads in barrier will be released.

   is_static: BOOLEAN
     -- is the size of the barrier fixed?

   set_number_of_threads(size: INTEGER)
     require
       is_static;
       size > 0;

   number_of_threads: INTEGER
       -- number of signed threads

   signed: BOOLEAN
       -- is calling thread already signed?
     require
       not is_static

   sign_on
       -- calling thread will be a new user of barrier
     require
       not is_static;
       not signed

   sign_off
       -- calling thread won't be a user of barrier anymore
     require
       not is_static;
       signed

   wait
       -- Calling thread will wait until 'number_of_threads'
       --  threads are waiting (then they will all unblock).
       -- On termination initializes new barrier (with the
       --  same threads if the barrier is dynamic)
     require
       is_static or else signed

end -- THREAD_BARRIER
```

D.9 Classe THREAD_PIPELINE

```
class THREAD_PIPELINE

 inherit
   THREAD_CONTROL

 creation
   make

 feature

   make

   add_thread(thr: THREAD)
       -- adds a new concurrent thread to current [last] "pipe".
     require
       thr /= Void

   empty_pipe: BOOLEAN
       -- is current pipe empty?

   new_pipe
       -- appends a new empty "pipe" to pipeline.
     require
       current_pipe_not_empty: not empty_pipe

   start
       -- starts pipeline thread execution.
       -- exits only on pipeline termination.

end -- THREAD_PIPELINE
```

D.10 Classe THREAD_ATTRIBUTE

```
expanded class THREAD_ATTRIBUTE[T]

 feature

   put(e: T)

   item: T

end -- THREAD_ATTRIBUTE
```

D.11 Classe GROUP_MUTEX

```
class GROUP_MUTEX

 creation
   make

 feature

   make(num_groups: INTEGER)
     require
       num_groups >= 2

   destroy
```

```
  number_of_groups: INTEGER

  lock(g: INTEGER)
    require
      g >= 1 and g <= number_of_groups

  try_lock(g: INTEGER): BOOLEAN
    -- on lock success returns true (false otherwise)
    require
      g >= 1 and g <= number_of_groups

  unlock(g: INTEGER)
    require
      g >= 1 and g <= number_of_groups

feature
  -- group priorities (default is by the number of the group,
  --  from the highest priority [group 1] to the lowest
  --  [group number_of_groups].

  highest_priority_group: INTEGER

  lowest_priority_group: INTEGER

  greater_than_group_priority(g1,g2:INTEGER): BOOLEAN
      -- priority(g1) > priority(g2) ?
    require
      g1 /= g2;
      g1 >= 1 and g1 <= number_of_groups;
      g2 >= 1 and g2 <= number_of_groups

  lower_than_group_priority(g1,g2:INTEGER): BOOLEAN
      -- priority(g1) < priority(g2) ?
    require
      g1 /= g2;
      g1 >= 1 and g1 <= number_of_groups;
      g2 >= 1 and g2 <= number_of_groups

  set_highest_priority(g: INTEGER)
      -- moves group g to highest priority (other groups
      --  maintain their relative ordering)
    require
      g >= 1 and g <= number_of_groups

  set_lowest_priority(g: INTEGER)
      -- moves group g to lowest priority (other groups
      --  maintain their relative ordering)
    require
      g >= 1 and g <= number_of_groups

  increase_group_priority(g: INTEGER)
    require
      (g >= 1 and g <= number_of_groups) and then
      g /= highest_priority_group

  decrease_group_priority(g: INTEGER)
    require
      (g >= 1 and g <= number_of_groups) and then
      g /= lowest_priority_group

  set_default_priorities

  print_priority_lock_list

invariant

  number_of_groups >= 2

end -- GROUP_MUTEX
```

Appendix E

Some classes supporting the compilation of MP-Eiffel

E.1 Classe PROCESSOR

```
deferred class PROCESSOR

inherit
  THREAD
    rename
      main as life
    end;

feature -- PROCESSOR main program

  main is
    deferred
    end;

feature

  life is
      -- processor (boring) life
      -- detection of no program activity not optimized!
    local
      msg: TRIGGER_MESSAGE
    do
      !!cnd_var.make;
      register_processor(Current);
      main;
      increment_waiting_processors;
      if program_with_no_activity then
        terminate_program
      else
        if not triggers_enabled then
            -- triggers might became enabled due to a
            -- sequential precondition failure response
            -- to a asynchronous trigger call
          mtx.lock;
          cnd_var.wait(mtx);
          mtx.unlock
        end;
        if triggers_enabled then
          from until trigger_queue.is_terminated loop
            msg := trigger_queue.fetch_trigger;
            decrement_waiting_processors;
            msg.execute_call;
            increment_waiting_processors;
            if program_with_no_activity then
              terminate_program
            end
          end
        end
      end
    end

feature
  -- exception (to be used when a trigger call is executed)
  precondition_failed: BOOLEAN;

  notify_precondition_failure is
    do
      precondition_failed := true
    end;

  reset_precondition_failure is
    do
      precondition_failed := false
    end;

feature -- triggers

  enable_triggers is
      -- to be called during main execution if an object
      --  with triggers is created by the processor.
    once {"object","processor"}
      global_mutex.lock;
      !!trigger_queue.make;
      triggers_enabled := true;
      cnd_var.signal;
      global_mutex.unlock
    end;

feature {NONE} -- triggers

  triggers_enabled: BOOLEAN; -- default is false

  trigger_queue: TRIGGER_QUEUE;

  mtx: MUTEX;

  cnd_var: CONDITION_VARIABLE;

feature {NONE} -- features shared by all processors!

  global_mutex: MUTEX is
    once {"class","program"}
      !!Result.make
    end;

  waiting_proc_ref: INTEGER_REF is
    once {"class","program"}
      !!Result
    end;

  waiting_processors: INTEGER is
    do
      global_mutex.lock;
      Result := waiting_proc_ref.item;
      global_mutex.unlock
    end;
```

```
  unlocked_increment_waiting_processors is
    do
      waiting_proc_ref.set_item(waiting_proc_ref.item+1);
      check
        waiting_proc_ref.item <= unlocked_number_of_processors
      end
    end;

  unlocked_decrement_waiting_processors is
    do
      waiting_proc_ref.set_item(waiting_proc_ref.item-1);
      check waiting_proc_ref.item >= 0 end
    end;

  increment_waiting_processors is
    do
      global_mutex.lock;
      unlocked_increment_waiting_processors;
      global_mutex.unlock
    end;

  decrement_waiting_processors is
    do
      global_mutex.lock;
      unlocked_decrement_waiting_processors;
      global_mutex.unlock
    end;

  terminate_program is
    do
      global_mutex.lock;
      from
        all_processors.start
      until
        all_processors.off
      loop
        if all_processors.item.triggers_enabled then
          all_processors.item.trigger_queue.terminate
        end;
        all_processors.item.cnd_var.signal;
        all_processors.forth
      end;
      global_mutex.unlock;
    end;

  program_with_no_activity: BOOLEAN is
      -- all trigger's queues empty and all processors waiting
    do
      global_mutex.lock;
      if unlocked_number_of_processors = waiting_proc_ref.item then
        from
          all_processors.start
        until
          all_processors.off or else
          (all_processors.item.triggers_enabled and then
           not all_processors.item.trigger_queue.is_empty)
        loop
          all_processors.forth
        end;
        Result := all_processors.off
      end;
      global_mutex.unlock;
    end;

  all_processors: DYNAMIC_LIST[PROCESSOR] is
    local
      factory: DYNAMIC_LIST_FACTORY[PROCESSOR]
    once {"class","program"}
      global_mutex.lock;
      !!factory;
      Result := factory.make_dynamic_list;
      global_mutex.unlock
    end;

  register_processor(p: PROCESSOR) is
    do
      global_mutex.lock;
      all_processors.append(p);
      global_mutex.unlock
    end;

  unlocked_number_of_processors: INTEGER is
    do
      Result := all_processors.count
    end;

end -- PROCESSOR
```

E.2 Classe TRIGGER_MESSAGE

```
deferred class TRIGGER_MESSAGE
--    A new class is created by the compiling system for
--  each possible trigger message. That class will include
--  all the required actual arguments necessary to execute
--  the call (actual_call). The compiling system implements
--  appropriately the deferred routines.

feature

  actual_call is
    deferred
    end;

  execute_call is
    local
      precond_fail: SEQUENTIAL_PRECONDITION_FAILURE
    do
      if not sequential_precondition then
        if is_synchronous then
            -- precondition failure is propagated to the caller,
            -- without affecting the callee
          caller.notify_precondition_failure
        else
          !!precond_fail;
          caller.enable_triggers;
          caller.trigger_queue.enqueue_trigger(precond_fail)
        end
      else
        wait_for_concurrent_precondition;
        actual_call
      end;
    end

  is_asynchronous: BOOLEAN is
      -- true is procedure call
      -- (redefined to the appropriate constant boolean value)
    deferred
    end;

  is_synchronous: BOOLEAN is
      -- true is valued feature call
      -- (redefined to the appropriate constant boolean value)
    deferred
    end;

  sequential_precondition: BOOLEAN is
    deferred
    end;

  wait_for_concurrent_precondition is
    deferred
    end;

  caller: PROCESSOR;

  set_caller(p: PROCESSOR) is
    do
      caller := p
    end;

end -- TRIGGER_MESSAGE
```

E.3 Classe TRIGGER_QUEUE

```
class TRIGGER_QUEUE

creation
  make

feature

  make is
    local
      factory: QUEUE_FACTORY[TRIGGER_MESSAGE];
    do
      !!mtx.make;
      !!cnd_var.make;
      !!factory;
      queue := factory.make_queue
    end;

  enqueue_trigger(tm: TRIGGER_MESSAGE) is
    require
      tm /= Void
    do
      mtx.lock;
      queue.enqueue(tm);
      cnd_var.signal;
      mtx.unlock;
    end;

  fetch_trigger: TRIGGER_MESSAGE is
    do
      mtx.lock;
      from until terminated or else not queue.empty loop
        cnd_var.wait(mtx);
      end;
      if not terminated then
        Result := queue.tail;
        queue.dequeue;
      end;
      mtx.unlock;
    end;

  is_empty: BOOLEAN is
    do
      mtx.lock;
      Result := queue.empty;
      mtx.unlock
    end;

  is_terminated: BOOLEAN is
    do
      mtx.lock;
      Result := terminated;
      mtx.unlock
    end;

  terminate is
    do
      mtx.lock;
      terminated := true;
      cnd_var.signal;
      mtx.unlock
    end;

feature {NONE}

  terminated: BOOLEAN;

  mtx: MUTEX;

  cnd_var: CONDITION_VARIABLE;

  queue: QUEUE[TRIGGER_MESSAGE];

end -- TRIGGER_QUEUE
```

E.4 Classe SEQUENTIAL_PRECONDITION_FAILURE

```
class SEQUENTIAL_PRECONDITION_FAILURE

inherit
  TRIGGER_MESSAGE

feature

  actual_call is
    require
      false
    do
    end;

  is_asynchronous: BOOLEAN is true;

  is_synchronous: BOOLEAN is false;

  sequential_precondition: BOOLEAN is true;

  wait_for_concurrent_precondition is
    do
    end;

end -- SEQUENTIAL_PRECONDITION_FAILURE
```

Glossary

Assertion: Boolean condition (predicate) to be checked at that point in the program so that it is not incorrect.

Concurrent assertion: Assertion with a concurrent condition.

Class assertion: Invariants, preconditions and postconditions.

Formal assertion: Part of an assertion that can be executed by the program.

Informal assertion: Part of an assertion not executable by the program.

Attribute: Record of information belonging to objects.

Trash collector: Automatic memory management method.

Command: Object modification service (procedure).

Concurrent condition: Predicate that may depend on another processor than the one testing it.

Query: Object observation service (function or attribute).

Typed entities: Syntactic elements of a language that are associated with a "type".

Scheduling: Strategy for selecting processors to run.

Threads: Concurrent processing units based on sharing memory and other operating system resources between them. They are characterized by minimizing the context switching required for scheduling different *threads* and being part of a single operating system process.

Pure structured instructions: Instructions whose semantics are explicitly defined "outside-in". They allow the composition and decomposition of algorithms by nested blocks.

Imperative languages: Languages whose algorithm is expressed as a sequence of commands that can explicitly modify the state of the system.

Pure object-oriented languages: Languages whose programs are composed only of objects.

Method: Routine.

Concurrent object: Object usable by more than one processor.

Subtype polymorphism (of inclusion): Mechanism that allows objects to be associated with an entity, as long as the types of the objects are subtypes of the entity's type.

Parametric polymorphism: Mechanism that allows classes to be specified according to generic types.

Polymorphism *ad-hoc*: Mechanism that allows the definition of different services with the same name, as long as they have a different static signature.

Abstract processor: Abstract notion of processor with no connection to any specific execution support.

Writer processor: Processor while executing impure commands or queries.

Reader processor: Processor while executing pure queries.

Processor: Autonomous processing unit capable of supporting sequential execution of instructions.

Heterogeneous processing: When processors can be associated with different execution media.

Homogeneous processing: When processors can only be associated with one execution medium.

Process: Concurrent processing unit of operating systems. They are characterized by low cohesion between different processes (unlike *threads*.)

Routine: Function or procedure of a class.

Abstract service: Service without implementation (only represented by its interface).

Class service: Service shared by all instances of a class.

Single execution service: Services executed only the first time they are invoked.

Feature: Routine or attribute of a class.

Conditional synchronization: Synchronization that makes the use of objects conditional on certain conditions being met.

Inter-object synchronism: Synchronism that allows several exclusive uses of one or more concurrent objects.

Intra-object synchronization: Synchronization that protects the internal services of a concurrent object from each other.

Program execution support system: The set formed by the hardware and the operating system(s) of the computer system where the program is executed.

Concurrent programming systems: Systems that support concurrent programming, either through *software* libraries, concurrent languages, or a mixture of both.

SMP (Symmetric MultiProcessing): Computer architecture based on multiple central processing units operating with shared memory.

Subclass: Descendant class.

Subtype: A `A` class is a subtype of a `B` class, if the instances of `A` can be used in entities of type `B`.

Superclass: Upperclass.

Supertype: Inverse subtype relationship.

ADT (Abstract Data Type): Abstract Data Type.

Bibliography

[Ada95 95] *Ada 95 Reference Manual (Language and Standard Libraries).* U.S. Government, 1995.

[Agha 86] G. A. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, Massachusetts, 1986.

[Agha 99] G. A. Agha and W. Kim, "Actors: A unifying model for parallel and distributed computing", *Journal of Systems Architecture*, 45(15), September 1999.

[America 87a] P. America, "Inheritance and subtyping in a parallel object-oriented language". In *European conference on object-oriented programming on ECOOP '87*, pages 234–242, Springer-Verlag, London, UK, 1987.

[America 87b] P. America, "Pool-t: A parallel object-oriented language". In A. Yonezawa and M. Tokoro, eds., *Object-Oriented Concurrent Programming*, pages 199–220, MIT Press, 1987.

[Anderson 97] J. H. Anderson, R. Jain, and S. Ramamurthy, "Wait-free object-sharing schemes for real-time uniprocessors and multiprocessors". In *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS '97)*, pages 111–122, December 1997.

[Andrews 83] G. R. Andrews and F. B. Schneider, "Concepts and notations for concurrent programming", *ACM Comput. Surv.*, 15(1):3–43, 1983.

[Arslan 06] V. Arslan and B. Meyer, "Asynchronous exceptions in concurrent object-oriented programming". In *Proceedings of the first Symposium on concurrency, Real-Time, and Distribution in Eiffel-Like Languages, CORDIE'06*, pages 62–70, University of York – Department of Computer Science, July 2006.

[Baquero 95] C. Baquero, R. Oliveira, and F. Moura, "Integration of concurrency control in a language with subtyping and subclassing". In *Proceedings of the USENIX Conference on Object-Oriented Technologies (COOTS'95)*, pages 173–184, USENIX Association, June 1995.

[BH 72] P. Brinch Hansen, "Structured multiprogramming", *Communications of the ACM*, 15(7):574–578, 1972.

[BH 73] P. Brinch Hansen, *Operating System Principles.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1973.

[BH 75] P. Brinch Hansen, "The programming language concurrent pascal.", *IEEE Trans. Software Eng.*, 1(2):199–207, 1975.

[BH 93] P. Brinch Hansen, "Monitors and concurrent pascal: a personal history". In *The second ACM SIGPLAN conference on History of programming languages*, pages 1–35, ACM Press, 1993.

[BH 99] P. Brinch Hansen, "Java's insecure parallelism", *ACM SIGPLAN Notices*, 34(4):38–45, 1999.

[Bobrow 88] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon, "Common lisp object system specification", *SIGPLAN Not.*, 23(SI):1–142, 1988.

[Bohm 66] C. Böhm and G. Jacopini, "Flow diagrams, turing machines and languages with only two formation rules", *Communications of the ACM*, 9(5):366–371, 1966.

[Borning 86] A. H. Borning, "Classes versus prototypes in object-oriented languages". In *ACM '86: Proceedings of 1986 ACM Fall joint computer conference*, pages 36–40, IEEE Computer Society Press, Los Alamitos, CA, USA, 1986.

[Briot 87] J.-P. Briot and A. Yonezawa, "Inheritance and synchronization in concurrent oop". In *European conference on object-oriented programming on ECOOP '87*, Springer-Verlag, London, UK, 1987.

[Briot 98] J.-P. Briot, R. Guerraoui, and K.-P. Lohr, "Concurrency and distribution in object-oriented programming", *ACM Computing Surveys (CSUR)*, 30(3):291–329, 1998.

[Bruce 02] K. B. Bruce, *Foundations of Object-Oriented Languages – Types and Semantics*. The MIT Press, Cambridge, Massachusetts, 2002.

[Bruce 93] K. B. Bruce, "Safe type checking in a statically-typed object-oriented programming language". In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 285–298, ACM Press, 1993.

[Butenhof 97] D. R. Butenhof, *Programming with POSIX Threads*. Addison-Wesley, 1997.

[Canning 89] P. Canning, W. Cook, W. Hill, W. Olthoff, and J. C. Mitchell, "F-bounded polymorphism for object-oriented programming". In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 273–280, ACM Press, New York, NY, USA, 1989.

[Cardelli 85] L. Cardelli and P. Wegner, "On understanding types, data abstraction, and polymorphism", *ACM Computing Surveys (CSUR)*, 17(4):471–523, 1985.

[Cardelli 88] L. Cardelli, "Structural subtyping and the notion of power type". In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 70–79, ACM Press, 1988.

[Caromel 89] D. Caromel, "Service, Asynchrony, and Wait-by-Necessity", *Journal of Object-Oriented Programming*, 2(4):12–18, 1989.

[Caromel 93] D. Caromel, "Toward a method of object-oriented concurrent programming", *Communications of the ACM*, 36(9):90–102, 1993.

[Chambers 04] C. Chambers and T. C. Group, *The Cecil Language: Specification & Rationale*. Technical Report, Department of Computer Science and Engineering, University of Washington, Feb 2004.

[Coffman 71] E. G. Coffman, M. Elphick, and A. Shoshani, "System deadlocks", *ACM Computing Surveys (CSUR)*, 3(2):67–78, 1971.

[Conway 63] M. E. Conway, "A multiprocessor system design". In *Conference Proceedings 1963 FJCC*, pages 139–146, AFIPS Press, 1963.

[Cook 90] W. R. Cook, W. Hill, and P. S. Canning, "Inheritance is not subtyping". In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 125–135, ACM Press, 1990.

[Courtois 71] P. J. Courtois, F. Heymans, and D. L. Parnas, "Concurrent control with "readers" and "writers"", *Communications of the ACM*, 14(10):667–668, 1971.

[Dahl 68] O.-J. Dahl, B. Myhrhaug, and K. Nygaard, "Some features of the simula 67 language". In *Proceedings of the second conference on Applications of simulations*, pages 29–31, 1968.

[Dennis 66] J. B. Dennis and E. C. V. Horn, "Programming semantics for multiprogrammed computations", *Commun. ACM*, 9(3):143–155, 1966.

[Dijkstra 68a] E. W. Dijkstra, *Cooperating Sequential Processes. Programming Languages*, Academic Press, New York, 1968.

[Dijkstra 68b] E. W. Dijkstra, "Cooperating sequential processes". 1968. published as [Dijkstra 68a].

[Dijkstra 68c] E. W. Dijkstra, "Letters to the editor: go to statement considered harmful", *Communications of the ACM*, 11(3):147–148, 1968.

[Dijkstra 72] E. W. Dijkstra, "Notes on structured programming". In O.-J. Dahl, E. W. Dijkstra, and C. Hoare, eds., *Structured Programming*, pages 1–82, Academic Press, London and New York, 1972.

[ECMA-367 05] "Eiffel analysis, design and programming language". Jun 2005. ECMA-367 Standard.

[Floyd 67] R. W. Floyd, "Assigning meanings to programs". In J. T. Schwartz, ed., *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics 19*, pages 19–32, American Mathematical Society, Providence, 1967.

[Forum 94] M. P. I. Forum, "MPI: A message-passing interface standard", *International Journal of Supercomputer Applications*, 8(UT-CS-94-230):165–414, 1994.

[Geist 94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge Massachusetts, 1994.

[Ghezzi 91] C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of Software Engineering*. Prentice-Hall, 1991.

[Goldberg 89] A. Goldberg and D. Robson, *Smalltalk-80: The Language*. Addison-Wesley, 1989.

[Gosling 05] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification*. Addison-Wesley, third edition, 2005.

[Gosling 96] J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*. Addison-Wesley, first edition, 1996.

[Gries 81] D. Gries, *The Science of Programming. Texts and Monographs in Computer Science*, Springer-Verlag, 1981.

[Guttag 77] J. Guttag, "Abstract data types and the development of data structures", *Commun. ACM*, 20(6):396–404, 1977.

[Habermann 69] A. N. Habermann, "Prevention of system deadlocks", *Communications of the ACM*, 12(7):373–377, 1969.

[Harris 03] T. Harris and K. Fraser, "Language support for lightweight transactions". In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 388–402, ACM Press, 2003.

[Herlihy 03] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer, "Software transactional memory for dynamic-sized data structures". In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101, ACM Press, 2003.

[Herlihy 87] M. P. Herlihy and J. M. Wing, "Axioms for concurrent objects". In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 13–26, ACM Press, 1987.

[Herlihy 90a] M. Herlihy, "A methodology for implementing highly concurrent data structures". In *PPOPP '90: Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, pages 197–206, ACM Press, 1990.

[Herlihy 90b] M. P. Herlihy and J. M. Wing, "Linearizability: a correctness condition for concurrent objects", *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[Herlihy 91] M. Herlihy, "Wait-free synchronization", *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

[Herlihy 93] M. Herlihy, "A methodology for implementing highly concurrent data objects", *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(5):745–770, 1993.

[Hoare 69] C. A. R. Hoare, "An axiomatic basis for computer programming", *Communications of the ACM*, 12(10):576–580, 1969.

[Hoare 73] C. A. R. Hoare, *Hints on Programming Language Design.* Technical Report STAN-CS-73-403, Stanford Artificial Intelligence Laboratory, Computer Science Department, Stanford University, 1973.

[Hoare 74] C. A. R. Hoare, "Monitors: an operating system structuring concept", *Communications of the ACM*, 17(10):549–557, 1974.

[Hoare 78] C. A. R. Hoare, "Communicating sequential processes", *Communications of the ACM*, 21(8):666–677, 1978.

[Holmes 97] D. Holmes, J. Noble, and J. Potter, "Aspects of synchronization". In *TOOLS '97: Proceedings of the Technology of Object-Oriented Languages and Systems - Tools-25*, page 2, IEEE Computer Society, Washington, DC, USA, 1997.

[Holmes 98] D. Holmes, J. Noble, and J. Potter, "Toward reusable synchronisation for object-oriented languages". In *ECOOP '98: Workshop ion on Object-Oriented Technology*, page 439, Springer-Verlag, London, UK, 1998.

[Holmes 99] D. Holmes, *Synchronization Rings – Composable Synchronization for Object-Oriented Systems.* PhD thesis, Macquarie University, Sydney, Sydney, Australia, 1999.

[Issarny 01] V. Issarny, "Concurrent exception handling", *Lecture Notes in Computer Science*, 111–127, 2001.

[Joung 00] Y.-J. Joung, "Asynchronous group mutual exclusion", *Distributed Computing*, 13(4):189–206, 2000.

[Kafura 89] D. G. Kafura and K. H. Lee, "Inheritance in actor based concurrent object-oriented languages". In *Proceedings of the Third European Conference on Object-Oriented Programming*, July 1989.

[Knuth 74] D. E. Knuth, "Structured programming with go to statements", *ACM Computing Surveys (CSUR)*, 6(4):261–301, 1974.

[Lamport 77] L. Lamport, "Concurrent reading and writing", *Communications of the ACM*, 20(11):806–811, 1977.

[Lamport 79] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs", *IEEE Transactions on Computers*, C-28(9):690–691, 1979.

[Lamport 83] L. Lamport, "Specifying concurrent program modules", *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(2):190–222, 1983.

[Lauer 78] H. C. Lauer and R. M. Needham, "On the duality of operating system structures". In *Proceedings of the Second International Symposium on Operating Systems*, October 1978. reprinted in Operating Systems Review, Vol. 13, No. 2, April 1979, pp. 3-19.

[Lea 00] D. Lea, *Concurrent Programming in Java.* Addison-Wesley, second edition, 2000.

[Lieberman 86] H. Lieberman, "Using prototypical objects to implement shared behavior in object-oriented systems". In *OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 214–223, ACM Press, New York, NY, USA, 1986.

[Liskov 74] B. Liskov and S. Zilles, "Programming with abstract data types". In *Proceedings of the ACM SIGPLAN symposium on Very high level languages*, pages 50–59, 1974.

[Liskov 77] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert, "Abstraction mechanisms in clu", *Communications of the ACM*, 20(8):564–576, 1977.

[Liskov 86] B. Liskov and J. Guttag, *Abstraction and Specification in Program Development.* MIT Press, Cambridge Massachusetts, 1986.

[Lu 01] J. Lu, M. Zhang, M. Xu, and D. Yang, "A two-layered-class approach for the reuse of synchronization code.", *Information & Software Technology*, 43(5):287–294, 2001.

[Madsen 93] O. L. Madsen, B. Moller-Pedersen, and K. Nygaard, *ObjectOriented Programming in the Beta Programming Language.* Addison-Wesley, Jun 1993.

[Matsuoka 93] S. Matsuoka and A. Yonezawa, "Analysis of inheritance anomaly in object-oriented concurrent programming languages". In G. Agha, P. Wegner, and A. Yonezawa, eds., *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150, MIT Press, 1993.

[McHale 94] C. McHale, *Synchronization in Concurrent, Object-Oriented Languages: Expressive Power, Genericity and Inheritance.* PhD thesis, University of Dublin, Trinity College, Dublin, Ireland, 1994.

[Meyer 05] B. Meyer, "Attached types and their application to three open problems of object-oriented programming". In *ECOOP 2005, Proceedings of European Conference on Object-Oriented Programming*, pages 1–32, Springer Verlag, July 2005.

[Meyer 86] B. Meyer, "Genericity versus inheritance". In *OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 391–405, ACM Press, New York, NY, USA, 1986.

[Meyer 88a] B. Meyer, *Object-Oriented Software Construction.* Prentice-Hall, Englewood Cliffs, N.J., 1988.

[Meyer 88b] B. Meyer, *Object-Oriented Software Construction.* Prentice-Hall, Englewood Cliffs, N.J., first edition, 1988.

[Meyer 92] B. Meyer, *Eiffel: The Language.* Prentice Hall, Englewood Cliffs, N.J., March 1992. 2nd printing.

[Meyer 97] B. Meyer, *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.

[Mitchell 01] S. E. Mitchell, A. Burns, and A. J. Wellings, "Mopping up exceptions", *ACM SIGAda Ada Letters*, XXI(3):80–92, 2001.

[Moessenboeck 93] H. Moessenboeck, "Object-oriented programming in oberon". 1993.

[Moore 65] G. E. Moore, "Cramming more components onto integrated circuits", *Electronics*, 38(8), April 1965.

[NC 87] H. Norman C, R. K. Raj, A. P. Black, H. M. Levy, and E. Jul, *The Emerald Programming Language*. Technical Report 87-10-07, Department of Computer Science, University of British Columbia, Seattle, WA (USA), 1987.

[Nienaltowski 06a] P. Nienaltowski, "Flexible locking in scoop". In *Proceedings of the first Symposium on concurrency, Real-Time, and Distribution in Eiffel-Like Languages, CORDIE'06*, pages 71–90, University of York – Department of Computer Science, July 2006.

[Nienaltowski 06b] P. Nienaltowski and B. Meyer, "Contracts for concurrency". In *Proceedings of the first Symposium on concurrency, Real-Time, and Distribution in Eiffel-Like Languages, CORDIE'06*, pages 27–49, University of York – Department of Computer Science, July 2006.

[OeS 04] M. Oliveira e Silva, "Concurrent object-oriented programming: The MP-Eiffel approach", *Journal of Object Technology: Special issue: TOOLS USA 2003*, 3(4):97–124, April 2004.

[OeS 06a] M. Oliveira e Silva, "Automatic realizations of statically safe intra-object synchronization schemes in MP-Eiffel". In *Proceedings of the first Symposium on concurrency, Real-Time, and Distribution in Eiffel-Like Languages, CORDIE'06*, pages 91–118, University of York – Department of Computer Science, July 2006. Available at `http://www.ieeta.pt/~mos/pubs`.

[OeS 06b] M. Oliveira e Silva, "Concurrent contracts and inter-object synchronization in MP-Eiffel". 2006. Draft version available at `http://www.ieeta.pt/~mos/pubs`.

[Parnas 72a] D. L. Parnas, "On the criteria to be used in decomposing systems into modules", *Communications of the ACM*, 15(12):1053–1058, 1972.

[Parnas 72b] D. L. Parnas, "A technique for software module specification with examples", *Communications of the ACM*, 15(5):330–336, 1972.

[Peterson 83] G. L. Peterson, "Concurrent reading while writing", *ACM Trans. Program. Lang. Syst.*, 5(1):46–55, 1983.

[Pierce 02] B. C. Pierce, *Types and Programming Languages*. The MIT Press, Cambridge, Massachusetts, 2002.

[Puntigam 05] F. Puntigam, "Client and server synchronization expressed in types". In *Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, San Diego, California, October 2005.

[Ruschitzka 77] M. Ruschitzka and R. S. Fabry, "A unifying approach to scheduling", *Communications of the ACM*, 20(7):469–477, 1977.

[Ryant 97] I. Ryant, "Why inheritance means extra trouble", *Communications of the ACM*, 40(10):118–119, 1997.

[Strachey 00] C. Strachey, "Fundamental concepts in programming languages", *Higher Order Symbol. Comput.*, 13(1-2):11–49, 2000. (reprinted from 1967 article).

[Stroustrup 85] B. Stroustrup, *The C++ Programming Language*. Addison-Wesley, first edition, 1985.

[Stroustrup 97] B. Stroustrup, *The C++ Programming Language*. Addison Wesley Longman, third edition, 1997.

[Sun Microsystems Java Specification Requests 04] Sun Microsystems, Java Specification Requests, "JSR166: Concurrency Utilities". 2004. (`http://www.jcp.org/en/jsr/detail?id=166`).

[Templ 93] J. Templ, "A systematic approach to multiple inheritance implementation", *SIGPLAN Not.*, 28(4):61–66, 1993.

[Ungar 87] D. Ungar and R. B. Smith, "Self: The power of simplicity". In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 227–242, ACM Press, New York, NY, USA, 1987.

[Ungar 91] D. Ungar, C. Chambers, B.-W. Chang, and U. Hölzle, "Organizing programs without classes", *Lisp and Symbolic Computation*, 4(3), June 1991.

[Wirth 71] N. Wirth, "Program development by stepwise refinement", *Communications of the ACM*, 14(4):221–227, 1971.

[Wirth 74] N. Wirth, "On the composition of well-structured programs", *ACM Computing Surveys (CSUR)*, 6(4):247–259, 1974.

[Wirth 85] N. Wirth, *Programming in MODULA-2 (3rd corrected ed.)*. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[Xu 95] J. Xu, B. Randell, A. B. Romanovsky, C. M. F. Rubira, R. J. Stroud, and Z. Wu, "Fault tolerance in concurrent object-oriented software through coordinated error recovery". In *Symposium on Fault-Tolerant Computing*, pages 499–508, 1995.