UNIVERSIDADE FEDERAL DE PERNAMBUCO

CENTRO DE INFORMÁTICA

PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

LEONARDO FREITAS

"JACK: A process algebra implementation in Java"

ESTE TRABALHO FOI APRESENTADO À PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DO CENTRO DE INFORMÁTICA DA UNIVERSIDADE FEDERAL DE PERNAMBUCO COMO REQUISITO PARCIAL PARA OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIA DA COMPUTAÇÃO.

ORIENTADOR(A): Augusto César Alves Sampaio

RECIFE, ABRIL/2002

Abstract

The construction of concurrent programs is especially complex due mainly to the inherent non-determinism of their execution, which makes it difficult to repeat test scenarios. Concurrency has proved to be a fascinating subject and there are many subtle distinctions which one can make. This dissertation presents an approach for constructing concurrent programs using a set of process algebra constructs (for CSP) implemented as an object-oriented framework in Java called JACK; it stands for *Java Architecture with CSP kernel*.

The main objective of the framework is an implementation of process algebra constructs that provides, as naturally as possible, the algebraic idiom as an extension to this concurrent object-oriented programming language. By naturally, we mean a design and implementation that provide the process abstraction as if it were included in the Java language itself (i.e. embedded in the Java language as an extension package).

JACK is a framework that implements a process algebra. A process algebra is a formal language that has notations for describing and reasoning about reactive systems. It implements a modern version of the Hoare's Communication Sequential Process (CSP) formalism. The framework is provided as a Java extension package that supplies CSP operators embedded in the Java language. The library is structured using UML, role modeling for framework design and construction, and make use of design patterns and pattern languages. Furthermore, JACK follows some of the most important software engenieering practices to build frameworks and as a result its design achievies important properties like reusability, simplicity, expressive power, modularity, extensibility, and so forth. JACK is provided as a *gray-box* (white, and black-box) framework tailored to run CSP specifications in Java; it can also be used to model unified specifications like Circus and CSP-OZ, that combines CSP with Z.

The implementation is built using separation of concerns in a way that is highly beneficial to class-based design of frameworks. This work empathizes the use of design patterns and pattern languages to properly build frameworks, achieve desired software engineering properties and software quality requirements. The user of the JACK framework is able to describe its process specification in Java, either in CSP or in a combined algebra one, like in CSP-OZ or in Circus.

Resumo

A construção de programas concorrentes é de natureza complexa principalmente por conta do não-determinismo herdado da sua forma de execução, o que dificulta a repetição de cenários de teste. Concorrência tem provado ser um tema fascinante existindo muitas sutis distinções que podem ser feitas. Esta dissertação apresenta uma proposta para construção de programas concorrentes utilizando um conjunto de construções de álgebra de processos (para CSP) implementadas como uma arquitetura orientada a objetos em Java chamada JACK.

O principal objetivo da arquitetura é a implementação de construtores de álgebra de processos que forneça, da maneira mais natural possível, o idioma algébrico como uma extensão desta linguagem de programação concorrente e orientada a objetos. Por natural, entede-se um projeto e implementação que forneça a abstração de processo como se ele estivesse incluído em Java como um pacote de extensão.

JACK é uma arquitetura que implementa uma álgebra de processos. Uma álgebra de processos é uma linguagem formal que possui notação para descrição e raciocínio sobre sistemas reativos. Ela implementa uma versão moderna do formalismo *Communication Sequential Process (CSP)* desenvolvido por Hoare. A arquitetura está disponível como um pacote de Java que provê operadores de CSP para esta linguagem. A biblioteca é estruturada utilizando UML e modelo de papéis para projeto e construção de arquiteturas, fazendo uso de padrões e linguagens de projeto. Além disso, JACK segue algumas dos mais importantes práticas em engenharia de software para construção de arquiteturas, e, como um resultado, seu projeto possui importantes propriedades como reusabilidade, simplicidade, poder de expressão, modularidade, extensibilidade, etc. JACK é disponibilizada como uma arquitetura *gray-box (white, e black-box)* construída para rodar especificações CSP em Java. Ela pode ainda ser utilizada para modelar especificações unificadas como Circus e CSP-OZ, que combinam CSP com Z.

A implementação é construida utilizando separação de facetas de uma forma altamente benéfica para o projeto de arquitetura baseado em classes. Este trabalho enfatiza o uso de padrões e linguagens de projeto para construir arquiteturas de forma adequada, alcançando propriedades de engenharia de software e requisitos de qualidade desejados.

Acknowledgments

ao outro Outro encarnado em nós mesmos
e que não sabemos como lidar (...).

This dissertation is dedicated to the death of *Dum*
and the rebirth of *Dede* and her metamorphose tramp.

I have two main group of people to acknowledge. One that provides me full proof technical support, and another one that gives enough courage and psychological incentive (and presure) to continue. Each group plays a specific and necessary role in this disseration work, without them, it would not be possible to happen.

In the first main group I must distinguish my supervisors Augusto Sampaio and Ana Cavalcanti for their competent technical and personal support for the completeness of this work. I like you very much! Despite this, I believe that the most important person that have influenced my academic route at a whole is my great friend Adolfo Duran. Without his initial trust in my work and operational incentive to follow that academic path, I would not be there. I cannot forget to mention another important person in my academic life: José Augustp Suruagy Monteiro; he was the first one that trusted in my work and gave me operational support and incentive to continue in this area. Behind the scenes, there are also more people that helped a lot in many other technical details like Alexandre Motta, Orlando Campos, Adalberto Cajueiro, and some other department university fellows. Their valuable opinions and discussions were very important. I would also like to thanks Professor Jim Woodcock, Augusto Sampaio, and Ana Cavalcanti for their encouragement in continue my dissertation work as PhD thesis on the same research field — tools for formal methods. Professor Peter Mosses gives me helpfull support in an Action Semantic research related to this work. A special thanks must be given to Jaqueira's reacecourse (a beautiful place here in Recife that I run everyday); the most important ideas of this dissertation were perceived there. Finally, I must mention CAPES, a brazilian state organization that gives financial support for my technical research at UFPE.

The second group have the most important people in my life. I must thank

my mother and godmother for their emotional support and important phylosofical discussion about life, poetry, and psychoanalysis. I would like to thank my father for their operational and practical support, and also for his complete trust in my capacity even when I did not. Other important persons must be mentioned. Helson Ramos and *Espaço Moebius*, for the valuable discussions about psychoanalysis and the analytical style of life at a whole. Adolfo Duran and Caio Mattos, for their complete friendship. Orlando Campos, for his serenity and tranquillity in the time that we have acquaintance her in Recife. Katiana Barauna, for her initial emotional support here in Recife and during some personal problems in Salvador. To Bolildos guys, my old group of friends from Salvador, specially mentioning Mario, for his tranquility and security; Bolildo and Borela for their determination and hilarious behaviour; Mira, for his organization; Zinho, for our interesting discussions about social human way of life; Clô, for his blind friendship, even when I was not worthy of it. To Virginea and Nina, for their emotional support and admiration about me and my analytical way of life. Finally, for the other important people that I have forgot to mention, please, receive my apologizes.

I must say that, despite of many technical, personal and emotional problems during this work, it was worthwhile. A final thanks go to Guy Moupassant and his incridible *Horla*; Friederich Nietzsche and Arthur Schopenhauer, for their valuable books about human ethics; Sigmund Frued and Jacques Lacan for their analytical support. Without these sources of reference, these referred problems could never be solved.

“à toda coisa sombria chamada reflexão”, disse o filósofo aforismático [103];
“acima da razão não há tribunal ao qual apelar”, disse o mestre da psiquê [46];
“tentemos então instâncias menores primeiro, como o amor.”, disse o poeta [39].

“to all obscure thing called reflection”, said the aphoristic philosopher;
“above reason there is no court to appeal”, said the psyche master;
“thus try lower instances first, like love”, said the poet.

Contents

List of Figures

Chapter 1

Introduction

Process algebras [61, 124, 93] are very attractive formalisms to describe concurrent and dynamic aspects of complex computer systems. However, the execution of specifications must deal with concurrent programming, a non-trivial task. This introductory Chapter presents the main motivation to deal with formal concurrent specification and programming languages with respect to software quality requirements.

Section 1.1 presents a short introduction on formal languages. Next in Section 1.2, the main motivation for this work is presented. Then in Section 1.3, important software engineering requirements to build object-oriented frameworks are listed. Finally, in Section 1.4, we present the general structure of the dissertation.

1.1 Formal Languages

A language is a set of signs that serves as a communication medium between people. To express that communication, these signs must be organized following a set of rules and principles. Those rules are called the language grammar; all languages must have a grammar.

To build something useful with the language, the user needs to know how to build expressions with these signs, and how to compose them to create an expressible discussion. The syntax of the language is the part of the grammar that defines those signs organization. Each signal has by itself a specific meaning that needs to conform between related signs in a sentence. The semantics of a language specifies that relationship of significance between the language signs[1]. Therefore, to completely understand and make use of any kind of language, we need to know the language signs, its syntax, and its semantics.

The main difference between a formal language (like CSP [124] or Z [147]) and a natural language (like English or Portuguese) is how their syntax and semantics

[1]Semantics comes from the Greek *semantiké*, that means *the art of significance*.

are defined. While the natural languages have an open possibility for different meanings and ambiguity in their grammars, formal languages, in the opposite side, have a strict and singular meaning for all their possible constructions: mathematical treatment of the language rules. This rigour of formal languages is necessary due to its intention: computational reasoning. Up to now, a computer could not infer a meaning without a specific rule to apply. Computers do not support ambiguity in their processing.

In this sense, to describe computer software we need a formal language. There are many types of formal languages. For instance, a concurrent object-oriented programming language like Java [4, 72]. Java is a formal language in the sense that it has a well-defined grammar with syntax and semantic definitions. Furthermore, there are other types of formal languages like specification languages, refinement languages, and so forth. For instance, there is CSP (Communicating Sequential Process) [61, 124], that is a specification language for behaviour description of both concurrent and sequential processes.

Nevertheless, for each kind of problem, there is the need of a specific sort of language to deal with that problem domain. It is important to choose the appropriate language for the referred problem. The language ought to provide enough expressive power to solve the problem. It is also important that there is tool support to that language in order to make it use widespread.

1.2 Motivation

The construction of concurrent programs is especially complex due mainly to the inherent non-determinism of their execution, which makes it difficult to repeat test scenarios. It should be noted that the execution order of two concurrent activities is influenced by external programming conditions, such as processor load and interruptions. Various techniques, tools, and methods have been proposed to help programmers write concurrent programs.

A process algebra is a formal language that has notations for describing and reasoning about reactive systems. A process algebra introduces the concept of a system of processes, each with its own private set of variables, interacting only by sending messages to each other via *handshaken* communication. All these languages use synchronization on atomic events as the foundation for process interaction, and all provide some way of expressing event occurrence, choice, parallel composition, abstraction, and recursion [130]. By event we mean an instantaneous communication that can only occur when the participant process and the external environment agree [124].

Process algebras are useful because they bring the problems of concurrency into sharp focus. Using them, it is possible to address the problems that arise, both

at the high level of constructing theories of concurrency, and at the lower level of specifying and designing concrete systems, without worrying about other issues [124]. A process algebra theory consists of a well-defined set of semantic laws that describe the behaviour of processes. These descriptions may use both operational laws [108] and denotational laws [127] to properly define different theoretic parts [126, Chapters 3 and 4]. Concurrency has proved to be a fascinating subject and there are many subtle distinctions which one can make, both at the level of choice of language constructs (i.e. its operators) and in the subtleties of the theories used to model them (i.e. its semantic laws).

This dissertation presents an approach for constructing concurrent programs using a set of process algebra constructs (for CSP) implemented as an object-oriented framework. It is implemented as a framework in order to achieve desired software engineering quality requirements and be ready for both *black-box* and *white-box* reuse. This framework is called JACK and stands for *Java Architecture with CSP kernel*. The main objective of the framework is an implementation in Java [4, 72] of process algebra constructs that provide, as naturally as possible, the algebraic idiom as an extension to this concurrent object-oriented programming language. By naturally, we mean a design and implementation that provide the process abstraction as if it were included in the Java language itself [126, 28] (i.e. embedded in the Java language as an extension package). The framework solutions are based on micro-architectures of cooperating objects which ought to be applied, combined, and customized to build a process specification using that programming language extension. The resultant program can be used either as a simulation/prototype of a CSP specification, or as an implementation of the CSP network in a programming language that can physically run.

As already mentioned in [31], Java is suitable for this kind of library implementation since it has built-in concepts of concurrent programming and communication. It is also platform independent, well-suited for the development of embedded systems, has many research efforts to give it a formal semantics [65, 27, 25, 24]

Object-oriented frameworks promise higher productivity of application development through design and code reuse. Object orientation comprises object-oriented analysis, design, and programming [90]. Using a small set of concepts (objects, classes, and their relationships), developers can model an application domain (analysis), define a software architecture to represent that model on a computer (design), and implement the architecture to let a computer execute the model (program) [113].

None of these activities (analysis, design, and implementation), nor the resulting models, are trivial. To carry them out effectively, developers have invented additional concepts that represent the conceptual entities they are dealing with. One such key concept is the object-oriented framework. An object-oriented framework is a reusable design together with an implementation and documentation. The design represents a model of an application domain or a pertinent aspect thereof; the

implementation defines how this model can be executed, at least partially; and the documentation states and guides how the framework can be properly used (black-box use) or extended (white-box use). Although, in principle, the JACK framework is not configured to a specific process algebra, our particular focus of attention is CSP [124, 126, 130].

Communicating Sequential Process (CSP) [124] has become a very attractive formalism to describe concurrent and dynamic aspects of computer systems. It is a process algebra that has a formal semantics [124, 126, 61], tool support [33, 34], up-to-date references [130], and in progress research [12, 36, 82]. Despite this, there are many research lines under development related to frameworks and CSP, such as unified language [62] specifications [148, 30, 29, 30, 29, 31, 100]; libraries that implements occam [53], a language that implements an old CSP [61, 60] version [107, 59, 51, 144]; description of real-world specification examples [99, 45]; formal implementation strategies for CSP and combined specifications [12, 35, 82, 1]; and so on.

The CSP under consideration in this dissertation is the one described in [126, 124]. This is the updated version of the initial C.A.R. Hoare [61, 60] CSP description. Its main well-known implementation is the model checker FDR (Failures and Divergences Refinement) [33] and specification animation ProBE [34] tools. In this new version, there are two main differences from the original approach. The processes alphabet needs to be explicitly declared, which gives a greater flexibility in the parallel construction of process networks [124, Chapter 2]. An important aspect is the multidimensional typed channel. Other minor features are added, like modeling decisions against process termination (skip rules) and value manipulation (functional expression language). In spite of this, there are two new aspects related to this new version that are quite complicate. It is the multisynchronization and backtrack problems. The former is related to the parallel operator that now can multi-synchronize in more than two processes, an old restriction. The latter occurs due to the need of dealing with infinite data types without expansion (i.e. through a symbolic approach); that is, when there is some data dependency related to the communication under consideration. These topics are detailed in Sections 2.3.3 and 2.3.2 respectively.

CSP has become well-suited to describe most requirements related to the dynamic behaviour of distributed and concurrent aspects of complex computer systems. With this in mind, tools like FDR are adequate for simple toy examples and large real-world industrial ones.

With the use of this tool, one can design a system and proof some important properties like deadlock freedom, livelock freedom, non-determinism, refinements between specifications, and so on. At the level of processes implementation, it is necessary to achieve the same results (i.e. the same properties) as in the modeling stage. Nevertheless, a process specification in FDR cannot run, but just be analysed

(i.e. one can prove properties, but cannot see a process physically running). In this sense, there is a gap between process specification design and process implementation execution, that needs to be solved. Therefore, there is the wish to run a specification in an environment that captures the correct semantics of the processes.

As far as we know, there is the lack of a framework that describes CSP [124] in a programming language in the formal methods field. There are only implementations of occam [53], a programming language based on an old CSP version [61]. There is only one available tool related to process animation called ProBE [34]; it cannot physically run a process, but just iterate through its expected semantics. Therefore, there is a clear motivation to build a framework to deal with CSP [124] specifications. Despite this, the process framework opens the possibility for the user to deal with not only CSP process specifications, but also in specifications of combined languages like Circus [148, 149] or CSP-OZ [31], and works related to refinement calculus [96, 95, 5] of such languages [12], due to its ability to allow users to define their own process behaviours. In this way, the CSP process algebra is selected to be the language under consideration for a framework implementation.

1.3 Software Engineering

Software engineering deals with the construction of systems whose complexity requires the intervention of teams of engineers. Due to the interaction and cooperation among team members, software engineering must have a set of principles applicable, not only to the developed products, but also to their development process. These principles are less relevant when applied in the context of application development by a single programmer, or a framework that will never be extended. Software engineering principles are the basis for the methods, techniques and tools used by the software engineer. In [133], seven quality principles related to software engineering are mentioned; they are briefly enumerated below.

1. **Rigor and Formalism** — The use of a mathematical formalism based on a combination of both theoretical and experimental results.

2. **Separation of Concerns** — Separate handling of different aspects of program construction, in order to better control complexity and also allow separation of responsibilities, thus improving teamwork.

3. **Modularity** — Decomposition of a program into modules; the composition of a program from existing modules; and understanding each part of a program separately.

4. **Abstraction** — Identification of relevant aspects, with details ignored, thus allowing better complexity management and stepwise development.

5. **Anticipation of Change** — A program must be prepared to anticipate change in order to reduce its impact. The identification of a program's future evolutions must be carried out at the requirements gathering stage. The isolation of parts to be changed may be done using modularity.

6. **Generality** — The construction or use of a tool or method to solve a more general problem may be reused in the context of different programs, like design patterns [28] and pattern languages [121].

7. **Incrementallity** — A program is built in successive increments, allowing in this way, incremental testing and debugging of a programs.

We try to follow these requirements in order to provide a well-defined framework implementation with evolution support.

From a software engineering perspective, and based on the aforementioned principles, an approach to the development of concurrent programs must satisfy, in addition to functional and non-functional requirements, a set of quality requirements [133]. Thus, the definition of an approach to the development of concurrent programs, capable of satisfying the previous stated software engineering quality requirements, constitutes an important goal of the JACK process algebra framework implementation. In what follows, we identify some JACK aspects related to each quality requirement. Most of them come from [133, 113, 126].

1. **Rigor and Formalism** — The process algebra constructs implementation follows strictly operational laws [126, 130, 124] and implementation guidelines [126, 61] of CSP, which leads the framework to implement the rigor and formalism of CSP.

2. **Separation of Concerns** — JACK uses separation of concerns inherited from DASCO [133], which is itself a framework that deals with concurrency, synchronization, and recovery as orthogonal matters.

3. **Modularity** — Since we use separation of concerns, it becomes easier to achieve modularity. Despite this, JACK was modeled using the very modular technique of role modeling [113]. Thus, the whole framework has highly cohesive modules presenting low coupling between layers. This topic is detailed in Chapter 4.

4. **Abstraction** — With DASCO's separation of concerns (see Chapter 6 and [133]), it is possible to completely abstract a specific concern like concurrency or synchronization. With the use of Hoare's solution to implement processes [61, pp. 38], it is also possible to abstract process algebra constructs before their complete implementation, thus providing also incrementallity.

5. **Anticipation of Change** — The use of role modeling [113], design patterns [28, 123, 102, 23, 81, 121], pattern languages [121], and framework modeling [22, 113] techniques caters for anticipation of changes.

6. **Generality** — Role modeling and design patterns used by JACK are an important basis for the framework ability to be general and extensible, by following rigorous steps through well-documented guidelines.

7. **Incrementallity** — Again, with DASCO's separation of concerns and Hoare's abstractions, it is possible to incrementally develop both low-level concurrency aspects (i.e. threads, locks, transactions) and high level process algebra construct aspects (i.e. CSP operators and user defined processes).

Satisfaction of these quality requirements by the approach implies a rigorous set of well-defined and clear stages. The proper definition of these stages results from an extensive analysis of existing solutions for each concern at the proper framework layer. The application of the results of this analysis reflects itself in each module's correction (i.e. proper layer model, design, and implementation) [133].

Other important aspect to be mentioned is documentation. The lack of proper model and source code documentation in other process algebra libraries [107, 59], limites the possibility of extension and proper use of them (see Chapter 3 and [36]).

Framework modules must be documented, so that it is possible to understand their function at different abstraction levels. For instance, it must be possible to understand a module's collaboration structure independently from its implementation. Documentation must allow different kinds of module users, according to the required knowledge level (i.e. user developer or extension developer).

JACK provides many sources of documentation. This varies from detailed JavaDoc source code documentation [38], UML and Role Models [38], test case and example classes [38], and tutorials specifically tailored for user developers [41], for extension developers [42], and so forth.

1.4 Structure of the Dissertation

This dissertation consists of seven Chapters and one Appendix. A short description of what follows is given.

- **Chapter 2** — Description of the CSP process algebra constructs. The Chapter also states two important problems to be solved by the framework, that is, backtracking and multisynchronization of processes.

- **Chapter 3** — Description of why to build JACK as an object-oriented framework with design patterns and pattern languages. A brief introduction to

these topics and framework role modeling is also given. It makes a comparison between JACK and other available process libraries.

- **Chapter 4** — Introduction to the JACK framework from the end user (i.e. a JACK client) point of view. It describes the framework documentation and packaging rules, the type system used to describe process elements, the set of CSP operators available, and so on.

- **Chapter 5** — Presents the JACK architecture. It specifies the main layers of the framework, how they are organized, and which role each layer must play.

- **Chapter 6** — Presents the JACK implementation project. It details the layers at the lower levels of abstraction, establishes how layers and sub-layers get composed, and how the framework can be configured and extended by some advanced user or extension developer.

- **Chapter 7** — Summarizes dissertation main contributions, future work, and final remarks. This Chapter also provides a range of expected improvements for JACK future releases, and possible tools that can use the framework.

- **Appendix A** — Provides web links for other sources of information related to JACK, like on-line documentation, UML model, tutorials, other documents, and so forth.

Chapter 2

CSP Presentation

Communicating Sequential Processes (CSP) [124] has become a very attractive formalism to describe concurrent and dynamic aspects of computer systems. It has a formal semantics, tool support, up to date references, and in progress research. The version of CSP considered in this work is the one described in [124]. This is the updated version of the initial C.A.R. Hoare's CSP description [61].

The CSP operators can be implemented in a concurrent object oriented programming language, such as Java [72], to allow construction of concurrent programs with these operators. This implementation would be done in order to provide more abstract concurrent constructors for the language to allow designers and programmers to describe their programs in CSP.

Section 2.1 makes a short introduction to CSP. Next, in Section 2.2, a description of the most important CSP operators are given, with some discussion about the main differences between Hoare's version and Roscoe's version of CSP. After that, in Section 2.3, the main problems observed to achieve the implementation of this version of the formalism, extended to deal with infinite data types, is shown. Finally, in Section 2.4, a summary and some final considerations are given.

2.1 CSP

The problem domain under consideration for this dissertation is the formal specification of concurrent systems. There are many specification languages used for this purpose, like for instance, CCS [94], LOTUS [10], and CSP [61, 124].

Our main objective is an implementation of a CSP framework that provides, as naturally as possible, the CSP idiom in Java [4, 72], in order to provide more abstract concurrent constructs to software developers. By naturally, we mean a kind of implementation that directly maps, as much as possible, the CSP language constructors into Java.

CSP [124] has become a very attractive formalism to describe concurrent and dy-

namic aspects of computer systems. One of the fundamental features of CSP is that it can serve as a notation for writing programs which are close to implementation, as well as more abstract specifications which may be remote from implementation.

CSP was designed to be a notation for describing and analysing systems whose primary interest arises from the ways in which different components interact at the level of communication. CSP is a notation and calculus designed to help us understand interaction. The primary applications areas are those where the main interest lies in the structure and consequences of interactions.

The most fundamental object in CSP is therefore a communication event. Events are assumed to be inside a set Σ (the Greek capital letter "Sigma") which contains all possible communications for processes in the universe under consideration. We should think of a communication as a transaction between two or more processes, rather than as necessarily being the transmission of data one way.

A CSP specification contains the definition of processes. A CSP process is completely described by the way it can communicate with its external environment. To construct a process the specifier needs to decide on an alphabet of communication events. The choice of the process alphabet determines the level of detail or abstraction in the final specification.

The fundamental assumptions about communications in CSP are the following:

- They are instantaneous: there is no consideration about the real time intervals during the performance of communication events. They occur atomically — conceptually at the moment when they become inevitable. This property leads the specification to be safe with respect to possible race conditions during event occurrences.

- They can occur only when both the process and its environment allow them; but when the process and its environment do agree on an event, then it must happen (handshaking communication). This guarantees the liveness property of the specification.

CSP defines a way to reason about the interaction of processes with their environment using this model of communication.

2.2 CSP Operators

CSP has a variety of operators specifically tailored to describe the behaviour of processes. There are primitive basic processes, and operators to describe sequential, concurrent, and non-deterministic aspects of a system. There are other operators that deal with more complex features like renaming, piping, labelling, timed aspects, and so on.

For a comprehensive description of CSP and those operators see [124, 130]. For yet another short description of CSP operators see [49, Appendix A] and [99, Appendix A].

2.2.1 Primitive Processes

CSP has three primitive processes described as follows:

- $STOP$ — Represents a deadlocked process, that cannot do anything (i.e. a broken machine).
- $SKIP$ — Represents a process that has finished its job successfully.
- DIV — Represents a livelocked process that only performs communications invisible to its outside environment.

2.2.2 Prefix Operator

To express a process that represents the occurrence of an event, and then follow behaving as another process, there is the prefix operator ($\rightarrow$, or $->$ in machine readable CSP[1]) as follows:

$$e \rightarrow P \quad \text{(read as "e then P")}.$$

In the process given above, e represents an event and P the process that follows the occurrence of this event. The meaning of this process is very simple, after the occurrence of the event e, the whole process then behaves like P. Some examples of pritimive processes and prefix are given below:

$$\begin{aligned} CSPIntroduction = \;& (primitiveElements \rightarrow description \rightarrow \\ & example \rightarrow SKIP) \\ BrokenWatch = \;& (tick \rightarrow tack \rightarrow STOP) \\ CrazyWatch = \;& (tick \rightarrow tack \rightarrow DIV) \\ JohnsWakeUpRoutine = \;& (wakesUp \rightarrow brushesTeeth \rightarrow \\ & takesBreakfast \rightarrow goesToWork \rightarrow SKIP) \end{aligned}$$

The first equation defines a well-behaved process $CSPIntroduction$ that starts performing $primitiveElements$, passing through their $description$, and then giving an $example$. It is finished by $SKIP$, indicating successful termination.

The second equation, $BrokenWatch$, presents a watch that has some problem. It only performs one $tick$ and $tack$ cycle, $STOP$ping indefinitely afterwards. It is said that the process is deadlocked, or that the watch is broken.

[1]This is the input format accepted by tools that uses Roscoe's version of CSP, like FDR [33] and ProBE [34]. It is normally called CSPm.

The third $CrazyWatch$, presents a watch that has a funny behaviour. It successfully performs one *tick* and *tack* cycle and after that the external environment cannot infer anything about that process. It can either continue to work without our knowledge or just be broken. It is said that the process is livelocked.

The last equation $JohnsWakeUpRoutine$, presents the behaviour of John every morning. Firstly he $wakesUp$, then he $brushesTeeth$ and $takesBreakfast$, then $goesToWork$, and finally he successfully finishes his waking up routine.

2.2.2.1 Prefix Communications

An event can also be a communication event, which is represented by the pair

$$c.v$$

where c is the name of the channel, and v is the value of the message that is passed. Inputs and outputs are, respectively, the read event $c?v$ and the write event $c!v$, which are performed when both processes are ready, and a communication over channel c is performed.

An input or read prefix process is defined as

$$c?x : T \rightarrow P(x)$$

This is a process that behaves as process $P(x)$ after the value x is read by channel c. The value x is restricted to be an element of the set T. In this sense, T acts as a value constraint over the possible values to be read through channel c.

An output or write prefix processes is defined as

$$c!v \rightarrow P$$

This is a process that behaves as process P after the value v is sent through channel c.

This tightly defined communication protocol is called rendezvous communication. The set of messages communicable on channel c is defined as

$$type(c) = \{v | c.v \in P\}$$

where P is a predicate which constrains the values accepted for communication.

2.2.3 Fixed Point Operator

Sometimes there is the need to introduce repetitive behaviour for processes specifications; to do so, we use recursion. Recursion in CSP can be achieved by a reference to the process name on the right side of the equation that defines it. For instance, in a equation like $P = (e \rightarrow P)$, P performs the event e and behaves like P again. Therefore, this process behaves like an infinite process that is always enabled to perform the event e.

As an other example, one can describe an analogic watch as follows:

$$AnalogicWatch = (tick \rightarrow tack \rightarrow AnalogicWatch)$$

which performs *tick* and *tack* forever.

There is an important point to be mentioned about recursive processes related to its initial event. If a recursive equation is prefixed by an event like in $P = e \rightarrow P$, it is called a guarded recursion. In this case, the equation has a single solution that can be described using another important CSP operator, called recursion. Uses of this operator takes the form $\mu x : Args.Body$, where x is a process name, $Args$ are the process arguments, and $Body$ is a process definition that typically refers to x to denote recursive behaviour. The solution of a recursive guarded equation $P = E(P)$ that has an alphabet $\alpha P = A$ is defined as $\mu C : A.E(C)$, where C is a name for the locally defined process. Therefore, the analogic watch process can be redefined as follows:

$$AnalogicWatch2 = \mu AW : \{tick, tack\}.(tick \rightarrow tack \rightarrow AW)$$

or simply as

$$AnalogicWatch2 = \mu AW.(tick \rightarrow tack \rightarrow AW)$$

where the alphabet is left implicit.

2.2.4 Concurrent Processes

The term *concurrent process* is used to designate actions performed by a set of processes at a given moment. Each process defines its own execution flow, therefore we reach systems where there is more than one execution flow at the same time. In this context, events shared by two or more processes could interact in the sense that they could agree on the occurrence of that event. This is called a synchronous communication model, because if a process that has this shared event in its alphabet and cannot perform it, the other participant processes that wish to engage in the event must wait. There is the possibility that this process interaction never occurs, leading the whole synchronization to fail, behaving like the canonical deadlock process $STOP$.

The operator used to denote concurrent behaviour with synchronous interactions is parallelism ($[|X|]$). To completely specify the parallel operator, one needs to provide two processes and a synchronization alphabet: $P[|X|]Q$, where X is a set of events such that $X \subseteq \alpha P \cup \alpha Q$.

For instance, suppose that two employees of a factory are arranging boxes inside a truck for delivery. The first employee, called John, gets a box from the floor and gives it to the second employee called Joseph. After that, Joseph arranges the box inside the truck. Both employees need to agree in one event: when John holds a box and gives it to Joseph, he must want and be prepared to take that box. Therefore, at

this point, a synchronization between John and Joseph must occur. We can model that system as follows:

$$\begin{aligned} John = &\ \mu C.(getBoxOnFloor \rightarrow boxOnHand \rightarrow C) \text{ and} \\ Joseph = &\ \mu D.(boxOnHand \rightarrow arrangeBox \rightarrow D), \text{ therefore} \\ TruckArrangement = &\ (John \| Joseph) \end{aligned}$$

which is equivalent to

$$\begin{aligned} TruckArrangement = &\ \mu E.(getBoxOnFloor \rightarrow boxOnHand \\ &\rightarrow arrangeBox \rightarrow E) \end{aligned}$$

The interpretation of the process $TruckArrangement$ is simple. The events that are not shared by both process operands can be performed independently, by whatever process, while for the common event both processes must agree. In this case, if John works faster than Joseph, he must hold on with the box N until Joseph finishes the arrangement of the previous box and is prepared again to have another box on hand. After that, we provide the same process equivalently defined as sequential recursive process that represents the same behaviour as the concurrent one.

An important note about the parallel operator must be given. This Section presents Roscoe's [124] version of it. Nevertheless, there is an older version defined by Hoare [61] that is quite different. Hoare's version of the parallelism operator makes an implicit use of the participant process alphabets, therefore the process $P \| Q$ is uniquely determined by the alphabets of P and Q (i.e. αP and αQ). Hoare's version can be described in terms of the Roscoe's version by just making the process alphabet explicit (i.e. $P[|\alpha P \cap \alpha Q|]Q$).

In the following, we present the example of the Truck using the Hoare's version of the parallel operator:

$$\begin{aligned} & John = \mu C.(getBoxOnFloor \rightarrow boxOnHand \rightarrow C) \\ \text{with alphabet} \quad & \alpha John = \{getBoxOnFloor, boxOnHand\} \text{ and} \\ \\ & Joseph = \mu D.(boxOnHand \rightarrow arrangeBox \rightarrow D) \\ \text{with alphabet} \quad & \alpha Joseph = \{boxOnHand, arrangeBox\}, \text{ therefore} \\ \\ TruckArrangement = \quad & (John \| Joseph) \end{aligned}$$

which is equivalent to

$$\begin{aligned} TruckArrangement2 = &\ \mu E.(getBoxOnFloor \rightarrow boxOnHand \\ &\rightarrow arrangeBox \rightarrow E) \end{aligned}$$

The equivalence between $TruckArrangement$ and $TruckArrangement2$ is achieved applying some algebraic transformation laws related to the parallel operator.

2.2.4.1 Generalized Choice — External Choice

The generalized choice, also called External Choice ($\Box$, or `[]` in machine readable CSP), can be used to express any choice operator of CSP: it can behave as a non-deterministic choice ($\sqcap$), or as a Simple Choice ($|$). The simple choice operator can only be used when processes with distinct initial events are described, while the Internal ($\sqcap$) and External ($\Box$) choice apply to any processes. The external choice is called generalized because in the case of prefixed processes with equal initial events, its semantic meaning is equivalent to the Internal Choice operator; otherwise, if their initial events differ, its semantic meaning is equivalent to the Simple Choice operator.

As a process algebra, CSP has a set of algebraic laws that define equivalencies between (syntactically) different processes. The following laws applies for $\Box$:

$$((a \rightarrow P)\Box(b \rightarrow Q)) = \begin{cases} ((a \rightarrow P)|(b \rightarrow Q)), & if a \neq b \\ ((a \rightarrow P) \sqcap (b \rightarrow Q)), & otherwise \end{cases}$$

For example, to model the keyboard of a single calculator, where the user can press a numeric key at any moment, one could specify the following process:

$$\begin{aligned} SingleCalcNumPad = \quad & \mu C. \quad (zero \rightarrow C) \\ & \Box \quad (one \rightarrow C) \\ & \Box \quad \vdots \\ & \Box \quad (nine \rightarrow C) \end{aligned}$$

The events *zero*, *one*, ..., *nine*, represent the press of the corresponding key.

2.2.5 Non-Deterministic Processes

A non-deterministic process performs events without the interference of its external environment. For instance, for abstraction, normally deterministic processes can be defined non-deterministically. This is convenient because the more deterministic a specification becomes, the more detailed and complex it is.

In CSP, non-determinism can be modelled in a variety of ways, depending on the desired results. The following subsections present these operators.

2.2.5.1 Non-deterministic *or* — Internal Choice

The non-deterministic *or* process, also called Internal Choice ($\sqcap$ or $|\tilde{}|$ in machine readable CSP), describes a behaviour that does not depend on its environment. The process $(P \sqcap Q)$ can behave like either P or Q, with the process selection not depending on the environment.

As already mentioned, non-determinism is useful to abstract specification details. In this sense, the introduction of non-determinism can be used to avoid detailing

behaviour. For example, a mail router program might offer one of two possible routes, considering the network traffic; the user is not concerned with the route, but simply in the correct delivery of the message, and is therefore happy to leave the responsibility for making the choice to the router program. Of course, looking at the router program internally, there is some shortest path algorithm implemented to select the best route. Nevertheless, it at a certain point of our design we are not worried about this internal detail, the router program can be described as follows:

$$ROUTER = (ROUTE_A \sqcap ROUTE_B)$$

As another example, one can define a process that neither initialises variables nor checks preconditions, just runs and does not guarantee any results. For instance, if a programmer wants to test or implement some desired behaviour as quickly as possible, just to see it working, not worrying if the program crashes at the first moment, the process below can be used.

2.2.5.2 Interleaving

The interleaving ($|||$) operator is a particular case of the parallelism operator. In fact it can be viewed as a parallel operator with an empty synchronization alphabet X, meaning that there are no interactions between the two concurrent processes.

The execution behaviour of $(P|||Q)$ is given by the execution of any event of either P or Q in any order. For example, a fax machine may be described as follows:

$$FAX = \mu C.(acceptDocument \rightarrow printIt \rightarrow C)$$

The machine is initially ready to accept any document. After accepting a document, the FAX must print it and start behaving like FAX again. Suppose that a collection of four fax machines may be connected to the same phone number (with four lines): any of them is suitable for processing incoming faxes.

$$FAXES = FAX|||FAX|||FAX|||FAX$$

The system provides the facility for processing up to four incoming faxes at the same time. There is neither the need to control which FAX will be selected nor to enforce any agreement between the FAX machines. The interleaving operator is associative. Thus, there is no need to put parenthesis in the above definition.

In the following we present two interesting algebraic laws about $|||$:

$$(P|||SKIP) = P$$
$$(SKIP|||SKIP) = SKIP$$

this means that an interleaving process terminates only when the two processes terminate.

2.2.6 Sequential Processes

The behaviour of the sequential operator (;) is very simple. It grants the permission to the second process to execute when the first one has successfully terminated.

For example, a purchase process has two well-defined sequential behaviours. First one needs to choose a set of products and after that pay for it. One can pay only after terminating selecting the products. First we define the $CHOOSE$ process that represents selection of products.

$$\begin{aligned} CHOOSE = \quad & (select \rightarrow \quad (keep \rightarrow SKIP \\ & \qquad\qquad \Box\ return \rightarrow CHOOSE)) \end{aligned}$$

Then the PAY process that represents the payment for the already selected products.

$$\begin{aligned} PAY = \quad & (cash \rightarrow receipt \rightarrow \quad SKIP \\ & \Box\ cheque \rightarrow receipt \rightarrow \quad SKIP \\ & \Box\ card \rightarrow swipe \qquad\quad (sign \rightarrow receipt \rightarrow SKIP \\ & \qquad\qquad\qquad\qquad\ \Box\ reject \rightarrow PAY)) \end{aligned}$$

Finally, we define the $PURCHASE$ process representing the sequential composition of product selection and payment.

$$PURCHASE = CHOOSE; PAY$$

The client can choose as many products as he or she wants, terminating when no more items are desired. The payment procedure can be achieved either by cash, cheque, or credit card.

Another interesting use of the sequential operator is to introduce recursion. Repeated execution of the same component or sequence of components can be described by means of a recursive loop. A recursive process that describes a recurrent spending is given below.

$$SPENDING = PURCHASE; SPENDING$$

The use of the sequential operator to describe recursive behaviour is also called iteration. It is described below.

$$P^* = P; P^*$$

Roscoe mentions the iteration operator [124, Chapter 6] as a particular case of the sequential operator.

2.3 Considerations for an Implementation of Roscoe's CSP

The CSP described here is the newer, updated version of the initial C.A.R. Hoare [60, 61] version, implemented in FDR and described in [126] and [124]. In this new version, we have two main differences from the original approach. The process alphabets need to be explicitly declared, which gives us a greater flexibility in the parallel construction of process networks, although looses its associative property. Another aspect is the multidimensional typed channels, where a prefix can mix both read and write operations. Minor features are added like modeling decisions involving process termination ($SKIP$ rules) and manipulation of values (functional expression language).

The available libraries, that are related to process algebra implementation (see Section 3.2) deal with occam [53]. This is a programming language with built-in Hoare's CSP constructs.

This new version of the formalism brings up new problems to be treated in an implementation of Roscoe's CSP in another target host language: Java. The following subsections present a discussion about some of these identified problems, in order to illustrate the main difficulties of this implementation work.

2.3.1 Complexity of New Operators

Roscoe's version of CSP has a variety of new operators that are much more complex to implement and reason about than the traditional Hoare's CSP. A brief description of those concepts and a discussion about them are given in the subsections below.

2.3.1.1 Behaviour and Data

In Hoare's version of the formalism there is no support to describe data structures. It covers only behavioural descriptions.

The new version of CSP makes it possible to the designer to specify both behaviour and data, with a companion functional expression language. CSP is not tailored to deal with formal description of data structures, as in languages such as Z [147]. Nevertheless, the ability to describe behaviour related to some specialized data structures increases the whole expressive power of the language. In [100], this is elegantly explored; that work proposes a strategy to model check combined specifications like CSP-OZ [31], using FDR, by transforming the original CSP-OZ input to normal CSPm[2] input, making use of the available functional programming language. In this way, a data structure in Z and a behaviour definition in CSP can be directly model checked using this transformation[3] and FDR.

[2]Machine readable version of CSP implemented by tools like FDR and ProBE.

[3]There is a semi automatic tool, that make a best effort approach to give an accurate translation from CSP-OZ to CSPm [1].

For instance, Roscoe's CSP allows the designer to define many sorts of types and type constraints, to build and use sets, lists and other structures related to functional languages [106].

2.3.1.2 Communication Concept

Hoare's communication input prefix ($c?x \rightarrow P$) and output prefix ($c!e \rightarrow P$) are treated differently in Roscoe's version of the formalism. Both input and output are translated to a so-called communication event in the form $c.v$. For the output prefix, the expression e is evaluated. The input prefix has a type constraint over the possible read values ($c?x : T$); it is converted into a simple choice between all possible read values. For instance if we consider the read prefix

$$(c?x : \{0 \ldots 2\} \rightarrow P)$$

this process is translated to become

$$(c.0 \rightarrow P \ \Box \ c.1 \rightarrow P \ \Box \ c.2 \rightarrow P)$$

This expansion brings up a big difference between implementations of the formalisms. For instance, in model checkers like FDR [33], that do analysis of some process properties like deadlock or livelock freedom, this expansion does not lead to many troubles, since model checking cannot deal with infinite data types. Nevertheless, for a programming language implementation of the formalism, this kind of expansion cannot be achieved for infinite data types nor it should be done, for instance, when we are using types like `Object` classes.

In this sense, a different solution must be provided to solve this expansion problem. Such a solution is the execution of the input operator without expansion using a symbolic execution approach, in order to avoid expansion at all. This is important in order to allow inifite data types in prefix communications.

2.3.1.3 Generalized Parallelism

In Roscoe's CSP two or more process can synchronize, which is called multi-synchronization of processes, and give rise to another sort of problem specifically discussed in Section 2.3.3. In Hoare's version of the parallelism operator, only two process can synchronize on a shared channel name. With Roscoe's generalized parallelism, it is possible to build more flexible networks than with Hoare's version

Another important difference is the fact that the new operator accepts synchronization on a communication (i.e. channel and value pair — $c.v$) instead of on a channel. This opens the possibility to make use of the formalism of data types, which brings up some data dependency problems what in turn gives rise to another sort of problem specifically discussed in Section 2.3.2.

An important point that must be observed is that Roscoe's version of the operator looses the associative property, which could be a problem. This leads the operator to have a weak associative property when both interfaces are the same

$$P[|X|](Q[|X|]R) = (P[|X|]Q)[|X|]R$$

but the possibility, in $P[|X|](Q[|Y|]R)$, of X containing an event not in Y that Q and R can both perform, makes it hard to construct a universally applicable and elegant associative law.

When a very complex network needs to be analysed by the FDR model checker, the state explosion of the network can starve the tool. With the associative law, a specification designer can check each part of the parallelism and infer that the whole composition satisfies the observed properties. This is not true if the associative law does not hold.

2.3.1.4 Multidimensional Prefix

The multidimensional prefix is an extended version of the normal prefix that accepts a mixture of input and output operations at the same communication. In this sense, it extends the concept of communication from a pair of a channel and a value ($c.v$) to become a channel and a list of values ($c?x!z?y!w$), each one following the underlying input or output law.

With this new operator, the expressive power of the language is again raised due to the possibility to include data type constraints and functional language expressions inside this kind of construct, leading to complex communication prefixed constructs like

$$c?x : \{0 \ldots 2\}!(y + z)?w : \{3 \ldots\}!(k * 5) \rightarrow P$$

This tightly defined communication protocol is called multi-way rendezvous [13, 131] communication.

2.3.2 Backtracking

In the following sections we mention some operational problems related to the imeplementation of these new considerations of the Roscoe's CSP. When we start dealing with infinite data types, expansion of all possible paths of the network are not more possible. In this sense, a different symbolic approach must be provided. This new solution gives raise to a new problem of communication ability that is called backtracking problem.

Backtrack here refer to the ability of discarding a selected network path to try another available one, without effectively communicating anything. By path here, we mean the external environment selection for a communication that fires a specific

```
channel aux, a, b, c: int

P = aux?x -> (x >= 0 && x <= 3) & a?x -> b!x -> P
Q = c?y -> aux!y -> a!y -> Q
R = (P || Q) \ {aux}
```

Figure 2.1: Data Dependency Example — Hoare's Version

operator operational semantic law. For analysis tools like FDR [33], this problem does not exist at all, since since they expand all possible paths that the network can reach. Doing so, FDR avoids the backtracking problem through expansion, with the trade-off to do not deal with infinite data types. However, in tools whose objective is to simulate, prototype, or implement an specification, it is not possible to expand all available paths of the network due to the possible presence of infinite data types.

Backtracking happens basically due to the constraint set of the read prefix ($c? : x : T$) operator. It occurs because, without expansion, the synchronization with a writer prefix occurs if both processes are enabled to communicate, but due to the constraint restriction, it may not effectively be performed.

In this sense, backtracking can be solved with the external choice ($\Box$) and/or generalized parallel ($[|X|]$) operators. When the external choice or generalized parallel operator becomes ready to run, it can preview whether or not some communication will or not put the whole network in a backtrack state, and what possible communication paths can be taken.

In the following subsections we present some illustrative examples of backtracking and how it can be solved using the two mentioned versions of the CSP formalism. The first example uses the data dependency condition for backtracking. The next one uses the combinatorial choice selection for the same purpose.

2.3.2.1 Data Dependency Example

Let P be a process that reads values between 0 and 3 (inclusive) on a channel a of integers. Then it writes the read value on another channel of integers named b. Finally it starts again behaving like P.

Let Q be another process which reads any value from a channel of integers named c. Then it writes the read value on channel a, that value is used by P for read operations. Finally, it starts again behaving like Q.

Let R be a third process which represents the parallel composition of P and Q synchronizing on all possible communications on the shared channel a. These two processes ought to behave like a one-sized buffer that communicates values sharing a channel; it is also called rendezvous communication.

In one possible execution path of the process R, process Q can read any value

```
channel a, b, c: int

% Build a set containing x' that comes from {0...3}
T = {x' | x' <- {0..3}}

P = a?x:T -> b!x -> P
Q = c?y -> a!y -> Q
R = P [|{|a|}|] Q
```

Figure 2.2: Data Dependency Example — Roscoe's Version

from channel c and write it on channel a. Since the Q process is writing in a channel inside the synchronization set, it must wait for P to read from this shared channel. Another possible execution path of R is the process P executing first. In this case, it also waits for process Q to write the value that it reads. When processes P and Q reach the synchronization condition, the communication takes place and the processes continue their executions.

Attention should be given to the fact that there is a data dependency between the execution descriptions. For instance, if Q reads from channel c some value outside the integer type constraint of P, a deadlock condition arises on the buffer.

The problem treated here is the abstraction of the data dependency and what must be done in order to express it in a higher level. The description of this example in Hoare's and Roscoe's CSP respectively, are given in machine readable CSP in Figures 2.1 and 2.2. This example is discussed below for each different version of CSP.

Hoare's CSP

In Hoare's CSP [60, 61], the data dependency problem may be solved adding new process guards to the data dependency restriction, and auxiliary channels for independent communication on the synchronization points. To avoid the possible deadlock situation, the process must accept any communicated value inside the *integer* type range through the auxiliary channel *aux*. After that, it must filter this value using a boolean guard that states the data restriction on the value communicated (in this case, to be inside the closed interval $[0 \ldots 3]$). This means that to solve the problem some actions are needed; they are given below.

1. Identify the synchronization points;
2. Auxiliary communication channels must be added for all identified synchronization points;

```
P = ( a.0 -> b.0 -> P ) [] ( a.1 -> b.1 -> P ) []
    ( a.2 -> b.2 -> P ) [] ( a.3 -> b.3 -> P )
```

Figure 2.3: Data Dependency Example — FDR expansion

3. Add boolean guards that applies the data restriction after the auxiliary communication channels for all identified synchronization points;

4. Hide the auxiliary channels[4].

This solution of boolean guards and auxiliary channels can be achieved easily for single process networks, but there seems to be no way to generalize it for arbitrary network in a compositional way. Also, with this kind of solution, the process specification becomes polluted with auxiliary channels outside the problem domain. With the proposed backtrack strategy, a more transparent solution to this problem is given.

This problem does not arise if we do not deal with data types as in the case in [61], which deals with control behaviour only. With the possibility to deal with data types, the formalism gains expressive power and also the ability to be part of unified languages [62] like Circus [148, 149] or combined ones like CSP-OZ [31].

Roscoe's CSP

Roscoe's version of CSP has built-in support for functional language expressions and data type descriptions. With this sort of constructor, set comprehension or data type definitions can be used as the type constraint to restrict the possibly communicable data.

In the example of Figure 2.2, the channel a of Q communicates some value y input by the channel c. The process P is enabled to communicate through the input channel a. However, when the value is received, the whole communication may fail, due to the type constraint T in a on P.

Just to make a comparison, in analysis tools like FDR, this situation is solved, provided the data type of the channel (in this case the interval 0..3 of int) is finite. The solution is a consequence of the expansion of all possible communicable events on the channel. That expansion strategy is guided by applying the step-laws of the CSP operators [124]. In this case, the external choice step-law is applied. The result of the expansion of P by FDR is shown in the Figure 2.3.

This expansion strategy elegantly avoids the problem for finite data types. It does not loose the compositional property of the specification, since it uses an algebraic (transformation) law that gives a semantically equivalent process. It also

[4]The hide operator ($P \setminus X$) is not discussed here; its definition in JACK is mentioned in Section 4.2.4.2. It is completely described by Hoare in [61].

```
channel aux, a: int

P = aux?x -> (x % 2 != 0) & a?x -> STOP
    []
    a!2 -> STOP
Q = (aux!2 -> a!2 -> STOP [] aux!3 -> a!3 -> STOP)
R = (P || Q) \ {aux}
```

Figure 2.4: Combinatorial Choice Example — Hoare's Version

does not add any idiosyncratic constructor (i.e. auxiliary channels) to the original CSP specification, as in the Hoare's version discussed previously. In this way, this solution is elegant and adequate to solve the problem for finite data types.

Nevertheless, this can be applied only for specifications with finite data types. How to expand an infinite type like `Object`, that can occur in a specification runner like JACK? This suggests that we need a general solution for any data type, either finite or infinite. This topic is discussed in Section 2.3.2.3.

2.3.2.2 Combinatorial Choice Example

Let P be a process that makes an external choice between two prefixes, one that reads some integer value x from a channel a with a type constraint T that restricts the communication to odd numbers only, and another that writes the value 4 on the same channel a. After the communication, both prefixes behave like a broken machine (i.e. $STOP$).

Let Q be a process which makes an external choice between two prefixes, one that writes an integer value 2 on a, and another that writes the value 3 on a. After the communication both prefixes behave like a $STOP$.

Let R be another process formed of the parallel composition of P and Q, synchronizing on all possible communications on the shared channel a. The description of this example in Hoare's and Roscoe's CSP, respectively, are given in machine readable CSP in Figures 2.4 and 2.5.

In one possible execution path of the process R of Figure 2.5, processes P and Q behave like one of the choice branches enabled. Process R selects the possible communications. In this case all branches are enabled, since all of them want to communicate through the channel a. For this example, suppose that the left side branch of P and Q are selected by the implementation strategy of the related operators. In this way, Q is wishing to communicate 2 on channel a, and P wishing to read from this shared channel a. So the synchronization is enabled to occur and the communication can proceed. Nevertheless, since P does not know the value that Q will communicate, it must wait for that communication. After that, it can perform

```
channel a: int

% Type constraint that builds a
% set containing odd numbers only.
T = {x' | x' <- int, (x % 2 != 0)}

P = a?x:T -> STOP
    []
    a!2 -> STOP
Q = (a!2 -> STOP [] a!3 -> STOP)
R = P [|{|a|}|] Q
```

Figure 2.5: Combinatorial Choice Example — Roscoe's Version

the type constraint check to accept odd numbers only. From an operational point of view, P must notify R that the communication could not occur, and a backtracking must be done. Some other possible path must be tried, and the already executed ones must be marked as invalid for this communication example.

The relevant problem here is the need to make a combinatorial analysis of possible choice selections. This is needed in order to either find a possible successful communication path, or to ensure that there are no available paths and, therefore, the process must deadlock. In this simple example the job seems easy, but this might not be the case in a more realistic example. In a generic machine that runs the process, a deep nested combination of choices must be analysed, one against all the others. This is what is done in order to consider all possible paths, before notifying a deadlock condition.

This example is discussed below for each different CSP approach available.

Hoare's CSP

Again, to avoid the possible deadlock situation, the process must accept any communicated value inside the integer type range through the auxiliary channel aux. After that, it must filter this value using a boolean guard that states the data restriction on the value communicated. This means that the specification will again become polluted with unexpected information.

In this case, the solution does not solve the problem. When the communication happens on the auxiliary channel aux, if the type constraint is not satisfied, it behaves like $STOP$, and no backtracking is possible due to the already communicated and synchronized value. Therefore, the solution described previously does not work for this problem. If there are nested choices, as in our present example, the solution

```
int = 0..3
channel a: int

P = (a.1 -> STOP [] a.3 -> STOP)
    []
    a!2 -> STOP
Q = (a!2 -> STOP [] a!3 -> STOP)
R = P [|{|a|}|] Q
```

Figure 2.6: Combinatorial Choice Example — FDR expansion

is even more complicated.

Roscoe's CSP

As considered before, the left branch of P and Q are selected in the execution, then the process P is enabled to communicate through the input channel a. However, when the value is received, the whole communication fails due to the type constraint T in a on P.

In analysis tools like FDR, this situation is indirectly solved, provided the data type of the channel (in this case an *int* range) is finite. The solution is achieved by the expansion of all possible communicable events on the channel. Assuming that *int* type ranges from 0 to 3, the type constraint yields the following constraint set: $\{1, 3\}$. The same example expanded by FDR is shown in the Figure 2.6.

Differently from the process obtained using Hoare's approach, no deadlock occurs because the left side of P is never selected, since there is no matching communication for it. In this way, the right side is chosen. In this sense, this solution is elegant and adequate to solve the backtrack problem for finite data types.

2.3.2.3 Infinite Data Types

As mentioned above, the expansion strategy is not appropriate to deal with backtrack for infinite data types. Instead of this, a symbolic approach that applies all operational semantic laws, as the expansion approach does, is considered.

The symbolic approach is definitely more adequate for both infinite data types, and finite as well. With the symbolic approach, the expansion problem for infinite data type is avoided. The problem is the starvation of the network due to the attempt to expand all possibilities.

In the formal methods research field, a symbolic solution does not seem to have yet been explored, for a process algebra implementation in a programming language. A general solution for the backtracking problem is an important motivation.

The symbolic approach seems to be more appropriate to solve the backtracking problem in general. It is also easier to operationally implement than full expansion. This is the approach for the operational implementation of CSP used in the JACK framework. This decision was made based on previous research on the available implementations [59, 107, 32], and on an experimental CSP system that has a quite small multi-dimension data type [45, 99, 100].

Another research was done in the direction of finding and defining how this solution should be implemented. In [36, 35, 82] it is stated which paths the implementation should follow, and which it must avoid. In [36], it is observed that available process algebra implementation libraries [59, 107] mix functional (i.e. operator semantics) and non-functional (i.e. concurrency and synchronization model) concerns. This breaks the reusability and extensibility property of object-oriented frameworks, since that mixture in those libraries code makes it complex and obscure. This topic is reviewed in Section 3.2.

In [82], we explain how to implement the operational semantics [126, 124] of CSP, using action semantics [97, 14]. Other important sources of information are [99, 98, 100, 12, 20, 31].

2.3.3 Multisynchronization

Multisynchronization is the situation where nested levels of parallelism need to be synchronized together. In the original version of CSP, this problem never occurs due to the restriction that only two processes can be involved in a communication. In that version, the parallel operator synchronizes independently of any other operator or state of the whole specification network. In this sense, it is completely autonomous with respect to the work that another parallel operators are doing.

Roscoe's version of the generalized parallel operator has a controlled autonomy, managed by its synchronization alphabet and the external environment event selection. Multisynchronization can lead the network to backtracking, due to a possible communication path that will not multi-synchronize. The implementation of the operator must backtracks to attempt another way, if one exists. This is better illustrated with examples. In the following sections we present some illustrative examples of the multisynchronization problem and how it is interpreted using the two mentioned versions of the formalism. We also show the consequences of the presence of multisynchronization and backtracking together.

2.3.3.1 Read Multisynchronization

Two illustrative examples of the read multisynchronization are given below.

Prefix Multisynchronization

```
channel c

A = (c?x -> STOP)
B = (c!1 -> STOP) || A
C = (A || B)
```

Figure 2.7: Multisynchronization Example — Hoare's Version

```
channel c: {0..1}

A = (c?x -> STOP)
B = (c!1 -> STOP) [|{|c|}|] A
C = (A [|{|c|}|] B)
```

Figure 2.8: Multisynchronization Example — Roscoe's Version

Let A be a process which reads any value from channel c and then behaves like $STOP$. Let B be a process which is the parallel composition of A and an unnamed process that writes the value 1 on channel c, and then behaves like $STOP$. The parallel composition in B must multi-synchronize on all possible communications on channel c. Let C be the top level process formed of the parallel composition of A and B synchronizing on all possible communications on the shared channel c. In Figures 2.7 and 2.8 we present the machine readable CSP description of this example in Hoare's and Roscoe's version of CSP, respectively. In the following paragraphs, an implementation view of how each CSP version can be solved is given.

In Hoare's version, the execution of process C starts both processes A and B. Then, process A waits for a communication on channel c and process B will synchronize performing the communication $c.1$ and follow the execution to synchronize and wake up process A. Process C in turns synchronizes, and the whole process finishes behaving like $STOP$. This does not mean multisynchronization. In Hoare's version, the synchronization of processes can only occur in pairs.

In Roscoe's version, process C must inform that both A and B that they must synchronize on all possible communications of c. This means that these processes have autonomy, but they are controlled by this information, which is given by an outer parallel operator, that acts like a supervisor environment. Process A waits for a communication on channel c. Process B does not synchronize performing the communication $c.1$, because it knows that its supervisor must also synchronize on that communication. Process B must agree with its supervisor and waits for a decision from it before the communication can occur. Finally, process C signals the decision about the communication $c.1$ for its pupils that in turn make the whole multi synchronization of this example, and then finish behaving like $STOP$.

```
channel c: {0..1}

P = (c!1 -> STOP)
Q = (c?x -> STOP) [|{|c|}|] (c?x -> STOP)
R = (P [|{|c|}|] Q)
```

Figure 2.9: Choice Multisynchronization Example — Roscoe's Version

There is an important difference between these two views of the parallel operator. When we include data dependency between processes, the supervisor can decide if the selected path leads to a non-desired situation, like the ones shown in the backtracking examples above. With this fine control concerning the occurrence of a communication, the multisynchronization becomes possible, which does not occur in the Hoare's version. Next, another example where this situation can cause trouble and why the operator raises its expressive power is shown.

2.3.3.1.1 Choice Multisynchronization

Let P be a process which writes the value 1 on a channel of integers named c and then it behaves like $STOP$. Let Q be a process which is the parallel composition of two unnamed processes that just read any value on channel c and then behave like $STOP$. The parallel composition in Q must synchronizing on all possible communications on channel c. Let R be another process formed of the parallel composition of P and Q synchronizing on all possible communications on the shared channel c.

In Figure 2.9, we present the machine readable CSP description of this example in Roscoe's version of CSP. It is not possible to describe this example using the Hoare's version of CSP.

In Roscoe's version, process R informs both P and Q that they must synchronize on all possible communications of c, before starting them. This means that these processes have autonomy, but they are controlled by this information given by an outer parallel operator that acts like a supervisor environment. Process P waits for a communication on channel c. Process Q informs the situation to its supervisor asking if it must synchronize on channel c. The supervisor process R knows that it has another process waiting to synchronize on the requested channel. In this case, the multisynchronization can take place. Process R informs both processes about the situation which in turn makes the multisynchronization on the communication $c.1$. Finally, after the successful communication $c.1$, it finishes behaving like $STOP$.

There is also another similar multisynchronization problem not mentioned here related to write multisynchronization. It is very similar to the examples shown

```
typeOfc = {0..4}
channel c: typeOfc

% Type constraint that builds a set
% containing even numbers only - {0,2,4}.
T = {x' | x' <- typeOfc, (x' % 2) == 0}

P = (c!1 -> STOP [] c?x:T -> STOP)
Q = (c!2 -> STOP [] c!3 -> STOP)
S = (c?x -> STOP)
X = (P [|{|c|}|] Q)
R = (S [|{|c|}|] X)
```

Figure 2.10: Multisynchronization with Backtracking — FDR input

above, but instead of read prefixes, write prefixes are playing. In the following, an interesting example that mix multisynchronization with backtracking is shown.

2.3.3.2 Multisynchronization with Backtrack

Backtracking can also occur due to a multisynchronization. For instance, when combining the examples of the previous subsections, such kind of situation can arise, as shown in Figure 2.10.

In one possible execution path of process R, both process P and Q can select the right operand of the external choice operator since they are both enabled to communicate and synchronize. Assuming that the process is not expanded, because $typeOfC$ can be infinite, the supervisor process R cannot previously know that there is a data restriction on the selected side of P that forbids communication on odd numbers. Process S waits for a notification from its supervisor process (R) to continue.

When process X is notified by one of its pupils about the impossible communication, X in turn notifies the situation to R. This procedure will follow the network structure until a process without a supervisor is found, which in turn must make the decision about the whole network. In the given example, R asks X if there is other execution possibility, then X forwards the request to its pupils in order to try another path.

This attempt-oriented approach is necessary since we cannot previously infer a right decision because no expansion was done. In this simple example, the only possible available path is to try the left operand of Q against the right operand of P. Luckily, this decision allows the communication to occur at X, which must be informed to the supervisor. Finally, process R signals every pupil that the multisyn-

chronization can take place, since process S is able to perform the communication desired by X.

Therefore, to run this specification example, we need to both make use of backtracking and multisynchronization. Backtracking is used to solve a data dependency problem, searching for alternative paths availability. Multisynchronization is used to allow the multi-level synchronization, and also to allow the backtracking to solve the whole synchronization, that otherwise leads the main process R to deadlock.

In this way, we briefly explain our approach to implement and solve these problems operationally in a programming language implementation of CSP. As mentioned in Chapters 5 and 6, this supervision-oriented approach follows the guidelines stated by Hoare in [61, pp. 38] to describe generic processes, and operational laws described in [126, 124] to describe CSP operators semantics.

2.3.3.3 Multisynchronization and Backtracking Implementation

It is important to deal with both multisynchronization and backtracking in a framework implementation of Roscoe's CSP. These problems arise because we are also dealing with infinite data types. Nevertheless, both tools do not allow infinite data types. Dealing with these sort of problems is, therefore, the trade off to have an implementation that accepts infinite data types. The problems mentioned are the mix of data dependency, combinatorial choice selection, and multi-level synchronization.

An implementation of this sort of features has shown to be quite complex and tricky. Such kind of implementation does not seem to have been adopted in any available process algebra library (see Section 3.2). In the following chapters, we present what tools are available, how to properly build a framework to deal with this sort of problem, and how the JACK framework is organized in order to achieve these objectives.

2.3.3.4 Strategy Selection

A detailed study [36] was done for each mentioned problem, in order to infer how a framework implementation of CSP with generic user process definitions can be done. The implementation must generically deal with both CSP and user process definitions without mixing functional semantic problems with non-functional concurrency problems.

Many strategies were considered for the implementation of the framework. For instance, we firstly try to strictly follow the operational semantics laws described by Scattergood [126], but it does not fit well for our problem domain due to our requisite to allow transparently both CSP operators and user processes. After that, a network search algorithm was tried, in order to properly implement the multisynchronization stop condition and the backtrack path availability and selection possibilities; this solution again fails because we loose the compositional property of

the operators. Finally, we found a solution that solves both CSP operators and user processes generically based on guidelines defined by Hoare [61, Chapter 1]. These guidelines are described in Chapters 5 and 6.

2.4 Final Considerations

The CSP language's most basic operators have been introduced as well as some implementation difficulties that arise in the new Roscoe's version of the formalism. Those problems have been exemplified.

This chapter has also mentioned the already known problems to be treated by an implementation of the new version of CSP, like backtracking, multisynchronization, and new operators complexity. The next chapter presents some discussion about how to deal with these problems in a framework that implements CSP in a concurrent object-oriented programming language. It also compares other available process algebra libraries and modeling techniques.

Chapter 3

Framework Construction

This Chapter presents an overview about framework construction using design patterns and pattern languages. It also shows related process algebras and general purpose framework studied in the process of our framework construction. The JACK framework's main objectives, the design pattern initiative, and other aspects related to those tools are also presented.

In Section 3.1, a brief description of frameworks, design patterns, and pattern languages are given; that section also provides some discussion about the adequacy and importance of frameworks. Next, in Section 3.2, a list of related component libraries, frameworks, and methodologies is given. In Section 3.3, it is presented a brief discussion about the Java language selection to build JACK. After that, in Section 3.5, the JACK main objectives are defined and compared in Section 3.6 with other process algebra implementations [59, 107]. In Section 3.4, the modeling technique selected to design JACK is briefly presented. In Section 3.7, final considerations are presented.

3.1 Relevance of Framework and Design Patterns

Frameworks are an important kind of class library organization that specifically describe how each library entity is defined and how it should be integrated. The process of framework construction involves many steps and guidelines.

Framework modeling techniques are also important in order to make the framework widespreaded available. This modeling is normally achieved by using well-known design patterns to solve selected problem of a specific domain.

3.1.1 Framework

A framework is represented by a set of recurring relationships, constraints, or design transformations in different aspects of object systems. Frameworks can be viewed as an implementation of a Design Pattern [28], or a related set of design patterns.

In the following, we give some definitions of what a framework is, what it can do, when or where it can be applied, and so forth. These definitions vary from more abstract to more concrete from a set of different sources.

1. "A framework is a generic architecture that provides an extensible template for applications within a domain" [74, Chapter 13 pp.284]. "It is a package consisting of design patterns"[74, Chapter 14 pp.504].

2. A framework is at the heart of many patterns, but a pattern usually also includes less-formal material about alternative strategies, advices on when to use it, and so on. When one keeps a framework in a class library, it should be packaged with this documentation.

   A framework is a template package; a package that is designed to be imported with substitutions. It "unfolds" to provide a version of its contents that is specialized based on the substitutions made. A framework can abstract a collaboration, a generic type, a design pattern, and even a bundle of generic properties.

   "Therefore, it is more than a collection of collaborating abstract classes; they have companion documentation, related modeling decisions, and well-defined ways of use and integration with other frameworks" [22, Chapter 9 pp.340—341]. "We use frameworks to explicitly document the mapping from domain terms to terms and roles in the abstract problem descriptions" [22, Chapter 6 pp.279].

   "In this way, model frameworks can be used to express relationships that straddle type boundaries and to encapsulate relationships made up of a collection of types, associations, and actions. They are a powerful tool for abstraction and a useful unit to reuse" [22, Chapter 9 pp.380].

3. "Use frameworks to build specifications and designs as well as refinements between the two. Used properly, frameworks let one better document the main refinement decisions. This results in a design that is traceable" [22, Pattern 6.5 pp.283].

4. "A common aspect about frameworks is that they are adaptable. The framework provides mechanisms by which it can be extended (white-box reuse), or directly used (black-box reuse)" [22, Chapter 11 pp.461]. "In this way, a framework implementor ought not to build models from scratch, but instead build them by composing frameworks" [22, Pattern 10.3 pp.448].

5. "A framework is a set of cooperating classes that make up a reusable design for a specification of software. The framework dictates the architecture of your application. It defines the overall structure, its partitioning into classes and

object instances, the key responsibilities, how the classes and objects collaborate, and the thread of control. The framework captures the design decisions that are common to its application domain. Frameworks thus emphasize design reuse over code reuse. An added benefit comes when the framework is documented with the design patterns it uses" [28, Chapter 2 pp.26—27].

6. A good framework can reduce the cost of developing an application by an order of magnitude, because it lets one reuse both design and code. Frameworks take a common path while they evolve. This can occur in the following steps [123, Chapter 26]:

   (a) Application Examples — How to start designing a framework?

   (b) White-box Framework — How to properly extend a framework after its construction?

   (c) Black-box Framework — How does the framework exposes its services to the final application domain?

   (d) Component Library — When one is using white-box frameworks, similar objects must be implemented for each problem the framework solves. How to avoid rewriting similar objects on different framework instantiations?

   (e) Hot Spots — How do one eliminate common code inside the mature framework while the code evolves?

   (f) Pluggable Objects — How to add components to frameworks?

   (g) Fine-Grained Objects — How to refactor the framework to increase reuse?

   (h) Separation of Concerns — How to combine frameworks?

   In this sense frameworks are adequate for the construction of evolutive, and complex object-oriented systems.

7. "A framework is a model of a particular domain or an important aspect thereof. A framework can model any domain, be it a technical domain like distribution or garbage collection, or an application domain like banking or insurance. A framework provides a reusable design and reusable implementations to clients. A framework is represented by a class model, together with" [113, Chapter 4 pp.54]:

   A free role type set — a set of roles in which framework clients may participate.

   A built-on class set — a set of classes from other frameworks that this framework depends on.

An extension-point class set — a set of classes that may be subclassed by framework-external classes.

Object-oriented frameworks promise higher productivity and shorter time-to-market for the development of object-oriented applications. These goals are achieved through design and code reuse. While many projects show that these promises can be met, failed projects also show that they are not always easy to reach. Role modeling for framework design addresses three pertinent technical problems of designing, learning, and using object-oriented frameworks: complexity of classes, complexity of object collaboration, and lack of clarity of requirements put upon use-clients of a framework

"Role modeling for framework design combines the strengths of role modeling with those of class-based modeling, while leaving out their weaknesses. It is therefore an evolutionary extension of current methods that preserves existing investments" [113, pp. iii].

"A good framework's design and implementation is the result of a deep understanding of the application domain, usually gained by developing several applications for that domain. The framework represents the cumulated experience of how the software architecture and its implementation for most applications in the domain should look like. It leaves enough room for customization to solve a particular problem in the application domain" [113, Chapter 1 pp. 2]. "A good framework has well-defined boundaries, along which it interacts with clients, and an implementation that is hidden from the outside" [113, Chapter 2 pp.8].

Attention should be given to another aspect that the framework solution approach raises. "If applications are hard to design, and toolkits are harder, then frameworks are the hardest of all" [28, Chapter 2 pp.27]. This trade off seems acceptable when comparing different approaches to process algebra implementations [59, 107] (see also Sections 3.2 and 3.6) against the resulting design.

3.1.2 Design Patterns and Frameworks

A design pattern represents a model of a specific domain or concern. Design patterns can be viewed as design artifacts used to document, abstract, and make discussing with a development team easier.

Because patterns and frameworks have some similarities, people often wonder how or even it they differ in some way. It is actually very important to clearly distinguish between them. Frameworks are different from design patterns due to three main reasons [28, Chapter 2 pp.27]:

- Design patterns are more abstract than frameworks — differently from design patterns, frameworks cannot only be studied, but also be executed and reused directly.
- Design patterns are smaller architectural elements than Frameworks — a framework normally implements several design patterns, the reverse is never true.
- Design patterns are less specialized than frameworks — frameworks comes from an application domain, design patterns [28] can be used in any kind of application.

In the following, we give some definitions of what a design pattern is, what it can do, when or where it can be applied, and so on. These definitions are given from more abstract to more concrete descriptions from a set of different sources.

1. "A design pattern is a common solution to a common problem in a given context" [56, Chapter 28 pp.383].

   "A design pattern is a parameterises collaboration that represents a set of classifiers, relationships, and behaviour that can be applied to multiple situations by binding elements from the model to the roles of the pattern. It is a collaboration template" [74, Chapter 13 pp.387].

2. "A design pattern is a set of ideas that can be applied to many situations" [22, Chapter 9 pp.341].

   "Design Patterns enable designers to discuss their designs clearly with their colleagues and to convey sophisticated ideas rapidly" [22, Chapter 6 pp.669].

3. "A design pattern systematically names, motivates, and explains a general design that addresses a recurring problem in object-oriented systems. It describes the problem, the solution, when to apply the solution, and its consequences. It also gives implementation hints and examples. The solution is a general arrangement of objects and classes that solve the problem. The solution is customized and implemented to solve the problem in a particular context" [28, pp.360].

   The design patterns show how to use primitive techniques such as objects, inheritance, and polymorphism. They show how to parameterise a system with an algorithm, a behaviour, a state, or the kind of objects it is supposed to create. "Design patterns provide a way to describe more of the *why* of a design and not just record the results of your decisions" [28, Chapter 6 pp. 353].

The description of a pattern is done in a consistent format. "Each pattern may be divided into sections which leads to a uniform structure to the information, making the design pattern easier to learn, compare, and use" [28, Chapter 1 pp.6].

These templates format evolved on time to acquire more complex and extended features of object-oriented design. Doug Lea proposes a new template format for concurrent systems [81]. Rito Silva advance in this field, proposing yet another extended template that captures the composition of complex patterns as a design pattern by itself [133].

4. A design pattern is the abstraction from a concrete form which keeps recurring in specific non-arbitrary contexts. A pattern is frequently described as a problem/context/solution triple. Design patterns identify, name, and abstract common themes in object-oriented design. They preserve design information by capturing the intent behind a design. They identify classes, instances, their roles, collaborations, and the distribution of responsibilities.

   Despite much recent work on design patterns, many misunderstandings about patterns remain. Perhaps the most common and most harmful misunderstanding is to take the structure diagram of a design pattern description as the expected structure of a design pattern implementation. The structure diagram and the description of its participants are an illustration of one common form of the pattern. Therefore, a design pattern is an abstract idea that defies formalization and precise definition [113, Chapter 3 pp. 49].

Frameworks are normally divided into layers that solves specific problem domains. The definition of framework layers is also an important aspect of the framework construction. Each layer must capture a specific aspect to solve, and provide the solved functionality at well-defined and widespreaded interface points. The framework team ought to start the development with these definitions very clear, in order to properly use these powerful concepts.

3.1.3 Pattern Languages

Aside from design patterns, there are pattern languages. A pattern language is more than a pattern collection or catalogue. It explains how patterns are applied to a more general problem than the one solved by a single pattern

Some definitions from a different set of sources of what a pattern language is, what it can do, when or where it can be applied, and so forth, are presented below.

1. "Design patterns describe solutions for design problems, but they do not define how they should be applied in the more general context of the development process. Thus, from an object-oriented programming perspective, the concept

of pattern language as defining a set of activities for program construction is also introduced. These activities integrate design patterns in a program generation process" [133, Chapter 3 pp. 36].

2. "A pattern language may be an alternative solution to parts of a problem described by a set of related design patterns. A pattern language is responsible, therefore, for factoring the problem and its solution into a number of related problem-solution pairs, capturing each as a pattern in a collection of patterns" [121, pp.102].

3. "A pattern language is a collection of patterns that build on each other to generate a system. A pattern in isolation solves an isolated design problem; a pattern language builds a system. It is through pattern languages that patterns achieve their fullest power" [19, Chapter 2 pp.17].

The use of pattern languages establishes a well-defined methodology for proper usage and composition of design patterns.

3.1.4 Conclusion

The definitions above are sufficient to motivate us to build our CSP library implementation using frameworks, design patterns, and pattern languages due to its complexity and size. The definitions also suggest that these solutions are very reasonable for our problem domain.

As discussed shortly (in Sections 3.2 and 3.6), there are many desired properties for this kind of implementation, and also a lot of mistakes and problematic situations to avoid. In order to clearly identify these points, the proposed library is strongly based on framework, design pattern, and pattern language concepts.

We think that this construction approach is very important to any complex object-oriented library. Despite this fact, dealing with CSP processes operational semantics [126] implemented using the language threading support is a complex problem by itself. Therefore, the so called *pattern initiative* seems to be imperative to build an extensible, reusable, modular, and yet simple process algebra implementation in Java [72].

With that decision, there are some important questions that need to be answered. Some of them are listed below:

1. How many and which layers will the framework need to have?

2. How many and which responsabilities does each identified layer deal with?

3. Which patterns and pattern languages to use in other to achieve the responsibility goals?

4. Are these patterns implementations or designs available?

5. Which modeling technique to use in order to accurately describe the framework roles?

6. Is the framework prepared for gray-box reuse?

These questions are answered in Chapters 5 and 6. They define the JACK framework architecture details which includes its layers, design patterns, and design pattern templates[1] [113, Chapter 3]. In Chapter 6, a detailed discussion about how those layers must interact and their services and configurations are given.

The JACK framework is built-on one other framework. It is called JDASCO[2]; it deals with the non-functional requirement of JACK: support for concurrency, synchronization, and recovery. This is done in order to leave the heart of JACK to deal only with its functional requirement: provide support for processes in Java with the operational semantics of CSP [124].

The most complex part of our framework implementation was identified [36] as the concurrency and synchronization schemes to execute CSP processes and deal with problems like multisynchronization (see Section 2.3.3); recovery schemes to solve the backtrack problem mentioned in Section 2.3.2; and implementation of CSP language operators and user processes. In other libraries [59, 107], these aspects are mixed with process semantics, which makes the understanding of the library internals for extensions very difficult, leading to a loss of modularity.

The decision to build the library as a framework with well-defined layers and problem domains has shown to be very important in order to achieve desired software engineering properties like modularity, reusability, simplicity, and so on. The selected thread support library must both be designed as a framework, with design patterns, and explained by a pattern language.

The composition of the non-functional layer exposed services with the functional layer responsibilities is another complex task. In other words, the composition of concurrency and synchronization schemes with CSP processes operational semantics is not trivial. A presentation of the layers is given in Chapter 5, as already mentioned. A detailed discussion about their composition is given in Chapter 6.

The Coplien's tutorial about pattern writing [19] is a philosophic discussion about design patterns and pattern languages. In that work, all concepts mentioned here are clearly identified and summarized, many different kinds of pattern and

[1] A design pattern template is a design pattern instantiated in a generic purpose programming language, in our case Java.

[2] Java Developing Applications with Separation of Concerns. This is a Java implementation of the original framework called DASCo [132, 133]. This framework is basically the composition of three complex design patterns: Object Synchronization [102, Chapter 8 pp.111], Object Recovery [123, Chapter 15 pp.261], and Object Concurrency [133, Chapter 4].

pattern languages formats are presented, and also some not mentioned (but not less important) concepts like antipatterns are provided.

3.2 Libraries Related to Process Algebras

In this section, a brief discussion about available process algebras and related implementations is given.

3.2.1 JCSP

Java Communicating Sequential Process (JCSP) [107] is a Java library that implements occam [53] primitives like guards, alternatives, parallelism, channels etc. Here, we present an overview of the library, its main advantages and deficiencies.

The terms Java threads and processes are very close. Assigning a thread to an object creates an active object [81, Section 4.5] (a process). In Java, more than one thread may be assigned to an object. The user can control each thread by a diversity of methods, which must be used in a proper way. However, to get synchronization between threads correct is very difficult and therefore error prone.

Programming with Java threads allow concurrent programs to be described but it introduces complexity. The thread of control flows through objects by method invocation. This results in a shift from object-oriented to method-oriented modeling. The complexity increases even more when multiple threads may meet each other on shared variables. To prevent race hazards and invalid states, these threads must be synchronized so that only one thread may update these variables. The Java `synchronized()` keyword provides a critical region around the shared variables. However, this synchronization construct is very expensive for just one single thread of control.

Thinking in terms of threads is less abstract and less cognitive for the developers than thinking in terms of processes. Processes have only one thread of control that is encapsulated within the process and does not overlap the thread of control of other processes. Cooperation between threads is done by communication events (i.e. by sending and receiving messages). Synchronization between processes is purely established by channel communication. Processes do not invoke each other's methods but they communicate through channels.

JCSP is a library tailored to build networks of communicating processes. It conforms to the CSP model of communicating systems. In this sense, it can be brought to bear in the support of Java multi-threaded applications.

In JCSP, processes interact solely via CSP synchronising primitives, such as channels. Processes do not invoke each other's methods. Processes may be defined to run in sequence or in parallel. Processes may be combined to wait passively on a number of alternative events, with one of them triggered into action only by the

external generation of that event. Such collections of events may be serviced fairly (guaranteeing no starvation of one event by the repeated arrival of its siblings), by a user-defined priority or in an arbitrary manner.

JCSP is an alternative to the built-in monitor model for Java threads [72, Chapter 14 and 17]. JCSP primitives should not normally be mixed into designs with synchronized method declarations, instances of the `java.lang.Runnable` interface or `java.lang.Thread` class, or invocations of the `wait()`, `notify()`, or `notifyAll()` methods from `java.lang.Object`. It is interesting to note that JCSP model processes using a simple extension to normal Java threads. Nevertheless, there is a mixture between non-functional controlling code and functional semantic code.

Finally, it is important to note that the JCSP library reflects the occam [53] realisation of CSP, with some extensions to take advantage of the dynamic nature of Java. An occam [53] process declaration maps simply into a class implementing the `CSProcess` interface, whose constructor parameters mirror the process parameters and whose run method mirrors the process body.

This brief introduction is based on the JavaDoc documentation of the JCSP [107] package version 1.0rc—2. For a detailed discussion about the JCSP packages and available functionality see [36, 145, 144].

3.2.1.1 Advantages

The JCSP approach to model processes trusts and uses the thread architecture and thread scheduling policy of Java [72, Chapter 17] to model its concurrent behaviour. With JCSP, existing Java frameworks that interact with user software via listener registration and callback interfaces (such as the standard AWT [55, 64, 63] and Swing [141, 64, 63]) can be easily tailored to operate as processes with channels based on JCSP interfaces.

JCSP also provides a wide range of plug and play components. These components are CSP processes normally used in many different kinds of specifications. This in many cases frees the user from having to start the specification from scratch. It also serves as usage examples and bugless implementation of common specification pieces.

Another important advantage is the separation of channel I/O operations from physical I/O via data store interfaces. Each channel can have a different data store, which can handle different data types of physical I/O. Channel data stores are also used to make a simple kind of multi synchronization between processes.

3.2.1.2 Deficiencies

An important limitation of JCSP is the `Alternative` constructor. It is not declared as a class that implements the `CSProcess` interface, so it looses the compositional property of process declarations expected in CSP specifications. In order to achieve

this composition, the user process is obliged to select and use the alternative guards manually. In this sense, an intuitive process that makes use of the alternative constructor cannot run naturally; the user does not call a `run()` method for an `Alternative`; it must use a different approach that is two low-level. In other words, the user must implement part of the operator, since the operator gives only the selection of process to us; its the user responsibility to run the selected process, which in turn breaks the compositionality of the CSP operators.

`Alternative` is not modeled in such a way to allow backtracking (see Section 2.3.2), which is an essential aspect of our work. When the decision about a communication is taken, there is no easily defined way to go back if some backtracking circumstance occurs. This is due to way the `Alternative` is implemented. Differently from what the JavaDoc introduction says, the `Alternative` implementation makes use of Java monitors (i.e. `synchronized` blocks, and `java.lang.Object` methods — `wait()`, `notify()`, and `notifyAll()`). With a locking scheme handled using notification instead of execution queuing, it is difficult to restore all the monitors state (i.e. involving `Channel`s, other `Alternative` constructs, or `Parallel` processes) consistently with respect to its previous state before the communication had taken place. We must say, however, that in occam [53], backtracking is not a concern; so, JCSP was not meant to handle this problem.

JCSP has other minor deficiencies and problems mentioned below.

- JCSP uses primitive types instead of wrapper class references. This closes the possibility to the user to change the underlying value after submitting the process to run.

- JCSP does not define a type model. This may not be a problem, since CSPm is also typeless. On the other hand, in order to allow type correctness in Java (a strong typed language), JCSP states that for each data type to be communicated, a different `Channel` implementation must be provided. This increases the necessary code to be used.

  For each `Channel` implementation, it is also necessary to define the underlying `ChannelDataStore` responsible for the physical I/O with the desired new type, due to the hard coupling between `Channel` and `ChannelDataStore`. This lets channel extension to be achieved trough the use of a copy-and-paste like programming technique, which makes the whole implementation both less modular, and not safely extensible. Moreover, the data stores have a hard coupling between channels which make them both less reusable and less extensible.

- The JCSP `Parallel` constructor uses `Channel` *names* to synchronize processes: internally it makes use of channels as Java monitors. This is adequate for an occam implementation that uses Hoare's version of the parallel operator, which does not have an explicit alphabet and in which synchronization

is achieved through channel names. Nevertheless, for the generalized parallel operator of Roscoe's CSP, a different approach must be available: we need a parallel operator with explicit synchronization alphabet declaration.

Despite this, it is difficult to leave the whole network stable in the case of a backtracking situation, due to the notification scheme used in JCSP. It uses Java monitor lock methods (i.e. `java.lang.Object.wait()`) instead of a kind of execution history record. This kind of notification makes it difficult to retrieve the state of all processes involved in a communication that needs to backtrack.

- As identified in [36], it is not be possible to make the necessary changes to avoid some of the mentioned disadvantages, since some necessary access methods and classes are declared to be package or private visible; for instance, class `jcsp.lang.Guard` is declared package visible.

It has been observed that, with JCSP, the backtracking problem can be partially solved with many internal changes, and restrictions in the available operators in the language. JCSP neither provides nor points solutions to the generalized parallel operator. This sounds far from a solution to our problem domain.

3.2.1.3 Discussion

We conclude that JCSP is modeled to implement only a subset of CSP rather than a robust extensible framework. This is interesting because it remains simple. In spite of this fact, it implements occam and not CSP. This makes a lot of difference due to the lack of some much more complex CSP operators like hiding, functional renaming, or generalized parallel (see Chapter 2 and [49, Appendix A]).

JCSP also lacks a documented framework, model or design pattern decisions. JCSP does not seem to be built to be extended, but to be used. Therefore, it is inadequate to our goals (see Section 3.5), since it neither solves our problems, nor provides a well-defined way to do so.

Nevertheless, we have learned a lot from its source code and we found and follow many interesting design decisions. For instance, studying the JCSP code while it executes, it was found that the `Alternative` and `Parallel` operators are important to the solution to the backtracking problem.

We strongly believe that a framework model approach is necessary in order to both provide a solution to these new sort of complex problems (and operators), and also to allow the component library to be extended and integrated with other frameworks in a guided way. On the other hand, JCSP is a very useful library, and it inspired our framework design and implementation in many points. We have learned with its advantages and problems in order to try to give a more robust solution.

3.2.2 CTJ

Concurrent Threads in Java (CTJ) [59], like JCSP, implements occam [53] primitives like guards, alternatives, parallelism, channels, and so on. Here, we present an overview of the library and its main advantages and deficiences.

CTJ follows the same philosophy of JCSP in the sense of the treatment of threads and processes; that is, they share the idea of the implementation of active objects [81, Chapter 4]. The concept of programming with processes and communication between processes via channels is also provided by this Communicating Threads for Java (CTJ) package. In CTJ the user needs to specify his or her object to be a CSP process.

A CSP process is an active object with a private thread of control. Processes communicate only by channels and never invokes methods on processes when they are active. A process is an active object when its `run()` method has been invoked by some thread of control and has not yet been returned (i.e. successfully terminated). A process is a passive object when its `run()` method is not invoked. Therefore, a parent process should only invoke `run()` on a child process when its child process is in passive state. Sharing a process by two or more processes is forbidden (design rule) and therefore the `run()` method can never be invoked simultaneously by multiple processes. This simple rule avoids race hazards and strictly separates each thread of control to enable a secure multithreading environment.

This discussion is based on one of the tutorials [50] of the CTJ [59] package version 0.9, revision 10. For a detailed discussion about CTJ packages and available functionality see [36, 50, 51, 49].

3.2.2.1 Advantages

The CTJ approach to model processes is very robust. It builds up a completely new thread architecture to deal with concurrency, as opposed to JCSP [107], which, as previously discussed, is based on the thread architecture of Java. CTJ builds-up a completely new processor and process scheduler abstractions. There are trade offs in both libraries, nevertheless one can find many good convergence points between CTJ and JCSP.

CTJ is a library modeled to capture real-time dependent aspects of systems. The new thread architecture that it implements is important to achieve this goal. For instance, it builds up completely new abstractions like threads, processor, context switcher, critical region, scheduler, and so on. This leads its implementation to be quite complex, but sufficiently accurate to simulate real-time properties. The new thread architecture of CTJ adds many advantages to the library, like fine control over process scheduling policies. For instance, there is a platform independent implementation of thread priorities, and time slicing between context switches.

Differently from JCSP, the CTJ `Alternative` constructor is declared as a process

so that it can `run()` directly, which makes its use very close to that natural for the alternative operator of occam [53]. This makes the use of the alternative more intuitive, easier, and without any user assistance.

The channel modeling has the same limitation as JCSP, that is the use of many channel classes: one for each data type. However, it is better than JCSP, because CTJ data types are always treated as object instances, instead of primitive variables. In spite of this, there is a tool to support the channel skeleton code creation. This is an advantage compared with JCSP, but being typeless is a disadvantage.

The selection of processes and guards is made by specialized queues that control the whole processes. This is far better to control and implement the backtracking solution than the simple arrays and monitor notifications used in JCSP. In CTJ, the user can create a specialized kind of semaphore to deal with concurrency, instead of using Java monitors. This is not an advantage, but makes it more flexible to implement more complex functionalities like backtracking.

An interesting contribution of CTJ is its I/O architecture for channels, with the use of link drivers. A link driver determines the actual way of communication. It can provide internal (i.e. via memory such as a rendezvous or buffered mechanism) or external (i.e. by peripherals such as RS232, PCI or TCP/IP) communication. The link driver can be plugged into a channel and provides a protocol for data-transfer via hardware (i.e. memory or peripherals). The channel provides the necessary synchronization and scheduling. The combination of channels and link drivers is powerful, in that the concept provides a general solution for communication between two or more threads in the same address space (i.e. same memory) or for communication between (possible distributed) systems. As a result, processes become highly independent of their physical location on the total system. This enlarges the portability of the CTJ package.

With CTJ link drivers, existing Java frameworks that interact with user software via listener registration and callback interfaces (such as the standard AWT [55, 64, 63]) can be easily tailored to operate as processes with channel based on CTJ link drivers.

3.2.2.2 Deficiencies

The main deficiency of CTJ is its code complexity. The package is easy to use, and there are very well-implemented and commented examples, but understanding and extending the library is very hard work. It also does not use the Java concurrent primitives, except in some low level constructions. In the background of the library, it uses its own semaphores and distributed monitor primitives in such a way that resembles old time concurrency programming, which is quite tricky and complex. This leads to confusion when extensions are necessary. In other words, extending CTJ is an error prone job.

Other interesting implementation features of CTJ is that it has queues for notifications and synchronization between threads and communicating processes. Nevertheless, the available functionality is neither clear nor seem to be prepared for extension. For instance, the queue implementation is confusing as far as method meaning (i.e. expected functionality from the method name) and intended documented behaviour are concerned.

CTJ has other minor problems mentioned below.

- The alternative constructor has a queue of available guards to control its possible execution paths of `Alternative` constructors. In principle, this can be used to deal with backtrack. Nevertheless, there is a very coupled definition between the alting queue behaviour and other internal scheduling queues: the ready queue used by the process dispatcher, and the waiting queue used by the locking monitor semaphore. To alter the alting queue, those queues must also be adjusted. In order to achieve this goal in practice, low-level details about the CTJ thread architecture must be well-understood to avoid implementation mistakes. Despite this fact, there is no sufficient documentation to completely understand those low-level classes, which brings us to a complicated situation in the way to extend CTJ to deal with backtrack.

  Another minor point about the alting queue is that it has package visibility, which forces us to alter the CTJ package instead of extending it. This breaks completely the code reuse property of object-oriented frameworks.

- The redefined monitor primitive used in CTJ constructs uses three controlling semaphores. Therefore, when some state needs to be restored due to a possible backtracking occurrence, there is the need to handle three waiting (semaphore) queues. To identify which one of them became altered and must be restored is very error prone and difficult task.

It was observed that, with CTJ, the backtracking problem can be partially solved with many internal changes. For multi-synchronization, the solution is even more complex since occam does not support it at all, and so neither CTJ. In this way, this is far from a solution to our problem domain.

3.2.2.3 Discussion

We conclude that CTJ is modeled to be a robust process algebra library that deals with real-time aspects, but not a robust extensible framework. Moreover, it implements occam and not CSP.

Nevertheless, we have learned a lot with the CTJ source code and tutorials freely available from its web site [59], following many of its design decisions. For instance, studying the CTJ source code, we decided to implement JACK channels and data

links, which is clearly a CTJ contribution to our work. Another important idea that comes after studying CTJ source code, is the observation that, to deal with backtracking, there is the need for some kind of queuing or history sensitive execution of processes in order to properly identify and solve backtracking situations. Finally, the idea of an active object that has its run method called by its own thread of control and a passive object that has its run method called by its process parent, is followed in the final JACK implementation.

3.2.3 Jass

Java with Assertions (Jass) [32] implements assertions, class and loop invariants, pre- and post-conditions, guarantee about method result, trace assertions, and so forth. Here, we present an overview of this framework[3].

Assertions are statements about the state of objects. They describe properties of these objects, which have to be true at specific points of time during the execution of the program.

Assertions are based on the concept of design by contract [89, 90], that is examined and transferred to Java. For this purpose, certain instructions have been included in the Jass language that make it possible to express assertions in the programs. These instructions have the form of comments, so that an extended program is still accepted by the Java compiler. A precompiler translates these comments into statements in Java, which then checks the assertions during runtime and trigger exceptions in the case of violations.

The result of this work is the extended language Jass, including the precompiler to translate the extensions. The chosen translation approach in the transference to Java operators leads to a completely transparent handling of assertions for the developer. At the same time, it includes numerous additional options for the precompiler, with which the degree of the translation can be regulated.

Nevertheless, what an assertion framework (i.e. language and precompiler) has in common with process algebras? The concept of trace assertions, which is a contribution of Fischer [31]. They help us to specify the dynamic behaviour of programs at runtime. Trace assertions lay down the order of valid method invocations. Furthermore a method invocation can be bound to certain conditions. Trace assertions are defined as a kind of Jass assertion, so it is also a regular Java comment to the Java compiler that has class global scope to the Jass precompiler.

The discussion above is based on one of the Jass companion documentation [32] and Fischer's PhD thesis [31]. The version under consideration is the Jass version 2.0. For a detailed discussion about the Jass package and available functionality see [32, 31].

[3]Unlike CTJ and JCSP, Jass was built as an object-oriented framework. It also has formal syntax and semantics descriptions [31], and also tool support as a Java precompiler [32].

3.2.3.1 Advantages

Jass is neither a Java extension package nor a process algebra library. It is a language with a precompiler to be used behind the Java compiler. The main advantage of Jass is that it adds the option of design by contract to Java. Therefore, it acts as a kind of software functionality checker.

Users of Jass neither need to extend nor to implement any Jass class or interface. They need only to follow its defined syntax and semantics [31]. In this sense Jass is a kind of black-box framework tailored to be used by the final user and not to be extended by him.

The two most important advantages of Jass for the process algebra implementation context are:

- Jass implements design by contract, and has a formal syntax and semantics.
- Jass provides trace assertions for process specifications.

The design by contract concept was first suggested by Bertrand Meyer [89, 90] and bundles a plethora of theoretical and practical insights. It combines the object-oriented approach, abstract data types and methods taken from software verification. The idea is made up of four major aims:

- Formal specification — A formal specification can be given as an intrinsic part of the language without requiring further efforts in the learning process nor in the actual use.
- Documentation — The specification is part of the program code and can be automatically extracted from it.
- Debugging — The program can check itself during runtime.
- Software-tolerance — In the case of a violation of the contract, an exception is triggered, which can itself be handled by the programmer.

The trace assertions facility of Jass can be viewed as an independent feature of the language. With trace assertions, users can ensure the expected execution order of their methods. In this way, the trace assertions act as a kind of class invariant, because they enforce the specified behaviour. At this point, Jass touches the idea of a process algebra package for Java.

Therefore, Jass acts as a kind of class dynamic behaviour debugger or execution checker, which is very different from an extension package that provides CSP operators.

3.2.3.2 Deficiencies

As mentioned in last paragraph, Jass acts as a kind of class dynamic behaviour debugger or execution checker. It is the user responsibility to deal with threads, concurrency, synchronization locks, and so on. The framework does not provide any abstraction facility to deal with concurrent or synchronized aspects of the execution. It adds debugging facilities to process dynamic behaviour execution, nevertheless it does not provide any support to properly define processes, which leaves the process specification at a low-level.

This conflicts with the expected process algebra framework behaviour, that is to provide an extension package to abstract non-functional aspects of processes execution (i.e. process algebras operators). Jass is also not tailored to be extended (i.e. white-box reuse) by the application user by inheriting from one of its classes or implementing some of its interfaces.

One may use Jass to check if the process execution is working as expected. It is important to note that Jass provides a powerful way to do this. Jass provides from class invariants, to quantifiers (i.e. $\forall$, $\exists$, etc.), which are very useful for verifying the implementation correctness with respect to unified languages [62] like CSP-OZ [31] or Circus [148, 149].

3.2.3.3 Discussion

We conclude that Jass is modeled to be a Java language extension instead of a process algebra extension package. For this reason, Jass does not implement any CSP operator.

Jass is implemented as a framework, it has clear documentation, design patterns decisions, and a formal syntax and semantics. Nevertheless, it is inadequate to our goals (see Section 3.5). Actually it is not tailored to deal with such goals.

Nevertheless, we have learned a lot with the Jass source code and tutorials [32]. For instance, JACK abstract syntax trees and interpreter are inspired in the Jass implementation. This whole structure is not different from what is expected following some well-known reference [143], nevertheless, it is an instantiated example that uses CSP operators. Yet another important contribution from Jass is the idea of execution inspection. Based on this idea, JACK provides a set of protocol interfaces that must implement the Failures and Divergencies [33, 124] CSP model. This first version of JACK does not provide a complete implementation of that feature, but it can be used in the future as inspector of the execution of a given CSP network. These are clearly Jass contributions to our work.

3.2.4 Other Frameworks

This Section presents some other frameworks studied during the JACK framework construction. They are neither strictly process algebra frameworks, nor implement CSP; nevertheless, they are related to sotware specification and concurrent execution of processes. The information about these libraries comes directly from their main references. The references are mentioned sorted by relevance to our work.

3.2.4.1 ACE

The Adaptive Communication Environment (ACE) [138, 128] is an object-oriented framework and toolkit that implements core concurrency and distribution patterns for communication software. ACE includes several components to help in the development of communication software and to achieve better flexibility, efficiency, reliability, and portability.

Components in ACE can be used for the following purposes:

- Concurrency and Synchronization.
- Interprocess communication (IPC).
- Memory Management.
- Timers and Signals
- File System management.
- Thread Management.
- Event demultiplexing and handler dispatching.
- Connection establishment and service initialization.
- Static and dynamic configuration and reconfiguration of software.
- Layered protocol construction and stream-based frameworks.
- Distributed communication services — naming, logging, time synchronization, event routing and network locking, etc.

There is a Java version of ACE called JACE that bridges Java applications to use the ACE framework, originally written in `C++`. One of the most important framework on which JACK is based is DASCo [132, 133]. DASCO and JACK also use some of the ACE ideas to implement low-level concurrency and synchronization features.

3.2.4.2 Alloy and Alloy Constraint Analyser

Alloy is a new language for object modeling [21, 69]. The Alloy Constraint Analyzer is a tool for analysing Alloy object models. The two were designed hand-in-hand: the Alloy Constraint Analyzer exploits the structure of Alloy models, and Alloy was designed to be analysable. Alloy has the same aim as object modeling languages such as Rational's UML [56, 74]. Its basic structuring mechanism is strongly influenced by these languages (and early work on semantic data models), but its semantic basis is taken from Z [11, 136, 137, 147].

The Alloy Constraint Analyzer can generate instances of states satisfying invariants, simulate executions of operations, and check properties of a model, such as whether one invariant implies another, or whether an operation preserves an invariant. Since Alloy is undecidable, the Alloy Constraint Analyzer cannot prove theorems. In practice though, the Alloy Constraint Analyzer can find counterexamples of properties that do not hold quickly.

The Alloy Constraint Analyzer is essentially a compiler. It translates the problem to be analysed into a (usually huge) boolean formula. This formula is handed to a solver, and the solution is translated back by the Alloy Constraint Analyzer into the language of the model. All problems are solved within a user-specified scope that bounds the size of the domains, and thus makes the problem finite (and reducible to a boolean formula). Some of the solvers are complete, and will eventually find a solution if one exists; others are incomplete, and may not find a solution even if one exists within the scope.

Alloy is neither a model checker, nor a theorem prover. The Alloy Constraint Analyzer's analysis is fully automatic, and when an assertion is found to be false, it generates a counterexample. What the Alloy Constraint Analyzer does is more like refutation than proof. Alloy is less expressive than the languages typically handled by theorem provers. Unlike most theorem provers, the Alloy Constraint Analyzer is not a general purpose engine for mathematical analysis, but is designed for the analysis of object models.

The JACK type system makes use of ideas from Alloy Constraint Analyser source code to implement set comprehension and value constraints for channel communications. This forms the base of a simple normal form reduction algorithm of a subset of the predicate calculus used to avoid communication expansion. For more details about the Alloy language and the Alloy Constraint Analyser see [67, 21, 66, 69]. For details about the method used to solve boolean formula constraints, see [68].

3.2.4.3 JValue

JValue [117, 44] is a value object framework for Java. Most software developers who hear the word *value* or *data* think of chars, integers, floats or strings. The value concept is not restricted to these primitive value types. When designing and

programming, we also use domain-specific value types like HTML tags and URL's in the Internet domain, Time and Date in the calendar domain, and Currency and Money in the finance domain.

Most programming languages, including Java, restrict programmers to using primitive value types only. Domain-specific value types are designed and implemented as regular classes. At runtime, their instances behave like regular objects rather than like values. However, because values have value semantics rather than object semantics, and because value semantics is much more restricted than object semantics, crucial opportunities are missed for making programs safer and faster.

Using value semantics, programs get safer, because value semantics guarantee freedom from side-effects. If we could properly implement value semantics for objects, thereby making them value objects, we could get rid of a whole class of nasty bugs. This is the objective of JValue. Some interesting and important characteristics of JValue are summarized below.

- Immutable objects — Implements value objects as immutable objects.

  Being immutable, value objects can not be changed (by definition), and hence one avoid unwanted side-effects.

- Shared objects — Shares immutable objects efficiently using the Flyweight [28, Chapter 4 pp.195] pattern. Sharing value objects can significantly reduce memory footprint and the time spent on garbage collection.

These implementation strategies for value objects allows the application of some or all of the performance enhancements mentioned below. In order to make it clear, think *integer* whenever one reads value in the following list, and then imagine the argument applies not only to integers but also to monetary amounts, that is double/string pairs, or other domain-specific value types.

- Concurrency — The user cannot directly change the *state* of a value object, because he or she only has reading access to it (but this can be relaxed if necessary). Hence one has no locking overhead, because one does not need to worry about concurrent write accesses.

- Distribution — Between processes, values are always copied as part of their embedding object and never referenced across process boundaries. Hence, one avoids unwanted round trips across process boundaries.

- Databases — When writing a value to a database, one can write it directly into a table as part of an object. Once written to a table, no outside references remain, so there is no need to maintain an id or primary key. This minimizes lookup time.

- Serialization — When writing a value, one can write the value directly into the buffer. One does not have to worry whether it has been written before, because the current information is sufficient to restore the value object when reading it later.

The JACK type system uses some of the concepts available in JValue to describe types and value semantics. The data link and value serializer architectures also use some of the ideas of JValue serialization, an implementation of the Serializer [123, Chapter 17 pp.293] design pattern. JValue lacks proper documentation and its full design is not publicly available. For more information about it, see [117, 44, 112, 111].

3.2.4.4 FSP

FSP [84] stands for Finite State Processes, and it is a process calculus like CSP [61, 124], and Calculus of Communicating Systems (CCS) [94], to describe process models. With FSP descriptions, the user can define processes, composition of processes (i.e. parallel, sequential, etc.), constants, labels, ranges, safety properties, progress properties, and so forth. Furthermore, with FSP one can describe a process specification. With FSP, it is possible to concisely describe and reason about concurrent programs. Its main objective is to provide a concise way to describe Labeling Transition Systems (LTS).

FSP has a formal syntax and semantic description of its textual input; it was designed to be machine readable. In addition, FSP has a companion tool freely available called LTSA Analyser [85]. The analyser translates FSP descriptions to an equivalent graphical representation of it using an LTS.

The internal representation of the JACK process network is built using a LTS. The FSP descriptions helped us to properly configure our LTS to better represent a process network with user processes and language operators mixed, and also to properly "label" the graph to solve the backtrack and multisynchronization problems (see references on this topic in Appendix A).

3.2.4.5 Triveni

The design of Triveni is based on a process algebra that adds preemption combinators [7] to the standard combinators from process algebra such as parallel composition, waiting for events, hiding events, and so forth. The aim of that research is to enhance the practice of thread programming with ideas from the theory of concurrency, such as process algebras [61, 94].

Triveni has a formal description with a compositional semantics (operational, denotational, and logical) for a process algebra enhanced with input/output actions and preemption combinators, in the presence of fairness. A case study in Java Triveni is described in [17], involving the reimplementation of a piece of telecommunication

software "the Carrier Group Alarms (CGA) software of Lucent Technologies" 5ESS switch.

In practice, Java Triveni is defined as a programming language and a programming environment that deals with processes, concurrency, and event based notification of processes built in Java. For more details about Java Triveni see [16, 18].

This work helped us to take a better understanding on the dynamic execution behaviour of a process network that represents a process algebra. Triveni is not an extension package of Java but a programming environment, which make its goal different from JACK's goals.

3.2.4.6 JOMP

JOMP [77, 104, 75] stands for "Java OpenMP". It is an effort to provide OpenMP primitives and directives in Java. The OpenMP is a collection of compiler directives, library functions, and environment variables that can be used for shared memory parallel programming.

The OpenMP programmer supplements his code with directives, which instruct an OpenMP-aware compiler to take certain actions. Some directives indicate pieces of code to be executed in parallel by a team of threads. Others indicate pieces of work capable of concurrent execution. Others provide synchronisation constructs, such as barriers and critical regions.

The directives are specified in such a way that they are ignored by a compiler without OpenMP support. This makes it easy to write portable code which exploits parallelism.

3.3 Programming Language Selection — Why Java?

Java is a fully object-oriented language. Everything in Java except primitive data types is an object. The Java syntax and semantics are also clear to understand, different from `C++`, that is error-prone. For instance, in Java one never needs to deal with memory leaks, pointers, pointer arithmetic, null-terminated character arrays, and so forth. Java eliminates multiple inheritance, replacing it with a new notion of interface. Java interfaces gives what is expected from multiple inheritance, without the complexity that comes with managing multiple inheritance hierarchies

The distinction between classes and interfaces is a well-established and undoubtedly important concept in software engineering. The introduction of interfaces in Java programming is an important improvement over other programming languages.

Java has many more important aspects to our problem domain more than just interfaces to abstract communication protocols. They are listed below.

- Built-in concurrent programming and communication primitives.

- Strongly typed and object-oriented concepts.
- Platform independency.
- Support for distribution and security.
- Since it is interpreted, Java is well-suited for the development of embedded systems.
- Has a continued research to give it a better formal description and a full proof compiler.

Java also has the package concept. With this, it is possible to clearly separate design decisions, framework layers, interface protocols from concrete implementations, and so on. This is very important in order to allow concurrent framework development. Each development team can be responsible for one package or specific functionality.

Another advantage of Java is its companion packages and good documentation that provide useful basic functionality for framework construction. These and other reasons are better described in [4, 64, 72].

3.4 Framework Modeling

Here we present a brief description of role modeling, the modeling technique selected to be used in JACK. The information is based on [113, Chapter 5] and [120].

3.4.1 Role Modeling

A primary problem in designing and integrating frameworks is related to the way they are modeled. A class-based approach is the dominant option, but it fails to adequately describe object collaboration behaviour. When designing a framework a set of guidelines ought to be observed and clarified. For instance, the responsibilities an object has, the contexts these responsibilities depend on, how these responsibilities can be combined, and so on. Another important definition states how a client interacts with a framework; it defines who are those clients and what they expect from the framework.

There are many problems involved in framework design [120]:

- Class Complexity — how a class interacts with its clients?
- Object Collaboration — how object instances interact with their partners?
- Separation of Concerns — complex functionality used for different purposes should be kept separate to ease understating and increase reuse.

- Reusable models and design patterns.

Framework integration also has problems like client constraints, and unanticipated use-contexts.

Role modeling for framework design and integration defines some new concepts like role, role type, role constraint, object collaboration task, role model, built-on class, extension points, and so on [113, Chapters 3 and 4]. The JACK framework was built based on these modeling concepts.

The macro structure of JACK layers is divided and defined based on some examples of frameworks that use role modeling (i.e. JValue [117], JHotDraw [113, Chapter 8]). One important aspect observed from these examples is the separation of concerns, and the separation of service interface protocols with respect to their corresponding implementation. The DASCO project [133, 132] also suggests this kind of modeling technique as a future extension work.

This modeling approach increases modularity, reusability, expressive power, clarity, and many other desired software engineering properties. It also minimizes the compilation interdependencies which decrease the whole framework coding and testing time. For a brief and clear discussion of role modeling for framework design, see [120]. For a detailed presentation of role modeling concepts and examples in industrial frameworks, see [113].

3.4.2 UML and Role Modeling

UML offers a rich metamodel for modeling object systems, which makes it easy to extend it with role modeling concepts. The extension of UML with role modeling concepts relies only on three basic UML concepts [56, 74]: Class, Interface, and Stereotype.

When working with UML Role Modeling, developers use UML classes and interfaces as usual. If needed, they add additional role type and role model information. They tag interfaces and classes as specific kinds of role types by using UML stereotypes, and make the role models explicit by connecting role type interfaces and classes.

From a UML perspective, the role type interfaces are just interfaces that relate to other interfaces and classes without further qualification. In this sense, role modeling extends UML information and does not contradict or conflict with.

3.4.3 Java and Role Modeling

Java provides interfaces, classes, and packages as concepts that can be used to implement role model based designs. Java interfaces and classes can be used to represent role types and classes, and Java packages can be used to provide a name space for role models, class models, and frameworks.

The use of abstract classes as a representation mechanism of a free role type does not make sense, because Java classes are restricted to single inheritance. A free no-op role type [113] of a class model may be represented as an empty Java interface (or not at all). A role type of a reusable role model is represented as a Java interface. The same argument that applies to a free role type of a class model applies here as well. Classes are simpler to represent.

In both cases, it is possible to tie in the different role types that a class must provide. If the class is represented as a Java interface, it inherits from the interfaces that represent (some of) the role types it is to provide. If the class is represented as a Java class, it simply implements them. In addition, non-reusable role types are textually and directly embedded into the Java interface or the Java class.

Role models that are considered reusable should get their own Java package. The package introduces a convenient name space for the role types of the role model. A framework should also get its own package. It ties in reusable role models by means of import statements.

3.5 The JACK Framework Main Objectives

The main objective of our framework is to provide processes embedded in the Java Language constructors as an extension package. These processes ought to be either CSP operators or user defined processes. JACK also ought to provide both CSP operators and other common CSP elements like types, alphabets, guards, channels, and so forth.

The CSP operators implemented are defined in [124], and briefly discussed in Section 2.2. JACK follows the operational semantics laws outlined and discussed in [130, 126]. A user process presents any user-defined functionality, that can be composed with the use of CSP operators with data structures. With this possibility, the implementation of unified formalisms is possible. For instance, CSP-OZ [31] (a combination of Object-Z [134] and CSP) or Circus [148, 149] specifications can be implemented, as discussed in [12].

Another important objective is to provide a framework modeled and prepared for either black-box, or white-box reuse; the so called grey-box reuse. Black-box reuse means that the framework can be used directly in some application domain, for instance, to model a user specification. White-box reuse means that the framework is prepared to be extended, for instance for the implementation of new operators.

The most important difference between JACK and other similar libraries are the fact that JACK deals with Roscoe's CSP [124], instead of occam [53]. This new version of the formalism requires that different kinds of problems are handled, like multisynchronization and backtracking, as already mentioned in the previous chapter.

Another goal of the JACK framework is to give a semantic meaning to the backtracking problem. This is expected to be done in such a way that the designer of the original specification does not need to be aware of this problem; he needs only to think and take care of the problems of his system domain.

The JACK implementation also ought to follow a well-defined implementation model in order to avoid obscure problems in its design. JACK follows the most recent framework modeling techniques, like role modeling [113], and implementation techniques using well-known design patterns [28, 123, 102, 121, 133]. This is necessary in order to achieve desired software engineering properties like expressive power, reusability, extensibility, simplicity, modularity, and so forth.

3.6 JACK versus other Frameworks and Libraries

In this Section, a comparison between JACK goals against similar process algebra implementations is presented to summarize our previous discussion. The differences between each approach is also exposed. The main differences between JACK and the other process algebra implementation libraries (i.e. CTJ and JCSP) are as follows:

- JACK is modeled as a framework instead of a class library. It has a well-defined set of layers that provides well-defined services to each other. Functionality and complexity are clearly distributed among those layers.

  With the separation of concerns and functionalities between layers, it was possible to solve each problem inside its own domain without making a mixture of concepts and goals, as it occurs in some points of the mentioned libraries. For instance, both CTJ and JCSP mix concurrency and synchronization related code with process semantics related code. With this approach, problems like inheritance anomaly [86] arise when extensions are made necessary.

- JACK processes and its CSP operators implementation are based on Roscoe's version [124] of CSP, instead of occam [53].

- JACK has a strong type system to define channels, alphabets, types, and communication patterns.

- JACK distinguishes the process execution environment, that is generic, from the process execution behaviour, that is specific. With this separation, it is observed that the final user process becomes light weight with respect to the corresponding process in the other two libraries, that mix those perspectives of the problem.

In what follows we mention the main identified deficiencies between JACK and the other library implementations.

- The objectives of JACK differs from the objectives of other libraries. It was designed to implement the Roscoe's version of CSP in Java instead of occam like other libraries. In this sense, the framework must deal with many new problems and dificulties which makes it more complex and bigger than the other libraries.
- The integration between JACK layers is a very complex task.
- There is no performance tests available for JACK. In this way, costy implementations like the multisynchronization and backtracking problems, can be reviewed.
- JACK provides a useful type system used to allow strong type checking for communications. This is interesting benefit in order to avoid abnormal program execution due to type inconsistency. Despite this, the end user must know how to use the type system which can delay the learning process to use the framework.

In the following, we present some discussion about specific points that JACK inherits or avoids from each referred library.

3.6.1 JACK and JCSP

Some aspects in which JACK clearly differs from JCSP are discussed below.

- JACK is modeled to be a grey-box framework, whereas JCSP is made to be black-box reused.
- JCSP is semantically weak in some points. For instance, the `Alternative` constructor is not built as a process, which makes it play a different role in the processes network construction. With this kind of modeling, the user must deal with alternative selection manually, which might be very cumbersome. Both CSP [124] and occam [53] idioms provide a well-defined semantics for this operator, that is not followed by the JCSP implementation.
- JCSP has prioritised versions of the alternative and parallel constructor, JACK does not.

In spite of this, JACK has many characteristics inspired by JCSP.

- JCSP uses the concept of barriers to synchronize parallel processes. The same concept is used in JACK to make the synchronization of the generalized parallelism operational. Nevertheless, instead of using an explicit locking scheme through Java monitors, JACK makes use of its execution (bottommost) layer, that provides those synchronization and concurrency services. The execution layer is JDASCO, a Java implementation of DASCo [133, 132].

- Channels in JCSP have a separate data store to perform the low-level physical I/O operation; the same concept is adopted in JACK.

Generally, JCSP has contributed to many points of the JACK design. A detailed comparison between an early prototype version of JACK and JCSP is given in [36].

3.6.2 JACK and CTJ

The CTJ library is more robust and expressive than JCSP. For instance, it deals with real time aspects of specifications. Nevertheless, it is more complex than JCSP due to its own thread architecture. Some points where JACK clearly differs from CTJ are listed below.

- CTJ implements a new scheduler, processor and thread abstractions in order to achieve its real time goal. JACK relies on its execution layer to provide those abstractions.
- CTJ captures real time aspects of specifications, JACK does not.
- CTJ has prioritised versions of the alternative and parallel constructor, JACK does not.
- CTJ does not use the Java concurrency primitives of monitors and scheduling policy for threads. It builds up a completely new model to this.

  On the other hand, JACK separates the use of concurrency and synchronization primitives at the execution layer . JACK also seems to have a much clearer lock scheme that is simple and powerful (thanks to DASCo framework [133]), in contrast to CTJ that has locks in many places with a proprietary lock scheme, which makes it much more complex.
- CTJ provides as default a environmental process to the alternative operator. In other words, a sole alternative operator can run without deadlocking due to a missed paired parallel operator. The CTJ execution engine detects this situation and starts a parallel environment that offers any communication, avoiding this way the alternative to become deadlocked.

  In JACK an external choice (the CSP operator correspondent to the occam alternative operator) without a paired parallel operator behaves like a broken machine (i.e. the $STOP$ process), following the semantics of CSP [124].

Despite this, JACK has many characteristics inspired by CTJ.

- The use of queuing structures to store execution information is observed inside the low-level CTJ scheduling primitives.

- CTJ Link drivers are very interesting in the sense that they completely abstract the semantics of communication from its underlying physical I/O operations. The same idea is followed in JACK, that also provides the same concept of link drivers for channels, called data links. This makes the library I/O completely flexible to any kind of medium, from local memory to remote machines through socket connections.

- CTJ uses event listeners (i.e. observer design pattern [28]) to register for some specifically defined events. This is an interesting design decision both for backtracking and for distributed termination condition notification. JACK makes use of a specialized version of that pattern for event notification [110] intra/inter object and across related layers.

More details comparing JACK and CTJ internals can be found in [36].

3.6.3 JACK and Jass

The most important intersection between JACK and Jass goals is the trace assertions facility. With trace assertions, Jass can guarantee the execution order of methods, which in turn means that it can guarantee the behaviour of some process specification.

Jass is provided as a black-box framework to be used as an extension to the Java language.JACK is a grey-box framework, that allows users to specify their processes using well-known CSP operators.

Despite this, JACK has some characteristics inspired by Jass. For instance, the JACK topmost layer (JCAST) is structured as an abstract syntax tree (called JCAST) that is firstly inspired by the Jass and Triveni abstract syntax tree representation of processes. The visitor pattern [28] used in Jass was observed to be a very useful way to provide process interpretation for the JACK framework. This is an important facility towards the semi-automatic translation [12, 1] from specification source code (i.e. CSPm, Circus, CSP-OZ) to Java classes using JACK.

3.7 Final Considerations

In this chapter, an overview of framework construction using design patterns and pattern languages was given. An interesting reflection about design pattern usage can also be found in [15, 2]. After that, an overview of the available related libraries was presented in order to show the state of art in this field of formal methods. Next, some of the most important JACK goals and modeling techniques used were presented. We concluded with a comparison between those referred libraries and JACK.

In the next chapter, JACK main objectives are discussed in more detail. We also show JACK usage from the user point of view. That chapter can be viewed as a short guided introduction to the framework.

Next, in Chapters 5 and 6, the required prerequisites of well-designed frameworks are discussed, like layering and the design pattern initiative. In Chapter 6, detailed decisions and pattern templates (i.e. instances of defined design patterns) are shown.

Chapter 4

Using JACK

As mentioned in the previous chapter, JACK is built to be a *gray-box* reuseable framework (i.e. white-box and black-box). This chapter aims at describing how a JACK framework client can use the available constructs, in order to properly define its process specifications. The chapter presents basic available constructs like processes, CSP operators, and some type system facilities. A more complete description of these facilities can be found in [38, 41].

In Section 4.1, an overview of the JACK processes structure is given. Then in Section 4.2, the structure of some of the most important CSP operators available in JACK, together with some code examples are presented. After that, in Section 4.3, more details about user processes definition are mentioned. Next, in Sections 4.4 and 4.5, an overview of the JACK type system and some details about the process infrastructure are provided. Finally, in Section 4.6, some final considerations are given.

4.1 JACK Processes

The most important aspect of JACK is the ability to deal with processes. A JACK process is an independent entity that defines a behaviour that can be executed. The process abstraction represents a piece of a specification, but can also be used to implement concurrent aspects of systems with a high-level concurrency abstraction.

The common use of a process is related to the operating system concept of a unit of resource allocation (normally executable assembled code) for both CPU time and memory [84, pp. 23]. The processes execute for a while, running independently of any other processes [139]. Processes have a controlled autonomy managed by a supervisor environment that selects events to be communicated. In JACK, a process acts like an entity that has concurrent behaviour from the operational system point of view. Thus, a JACK process object instance is a kind of active object [78].

In the object-oriented world, different threads of execution are used to propel

conceptually active messages (i.e. method calling) sent among conceptually passive objects (i.e. object instances). However, it can be productive to approach some design problems from the opposite perspective: active objects sending each other passive messages [81, pp. 367]. In this sense, a JACK process can be constructed as a special kind of object, in which:

- Processes have no externally invocable methods related to their semantics. Since there are no directly invocable methods, it is impossible for methods to be invoked by different threads. Thus, there is no need for dealing explicitly with locks.
- Processes communicate only by signaling, reading, or writing data through communication channels.
- Processes need not spin forever in a loop accepting messages (although many do). They may signal, read, or write messages on various (possible shared) channels as desired.
- Channels are synchronous and can perform either signal, read or write operations.

There are two main groups of processes in JACK: CSP operators, and user defined processes. The former implements the well-known CSP [124] operators, and the latter provides a way to a user to define its own active object to execute the desired behaviour.

The state transition diagram in Figure 4.1 expresses the transitions between possible process states. A process can be either in *user* or *kernel* mode of execution. In the user mode, the process is completely prepared to run, but it is not physically running yet. During the physical execution of a process, it is in *kernel* mode. At this stage, the JACK execution layer assigns activities and locks to the process, in order to allow the execution of its functionality according to the expected semantics. These states are mentioned below.

- Initialized — the process is allocated and prepared to run, or successfully terminated. The process is in *user* mode when instantiated.
- Started — the process is notified that it will enter in *kernel* mode of execution. At this point, users ought to initialize any entity related to the running procedure, like other process or any data structure.
- Running — the process enters in *kernel* mode and becomes an active object with its own thread of control. At this stage, the process gets executed.
- Ready — the process is not running and is ready to execute. This state is achieved when the operating system schedules other processes to run.

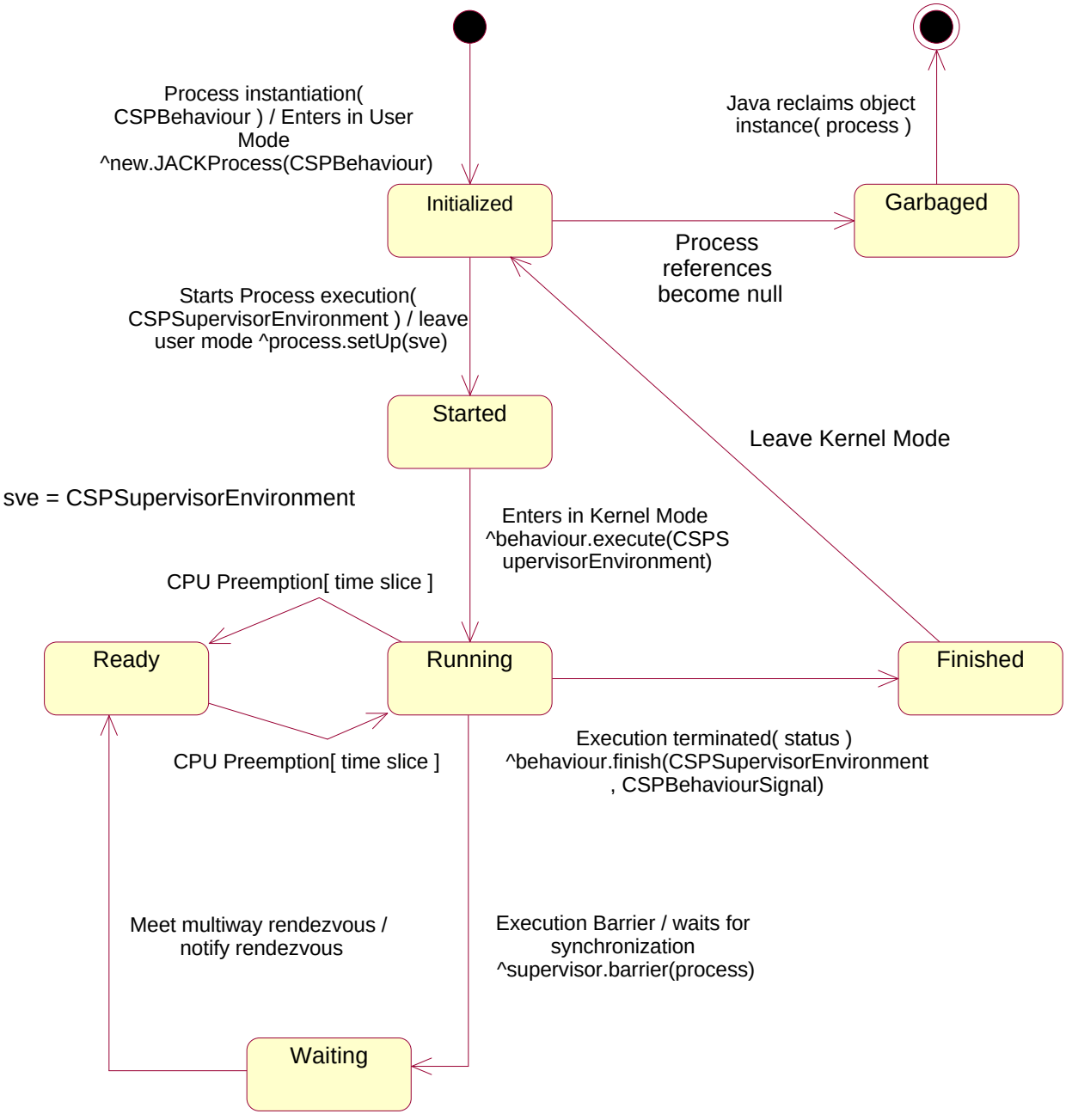

Figure 4.1: JACK Process Finite State Machine

- Waiting — the process is waiting for some execution condition to be reached. For instance, a process that represents a prefix operator may be put to sleep by a process that represents a generalized parallel operator (i.e. barrier on the generalized parallel operator to form a multiway rendezvous [13, 131], until the synchronization condition holds). This situation occurs due to the implicit semantic implementation of processes, the user never needs to directly deal with this.

- Finished — the process has finished its execution and leaves the *kernel* mode. This can happen due to either successful termination, or some abnormal execution condition.

- Garbage — the process is terminated and will never be assigned to execute again. The Java garbage collector may clean it up.

A complete definition of processes and how they are implemented in JACK are given in Chapters 5 and 6.

4.1.1 JACK Process Structure

A JACK process is composed by four main aspects:

1. The process interface — responsible for dealing with process infrastructure and generic information.

2. The process behaviour — responsible for dealing with process behavioural and specific information.

3. The process semantic model — responsible for dealing with proper process semantic execution with respect to its defined semantics.

4. The process execution support — responsible for dealing with the process physical execution, framework layer integration, and communication selection support.

The JACK client must deal only with the second aspect. It directly represents the specific behaviour related to the user process specification under consideration. The other aspects are provided by the framework. In order to provide a complete overview of process execution, in the following subsections the most important of these aspects are explained.

4.1.1.1 JACK Process Interface

This part is responsible for dealing with process infrastructure and generic information. The JACK client never needs to directly deal with this in code, but should be aware of this process structure, in order to better use the expressive power of the framework implementation. In what follows, we mention the main generic aspects related to all sort of processes that the framework must deal with, in order to properly provide an implementation of processes.

- Proper composition with the process network. The process network is the lower level execution environment responsible for dealing with threads, locks, and transactions;
- Definition of process information like level, unique name, parentship, upward searching facilities, and so on;
- Debugging and tracing facilities;
- Supervision operations like immediately available event inspection, and strategies [28, pp. 315] used for external environmental event selection (see Section 4.5);
- Process cloning support;
- Event registration for exit, event, and exception occurrence;

The process interface is described by the `jack.jacs.japs.csp.CSPProcess` JACK interface.

There is only one implementation of it, called `jack.jacs.japs.csp.impl.JACKProcess`. User processes and CSP operators make use of it to implement the specific part of the process behaviour. Thus, the process interface is responsible for concentrating and abstracting the generic aspect of JACK processes for the final user. The package `jack.jacs.japs.csp.tests` [38], provides useful examples of these mentioned functionalities.

4.1.1.2 JACK Process Behaviour Interface

This part is responsible for dealing with process behavioural and domain specific information. The JACK client directly deals only with this part. Through the use of the process behaviour interface the user can construct its processes specification. An example of this feature is shown later in this chapter. In what follows, we mention the main specific aspects related to process behaviour implementation.

- Process set up procedure, like object instantiation or resource allocation;

- Process execution behaviour.
- Inspection of currently immediately available events that can be communicated by this process.
- Process cleaning-up (or finish) procedure, like signaling, resource deallocation or event handling.

The process behaviour is described by the `jack.jacs.japs.csp.CSPBehaviour` and `jack.jacs.japs.csp.CSPExtendedBehaviour` JACK interfaces. Every user process must define its own specific execution and inspection behaviour. Thus, the `CSPBehaviour` interface is used to segment the specific from the generic process execution functionality. User processes are detailed in Section 4.3 after the CSP operators description.

4.1.1.3 Creating user processes in Java

The user must provide an implementation of one of the behaviour interfaces, in order to properly define its process in Java. The basic behaviour interface (`CSPBehaviour`) has two methods.

The first one, called `inspect()`, is responsible for dealing with event selection over the path possibilities. The JACK process execution environment calls this behaviour method in order to get information about the events that the process can engage initially (i.e. domain D).

The second method is called `execute()`; it is responsible for dealing with process execution related to its expected semantics. Figure 4.2 shows a class code template of a user process defining a process behaviour. Some details like supervisor environment, signals, and alphabets are mentioned latter.

There are some Java packages that ought to be imported by the user classes. The first two of them are related to process service interfaces, process service implementations, and CSP operators; they are called `jack.jacs.japs.csp` and `jack.jacs.japs.csp.impl`. The last two are related to the JACK type system, like values, types, and so on. They are called `jack.jacs.japs.typesystem` and `jack.jacs.japs.typesystem.impl`.

These method templates are invoked by JACK, normally through one of the CSP operators constructs like choice, prefixe, or generalized parallel. Since a process is an active object, the user must not deal with method calling. Just specific functionality needs to be provided. The complete internal execution sequence is a topic detailed in Chapter 6.

The `execute()` method must implement the process specification under consideration. At instantiation of the process, the constructor must setup all resources, such as channel inputs/outputs, parameters, and other used processes before the

```
package user;

import jack.jacs.japs.csp.*;
import jack.jacs.japs.csp.impl.*;
import jack.jacs.japs.typesystem.*;
import jack.jacs.japs.typesystem.impl.*;

/** User process code class template  */
public class MyProcess implements CSPBehaviour {
    public MyProcess(Arguments) {
        // User specific initialization code, ...
    }

    // ********** User defined methods **********
        ...

    // ********** CSPBehvaiour Interface Implementation **********
    public CSPAlphabet inspect() {
        /* Returns an alphabet of possible communications.*/
    }

    public CSPBehaviourSignal execute(CSPSupervisorEnvironment sve) {
        //Start the execution of other related processes.
        //Returns a signal indicating whether the execution was successful or if some error has occurred.
    }
}
```

Figure 4.2: JACK User Process Class — Java Code Template

`execute()` method is called by a composition construct. The arguments of the constructor of `MyProcess` specifies the process interface of the user process. The process interface contains the process name, channel input and output interfaces, and additional parameters to setup the process. These parameters can be used to initiate the process state.

Processes must never invoke each other's methods when they are running; they must cooperate via one of the available channel interfaces as defined by the process interface. For instance, if a user process tries to call the `execute()` method of any of its related process, an invalid state exception is thrown. A process may invoke only the `CSPProcess.start()` method of one of its related composed processes, which is safe because this process is not yet running (i.e. the process is at the *Initialized* state). The `CSPProcess.start()` method is the only JACK process related method a user process may invoke inside the `execute()` method.

The extended behaviour interface provides a set of useful methods that the user can use to implement special functionality. For instance, the extended interface provides `setUp()` and `finish()` methods that can be used by processes that are started more than once. The `setUp()` method is called every time the process is started (i.e. enters in Initialized state); and the `finish()` method is called every time the process terminates, either successfully or due to some execution error (i.e. enters in Finished state).

4.2 JACK CSP Operators

A user must know how to use some JACK structures, like CSP operators, alphabets and channels, in order to properly create its processes in Java using JACK. We provide a description of the most common CSP operators available in the framework. Together with CSP operators, we mention the structures related to the building procedure of process specifications. The user needs to directly deal with these structures. In what follows, we present a list of groups of functionality related to the structure of CSP operators.

- Auxiliary Functionality Group — describes two important CSP concepts: alphabets and channels.
- Primitive Group — describes the standard CSP primitive processes.
- Unary Operators Group — describes the standard CSP unary operators.
- Binary Operators Group — describes the standard CSP binary operators.
- Extended Operators Group — describes extended operators like labelling, renaming, and interruption [130, 126].

- Replicated Operators Group — describes the replicated version of the above operators. They are commonly used in CSPm based tools like FDR [33] and ProBE [34].

JACK operators are implemented by the `jack.jacs.japs.csp.CSPOperator` interface. It is a direct subinterface of `jack.jacs.japs.csp.CSPProcess` that provides a CSPm string representation of any given JACK process. According with the groups above, there are related `CSPOperator` subinterfaces for unary (`jack.jacs.japs.csp.CSPUnaryOperator`) and binary (`jack.jacs.japs.csp.CSPBinaryOperator`) operators. These interfaces provide access methods to the unary `process()` operand, and to the binary `left()` and `right()` operands. These access methods return an instance of a `CSPProcess` instead of a `CSPOperator`. This allows the mix of both CSP operators and user process definitions on the same process tree. The CSP operators are briefly described in Chapter 2.

4.2.1 Channels

An important design decision for a passive message environment is how to designate the sources and destinations of messages. All messages in CSP are synchronous, and processes can never be the direct source or target of a message, in order to avoid race hazards. The most common synchronous message-based scheme used in such designs is the one whose channels are used to send and receive messages.

Channels are entities where message synchronization occurs. In other library implementations of CSP (i.e. JCSP and CTJ), the `Channel` class is responsible for the synchronization between partner processes. However, these libraries model a variant of the occam communication scheme, which states that only two processes can communicate through a given channel name. Thus, communication is said to be one-to-one, also called *Rendezvous* [84, Chapter 10]. In JACK a channel accepts many-to-many communication, also called *Multiway Rendezvous* [13, 131]. The channel is just a gateway used by processes to exchange information, either data or signals.

Depending on the kind of communication in which a process engages, it must make use of a different JACK channel interface. There are three main basic channel interfaces enumerated below.

1. `jack.jacs.japs.csp.CSPChannelInput` — input communications.
2. `jack.jacs.japs.csp.CSPChannelOutput` — output communications.
3. `jack.jacs.japs.csp.CSPChannelSignal` — signalling events.

These interfaces directly inherits from a base interface called `jack.jacs.japs.csp.CSPChannel`, which provides generic basic functionality related to channels.

Every channel must have a `jack.jacs.japs.typesystem.CSPType`. The channel only accepts communications of this specified JACK type. In this way, JACK clients must be aware of some aspects of the JACK type system. If the user does not provide any channel type, the type `java.lang.Object` is used by default. An overview of the JACK type system is given in Section 4.4.

Channel Usage — Java Code Example

Figure 4.3 shows the construction of channels of different types. More examples can be found in test case classes in package `jack.jacs.japs.csp.tests.processes`.

The class constructor instantiates three channels called `a`, `b`, and `c` with types `EVENT_TYPE`, $\{0 \ldots 20\}$, and `java.lang.Object` respectively. Users can provide channel names for any desired purpose; if a name is not given, a default unique name is assigned. The `EVENT_TYPE` is a special type that represents CSP signals. If the user does not provide any type to the channel constructor, the default `Object` type is used. The constructor also instantiate two prefix processes that shares one of these recently created channels. The channel attributes of this code example ought to be used as parameters of process constructs that expects a channel, like prefixes. Prefix operators are explained in Section 4.2.4.1.

The available implementation of channels implements all channel interfaces, as shown in Figure 4.4. This means that a channel instance can represent any channel interface and be passed to processes that expect a specific channel kind. This is an important feature since channels are the only entities shared among processes. For instance, the read prefix process `fRP` expects a `CSPChannelInput` instance and the write prefix process `fWP` expects a `CSPChannelOutput` instance. Since the `Channel` implementation class implements both interfaces, it can be used in both operator constructors (see Figure 4.4).

4.2.2 Alphabets

The alphabet of a process is the set of all possible communications that it can perform. It acts as the type of the process [124, pp. 76]. In other words, the alphabet of a process is the set of actions in which it can engage [84, pp. 22].

Alphabets in JACK are used by the user basically to properly implement the `CSPBehaviour` interface `inspect()` method. It ought also to be used to instantiate operators that depend on alphabets, like the generalized parallel or hiding operator. An alphabet is defined by the interface `jack.jacs.japs.csp.CSPAlphabet` and implemented by the class `jack.jacs.japs.csp.impl.Alphabet`.

```
/** Dealing with Channels - Simple Example */
public class MyProcess implements CSPBehaviour {
    private CSPOperator fRP, fWP;
    private CSPChannel fCa, fCb, fCc;

    public MyProcess(Arguments) {
        // User specific initialization code, ...
        fCb = new Channel("b", EVENT_TYPE);//Channel used for signals
        fCc = new Channel("c", new IntType(0,20));//Int range typed channel
        fCd = new Channel("d");//Object typed channel — this is the default channel type

        // c!1 → c?x → STOP
        // Process RP = c?x → STOP
        // Process WP = c!1 → RP
        fRP = new ReadPrefix((CSPChannelInput)fCc, new Stop());
        fWP = new WritePrefix((CSPChannelOutput)fCc, fRP, new IntValue(1));
    }

    // ********** CSPBehvaiour Interface Implementation **********

    /** The possible initial communication domain of MyProcess is the same as the fWP process. */
    public CSPAlphabet inspect() {
        return fWP.behaviour().inspect();
    }

    /** * The execution of MyProcess just starts the write prefix process and terminates successfully.
     * The prefix executes on the calling thread, thus the execution here is serialized.
     */
    public CSPBehaviourSignal execute(CSPSupervisorEnvironment sve) {
        fWP.start(sve);
        return B_EXECUTE_SUCCESSFULLY;
    }

    // ********** User defined methods **********
        ...
}
```

Figure 4.3: JACK Channel Usage — Java Code Example

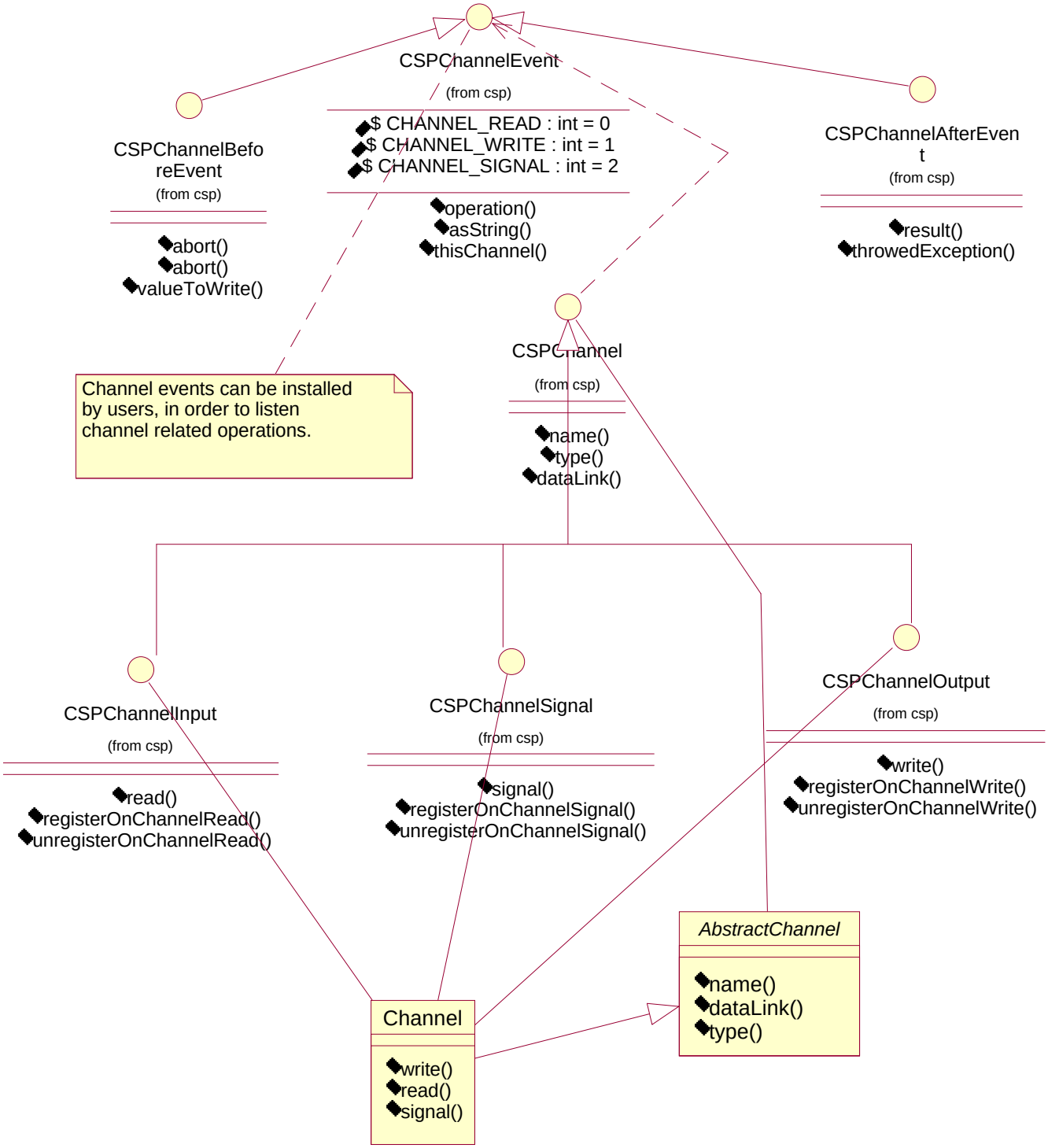

Figure 4.4: JACK Channels Structure

Alphabet Usage — Java Code Example

Figure 4.5 shows the construction of processes that depend on alphabets. More examples can be found in test case classes in package `jack.jacs.japs.csp.tests.processes` [38].

The class constructor instantiates an alphabet to be used by a generalized parallel operator. An alphabet is structured as a set of channel and value pairs. The given example, shows a generalized parallel that must synchronize in all values of the type of the given channel; it means that it must synchronize on $\{c.0, \ldots, c.5\}$. These set of values can also be represented by $\{|c|\}$, indicating the whole type range of the channel type. This notation comes from FDR [33] and denotes the productions of a channel (i.e. all possible value that the channel can communicates). When one adds a channel to an alphabet, it is adding the whole value range of the type of the channel to the related alphabet.

This code example shows how to create a simple generalized parallel synchronizing on all possible values of the type of the channel `c`. More detailed examples about parallel processes and alphabets are given in the next Sections.

4.2.3 Primitive Processes

JACK provides the implementation of standard CSP primitive processes: $STOP$, $SKIP$, and DIV. The user just needs to instantiate them using a default constructor when necessary. They are respectively represented by the `Stop`, `Skip`, and `Div` classes localized in the `jack.jacs.japs.csp.impl` package. An example of their use is given in Figure 4.6; it comes from a test case class method localized in package `jack.jacs.japs.csp.tests.processes`.

This code snippet just creates one instance of each available primitive process and outputs their correspondent CSPm description on the standard output. That description is the equivalent FDR CSPm code that this operator represents.

4.2.4 Unary Operators

The operators representations aims at making their use as close as possible, to that in a CSP specification. In the following paragraphs, the most common CSP unary operators are presented together with an example that shows their usage.

4.2.4.1 Prefixes

There are three prefix process in JACK: read, write, and event prefix. Each one representing respectively, input, output, and signalling communication.

The structure of prefixes is very close to the one expected by CSPm users. All these prefixes must have a channel and process to follow. For instance, the read

```
/** Dealing with Channels - Simple Example */
public class MyProcess implements CSPBehaviour {

//Local Declarations and Attributes
    private CSPChannel fCc;
    private CSPAlphabet fGenParAlpha;
    private CSPOperator fRP, fWP, fGP;

    public MyProcess(Arguments) {
        // User specific initialization code, ...
        fCc = new Channel("c", new IntType(0,5));//Int range typed channel

        fGenParAlpha = new Alphabet(fCc);//Creates the alphabet {|c|}

        // c?x → STOP[|{|c|}|]c!2 → STOP
        // Process RP = c?x → STOP
        // Process WP = c!2 → STOP
        // Process GP = RP[|{|c|}|]WP
        fRP = new ReadPrefix((CSPChannelInput)fCc, new Stop());
        fWP = new WritePrefix((CSPChannelOutput)fCc, new Stop(), new IntValue(2));
        fGP = new GeneralizedParallel(fRP, fGenParAlpha, fWP);
    }

// ********** CSPBehvaiour Interface Implementation **********

    // The possible initial communication domain of MyProcess is the same as the fGP process.
    public CSPAlphabet inspect() {
        return fGP.behaviour().inspect();
    }

    // The execution of MyProcess just starts the parallel process and terminates successfully.
    // The parallel operator executes in their own thread of control.
    public CSPBehaviourSignal execute(CSPSupervisorEnvironment sve) {
        fGP.start(sve);
        return B_EXECUTE_SUCCESSFULLY;
    }

// ********** User defined methods **********
        ...
}
```

Figure 4.5: JACK Alphabet Usage — Java Code Example

```
public void testPrimitiveOperatorConstruction() {
    CSPOperator stop = new Stop();
    System.out.println(stop.asCSPm());

    CSPOperator skip = new Skip();
    System.out.println(skip.asCSPm());

    CSPOperator div = new Div();
    System.out.println(div.asCSPm());
}
```

Figure 4.6: JACK Primitive Processes — Java Code Example

prefix constructor of CSPm has, as mandatory parameters, a channel for input and a process to follow that communication. It also has, as an optional parameter, a boolean guard, and a logical predicate constraint over the type of the channel input. The operator can also be optionally named for recursion definition purposes. The same occurs for the write and event prefix processes. Some simple examples of prefixes are also given in Figures 4.3 and 4.5. The read, write, and event prefix processes are represented by the `ReadPrefix`, `WritePrefix`, and `EventPrefix` classes of the `jack.jacs.japs.csp.impl` package.

The write prefix has an additional "value to write" parameter. An important note must be mentioned about linking of prefixes. In CSP it is possible to link a value read by some prefix to be written by another prefix (i.e. $c?x \rightarrow c!x \rightarrow STOP$). When one likes to make such construction, the user must create a write prefix without passing the "value to write" parameter, since it is unknown. Then, he must pass it as the linking write prefix as a parameter of the read prefix operator. This is shown in the example of Section 4.2.4.4.

4.2.4.2 Hiding

The hiding operator is built in a straightforward way. As with the CSPm operator, the JACK hiding operator just expects a process to hide an alphabet of communications. It is represented by the class `jack.jacs.japs.csp.impl.Hiding`. It expects a process and an alphabet as mandatory parameters. The operator can be used to encapsulate some internal set of events that the external environment must neither view nor interact with.

4.2.4.3 Mu and Recursion

These operators represent a recursive process definition and a recursive call to that definition. They are represented by the classes `Mu` and `Recursion` of the package

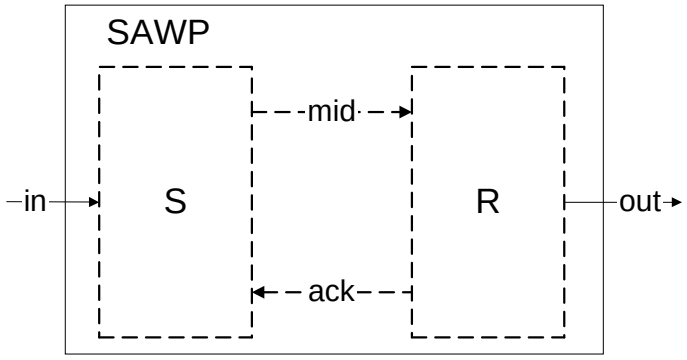

Figure 4.7: Stop-and-Wait Protocol

`jack.jacs.japs.csp.impl`, respectively. The `Mu` process expects a process body as a parameter. The `Recursion` process expects just the desired name of the `Mu` process definition to refer to.

The name of the `Mu` process must be used by the recursion process in order to properly link the recursive definition and the recursive call. JACK provides a dynamic check of `Mu`/`Recursion` relationship while in *kernel* mode, in such a way to ensure that recursive definitions and recursive calls are always consistent with each other (i.e. one cannot make a recursive call to an undefined `Mu` process). The checking procedure is dynamic because the network cannot infer its structure while in *user* mode; only when the network starts running (i.e. enters in *kernel* mode), and reaches the `Mu` definition we can check the consistency of the recursion.

4.2.4.4 Stop-and-Wait Protocol Example

To exemplify the use of the unary operators mentioned in this section, we provide the implementation of a stop-and-wait protocol specification in CSPm. The stop-and-wait protocol implements an one-place buffer. It consists of two processes, the sender process S and the receiver process R: a message is input by S, passed to R by S, and finally output by R. The protocol is illustrated by Figure 4.7.

Having accepted a message throught *in*, the sender process S passes the message to R along the *mid* channel, and then wait for an acknowledge before accepting the next message. The receiver process R accepts messages along *mid*, and sends an acknowledgement once a message has been output throught *out*. The channel *mid* and the acknowledgement signal event *ack* are private connections and should have no participants other than S and R: they are internal events.

The two processes are designed to be combined in parallel; parallel constructs are binary operators, they are mentioned in next Section. The processes S and R must wait for synchronization on their *in* and *out* channels from the external environment. This means that they ought to synchronize on these channels with other process that

channel ack%Event Type signal.
channel in, out, mid: int

% Build a set containing x' that comes from $\{0, 1\}$
$T \quad = \{x^{'} | x^{'} \leftarrow \{0 \ldots 1\}\}$

$S \quad = in?x : T \rightarrow mid!x \rightarrow ack \rightarrow S$
$R \quad = mid?y \rightarrow out!y \rightarrow ack \rightarrow R$
$SAWP \quad = S[|\{|in, out, mid, ack|\}|]R \setminus (\{|mid|\} \cup \{ack\})$

Figure 4.8: Stop-and-Wait Protocol — CSP Specification

perform the physical input and output operations. The specification of the protocol is given in Figure 4.8; this is taken from [130, Chapter 3]. The Java code that represents the specification of Figure 4.8 is given below.

```
package jack.jcase.sawp;

import jack.jacs.japs.csp.*;
import jack.jacs.japs.csp.impl.*;
import jack.jacs.japs.typesystem.*;
import jack.jacs.japs.typesystem.impl.*;

/**
 * Process that represents the stop-and-wait protocol of Figure 4.8 in JACK
 *
 * The CSPEnvironment interface defines some useful variables used in the example like
 * the signal B_EXECUTE_SUCCESSFULL and the DEFAULT_SUPERVISOR_ENVIRONMENT instance.
 */
public class StopAndWaitProtocol implements CSPBehaviour, CSPEnvironment {
    private static final boolean READONLY = true;

    private CSPType fChannelType;
    private CSPChannel fIn, fOut, fMid, fAck;
    private CSPAlphabet fSAWPSynchAlpha, fSAWPHiddenhAlpha;

    private Recursion fSr, fRr;
    private EventPrefix fEPS, fEPR;
    private ReadPrefix fRPS, fRPR;
    private WritePrefix fWPS, fWPR;
    private CSPOperator fS, fR, fSAWP, fSAWPGP;

    public StopAndWaitProtocol() {
        fChannelType = new IntType(0, 1);// Int type {0 ... 1}
        fIn = new Channel("in", fChannelType);
        fOut = new Channel("out", fChannelType);//Int range typed channel
        fMid = new Channel("mid", fChannelType);
```

```
fAck = new Channel("ack", EVENT_TYPE);//Event typed channel

/* Definition of Process S = in?x : T → mid!x → ack → S
 *
 * 1) Links Mu process S with the unwind process name of recursion process Sr.
 * 2) Pass the value to write of the write prefix to the read prefix (fRPS);
 * when the read prefix executes, it fills in the given value using value
 * serialization facility (see Section 4.4).
 */

fSr = new Recursion("S");//Creates a recursive call link to a Mu process named "S".

/* The prefix processes expect a channel and a process to follow.
 * The write prefix expects a value to write, but since its value is linked with the value of the read
 * prefix, it must inform this to it. To do so, just create a write prefix without an explicity value
 * to write, then pass the created write prefix as the linked prefix with the read prefix.
 * The read prefix for this case accepts two process parameters, one that is the process that the prefix
 * follows, and another that is the write prefix linked with the value read by this read prefix. In our case
 * they are the same process.
 */

fEPS = new EventPrefix((CSPChannelSignal)fAck, fSr);
fWPS = new WritePrefix((CSPChannelOutput)fMid, fEPS);
fRPS = new ReadPrefix((CSPChannelInput)fIn, fWPS, fWPS);
fS = new Mu(fRPS, fSr);

/* Definition of Process R = mid?y → out!y → ack → R */

fRr = new Recursion("R");

fEPR = new EventPrefix((CSPChannelSignal)fAck, fRr);
fWPR = new WritePrefix((CSPChannelOutput)fOut, fEPR);
fRPR = new ReadPrefix((CSPChannelInput)fMid, fWPR, fWPR);
fR = new Mu(fRPR, fRr);

/* Definition of Process SAWP = S[|{|in, out, mid, ack|}|]R \ {|mid, ack|})
 *
 * 1) Definition of alphabets
 * 2) Linking of already constructed processes
 */

fSAWPSynchAlpha = new Alphabet();//Builds the alphabet: {|in, out, mid, ack|}
fSAWPSynchAlpha.add(new CSPChannel[] { fIn, fOut, fMid, fAck });

fSAWPHiddenhAlpha = new Alphabet();//Builds the alphabet: {|mid, ack|}

fSAWPHiddenhAlpha.add(new CSPChannel[] { fMid, fAck });

fSAWPGP = new GeneralizedParallel(fS, fSAWPSynchAlpha, fR);
```

```
        fSAWP = new Hiding(fSAWPGP, fSAWPHiddenhAlpha);

    // *********** CSPBehvaiour Interface Implementation ***********

    public CSPAlphabet inspect() {
        return fSAWP.behaviour().inspect();
    }

    public CSPBehaviourSignal execute(CSPSupervisorEnvironment sve) {
        fSAWP.start(sve);
        return B_EXECUTE_SUCCESSFULLY;
    }

    // *********** User defined methods ***********

    /**
     * Creates a stop-and-wait behaviour, attach it to a process instance and start it using the default JACK
     * supervisor environment. Since a JACK process is an active object, after the start() call, the main thread
     * finishes its execution, and only the threads related to JACK processes remain active.
     */
    public static void main(String[] args) {
        CSPBehaviour swpBehaviour = new StopAndWaitProtocol();
        CSPProcess SWP = new JACKProcess(swpBehaviour);

        SWP.start(DEFAULT_SUPERVISOR_ENVIRONMENT);
    }
}
```

In this example, we use some structures of the JACK type system (see Section 4.4), like values and types.

4.2.5 Binary Operators

In this section, the most common CSP binary operators are presented together with an extended example of the stop-and-wait protocol.

4.2.5.1 Choice Operators

JACK provides three different choice operators. Each accepts two operand processes as mandatory arguments. They are implemented in package `jack.jacs.japs.csp.impl` and are summarized below.

1. `ExternalChoice` ($P \Box Q$) — represents choice over a set of events that the user environment can select. The implementation of this operator is an important point in the resolution of the backtracking problem mentioned in Section 2.3.2.

2. `InternalChoice` $(P \sqcap Q)$ — represents an abstraction over a set of events. That is, it internalizes the selection of one of its process in order to abstract the choice decision from the external environment.

3. `ConditionalChoice` $(P \lhd b \rhd Q)$ — represents a choice based on a conditional boolean guard. It is similar to the **if** b **then** P **else** Q selection constructor of programming languages. Furthermore, it has a boolean guard defined by a `jack.jacs.japs.csp.CSPGuard` interface to represent the boolean query b of the **if** operator.

There is another choice operator in CSP not available in JACK; it is called sliding choice $(P \rhd Q)$. It is omitted since it can be constructed using the other available choice operators (i.e. $P \rhd Q \equiv (P \sqcap STOP) \Box Q$). Roscoe mentions [124, Chapter 11] that the implementation of the sliding choice operator can be explicitly made (i.e. without the transformation). JACK allows this explicitly construction through especialization of the supervisor environment event selection policy. Figure 4.11 presents an example of the choice operators.

4.2.5.2 Parallel Operators

There are four different parallel operators in JACK. They take two operand processes as mandatory arguments. The generalized and alphabetized parallel operators expect the synchronization alphabets as well. They are summarized below.

1. `GeneralizedParallel` $(P[|X|]Q)$ — represents the parallel composition of two processes P and Q, synchronizing on the events inside the alphabet X. This is the most generic parallel operator; all other operators can be defined in term of this one [124, Chapter 2].

2. `AlphabetizedParallel` $(P[X||Y]Q)$ — represents the parallel composition of P and Q, and establishes the alphabet of events that they are allowed to communicate. It means that P and Q can only communicates events inside X and Y alphabets respectively, deadlocking otherwise. They must synchronize on events inside the alphabet $X \cap Y$.

3. `Interleaving` $(P|||Q)$ — a generalized parallel where the alphabet X is empty: they never need to synchronize and run completely independent of each other. This operator can be used to include non-determinism in parallel systems.

4. `SynchronousParallel` $(P||Q)$ — represents a generalized parallel where the alphabet X is the set of all possible events defined as available for communication; in CSP this is captured by the Σ alphabet. The operator can be

described in terms of the generalized parallel as $P[|\{|\Sigma|\}|]Q$. This means that these processes must synchronize in every communication that they perform. Definitions like this can be used to implement replication on distributed systems.

The implementation of the parallel operators is an important point on the resolution of the multisynchronization problem mentioned in Section 2.3.3.

4.2.5.3 Sequential Composition

The sequential composition $P;Q$ behaves as P until it terminates successfully (i.e. performs $SKIP$), at which point it passes control to Q. Almost all programming languages provide a sequential composition operator. We need to be careful, however, as to what successful termination means. In CSP, a process terminates successfully if it performs the $SKIP$ primitive process.

Another important note is related to termination and parallel composition. As mentioned in [130], the treatment of termination is a design decision, and so is implementation dependent, although parallel compositions are always required to synchronize on $\surd$ (i.e. $SKIP[|X|]P \equiv P$). For instance, Roscoe's treatment of termination ensures that, if a process can possibly terminate, then the process itself can choose to terminate and refuse all subsequent interaction (i.e. it terminates independently of its external environment). On the other hand, Hoare's treatment achieves the same results imposing a restriction requiring that $SKIP$ should never be offered as an alternative in an external choice (i.e. the process $P \Box SKIP$ is denied; in Roscoe's CSP this process is equivalent to $P \rhd SKIP$).

JACK follows the guidelines established in [130] to properly implement termination. These guidelines are more general and reflect both approaches. They are the same as Roscoe's, but with the agreement of the external environment on termination. This does not mean that the user must decide to terminate when possible, but that the environment has the ability to either leave the decision to the process, or open the decision possibility to the user. Actually, the supervisor environment does not open this possibility to the user, so it implements Roscoe's version. Nevertheless, the generalization of this is straightforward, since the supervisor was designed with the Schneider's solution [130, Chapter 3] in mind.

4.2.5.4 Stop-and-Wait Protocol Extended Example

A stop-and-wait protocol, which permits its input to be overwritten once, if it has not already passed along the *mid* channel, is specified in Figure 4.9 and illustrated in Figure 4.10. Figure 4.11 shows the JACK code for this extended version, presenting the different relevant parts.

channel ack%Event Type signal.
channel in, out, mid: int

% Build a set containing x' that comes from $\{0, 1\}$

$$T = \{x^{'} | x^{'} \leftarrow \{0 \ldots 1\}\}$$

$$S2 = in?x : T \rightarrow (S2 \Box (mid!x \rightarrow ack \rightarrow S2))$$
$$R2 = mid?y \rightarrow out!y \rightarrow ack \rightarrow R2$$
$$SAWP2 = S2[|\{|in, out, mid, ack|\}|]R2 \setminus (\{|mid|\} \cup \{ack\})$$

Figure 4.9: Stop-and-Wait Protocol Extended — CSP Specification

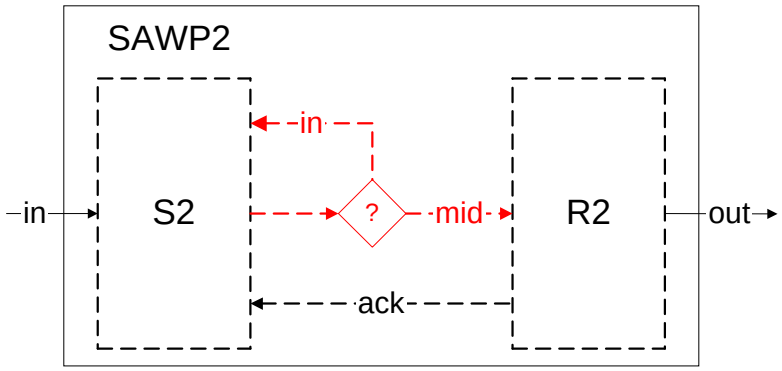

Figure 4.10: Extended Stop-and-Wait Protocol

After its first input $in.x$, the process $SAWP2$ is in the position where a choice is to be made between an external event $in.w$ and an internal event $mid.x$; that is represented in Figure 4.10 as diamond shape. If at this point the environment simply waits for output to be offered, and offers no further input, then the internal event must occur, and output is indeed offered. On the other hand, if the environment offers a second input, then there are two possibilities: the internal event has not yet occurred, and the second input is accepted; or the choice has already been made in favour of the internal communication, in which case the second input will be refused. The environment is unable to prevent this second possibility [130, Chapter 3].

In Figure 4.12, the graph that represents this process network is given.

4.2.6 Extended Operators

JACK provides some extend CSP operators that are very useful. For instance, the Iteration ($P^* = P; P^*$) operator is provided. There are interesting operators not yet available but considered for future relases like labelling ($l : P$), piping ($P \gg Q$), and event renaming (forward $f(P)$, and backward $f^{-1}(P)$). Another interesting set of operators that could be implemented in JACK are the timed CSP operators [130] and the replicated version of all CSP operators.

4.3 JACK User Processes

We already know how to create user processes, how to use CSP operators, and user defined behaviour. Now, let us give some details about user processes definition.

In JACK, it is possible to have user defined processes through the implementation of the `CSPBehaviour` interface. A user defined process can specify any sort of behaviour, both using available CSP operators or some other necessary construct like structured data types.

A JACK user process has the same structure of a CSP operator. The main difference is the fact the behaviour of CSP operators is the implementation of the operational semantics [126] of that operator, while the behaviour of user processes is specific.

JACK already provides a process interface implementation for the generic part of a process specification, as already mentioned in Section 4.1.1.1. The user must just provide an implementation of its specific desired behaviour, and submit that behaviour to the already implemented process interface. Thus, the process interface deals with structural (generic) properties, and the user behaviour interface deals with behavioural (specific) properties.

```
/** Process that represents in JACK the SAWP protocol of Figure 4.9 */
public class StopAndWaitProtocolV2 implements CSPBehaviour, CSPEnvironment {
    public StopAndWaitProtocolV2() {
        ...
        /* Definition of Process S2 = in?x → (S2□mid!x → ack → S2)) */

        fS2r = new Recursion("S2");
        fEPS = new EventPrefix((CSPChannelSignal)fAck, fS2r);
        fWPS = new WritePrefix((CSPChannelOutput)fMid, fEPS);
        fECS = new ExternalChoice(fS2r, fWPS);
        fRPS = new ReadPrefix((CSPChannelInput)fIn, fWPS, fECS);
        fS2 = new Mu(fRPS, fS2r);

        /* Definition of Process R2 = mid?y → out!y → ack → R2 */

        fR2r = new Recursion("R2");

        fEPR = new EventPrefix((CSPChannelSignal)fAck, fRr);
        fWPR = new WritePrefix((CSPChannelOutput)fOut, fEPR);
        fRPR = new ReadPrefix((CSPChannelInput)fMid, fWPR, fWPR);
        fR2 = new Mu(fRPR, fR2r);

        /* Definition of Process SAWP2 = S2[|{|in, out, mid, ack|}|]R2 {|mid, ack|})
         *
         * 1) Definition of alphabets
         * 2) Linking of already constructed processes
         */

        //Builds the alphabet: {|in, out, mid, ack|}
        fSAWPSynchAlpha = new Alphabet();
        fSAWPSynchAlpha.add(new CSPChannel[] { fIn, fOut, fMid, fAck });

        //Builds the alphabet: {|mid, ack|}
        fSAWPHiddenhAlpha = new Alphabet();
        fSAWPHiddenhAlpha.add(new CSPChannel[] { fMid, fAck });

        fSAWPGP2 = new GeneralizedParallel(fS2, fSAWPSynchAlpha, fR2);
        fSAWP2 = new Hiding(fSAWPGP2, fSAWPHiddenhAlpha);
    }
```

Figure 4.11: Extended Stop-and-Wait Protocol using JACK — Java code

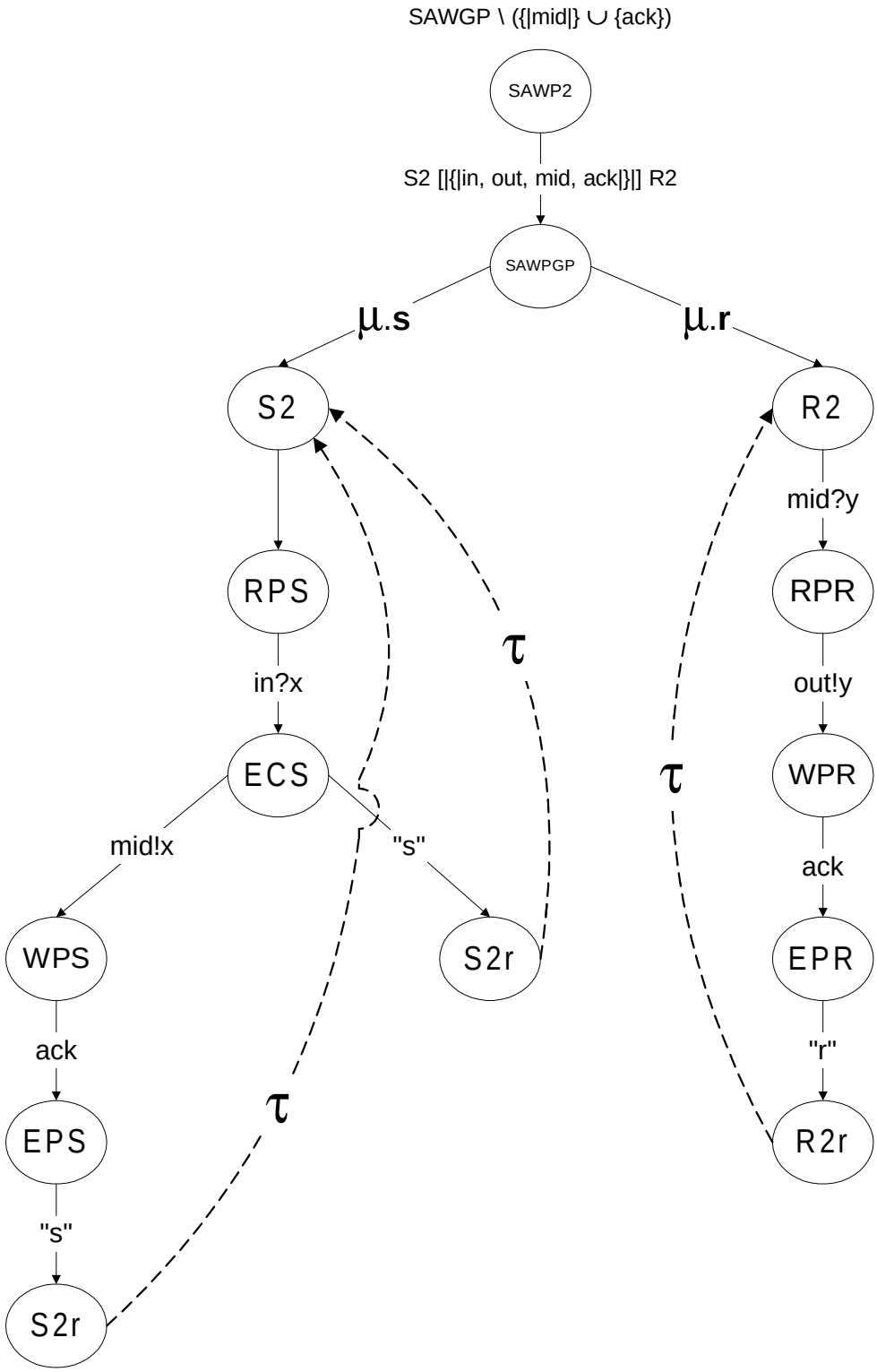

Figure 4.12: Stop-and-Wait Protocol Extended Version — LTS

4.3.1 User Process Behaviour

The user process behaviour is defined by the `jack.jacs.japs.csp.CSPBehaviour` or `jack.jacs.japs.csp.CSPExtendedBehaviour` interfaces. The user must implement one of these interfaces and submit it to the JACK process interface implementation (`jack.jacs.japs.csp.impl.JACKProcess`) to run.

Hoare [61, pp. 38] states that, for the implementation of processes, two structures must be defined.

1. A function f describing the process behaviour for an event selected;
2. A domain D defining the set of events which the process is initially prepared to engage. For each $x \in D$, $f(x)$ defines the future behaviour of the process if the first event to be communicated is x.

In this sense, a JACK process must provide this necessary information, in order to be properly implemented. To do so, a JACK process implementation must be able to provide an answer to the questions below.

1. What a process wants to communicate immediately (i.e. domain D)?
2. From the available events inside the domain D, what communication was selected by the external environment (see communication decision functions in Section 4.5)?
3. What is the behaviour of a process after the selected communication has occurred (i.e. function f)?

Thus, the `CSPBehaviour` interface is responsible for implementing both functionality through its methods. The `execute()` method is directly related to the function f, and the `inspect()` method is directly related to the domain D. The former must specify the desired conduct of a process execution, and the latter must returns the alphabet of possible events in which the process is prepared to engage immediately.

These methods execute under the observation of the `CSPSupervisorEnvironment`, in order to properly implement the multisynchronization and backtracking protocols mentioned in Sections 2.3.3 and 2.3.2 respectively. This parameter extends the Hoare's description of process implementation [61, pp. 38—39], in the sense that process execution is now controlled by an external supervisor environment. Thus, processes have a controlled autonomy of their execution thread. For instance, the CSP operators implementation makes use of these supervision information, in order to properly implement their semantics. Specialized user processes can also use these information. For instance to inspect the current execution frame information like backtrack points, multisynchronization points, hidden points, or recursive definition points; or to inspect the execution history of the process. For details about the supervision information see Chapter 6 and [38].

4.3.2 User Behaviour Interface Definition

User process behaviour can be defined in two ways, as mentioned below.

1. Via interface implementation — the user class implements the `jack.jacs.japs.csp.CSPBehaviour` interface and submits it to the `jack.jacs.japs.csp.CSPProcess` interface implementation to run; the implementation is called `jack.jacs.japs.csp.JACKProcess`. Thus, the user behaviour is decoupled from the JACK process hierarchy.

2. Via class inheritance — the user class extends the `CSPProcess` interface implementation provided by JACK. Thus, the user behaviour is coupled with the JACK process hierarchy, and it must not have any other superclass.

These two versions of user behaviour definition is similar to the Java Thread architecture definition [63, Chapter 2]. Actually, the relationship between the `JACKProcess` class and the `CSPBehaviour` interface is implemented in the same way as Java `Thread` class and `Runnable` interface (i.e. the `JACKProcess` class implements the `CSPBehaviour` interface and provides a no-op versions of its methods, see Figure 5.4 in Section 5.2.2.1).

4.3.3 User Behaviour Definition for Combined Specifications

Users can implement behaviours that represent combined specifications, like CSP-OZ [31] or Circus [148, 149]. Doing so, the user may want to interact with the supervisor environment, in order to properly know what the supervisor wants to communicate (i.e. which events are actually available). The user ought to know how to use alphabets and some of the supervisor environment methods, in order to properly implement this kind of specialized specifications. Some examples of them can be found under the `jack.jcase` package. The complete and detailed documentation of these entities can be found in [38].

An important note must be mentioned about user behaviours that specify combined specifications. If one uses other constructs than the available operators, that can generate a τ signal, the user must deal with this τ signal at process execution. That is, the supervisor environment can select the user's τ action without calling the user decision function since it is an internal event; thus users must be aware of any τ signal that it generates outisde the CSP operators domain. This topic is detailed in [41, 38].

4.4 JACK Type System Overview

JACK provides a robust type system to implement CSP specifications. An important aspect that demands a strong type system is the symbolic approach to deal with infinite data types. Since we do not make type domain expansion, we must have some way to infer the type under consideration, in order deal with operations on it symbolically.

With the type system, the user can define various different entities useful to describe processes. In what follows, we briefly summarize some of the most important aspects of the JACK type system directly related to the basic user point of view of the framework. For more details and advanced features of the type system, see [38, 41].

Types — It is possible for the user to define its own data type. The framework provides ways to define many sort of types. It has multidimentional, ranges, enumerable, comparable, primitive, and variant types. The type system also provides a series of useful type related operations, like type structure inspection, value acceptance (domain) check, prototyped value construction, type compatibility, event notifications, and so forth. An interesting aspect of type implementation is the fact that it is the place where the physical symbolic dealing with CSP values is achieved.

For instance, one can create an infinite range like the natural numbers with

```
CSPRangeBound lowerB = new IntRangeBound(0, INCLUSIVE);
CSPRangeBound upperB = IntRangeBound.getUnlimitedBound(ISPOSITIVE);
IntType naturals = new IntType(lowerB, upperB); // {0 . . . ∞+{
```

Values — JACK clients need to use values to properly define type constraints, values to output on write prefixes, or linked values for inputs on read prefixes. Every value in JACK can infer its type. There are comparable values, related to comparable types; range bound values, that establish the limits of a range type; enumerable values, related to finite enumerable types; number values, a special sort of value that has built-in arithmetic operations that can be used as a primitive expression language (i.e. integer addition, subtraction, multiplication, and division). In the same way as with types, there are some value operations of interest to the JACK user. For instance, value type inference, value structure inspection, value alteration, event listening, physical serialization, and so on. Most parts of the JACK values are inspired on the structure and architecture of the JValue framework [117, 44, 112, 111].

Value Sets — A value set can be viewed as a type domain restriction through the use of a logical predicate; currently, only the propositional calculus is available

(i.e. $\neg$, $\vee$, and $\wedge$ logical operators). With the use of value sets, and value constraints, the implementation of the JACK type system includes a normal form reduction algorithm of a subset of the predicate calculus. A value set is composed of two main parts: the type domain under consideration and a value constraint predicate for that type domain. We can use set operations like union ($\cup$), intersection ($\cap$), difference ($\setminus$), generalized union, and generalized intersection. Other test operations like membership ($x \in S$), subset ($\subseteq$), proper subset ($\subset$), and equality ($=$) are also provided. There are many sorts of value sets, varying from normal finite ones to optimized sorted set infinite ones.

Value Set Constraints — Value set constraints are predicates over a value set type domain. With them, it is possible to restrict the value set domain with some propositional logic predicate, an enumerated set of values, a range, and so on. The value set constraint classes are responsible for the implementation of the normal form algorithm of a subset of the predicate calculus; this algorithm is based on [67, 68, 21], and is detailed in [38, 41]. The constraint also provides some useful operations like value acceptance check, equality test, and reduction operation. The JACK client can use these features to implement a complex data structure of its processes specification.

In Figure 4.13, we show some examples in a Java method that uses some of the mentioned structures of type system; they come from test case classes in package `jack.jacs.japs.typesystem.tests`. The method shows how to create range bound values and range types, and how to use some of the range type methods. These methods asserts that one range encloses the others, that some of them overlaps, and so on.

4.5 JACK Process Subsystem Overview

The JACK process subsystem deals with the non-functional aspects and requirements of process execution. These aspects are basically supervised execution and low-level concurrency support.

Users may never need to deal with such structures directly, but it is interesting for them to know the roles that these structures play, in order to have a better understanding of the framework as a whole. Despite this fact, this information is neither necessary, nor vital to be knowledge by the common user to properly describe their processes. Advanced users and framework extension developers however, must completely understand these entities and the protocols interfaces that controls their interaction between the other main entities already mentioned.

In what follows, we briefly summarize some of the most important elements of the JACK process system.

```
import jack.jacs.japs.typesystem.*;
import jack.jacs.japs.typesystem.impl.*;

/**
 * Test range type and range bound value construction and related operations
 */
public void testRangeTypeAndValue() {
    boolean READONLY = true;
    boolean POSITIVE = true;
    boolean INCLUSIVE = true;

    /* Comparable values instantiation. Represents the int values 0 and 1.*/
    CSPComparableValue v0 = new ComparableValue(new Integer(0), READONLY);
    CSPComparableValue v1 = new ComparableValue(new Integer(1), READONLY);

    /* Creates a closed range bound value representing the bound of some range, i.e. {0...??}. */
    CSPRangeBound rb0 = new RangeBound(v0, INCLUSIVE);

    /* Creates a opened range bound value representing the bound of some range, i.e. {??...1{. */
    CSPRangeBound rb1o = new RangeBound(v1, !INCLUSIVE);

    /* Creates a unlimited positive int range bound, i.e. {??...infty+{ */
    CSPRangeBound rbPos = new RangeBound(v0.type(), POSITIVE);

    /* Creates the range types {0...infty+{, {0...1{, and }1...∞+{*/
    CSPRangeType rt0Pos = new RangeType(rb0, rbPos);
    CSPRangeType rt01o = new RangeType(rb0, rb1o);
    CSPRangeType rt1oPos = new RangeType(rb1o, rbPos);

    /* Asserts that the operation returns true.If the operation returns false, the method throws an exception. */
    assertTrue(rt0Pos.encloses(rt01o));        // {0...1{ ⊆ {0...∞+{
    assertTrue(!rt01o.encloses(rt0Pos));       // {0...∞+{ ⊈ {0...1{
    assertTrue(rt0Pos.overlaps(rt1Pos));       // {0...1{ ∩ {0...∞+{ ≠ ∅
    assertTrue(rt0Pos.accept(v1));             // 1 ∈ {0...∞+{
    assertTrue(!rt1oPos.accept(v0));           // 0 ∉ }1...∞+{
}
```

Figure 4.13: JACK Type System — Java Code Example

Process Supervisor Environment — This environment is responsible for controlling the occurrence and selection of possible communicable events. It is also responsible for implementing two important aspects of the JACK framework: the ability to deal with multisynchronization and backtrack without loosing the compositionality property of CSP; and to connect the functional JACK semantic layer with the non-functional JACK execution layer. The supervisor environment is represented by the `jack.jacs.japs.csp.CSPSupervisorEnvironment` interface and is passed as a parameter of the `CSPBehaviour` interface methods. Users that just compose CSP operators may never need to deal with supervisors directly. On the other hand, advanced users that compose processes in a non-trivial way (i.e. with some data dependent operation or processes that can deal with internal events like τ), or extension developers that may provide other CSP operator implementations like Interrupt ($P \triangle_{event} Q$, see [126, 124, 130]), must be aware of the supervisor interface.

Communication Decision Function — A communication decision function is a strategy [28, pp. 315] used by the supervisor environment to decouple the event selection functionality. The `DEFAULT_SUPERVISOR_ENVIRONMENT` instance provides a default random decision function that randomically selects events for the inspected alphabet of initially available events. Users can install different decision functions on the supervisor environment, in order to take control of the event selection routine. For instance, one can implement a Java Swing [141] user interface that selects events pushing buttons available according to the supervisor inspection. The communication decision function is represented by the `jack.jacs.japs.csp.CSPCommunicationDecisionFunction` and defines only one method called `select()`, that receives an inspected alphabet and returns a selected communication. Decision functions and supervisor environment details are mentioned in Chapter 6 and in the on-line documentation [38].

Process Execution Network — The process execution network is responsible for providing the non-functional aspects of process execution. The process network represents the encapsulation of the low-level concurrency world that needs to deal with threads, synchronization locks, and recovery schemes. The network acts like the client of the JACK bottom-most low-level execution layer. Together with the supervisor environment, it represents the link between the semantic layer and the execution layer (see Figure 5.3 in Section 5.2.2). This layering scheme of the framework is completely detailed in Chapters 5 and 6.

These entities are mentioned in Chapter 6. For more details and related advanced features, see [38, 41].

4.6 Final Considerations

In this chapter, a description of some of the most important JACK usage guidelines have been given. We described the structure of processes, the user behaviour definition, the CSP operators, and the JACK type system. Some simple CSP specifications and corresponding Java class code examples were given, in order to make it clear how JACK implements the expected functionalities.

The next two Chapters present the JACK framework architecture and implementation, respectively.

Chapter 5

JACK Framework Architecture

This chapter presents the JACK architecture and design rationale. JACK stands for *Java Architecture with CSP Kernel*. This framework implements the CSP [124] process algebra in Java as if the CSP operators were available as part of the language. The framework aims at making concurrency primitives more abstract and the related low level functionality to be hidden, in order to avoid the idiosyncratic situations that users of concurrent programming environments normally face.

In Section 5.1, important framework requirements are discussed. Then in Section 5.2, a description of each framework layer is given; that description gives an overview of the framework infrastructure; details about layer composition are presented in the next Chapter. After that, in Section 5.3, a general discussion about design pattern usage and its importance in the framework construction process is presented. Finally, in Section 5.4, final considerations are given.

5.1 Requirements

As discussed in the last chapter, the existing library solutions for implementing CSP in Java(CTJ [59] and JCSP [107]) do not attend all our needs. Despite this, the lack of a framework model and well-defined design patterns make the adaptation and extension of these libraries very difficult.A detailed study of these library solutions was carried out [36], but it shows that the libraries were not built to deal with the kind of problem that JACK aims to solve, nor it is clear how to alter or extend their functionality to implement most CSP [124] operators, our desired result.

The most complex part of our framework implementation was identified as the definition of a generic and safe locking and threading scheme to execute processes. Both in CTJ and JCSP, this is mixed with process semantics, which makes the understanding of the library internals for extensions very difficult, which in turn leads to modularity loss.

In order to avoid this problem, the decision to build a new library as a frame-

work with well-defined layers and problem domains was done as discussed in Sections 3.1.1 and 3.1.2. Therefore, the selected thread support library must both be designed as a framework, with design patterns, and explained by a pattern language in order to achieve these desired goals.

We investigated the framework modeling techniques [133, 113, 90, 22], design patterns and pattern languages [121, 28, 73, 76, 123, 102, 81, 57, 23]. Our aim is a reuseable, simple, expressive, and extensible architecture, that supports incremental development. As already mentioned in Chapter 3, the available implementation mixes functional (i.e. process semantics) and non-functional (thread scheme) properties that leads to obscurity and difficulty to understand and extend.

It is imperative to JACK that it offers constructs with high expressive power. Nevertheless, expressiveness should be achieved with simplicity. For instance, it is well-known [8] that semaphores can emulate monitors and vice-versa. However, such emulations are complex. Thus, constructs provided to the programmer should result from a compromise between expressiveness and simplicity (i.e. easy of use).

JACK must also allow the extension of CSP operators by using traditional reuse mechanisms of object-oriented programming: class inheritance and object composition [28]. It must also support incremental development of operators. Moreover, synchronization leads to a well-known problem with object-oriented frameworks and synchronization schemes called the inheritance anomaly problem [86, 87]. This occurs most commonly because there is no language support for a kind of *lock inheritance*; this way proper subclass locking becomes obscure or even impossible. McHale [87] also shows that, even when synchronization code is structurally separated from functionality code, if synchronization constructs have a limited expressive power, not supporting, for instance, the six types of synchronization information enumerated by Bloom [9], then it is necessary to use functional code to preserve desired synchronization information, hence losing the separation between functionality and synchronization [133]. That vital non-functional problem is delegated to be treated by a framework called DASCO [133, 132] due to its separation of inheritance for functionality reuse, and synchronization reuse. DASCO stands for *Development of Distributed Application with Separation and Composition of Concerns*.

Our decision to use separation of concerns is motivated by the clearly observed property of modularity and well-defined dependencies. The separation of concerns approach states that the framework should be broken into well-defined layers that must capture a specific concern of the design and functionality. The JACK framework implementation deals with integration of concurrency, synchronization, recovery and processes. It provides an implementation of those concerns and also consider concern composition. The DASCO implementation provides each concern without loosing desired object-oriented framework properties like expressiveness, simplicity, reusability, modularity, and incremental development. DASCO also was designed as a framework, using a detailed and well-defined set of design patterns, composed

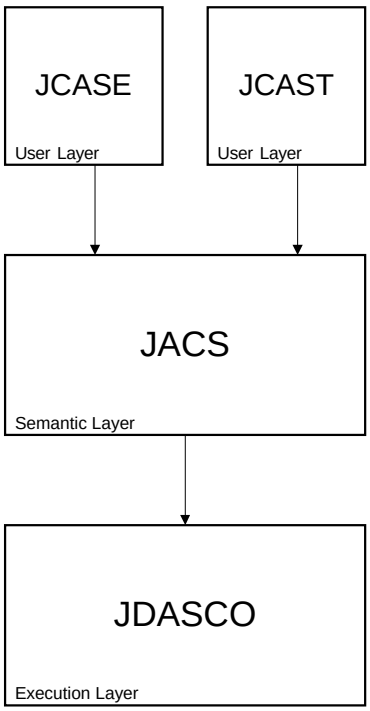

Figure 5.1: JACK main layers

together using a pattern language [133].

Composed CSP operators are difficult to test, implement, and debug due to the non-determinism inherent to their execution, or semantic complexities. Incremental development must allow incremental program construction in such a way that functionality is developed first and tested in a sequential environment. Object concurrency, synchronization, and interaction are introduced later. It should be noted that incremental development assumes the reusability requirement, but the reverse is not necessarily true.

5.2 Layer Description

In this Section a description of JACK layers is provided. Each layer represents a piece of functionality that grants service implementation at well-defined points, in order to allow other layers to use this functionality independently. The layering strategy is very important to decouple and distribute the complexity of a large object-oriented system, making each layer to contribute with its partner, as mentioned in [28, Chapter 2].

JACK has three main layers that have themselves some sub-layers. Figure 5.1 shows how the main layers are organized.

Execution Layer: JDASCO — Java Development of Distributed Application with

Separation of Concerns

The bottommost layer is responsible to deal with low-level functionality, and to provide non-functional features like threads, monitor locks, etc. It is a Java extended version of DASCO [133, 132].

Semantic Layer: JACS — Java Architecture with CSP Semantics

The semantic layer acts as the user of JDASCO. It provides user processes, CSP operators, and CSP auxiliary constructors. It is the heart of the JACK framework, since it provides most of the functionality observed by the final user (i.e. a JACK client).

User Layer The user layer is open and should be used by the JACK client to implement its specification or desired functionality. In other words, it is just a design layer that represents the entry point of the JACK framework, it does not provide any functionality related to processes, or CSP operators semantics.

Here, we present two examples of possible user layer configurations:

JCAST — Java CSP Abstract Syntax Trees

The AST representation layer acts as a user of JACS. It provides AST representation of CSP operators. It could be used by a parser to build an AST that represents a pure CSPm specification in JACK.

JCASE — Java CSP Applications, Samples and Experiments

This layer appears just as a case study of the JACK framework to build JACK processes from a simple CSP specification example. It also acts as a user of JACS that builds Java classes from specification source code. A formal translation from CSP to Java using CTJ [59] is presented in [12]. A similar work for JACK is under consideration as a possible future work.

A detailed description of each layer is given in the next subsections.

5.2.1 Execution Layer — JDASCO

The execution layer is the bottommost layer. It is responsible for dealing with non-functional properties of processes. This layer provides and combines concurrency with thread management; synchronization with monitor and locks coordination; and recovery through commitment, abortion, and preparation operations, and also a walkable execution history.

The intention here is to generally describe the configuration options of DASCO functionality and some minor design details. For full details about DASCO design, see the main reference [133], or the description of its published design patterns in [122, 101].

JDASCO is divided in three sub-layers, as shown in Figure 5.3. The bottom layer is called the concern layer; it provides the concerns under consideration: concurrency, synchronization, and recovery. The middle layer is called the composition layer; it provides the combination of those concerns: concurrency with synchronization, synchronization with recovery, concurrency with recovery, and so forth. Finally, there is a top application layer example, showing how to use the concern combination. The JDASCO client ought to make use of its composition services filling in all user dependent roles, and following its strict usage guidelines; these are fully detailed in [133].

There are initial prototype versions of DASCO [132, 37] that provide all possible composition of concerns, and application examples. The implementation developed in JACK considers, and extends only the composition of interest to it: the concurrent synchronized recoverable one. That implementation also restructures the framework to become role-modeling (i.e. avoid multiple inheritance, creating protocol interfaces, etc.), and event-oriented (i.e. make use of event notifications [110] to allow interactive access to many important points of functionality).

The service offered and the composition alternatives define a number of policies with configuration possibilities. They must be strictly defined by the application layer which, in the case of JACK, is one of the semantic sub-layers (`JACS.JDASCO`, see Section 6.2.3). The policy selection of each concern and composition must follow some restrictions and usage guidelines defined in [133] and in Chapter 6.

The main motivation for this kind of service arrangement using separation of concerns was based on a thorough research in the concurrency development field. It was found that the main and most complex problem to make a CSP or occam implementation are both the concurrency and synchronization designs.

Many design patterns [122, 101, 81, 28], algorithms and formal descriptions [52, 6, 3, 129, 88, 58, 83], and semantic descriptions [126, 130, 14, 82] that have been trying to solve this kind of problem were found. DASCO [133, Chapter 2][1] directly deals with most important problems related to concurrency and synchronization, like partitioning of states, history sensitiveness, inheritance anomaly, and so on. DASCO also deals with another very important aspect for JACK: recovery in the case of backtrack occurrence (see Section 2.3.2).

Therefore, the use of this framework is adequate. DASCO does not solve all necessary problems and presents a few minor difficulties for us. Nevertheless, it has shown to be well-suited as the base of low-level JACK design and implementation

[1] In Rito Silva PhD thesis Chapter 2, there are detailed description about all these properties.

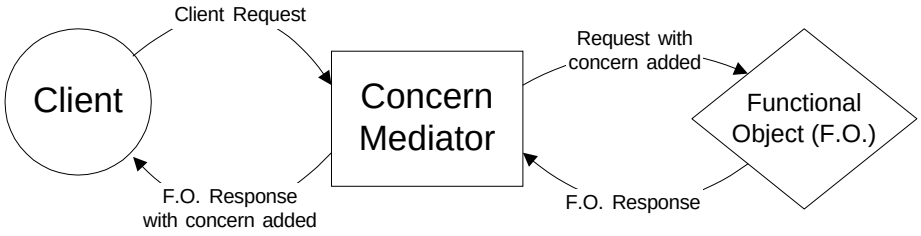

Figure 5.2: JDASCO Main Participants

architecture.

Our implementation of DASCO is a role modeling [113, 120] redesign of the original DASCO framework [133, 132]. The original DASCO was built using `C++` and it was translated to Java using role modeling as suggested in the future work Section of [133]. In this remodeling we introduced some design patterns and pattern languages, in order to generalize and normalize the original implementation. For a detailed model of JACK processes see [38]; it provides a Role and UML model descriptions of the framework, links to related (draft) documents, like the description of the backtrack and multisynchronization solution, the structure of the process network graph, tutorials for the framework, and so on.

5.2.1.1 Main Design Patterns Used

The main design patterns used in the JDASCO layer were those defined in [81, 123, 102, 133]. They are called Concurrent Object, Synchronized Object, and Recovery Object and are related to concurrency, synchronization, and recovery. These patterns have a uniform set of participants: a JDASCO client, a concern mediator, and a functional object.

The scenario was drawn as a client requesting some service of a functional object, as shown in Figure 5.2. That request is intercepted by the concern mediator responsible to introduce the desired concern service (i.e. Concurrency, Synchronization, or Recovery) or compositions of it. By composing the pattern entities in this way, neither the functional object nor the client need to deal with each concern functionality directly, but just with its own related behaviour and responsabilities.

For instance, there is no thread construction or locking scheme under control of the functional object. In this sense, the concern mediator abstracts the concern implementation from the JDASCO client. This separation allows the implementation of the JDASCO client and the functional object to concentrate on their functional behaviour instead of non-functional responsabilities. However, this does not mean that the JACK or JDASCO client will never need to deal with threads or locks. What it does mean is that the user never needs to deal with locking and threading related to the desired functionality under consideration, in our case, the thread-

ing and locking schemes of processes implementation (i.e. CSP operators and user processes).

For instance, patterns like mutex [81], reentrant mutex, conditional variable, latch, etc, are used to build low-level functionality. Basically, the JDASCO code for those patterns is inspired on the *CPJ* (Concurrent Programming in Java) [81, 79] framework, and *ACE* (The Adaptative Communication Environment) [138, 128] (see Section 3.2.4.1) framework source code and available documentation.

The composition and usage guidelines of these patterns were explained and based on a pattern language defined in [133]. Stepwise implementation and integration of policies and concerns is proposed, justified, and explained in Chapter 6.

5.2.2 Semantic Layer — JACS

Immediately above the execution layer, there is the semantic layer. Its main objective is to provide a high-level process representation. It makes use of the services of the execution layer. The implementation of the semantics of the processes follows the operational view of them described in [124, 126, 130, 61]. Many other implementation details are collected from other libraries [107, 59, 32, 18, 16], algorithms [13, 131], CSP related articles [52, 3, 129, 83], and other sources [6, 58, 88], as already mentioned in Chapter 3.

The process network is part of the semantics layer and it is represented by a LTS graph marked with some tag interfaces. The complete description of it includes too low-level details related to JDASCO composition of concerns and JDASCO concern collaborations. These details about them can only be mentioned after the next Chapter. They are not mentioned in the dissertation, but can be found in a draft document in [38, 40] and in JACK JavaDoc on-line documentation in the same source.

Constructors for types, channels, communication, alphabets, etc, are also provided (see Chapter 4). With these constructors, building a process that represents a CSP specification is easy, and results in readable and safe contained code.

Currently, the framework implements the most common operators like prefixes, choices, recursion, sequential composition, and parallel composition operators. In Chapter 2 these operators are briefly described, and in Chapter 4 there are many examples and situations illustrating useful usage guidelines of these operators in JACK.

Attention should be given to the possibility of user defined process descriptions. JACK gives the user the ability to compose and define their own processes in whatever way they want (i.e. not only using CSP operators). For instance, the user process can make use of any complex Java data structure. This is important to implement specifications in integrated languages like CSP-OZ and Circus.

The JACS semantic layer has two sub-layers, as shown in Figure 5.3. The one

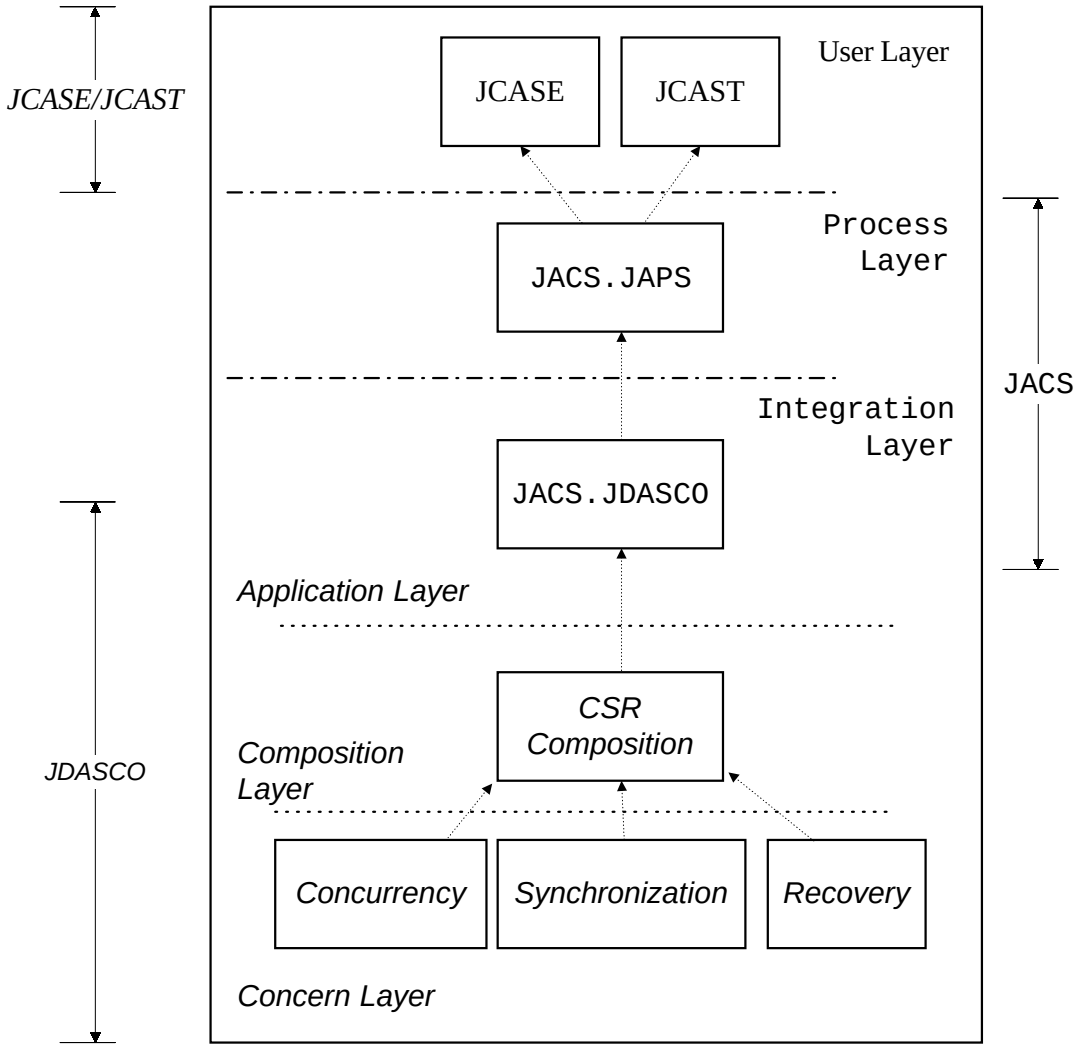

Figure 5.3: JACS and JDASCO sub-layers integration

called JACS.JAPS (process sub-layer) provides the semantic machinery to describe CSP specifications (i.e. CSP operators and auxiliary constructs) and processes. JAPS stands for *Java Applications for Processes Surveyor*. The other layer, called JACS.JDASCO, links the CSP semantic elements with JDASCO, acting as the JDASCO application layer; it is called the integration sub-layer.

JAPS is the heart of the framework from the end user point of view. It is implemented based on role modeling [113] ideas. JAPS is divided into groups of functionality (i.e. Java packages) as follows:

- JAPS.CSP — represents the CSP operators, alphabet, process supervisor environment, process network, etc.

  It defines the service interfaces, the default implementation of them, and test case classes.

- JAPS.TYPESYSTEM — represents the JACK type system.

  It defines interfaces default implementation, and test cases for types, values, value sets, and so on.

This kind of functionality division, as commented in [117, 119, 118], has shown to be very useful in the source code organization and class files interdependency during the compilation procedure.

The following subsections present the representation of the structure of processes, operators and user defined processes. It shows the class diagrams relating each entity and some construction details related to the architecture organization.

5.2.2.1 Process Representation

This subsection presents the representation of the already defined process structure (see 4.1.1). Processes are the main entity in the framework and are provided as a Java extension package. They represent the infrastructure stream of execution, so the package acts as an execution environment. To completely define a process, it must have a companion behaviour (i.e. a specific definition to be plugged inside a generic execution environment). This way, a CSP operator can be viewed as a JACK process environment with a behaviour already defined and provided by the framework.

JACK users that want to describe processes other than CSP operators must provide a behaviour interface implementation to achieve their needs. The user behaviour interface must provide a function f describing the process behaviour and a domain D defining the set of events which the process is initially prepared to engage.

The definition of the function f can be either a user defined process or a composition of CSP operators. In Section 4.3, we mentioned that JACK allows its users to either derive from a JACK process environment directly or to implement a behaviour interface and submit it to the default process implementation for execution. Figure 5.4 shows the relationship between a JACK process, a user defined behaviour, CSP operators and default process execution environment.

All JACK processes have a behaviour that defines their functionality (function f and domain D) and a unique name. The behaviour is used to define its specific functionality. The unique name is used by the process network and supervisor environment to control the execution status of processes. If the names are not provided by the user, a default one is given. For details on this modeling scheme, see the published UML model at [38].

5.2.2.2 CSP Operators Representation

The operator representation aims at making their use as close as possible to that in a CSP specification. For instance, the read prefix constructor of CSPm has a

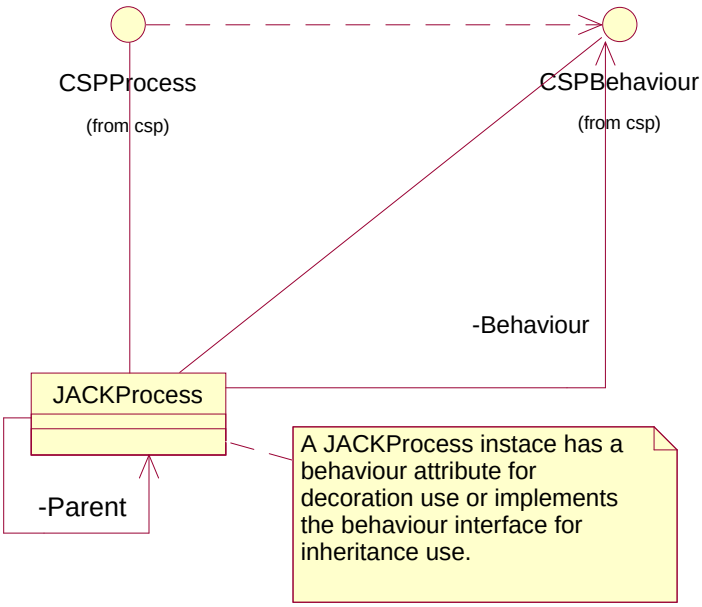

Figure 5.4: JACK Process x Behaviour Relationship

a) $ch?x \rightarrow STOP$

b) $N = ch?x : T \rightarrow P$

c) $N = b\&ch?x : T \rightarrow P$

Figure 5.5: Read Prefix Examples: a) Single, b) Complete, c) Extended

channel for input and a process to follow the channel communication. Optionally It also has a value set constraint over the type of the channel input. The operator can also be named for any desired purpose. Another well-known and widely used extended construction is a guarded prefix. In Figure 5.5, these three forms of the read prefix operator are presented.

Other operators are implemented as expected: interleaving, external choice, internal choice, conditional choice, and sequential composition have two argument processes; generalized parallel has two processes and one synchronization alphabet; and so forth. In Figure 5.6, a brief view the CSP operators dependencies and relationships is given.

5.2.2.3 User Process Representation

Users can define specialized behaviour to be attached to a process execution environment. In doing so, a user can mix canonical CSP specifications (i.e. using CSP operators, and auxiliary constructors) with any specialized data structure or exe-

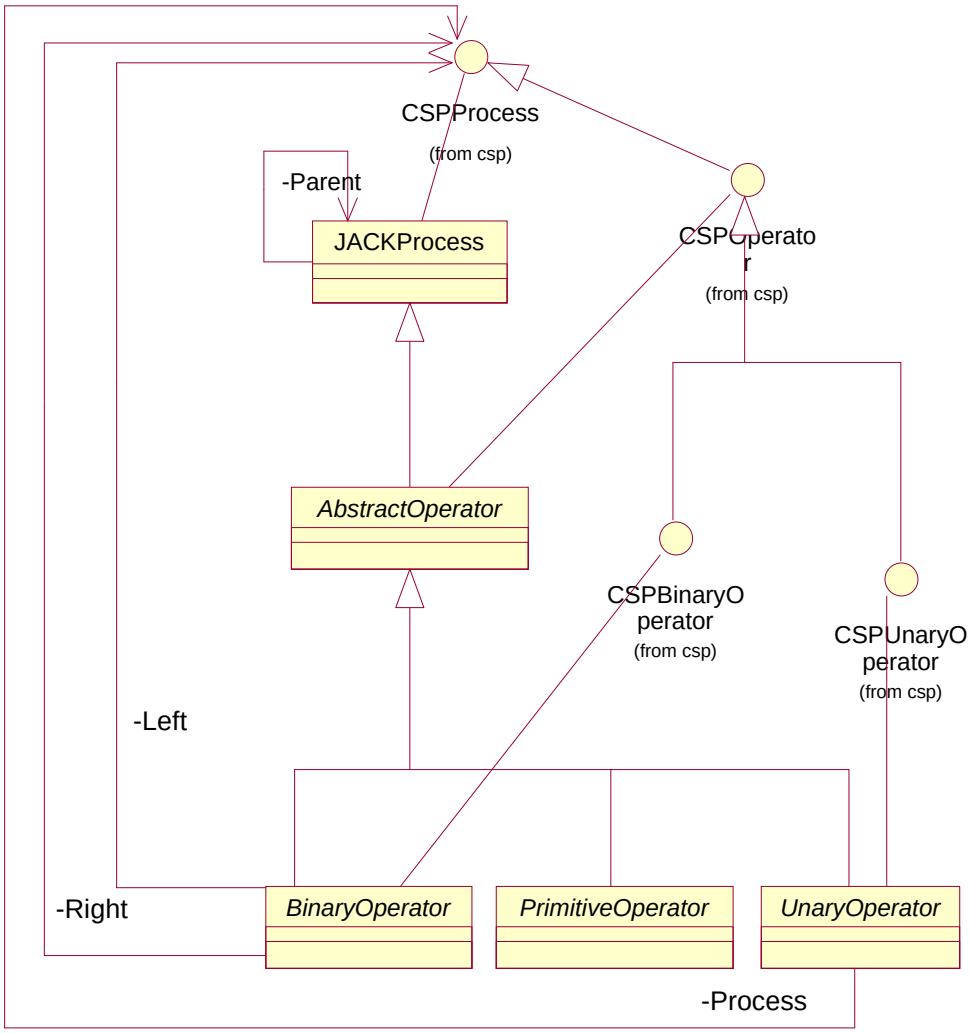

Figure 5.6: JACK CSP Operators Relationship Overview

```
channel a, b
P = a -> (P ||| b -> SKIP)
```

Figure 5.7: Process dynamic growing CSPm example.

cution behaviour. This kind of specification description was used in other libraries that implement occam [53], like CTJ [59, 50, 51, 49] and JCSP [107, 145, 144].

Since JACK needs to deal with CSP instead of occam, some complex problems arose, like backtrack (see Section 2.3.2) and multisynchronization (see Section 2.3.3). Thus, the user must be careful with the possibility of dynamic growing of the network while any process is running. Dynamic growing means processes that while running creates other processes.

This does not mean that the user process cannot grow dynamically. However, to properly build process networks that grow dynamically, the user must not forget to alter the domain D to reflect the dynamic growing pattern. It means that this dynamic growing must be taken into account, in order to correctly represent the domain D of possible initial events that the process can engage in. Thus, to avoid confusion in the proper correspondence of both method implementations, this kind of *on-the-fly* dynamic growing ought to be avoided.

The user behaviour interface provides a set up routine that can be used to instantiate all necessary processes before it starts running. This method can be used to create any necessary processes. With the use of this set up feature, it is easier to correctly implement both the desired process growing pattern and also the imperative correspondence between behaviour function f and domain D.

For instance, the process shown in Figure 5.7 can grow dynamically by instantiating the unnamed process $b \rightarrow SKIP$ on the fly as one of the interleaving operators. Examples of such usage can be found in Section 4.2. To avoid this, that unnamed process should be created before the execution. To do so, the user must provide a set up routine used for this creation purpose (see Section 4.1.1.2). The SKIP-termination rule [130] states that: when one side of the interleaving becomes SKIP, the whole processes could be viewed as the other execution side only (i.e. $P|||SKIP \equiv P$).

In Figure 5.8 a brief view of the user behaviour dependencies diagram are given; they are detailed in the next Chapter.

5.2.3 User Layer

The user layer is the topmost JACK layer and it acts as a placeholder; it is not an architectural entity, but just an entry point for the services of the framework exposed by the JACS semantic layer. It is not part of the functional framework. It is the entry point for JACK clients to start building their descriptions using JACK.

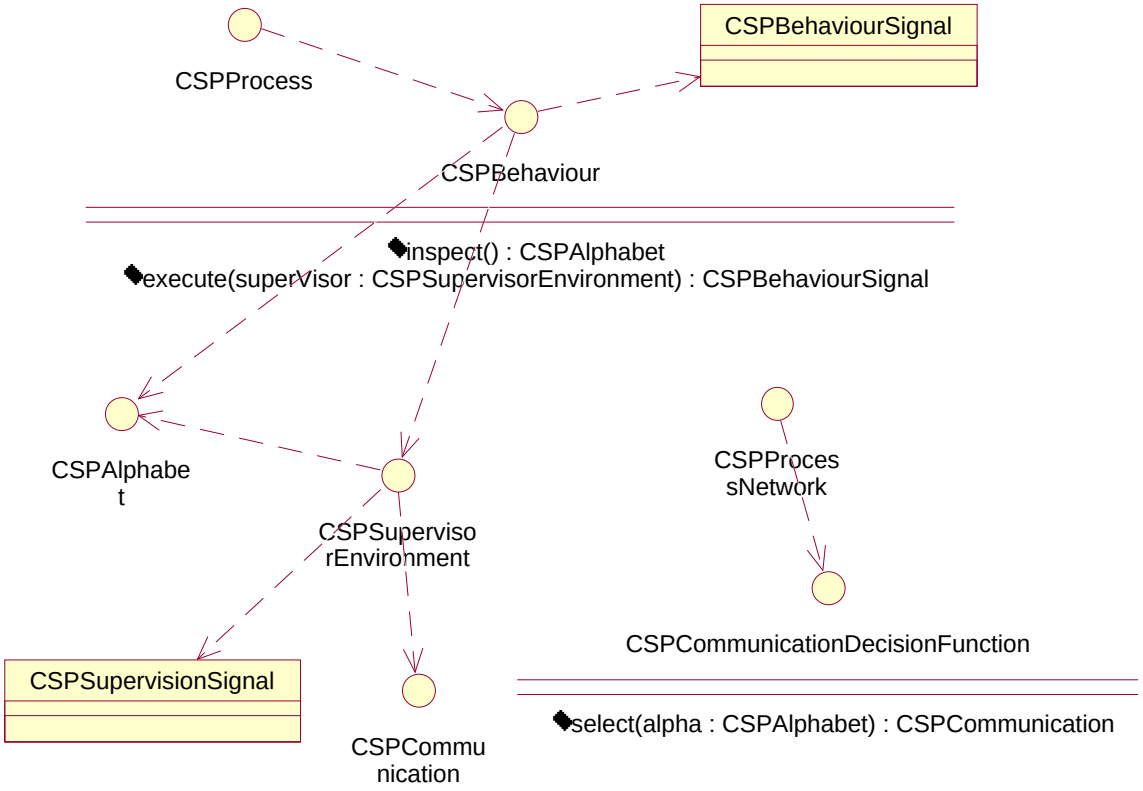

Figure 5.8: JACK User Process Behaviour Structure

JACK provides two examples of user layers; they are summarized below.

JCASE — Describes some simple case studies provided as Java classes that implements CSPm specifications.

JCAST — Describes an initial version of a CSPm interpreter based on the abstract syntax tree of CSPm described in [126]. The interpreter and the AST use the acyclic visitor [123, Chapter 7] design pattern. This opens the possibility of creating an interpreter for a pure CSPm specification as a final application for the framework. That specification can be translated directly to JACK with some possible necessary user interaction, as noted in a similar translation work [1]. Provided some well-known CSP parser [126, 1, 125] implementation is available, and given some adjustments to become compatible with the designed AST, JACK has a semi-complete specification running tool.

As already mentioned, JACK is a process-oriented framework. This means that it provides process functionality to the final user that uses it for any purpose. The stated purpose in this view of the framework is to describe some process specification, possibly in pure CSPm [33], or in CSP-OZ [31], or even in Circus [148, 149] specifications. For instance, JACK can be used by the final user to describe CSP related specifications (i.e. CSPm, CSP-OZ, Circus) written directly as a Java class. There is a work [12] that provides formal translation rules from CSP-OZ to CTJ.

5.3 Design Patterns Discussion

Some of the most important patterns at the user layer are the abstract syntax tree, that represents the CSP grammar following Watt guidelines [143, 142] to build interpreters; and the acyclic visitor [123], that makes the AST interpretation modular and easier to extended. With this modeling scheme, extensions of the grammar become straightforward. For instance, an extended example with the replicated version of some CSP operators was provided under the JAPS.CSP sub-layer (i.e. `jack.jacs.japs.csp.replicated` Java package).

The JACK framework architecture is based on specific, and well-suited design patterns, as much as possible. A thorough research was done for each selected pattern, basically following the guidelines of [121].

The use of design patterns has improved the quality of the final implementation. Some of the most important properties according to [90] are modularity, robustness, consistency, reusability, incremental development, simplicity, and so on. Sometimes, to achieve these properties, the design and code increase in complexity, but it is an acceptable trade off, since they continue relatively simple.

Generic patterns are implemented as common utilities. For instance, the event notification pattern [110] is used for detailed announcement of JDASCO execution

information. Without that resource, debugging and some inter-layer functionality the framework would be very difficult to implement and control.

Another important generic pattern is an enhanced version of the Double Check Lock pattern [23], that correctly implements the Singleton pattern [28] on concurrent execution environments using thread local storage. A diagnostic logger and a diagnostic message [102, 117] used for debugging and logging facilities are also provided. Finally, a set of template collections that extend the Java collections [54, 63] framework with template class assertions are implemented. These utilities are provided under the `jack.common` package. For a detailed description of these common patterns and utilities, see [38].

5.4 Final Considerations

In this chapter a description of some of the most important JACK architectural requirements are given. They are the layer and sub-layer descriptions, and the main design patterns used. The use of design patterns has shown to be very important in the design and implementation of complex functionality for an object-oriented framework. The JACK framework as a whole is based on design patterns.

As suggested extension operators, there are the labeling and functional renaming CSP operators [130]. The most important and expected infrastructure extension is a functional expression language support to deal with operations against values. For operator extensions one should also have in mind a probably necessary extension of the AST representation in the user layer (JCAST) above, since the AST represents the semantic structure of processes.

The next chapter presents a detailed description of the JACK layers implementation project.

Chapter 6

JACK Framework Implementation

This chapter presents the description of each relevant layer characteristic and layer composition of the JACK framework. The aim of this chapter is to detail each layer implementation decision and composition structure. In order to make the description more clear, for each layer entity a stepwise procedure to implement the `ExternalChoice` ($\Box$) CSP operator is shown.

Section 6.1 describes the JDASCO execution layer. Next, in Section 6.2, the JACS semantic layer and its integration with the JDASCO execution layer are presented. In Section 6.3, final considerations are given.

6.1 JDASCO — Execution Layer

As already presented in the last Chapter, JDASCO has three main participants (a JDASCO client, a concern mediator, and a functional object — see Figure 5.2) and three concerns (concurrency, synchronization, and recovery — see Figure 5.3). The next subsection presents the main JDASCO participants with some details related to their roles. After that, for each concern provided by JDASCO, a brief description of its execution phases available policies and management classes implementation are shown; for a detailed description of JDASCO separation concerns and composition of concerns see [43] and [133] for DASCO as well. Finally, a discussion about the selected JDASCO composition for JACK processes is presented.

The JACK user expected behaviour is to ask for some process to run. The process expected behaviour is to satisfy the CSP operator semantics or user defined behaviour.. These two expectancy need to be organized and distributed in the three main JDASCO roles defined above.

6.1.1 JDASCO Client

The JDASCO client is the entity that request the service under consideration. It ought not to be aware of any non-functional concern detail like concurrency or

synchronization. In this sense, it neither has to create, nor to manage an activity or lock related to the functional object functionality (i.e. CSP operators locks and threads). However, JACK clients should need to create and manege their own internal threads and locks. On the other hand, JDASCO abstracts and manages the concern control related to the functional object from its clients [133], but not the whole concern model. Thus, abstraction does not mean transparency.

On the other hand, JDASCO covers the concern, but not the concern model. This means that, at some points, JDASCO clients need to be aware of the concern model. For instance, the concurrency concern can be used in a synchronous or asynchronous way through the use of the future design pattern [23]. Moreover, it is the JDASCO client responsibility to correctly interact with futures in the sense of awareness of activity interruption. This breaks the modularity property a little. Another similar example occurs with the recovery concern, when the compensation operation policy is used, because the concern mediator must be aware of the functional object interface. These are topics discussed in following subsections.

The JACK entity that plays the role of a JDASCO client is a process. A process is a JDASCO client in the sense that is a process that needs to be synchronized, needs to execute concurrently, and needs to be recovered in some circumstances. Nevertheless, a process acts as a JDASCO client only when it is an active entity. It is an active entity when it runs independently (i.e. on its own thread of control) or when it is the root process. This occurs only for the parallel operator and for any other operator that has been the root process of the network.

In other words, a JACK client wants to start a process; to do so it need to directly interact with the them. Nevertheless, any kind of client never need to interact with JDASCO functional objects, thus a process acts, in this sense, as a special kind of JDASCO client provinding some way to be started.

Also note that, at some JDASCO points, the JDASCO clients must be aware of the concern model (these topics are mentioned in the next subsections). This means, for instance, that the JDASCO client must be aware of the concurrency model defining if it will be synchronous or asynchronous. Since JACK implements CSP, it surely has synchronous behaviour. This control must be hidden from the end JACK user, which again leads the JACK process to become the JDASCO client in order to properly hide the concern model awareness from end users.

6.1.2 JDASCO Functional Object

The functional object is the service provider under consideration. It ought to be implemented independently from any non-functional requirements, like multi-threading or synchronization locks. These non-functional requirements are handled and completely abstracted by the concern mediator. For instance, consider a bounded buffer with concurrent access, that has three methods: one to remove from the buffer, an-

other to write to the buffer, and a final one to inspect the content of the buffer head. It does not have to deal with critical regions in the underlying buffer data structure, nor it needs to take any recovery procedure if some corruption on the buffer occurs due to the concurrent access to it. These features ought to be carried out by the concern mediator, which in turn simplifies the functional object implementation leaving it to deal only with its functional requirements.

The concern mediator (called Interface in original DASCO) deals with non-functional requirements of framework execution. It is responsible for providing these non-functional services to functional objects and hide these same services from JDASCO clients. In this sense, it is the heart of the JDASCO infrastructure, providing the implementation of each available concern and their composition.

A JACK process also plays the role of a JDASCO functional object when it is performing its underlying semantic execution behaviour. This leads us to a very special case of JDASCO configuration where the functional object and the client is been implemented by parts of the same object: a JACK process. This organization is justified by the fact that functional objects expose some methods (functionality) that need to be mapped to the JDASCO management classes. These objects is responsible to take care about all concern issues of the functional object. Unfortunately, the behaviour of CSP operator semantics and user processes behaviour are to describe and control, concurrency, synchronization, and recovery issues. In this sense, the management classes implement parts of the operator semantics and the operators accessed by the JDASCO clients, that are by theirselfs other processes.

6.1.3 JDASCO Management Classes

In a detailed level there are, for each concern, three other entities: a first level policy manager, a second level policy manager, and structural management classes. These entities are mainly related with the concern mediator, but, at some points, they are also related to the functional object or the JDASCO client. Figure 6.1 illustrates this detailed scenario; it is directly related to Figure 5.2 shown in Section 5.2.1. This relation with boundary entities makes the implementation to loose some modularity, but this is necessary and it is an acceptable trade-off. We further discuss these detailed entities latter in this section.

The first level policy manager is responsible for providing concern configurations of the JDASCO client, concern mediator, and functional object interaction. The second level policy manager is responsible for taking a fine control over the execution path. The management classes are responsible for integrating the calling path between the JDASCO client and the functional object through the concern mediator. These classes also represent the entities to be implemented by the framework designer to expose the implemented functional object services to JDASCO clients through the concern mediator. In Section 6.2, these entities and the selected

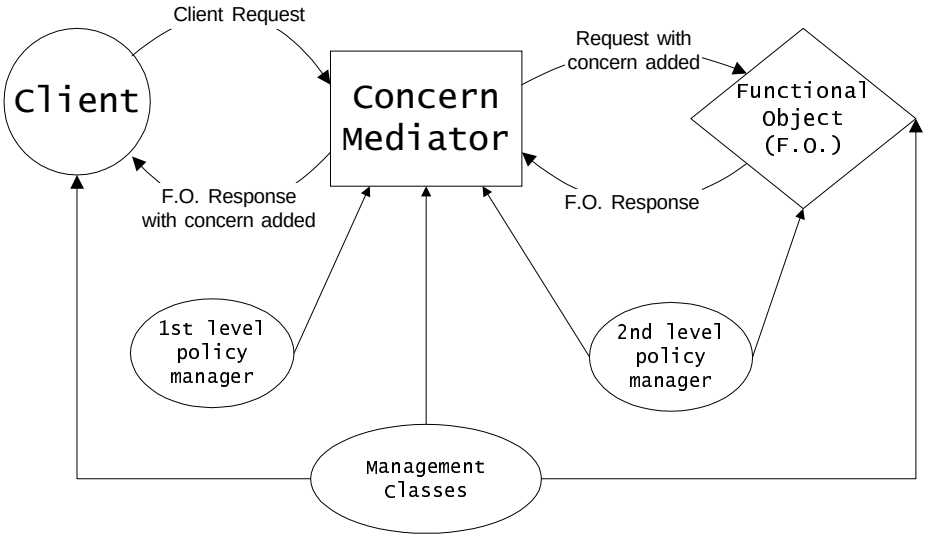

Figure 6.1: JDASCO Framework — Main Structure Detailed

configuration policies for integration with the semantic layer are presented.

For each concern, it is necessary to define specialized management classes to map the implementation of each CSP operator and user process. This leads us to a class explosion problem inherited from JDASCO. Luckly, since most management functionality is similar, and there is only one functional object method to be exposed (i.e. the method that runs a process), we can use inheritance and abstract base classes to simplify the implementation of the integrarion between JACK.JACS and JDASCO.

6.1.4 Separation of Concerns

"Separation of concerns is considered a key technique in software engineering. This technique consists of treating separately different aspects of program construction in order to control complexity, in accordance with a divide-and-conquer strategy. For instance, the separation between a program's functional and non-functional concerns allows the construction of an initial version supporting only its functionality in terms of expected input and output, and its later enrichment to support other aspects, such as performance, reliability and computational constraints. Problems handled by functional and non-functional concerns are specified, respectively, by a program's functional and non-functional requirements. The construction of concurrent programs is especially complex due mainly to the inherent non-determinism of their execution, which makes it difficult to repeat test scenarios." [133, Chapter 1]

JDASCO directly deals with three different concerns: concurrency, synchronization, and recovery. Each concern is organized as a different design pattern (i.e. Object Concurrency [133, Chapter 4], Object Synchronization [102, Chapter 8], and

Object Recovery [123, Chapter 15]) that deals with a specific problem domain.

Both patterns share the same structure detailed in Figure 6.1. They have management classes used to integrate functional object features with JDASCO concern mediator, concern phases that separates and establishes the roles that each concern must play, configuration policies divided in two levels of control for each concern, and some auxiliary entities used to control the concern execution. It is important to note that each concern encapsulates the concern policies, but not the concern model. For instance, JDASCO clients need to be aware of concurrency concern synchronous or asynchronous execution of functional object services, in order to control when they want to have their activity interrupted or not; this is done through future objects.

In what follows, we give a brief description of the most relevant topics for each concern. For a complete description about JDASCO concerns see [43] and [133] for DASCO.

6.1.4.1 Concurrency Concern

The concurrency concern deals with activity creation, management, and synchronous or asynchronous communication between them; this guarantees correct concurrent access to those services. An activity in JDASCO is equivalent to a specialized kind of Java `Thread` [4, 54, 63].

The management class related to concurrency concern is a method object. A method object class implementation contains the client invocation context. This involves the functional object method to be called (i.e. a functional object instance and a method of it) and the invocation arguments. For each functional object exposed method, there is a method object to implement the execution scheme for that method. The execution arguments must be given during the method object construction.

This concern has four execution phases: `CREATE`, `EXECUTE`, `FINISH`, and `DELETE`. A brief description of each one is given below.

1. {`CREATE`} — Responsible for instantiating the method object and future instances.

2. `EXECUTE` — Corresponds to method object execution.

3. {`FINISH`} — State where the method object notifies the corresponding future, if one exists, about method termination and returned values. With this notification scheme, JDASCO client objects synchronously waiting or asynchronously pooling on futures are notified.

4. {`DELETE`} — The method object and its underlying invocation context can be garbage collected.

The phases between braces indicate that they execute on the same shared synchronization lock session. This guarantees execution safety, minimizes lock acquisition, and increases execution performance. The `EXECUTE` phase must not be under any lock here, since the concurrency concern is related to method execution and not method synchronization; this execution synchronization is carried out by the synchronization concern. Therefore, when composing concerns, the synchronization concern must deals with this.

The concurrency concern configuration first level policies define how the concern mediator executes the client call and the functional object respective service. There are concurrency generators for configuring and controlling these policies.. The second level policies are captured by future instances, in order to allow JDASCO clients to control the underlying concurrency model either synchronously or asynchronously. Below, the description of each kind of concurrent policy follows.

1. First level — Implemented and managed by concurrency generators

   `NON_CONCURRENT` — JDASCO client calls are started on the calling activity, which means serial execution. This policy is implemented by the mediator by just sequencing the user call to the functional object through the client thread.

   `SEQUENTIAL` — JDASCO client calls are queued for execution on a single background activity, leaving the client activity free to proceed. This policy is implemented by the generator using a separate thread that queues each client call; in this way, the client thread is free to continue and its request will be under processing by the background queue thread.

   `CONCURRENT` — For all JDASCO client calls, a separate activity gets created to run the method. This policy is implemented by the generator that creates one new thread for each client call. In this way, we get the highest concurrency level. Figure 6.2 shows the concurrent generator policy. This is the selected policy used in JACK because it has better execution performance and is adequate to implement a concurrent language. Nevertheless, the `NON_CONCURRENT` policy is used in early stages of the development process due to its adequacy for debugging purposes.

2. Second level — Implemented and mangaged by future objects

   Synchronous — JDASCO client activity is interrupted while the mediator is running the request. Since CSP is synchronous this is the selected policy for JACK.

   Asynchronous — JDASCO client activity is not interrupted, it ought to pool the future object in order to inspect request termination. Concern mediator ought to notify that future about termination.

JDASCO clients must be aware of the second level policies only; it represents the concurrency model that the client can follow. It means that the second level policy configuration will depending on how clients interact with future objects; if they pool future objects for results, clients are asynchronous, and they are synchronous if wait on future objects.

6.1.4.2 Synchronization Concern

The synchronization concern deals with locks and execution flow control. The concern mediator adds these non-functional requirements to the service exposed by the functional object in order to guarantee safety access (i.e. avoid resource sharing conflicts due to critical regions contention) to those services.

The management classes related to synchronization concern are the synchronization predicate and the synchronization data. For each functional object exposed method, there is a synchronization predicate and data pair to implement the synchronization scheme for that method.

A synchronization predicate class represents the information exchanged and the control of the execution of a JDASCO invocation to a functional object method. The synchronization data responsibility is to provide and centralize (possibly duplicating, or referencing) functional object synchronization information, in order to abstract domain dependent information.

Bloom [9] states that a synchronization scheme ought to provide six types of synchronization information support: invocation type; invocation state; invocation arguments; state of pending predicates; state of executing predicates; state of functional object; and invocation history. The synchronization predicate interface provides the first four information types and synchronization data provides the last two. For a complete and detailed discussion about this topic see [133, Chapter 2 p.23; and Chapter 5 p.85] and [43].

The synchronization concern has five execution phases: `CREATE`, `PRE-CONTROL`, `EXECUTE`, `POST-CONTROL`, and `DELETE`. A brief description of each one is given below. Figure 6.3 shows synchronization predicate and synchronization data execution sequence with respect to each synchronization phase.

1. {`CREATE`} — Phase responsible for creating the synchronization predicate and initialise the predicate queue, in order to allow a centralized queue of predicates used for investigation during their execution.

2. {`PRE-CONTROL`} — Phase responsible for synchronizing a functional object invocation before its method execution occurs, in order to guarantee safe access to it. The phase is sub-divided in two parts: the first one inspects if the functional object state is prepared for synchronization; the second one inspects if the executing predicate queue is prepared for synchronization.

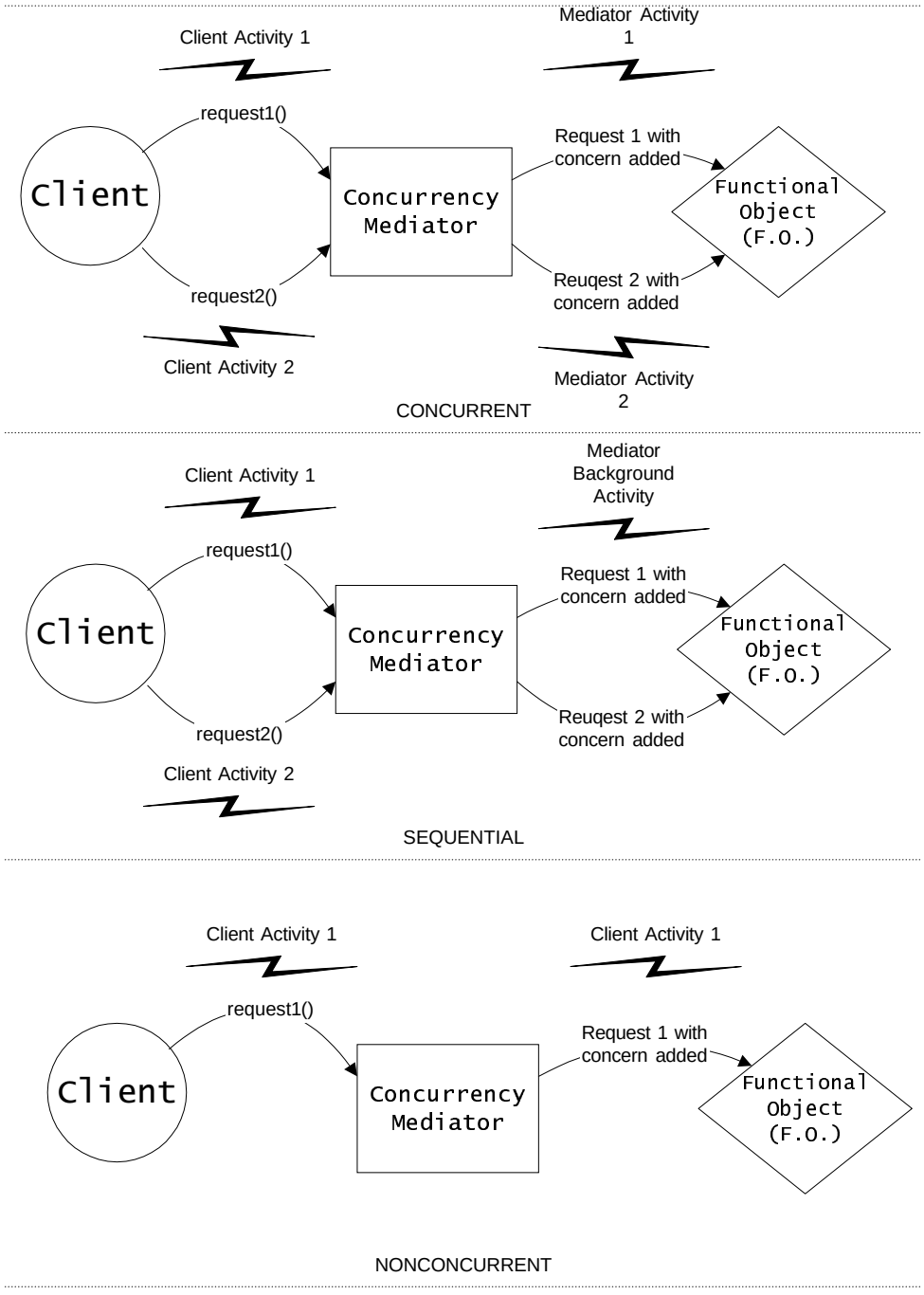

Figure 6.2: Generator Policies — CONCURRENT

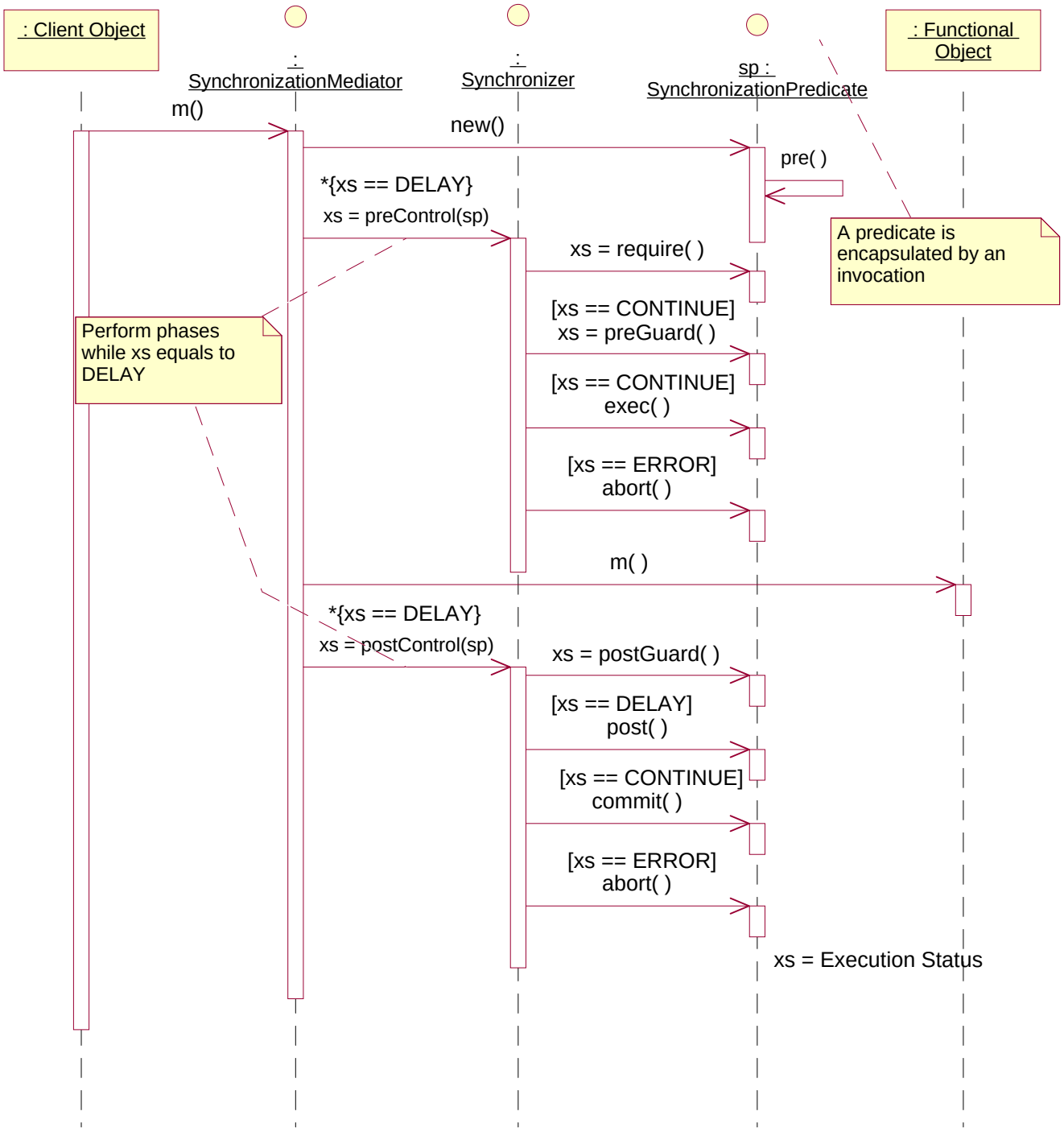

Figure 6.3: Synchronization Concern Generic Collaborations

3. `EXECUTE` — Phase responsible for notifying the underlying functional object method its execution has granted permission and it is allowed to proceed.

4. {`POST-CONTROL`} — Phase responsible for synchronizing a functional object invocation after its method execution occurs. It can post the execution, if it must be delayed due to the need to wait for a synchronization lock; abort it, if some synchronization error occurs; or leave it to finish successfully.

5. {`DELETE`} — The synchronization predicate and its underlying invocation are signalled to be garbage collected.

The synchronization concern first level policies define how the concern mediator synchronizes the JDASCO client call and the functional object respective service. The second level policies are captured by how synchronization predicates and synchronization data interacts with the functional object. A brief description of each kind of synchronization policy follows.

1. First level — Implemented by synchronizers

   `PESSIMISTIC` — Allows synchronization predicate execution only when all necessary locks were acquired. This policy is best suited for high contention lock environments.

   `OPTIMISTIC` — Always allows execution of synchronization predicate execution. It aborts if contention for any lock occurs in the middle of the execution. This policy is better suited for low contention lock environments.

   `GENERIC` — A mix of the two other policies varying according to the user circumstances.

2. Second level — Implemented by synchronization predicates and synchronization data

   Readers/Writers — Synchronizes the JDASCO client request based on the current state of the running synchronization predicate requests inspected through the predicate invocation queue.

   Producer/Consumer — Synchronizes the JDASCO client request based on the current state of the functional object synchronization data information.

   Dynamic Priority — This policy is similar to Readers/Writers, but with priority information associated with all invocations; it is useful, for instance, to avoid reading starvation.

History Sensitive — This is a newly created policy to be used by JACS, not defined in [133]. It is a combination of Readers/Writers with dynamic priority, and an execution history with the current executing functional object. The complete finite state machine is stored in order to open the possibility to inspect the execution state of other concerns, while doing the synchronization of predicates. Details about this policy can be found in [38, 40, 43].

These two level policies were created in order to satisfy the already mentioned six Bloom's synchronization requirements stated in [9].

6.1.4.3 Recovery Concern

The recovery concern deals with functional object recoverability using transaction sessions, with commitment and abortion ability; this guarantees recoverable transaction support to those services.

The recovery concern has a recoverable object point management class. For all functional object methods that are exposed, there is a single recoverable object point that abstracts the whole recovery procedure using one of the second level policies presented below. A recoverable object point class represents the recovery procedure to be done on the underlying functional object that it handles. This involves functional object preparation for execution, commitment of execution results, or abortion on execution failure.

The recovery concern has five phases: `CREATE`, `PREPARE`, `EXECUTE`, `FINISH`, and `DELETE`. A brief description of each one follows.

1. {`CREATE`} — Phase responsible for creating the recovery object instance.
2. {`PREPARE`} — Phase responsible for associating a recovery object instance with a functional object, obtaining the functional object where the invocation must take place; that is a functional object prepared for recovery.
3. `EXECUTE` — In this phase the functional object returned by the `PREPARE` phase gets executed.
4. {`FINISH`} — This phase is responsible for either committing or aborting the invocation execution.
5. {`DELETE`} — The recovery object instance is signalled to be garbage collected.

The recovery concern first level policies define how the concern mediator recovers the JDASCO client calls against the functional object respective service. Some restrictions where applied for recovery policies when composing concerns as defined in [133, 146]. The second level policies are captured by recoverable object points

to control how JDASCO clients manages the underlying recovery model. In what follows, a brief description of each kind of recovery policy is given.

1. First level — Implemented by recoverers

   `UPDATE_IN_PLACE` — Always has the most recent changes, which makes it simpler to commit executions. It is complex to abort (from an implementation perspective) due to recovery operations to be performed after abortion.

   `DEFERRED_UPDATE` — Always reflects a consistent state storing uncommitted changes, which makes it suitable to abort executions, because it can simply set the uncommitted object as the current one. It is also suitable to operation commitment since it simply needs to update the uncommitted changes.

   `BACKTRACK` — Specialization of `DEFERRED_UPDATE` to deal with backtracking. It interacts with the supervisor environment to properly implement this feature [40].

2. Second level — Implemented by recovery object points

   Compensating Operations — The functional object must provide methods that compensate the recovery operations of commitment and abortion; this can break modularity.

   Object Copying — The recovery object point makes a copy of the functional object at the preparation stage of the recovery procedure.

These recovery policies are directly related to backtrack implementation.

6.1.5 Composition of Concerns

JACK is interested only in the concurrent synchronized recoverable concern composition, since it is the required composition to use JDASCO for implementing JACK processes. Thus, this is the only composition implemented. Nevertheless, it is interesting to show which new role each concern composition rises; this is done in what follows. An initial prototype version of JDASCO implements all composition possibilities with respective examples of their use [37].

This is the selected composition policy due to the nature and prerequisites of our framework implementation. The concurrency concern is necessary for dealing with processes running concurrently, a nature aspect of CSP. The synchronization concern is necessary for dealing with the multisynchronization problem and to implement the parallel operator. The recovery concern is necessary for dealing with the backtracking recovery procedure implementation.

6.1.5.1 Synchronization and Concurrency

When composing concerns some additional policies and management classes arise. The composition of synchronization with concurrency adds the necessity of a concurrency context at the synchronization concern. This context is responsible for controlling lock management during concurrent accesses of JDASCO activities in order to avoid busy looping for lock acquisition. The concern activity is suspended until some other activity wakes it up, due to the relinquish of the shared synchronization lock at the synchronization `POST-CONTROL` phase. Depending on the selected synchronization policy, a different concurrency context policy is automatically selected and used.

When composing concurrency with synchronization, method objects need to notify futures about synchronization execution errors. On the other hand, concurrency opens the possibility of only creating and associating activities before or after the synchronization `PRE-CONTROL` phase lock acquisition, in order to allow better resource usage. This is called the association policy. There are two association policies for concurrency and synchronization: `IMMEDIATE` and `LAZY`. They are captured by the `Associator` interface at the JDASCO composition sub-layer and are described below. Figure 6.4 shows the JDASCO composition layer structure.

`IMMEDIATE` — This policy creates and associates an activity for execution before the synchronization `PRE-CONTROL` phase, which can block the activity, wasting resources in high contention environments. This policy can waste resources, but it increases concurrency.

`LAZY` — This policy creates and associates an activity for execution after the synchronization `PRE-CONTROL` synchronization phase, which can avoid unnecessary activity creation and locking acquisition, which saves machine resources. This policy can save resources, but it decreases concurrency.

The best choice of association policies is determined by the underlying execution environment pattern. There is a restriction about the `LAZY` association policy. It can be used if, and only if, the first level concurrency policy was `CONCURRENT`.

6.1.5.2 Recovery and Concurrency

The composition of recovery and concurrency adds the necessity of method objects to allow recoverers to set their underlying functional object before execution at the recovery PREPARE phase. Thus, a recoverable method object instance must always be used in JACK.

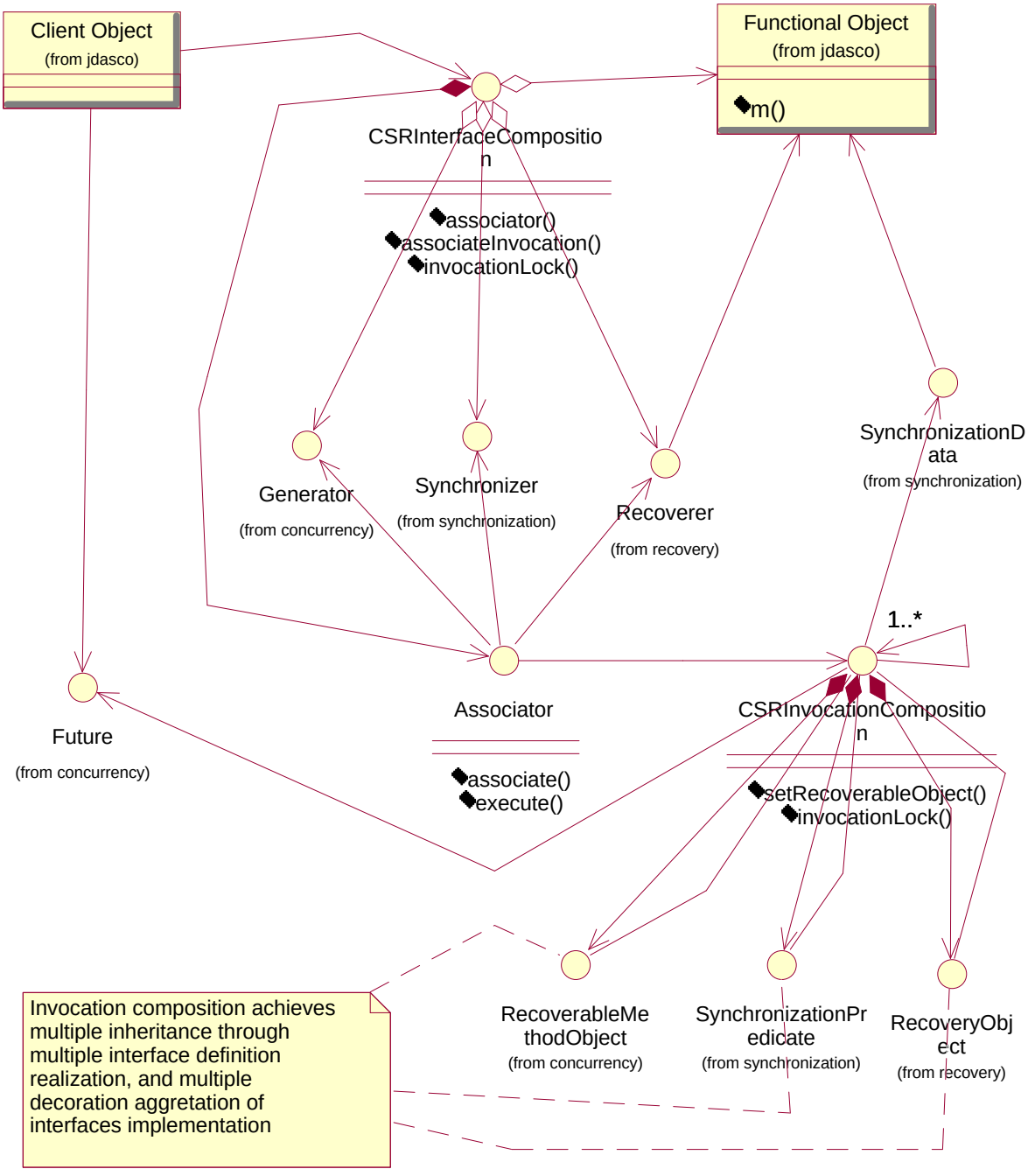

Figure 6.4: JDASCO Composition Layer Structure

6.1.5.3 Synchronization and Recovery

The composition of synchronization and recovery has some restrictions [146] on the policies adopted. For instance it is not possible to compose synchronized `OPTIMISTIC` producer/consumer policies with `DEFFERED_UPDATE` recovery policy. This is a theoric restriction mentioned in [133, Chapter 7] and proved in [146]. For a detailed discussion about this composition details, see [133, Chapter 7].

6.1.5.4 Composition of Concerns Phases

The composition of concerns also composes the phases of each concern. The composed version of phases is divided as follows (a collaborations showing the result of the concern composition is shown in Figure 6.5):

1. {`CREATE`} — Composition of the `CREATE` phase of each concern. This means that these phases, at the composition layer, occur atomically. It also initialises composition entities like the `Associator` and the `ConcurrencyContext`.

2. {`PRE-CONTROL;PREPARE`} — Composition of the synchronization `PRE-CONTROL` and the recovery `PREPARE` phases. The functional object preparation for execution can only occur when the synchronization lock is acquired and passed in the `PRE-CONTROL` guard methods (i.e. `require()` and `preGuard()`). When the `PREPARE` phase finishes, the lock is relinquished.

3. `EXECUTE` — Composition of the `EXECUTE` phase of each concern, which occur atomically.

4. {`POST-CONTROL;FINISH`} — Composition of the synchronization `POST-CONTROL` and the recovery `FINISH` phases. The functional object execution commitment or abortion can only occur when the synchronization lock is acquired and passed in the `POST-CONTROL` guard method (i.e. `postGuard()`).

5. {`DELETE`} — Atomic composition of the `DELETE` phase of each concern.

The finite state machine of the concern composition is directly related to each concern finite state machine. It is available in [38].

6.2 JACS — Semantic Layer

The implementation project of the composition layer of JDASCO and its integration with the JACPS semantic layer is a significant part of this dissertation work. It models the processes semantic behaviour to be used by the JACK user layer. In the following subsections, we have a detailed discussion about how JACS implements each role stated for the JDASCO integration procedure. As already mentioned in Section 5.2.2, Figure 6.6 shows the sub-layers related to the following subsections.

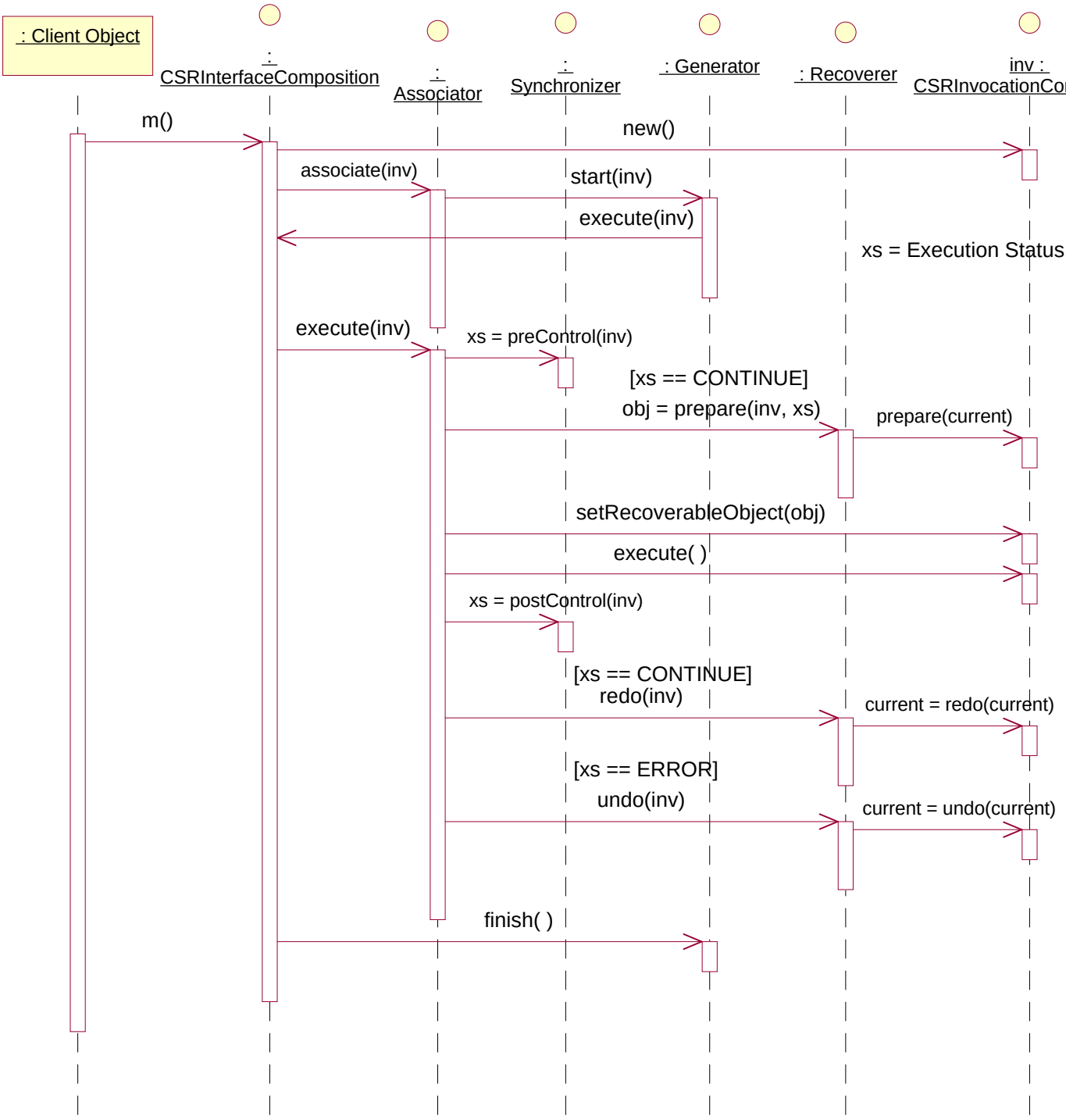

Figure 6.5: Composition of Concerns Collaborations

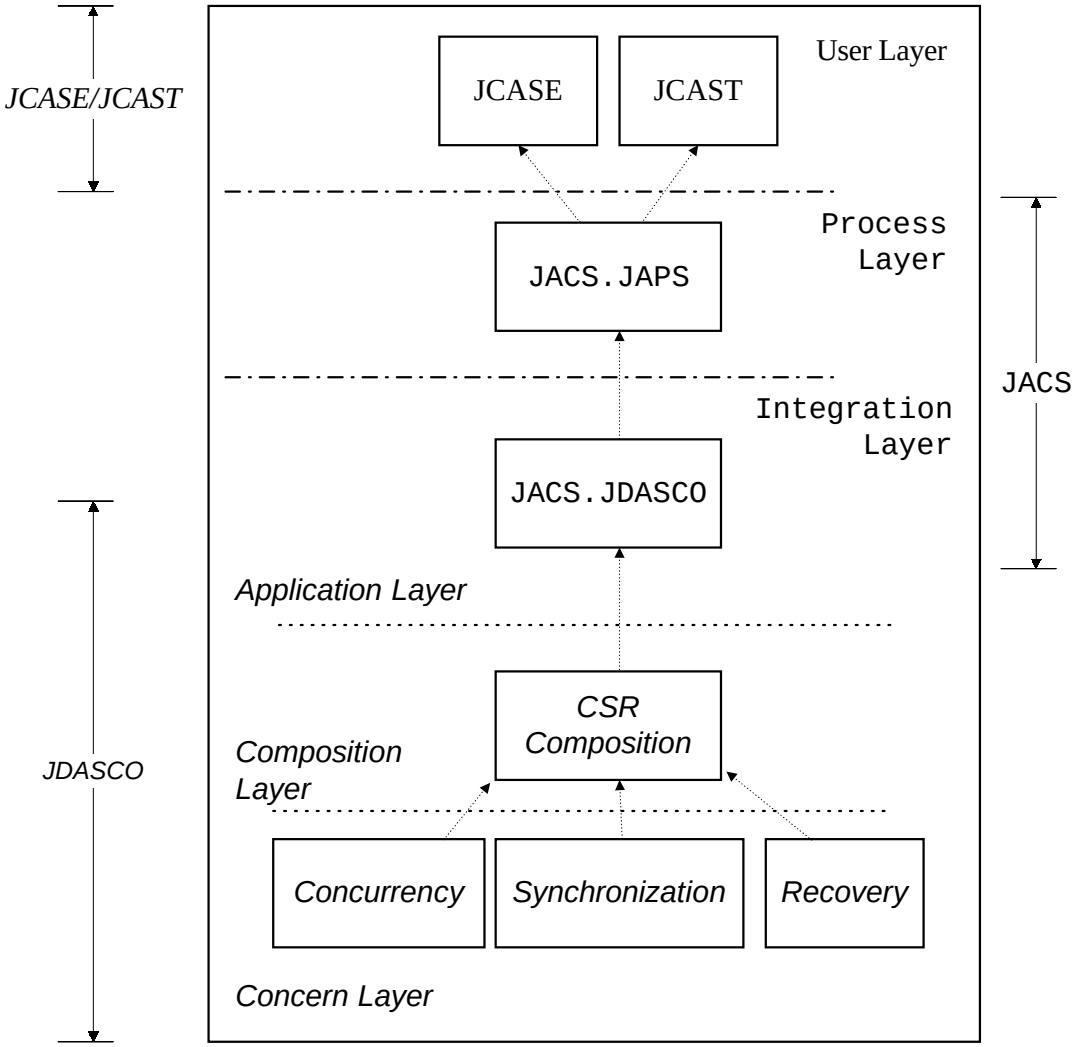

Figure 6.6: JACS and JDASCO sub-layers integration

6.2.1 An Example — `ExternalChoice` Implementation

Now that we have already seen how JDASCO can be configured, let us take a look at parts of the implementation of one CSP operator, in order to make the description of the implementation more clear. Some aspects of the implementation, like backtracking combinatorial choice selection, is hidden here due to its inherited complexity To understand this topic, more details about JDASCO is necessary; this is provided in [43]. In what follows, one can identify the stated JDASCO roles played by the operator and the semantic layer roles mentioned in following subsections. With JDASCO roles, the execution of any operator or user process is fragmented in many implementation classes and protocol interfaces.

6.2.1.1 `ExternalChoice` as JDASCO Client

The `ExternalChoice`, all other operators, and all user processes have the JDASCO client implementation abstracted by their base class, the `JACKProcess` class. The implementation is provided by the `CSPProdcess.start()` method. Details about this topic can be found in Section 6.2.3.5.

6.2.1.2 `ExternalChoice` as JDASCO Functional Object

The only functional object method exposed is the `CSPBehaviour.execute()` behaviour method. This method defines the behaviour of the process when some communication has been selected by the supervisor environment. Thus, for each user process or CSP operator, a specific `execute()` method implementation must be provided.

The `ExternalChoice` operator operational semantics states that it must selects and starts the appropriated process, according with the environment desired communication. A special procedeure must be carried out when both operands of the external choice can perform the given desired communication; in this case, an internal (non-deterministic) selection must be done. In what follows, the code of the `execute()` method implementation of the `ExternalChoice` operator is given.

```
public class ExternalChoice extends AbstractChoice implements CSPBacktrackable {

    ...

    private static final int CHOICE_ERROR = 0;
    private static final int RIGHT_CHOICE = 1;
    private static final int LEFT_CHOICE = 2;
    private static final int NONDETERMINISTIC_CHOICE = 3;

    private static final int FALSE_OPTION = 0;
    private static final int TRUE_OPTION = 1;
```

```
    private static final int[][] SELECTION_MATRIX = {
      // Rows=leftContains;Cols=rightContains
      // False , True
      /*False*/{ CHOICE_ERROR, RIGHT_CHOICE }, // False
      /*True */{ LEFT_CHOICE , NONDETERMINISTIC_CHOICE }, // True
      // False , True
    };

    public CSPBehaviourSignal execute(CSPSupervisorEnvironment sve) {
      CSPProcess l, r;
      l = left();
      r = right();
      CSPCommunication comm = sve.selectedCommunication();
      intlContains = (childInspectedAlphabet(l).containsCommunication(comm) ?
        TRUE_OPTION : FALSE_OPTION);
      int rContains = (childInspectedAlphabet(r).containsCommunication(comm) ?
        TRUE_OPTION : FALSE_OPTION);

      switch (SELECTION_MATRIX[lContains][rContains]) {
        default:
        caseCHOICE_ERROR:
            return B_SUPERVISOR_EVENT_SELECTION_ERROR;
        case RIGHT_CHOICE:
            notifySelect(r);
            r.start(sve);
            break;
        case LEFT_CHOICE:
            notifySelect(l);
            l.start(sve);
            break;
        case NONDETERMINISTIC_CHOICE:
        //comm → P□ comm → Q  ⇒   comm → (P ⊓ Q) by law □ —step
            notifyInternalCommunication();//notify TAU
            select().start(sve);//the select method will notify selection...
            break;
      }
      return B_EXECUTE_SUCCESSFULLY;
    }
    ...
}
```

Initially, it retrieves the left and right side operands, and the environment selected communication. Then, it checks if the selected communication is inside in one of the retrieved operands; these information was previously stored in a call to method `inspect()` by the supervisor environment in the execution process. After that, it checks the returned value against the selection matrix to infer the proper execution condition, either a successfull selection, a selection error, or a non-deterministic

selection. Next, it notifies the process selection, in order to generate the appropriated event, and starts the selected process to run returning the behaviour signal respectively related to the execution condition.

The code snipet shown above represents the normal execution of the semantics of an `ExternalChoice` operator. The backtrack procedure is carried out by layer integration points at the JDASCO.JACS integration sublayer and it execution and instantiation are managed by the supervisor environment. These are topics detailed in [40].

6.2.1.3 `ExternalChoice` Related JDASCO Management Classes

`ExternalChoice` Method Object

JDASCO states that, for each functional object exposed method there must have a respective method object implementation. In this sense, for each `execute()` method of each CSP operator and user process, a specific implementation of method object must be done. Fortunetely, the procedure is very simple and can abstracted to become generic for all CSP operators and user process.

There is only one method object implementation called `JACKProcessRMO`. It receives the execution context and parameter at construction and just call the `execute()` method of the process under consideration for it. The result of the execution call is a behaviour signal. This signal is sent to the `TemplateFuture` instance held by the method object, in order to inform the JDASCO client about the execution status, in the case the `ExternalChoice` operator acting as the JDASCO client.

`ExternalChoice` Synchronization Predicate

As occured with the method object, the synchronization predicate for the `ExternalChoice` operator is also abstracted by another JACK class called `JACKProcessSP`. This generic predicate controls the execution flow between JACK.JACS process sub-layer, JACK.JACS integration sub-layer, and JDASCO application layer. The interaction between the supervisor environment and its pupil process are also managed here. This is the most complex JDASCO generic role played by all JACK processes.

`ExternalChoice` Recovery Object

JDASCO states that only one recovery object implementation is necessary for all functional object managed by it. This is acceptable since it abstracts the recovery protocol and not the physical recovery operation. In this sense, it acts as a

mediator [28] between the functional object and the recovery concern mediator.

JACK uses the compensation operations second level policy due to its intrinsic necessity to check the occurrance and perform the backtracking algorithm. Thus, object copying is not adequate, moreover it is simpler than compensation operations through Java clonning support. The generic recovery object implementation is called `JACKProcessRO` and it interacts directly with the supervisor environment and the underlying managed process, in order to properly implement the backtracking algorithm.

6.2.2 JAPS — Processes Sub-Layer

The processes behaviour is modeled by the JAPS process semantics sub-layer. It implements user processes, CSP operators, and the JACK type system. The processes semantics implementation is divided in two parts: the static part, which provides process execution information, like their possible communication paths; and the dynamic part, which provides physical execution guidance, like activities (threads) to be created, locks that need to be acquired, or processes stack to be recovered (or backtracked). These two parts are treated in JAPS (process sub-layer), and in JACS.JDASCO (integration sub-layer) respectively (see Figure 6.6). In this sense, there are two levels of processes execution: one at the JAPS sub-layer, that prepares the execution environment providing any necessary information (i.e. processes operands, relationed alphabets, etc.); and another at the JACS.JDASCO sub-layer, that provides the physical execution of the process network using that information (i.e. JDASCO management classes).

Processes representation in operating systems [139] executes in two level: *user* mode and *kernel* mode. The JACK processes execution levels can be viewed using this operating systems process execution analogy. The JAPS process semantics sub-layer represents the process *user* mode of execution, and the JACS.JDASCO integration sub-layer described below represents the process *kernel* mode of execution.

An important feature about processes implementation is the way they are submitted to execute by JDASCO. The JDASCO configuration (i.e. policies and management classes) for JACK processes expects some awareness about the concurrency and synchronization models. For instance, the JDASCO client of the concurrency model must be aware of futures and their possible interruption.

In this sense, it is interesting, though not an obligation, that a JACK process implementation consider these JDASCO execution and composition requirements. On the other hand, JACK processes are implemented following Scattergood's [126, Chapters 4 and 8], Schneider's [130, Chapters 1, 2, and 3], and Hoare's [61, Chapter 1, pp. 38] operational semantics description and guidelines. These guidelines strongly suggest the use of well-defined semantic entities, in order to properly im-

plement the compositional behaviour of processes.

Hoare [61, Chapter 1 pp. 38—39] states that to properly implement a process just a function f describing the process behaviour, and a domain D defining the set of events in which the process is initially prepared to engage, are necessary (see Section 4.1.1). The other references establishes two semantic functions to be defined: one that represents the same domain D mentioned by Hoare, called `initials()`, and one that represents a list of CSP operators after the occurrence of some communication event called $after$.

Practical experiments has shown that the Hoare's approach to implement processes is more adequate to operationally implement generic processes (i.e. CSP operator processes and user defined processes) than the `initials` and `after()` semantic functions defined by Scattergood [126]. The Scattergood's approach is more adequate to operationally implement CSP operators only, since they do not consider user defined processes in their semantic functions definition.

Despite this fact, the `initials()` semantic function behaviour is similar to the `CSPBehaviour.inspect()` method that represents the Hoare's domain D. The operational semantic rules described in [130, Chapter 1, 2, and 3] are similar to the `CSPBehaviour.execute()` method, that represents the Hoare's function f. The `CSPBehaviour` interface is mentioned in Sections 4.3.1, 4.1.1.2, and 6.2.2.4.

JDASCO needs some guidelines in order to function properly as noted in Section 6.1. The function f and the domain D are these guidelines. The possible different paths returned by domain D act as compensating operations for the recovery concern, as execution flow controller for the synchronization concern, and establish the concurrent behaviour pattern for the concurrency concern.

The implementation of the `CSPBehaviour` interface for CSP operators are built following a compositional approach. So, each CSP operator implementation solves its corresponding semantic role in the execution flow and then forwards the job to its operands. In this way, all CSP operators are semantically defined. The interface signature is given in Figure 6.7; its relationship with a process is shown in Figure 5.4. The relationship of processes and CSP operators is shown in Figure 5.6.

Processes are entities that can run. To run, a process needs an execution environment and a behaviour pattern. The execution environment can be abstracted and acts as the static part of the process execution. On the other hand, a behaviour pattern must be defined for each process specification. In this way, a JACK process must have an attached behaviour pattern. Operators are processes with an already defined and implemented behaviour pattern. User processes, however, are processes that lacks a predefined behaviour pattern, which means that they need to be provided by the JACK client when instantiating the JACK process, in order to completely define the process execution semantics.

With this kind of process modeling, some interesting properties and implementation benefits arise. Since the execution environment scenario of a process is indepen-

```
package jack.jacs.japs.csp;

/**
 * This interface represents the user defined behaviour for user processes.
 *
 * @pattern
 * @author Leonardo Freitas ¡ljsf@cin.ufpe.br¿
 * @version 0.1 31aug2001
 * @since JDK1.2.2
 */
public interface CSPBehaviour extends CSPBehaviourSignalFactory {

    public CSPAlphabet inspect();
    public CSPBehaviourSignal execute(CSPSupervisorEnvironment superVisor);
}
```

Figure 6.7: JACK `CSPBehaviour` Interface Signature

dent of whether it is a user process or a CSP operator, we can generalize this part of the process execution. On the other hand, the behaviour pattern of processes is the dynamic part of its implementation: for each kind of process, there is a possible different execution behaviour to be implemented.

These two parts are reflected by the division of a generic JACK process execution environment, represented by the `JACKProcess` class implementation of the `CSPProcess` interface (see Section 4.1.1.1); and a specific JACK process behaviour pattern, represented by some implementation of one of the `CSPBehaviour` interfaces (see Sections 4.3.1 and 4.1.1.2). The former is provided as a default environment implementation by the JAPS sub-layer and the latter needs to be defined by the JACK client defining user processes. The CSP operators behaviour patterns however, have a default implementation at the JAPS sub-layer. Since there is a separation of service interfaces and service implementation, framework extensions or the completely provision of different implementations is facilitated.

In what follows, we present some of the main JAPS roles to be used by JACK user layer and the integration with JDASCO. In Figure 6.9, the main interaction between JACS and JDASCO is shown. The figure shows the JACK synchronization predicate (`JACKProcessSP`) been called at the `PRE-CONTROL` phase. Then, in Figure 6.10, the default inspection procedure for process communication selection is shown; it occurs when the supervisor environment calls the `inspect()` method of the inspection solver (see Figure 6.11). After that, in Figure 6.8, the relationship between these integration entities is given in a class diagram.

The relationship between these entities and how they interact with JDASCO concerns [43] are described in detail in a draft document [40] available at [38]. This

document explains the meaning and structure of a supervision frame. It is important to clearly understand JDASCO internals [43] before looking at this reference. Nevertheless, it is the heart of the solution of most important problems like backtracking (see Section 2.3.2) an multisynchronization (see Section 2.3.3).

6.2.2.1 Process Supervisor Environment — `CSPSupervisorEnvironment` Interface

The process supervisor environment performs the role of the JACK client user selecting the communication path to be followed. For instance, it represents the human interaction with a cash machine, which means the user selecting some buttons, in order to properly perform the desired action. The semantics of the involved processes guarantee the expected machine behaviour, forbidding, for instance, withdrawing money before login validation. In Figure 6.11, we present parts of the interface signature of the supervisor environment: in the on-line documentation [38], one can found the complete interface signature with helpful documentation about each one.

Each process execution must be under control of at least one supervisor environment. The environment is responsible for controlling vital process execution operations. These operations were firstly studied in an action semantics [97] description of the operational semantics of CSPm [35, 82]. They are presented below.

JACK Integration — responsible for integrating the semantic layer of JACK (JACS) with the execution layer (JDASCO).

The environment fills the gap between these two layers. It holds a reference to a process network instance, which in turn controls the JDASCO composition layer through an instance of a concurrency synchronized recoverable composed concern mediator.

In this way, the supervisor controls the way the semantic layer interacts with the execution layer. In order to properly integrate these layers, the JDASCO management classes must be aware of the supervision operations.

Communication Selection — selects the immediate available events to communicate.

When a process starts running and enters in *kernel* mode, its execution at the JDASCO synchronization concern asks the supervisor to determine communications that must be selected for this execution round. The supervisor then, forwards the selection request inspecting its pupil processes through the `CSPProcess.inspect()` method about possible immediate available events to engage. At this point, some trivial backtracking cases are prematurely avoided, like prefix guards pre-condition. It must be mentioned that, semantically, prefixes that fail to pass in their guard pre-conditions are "readyless". Nevertheless, operationally, at JDASCO low-level, this situation is treated the same

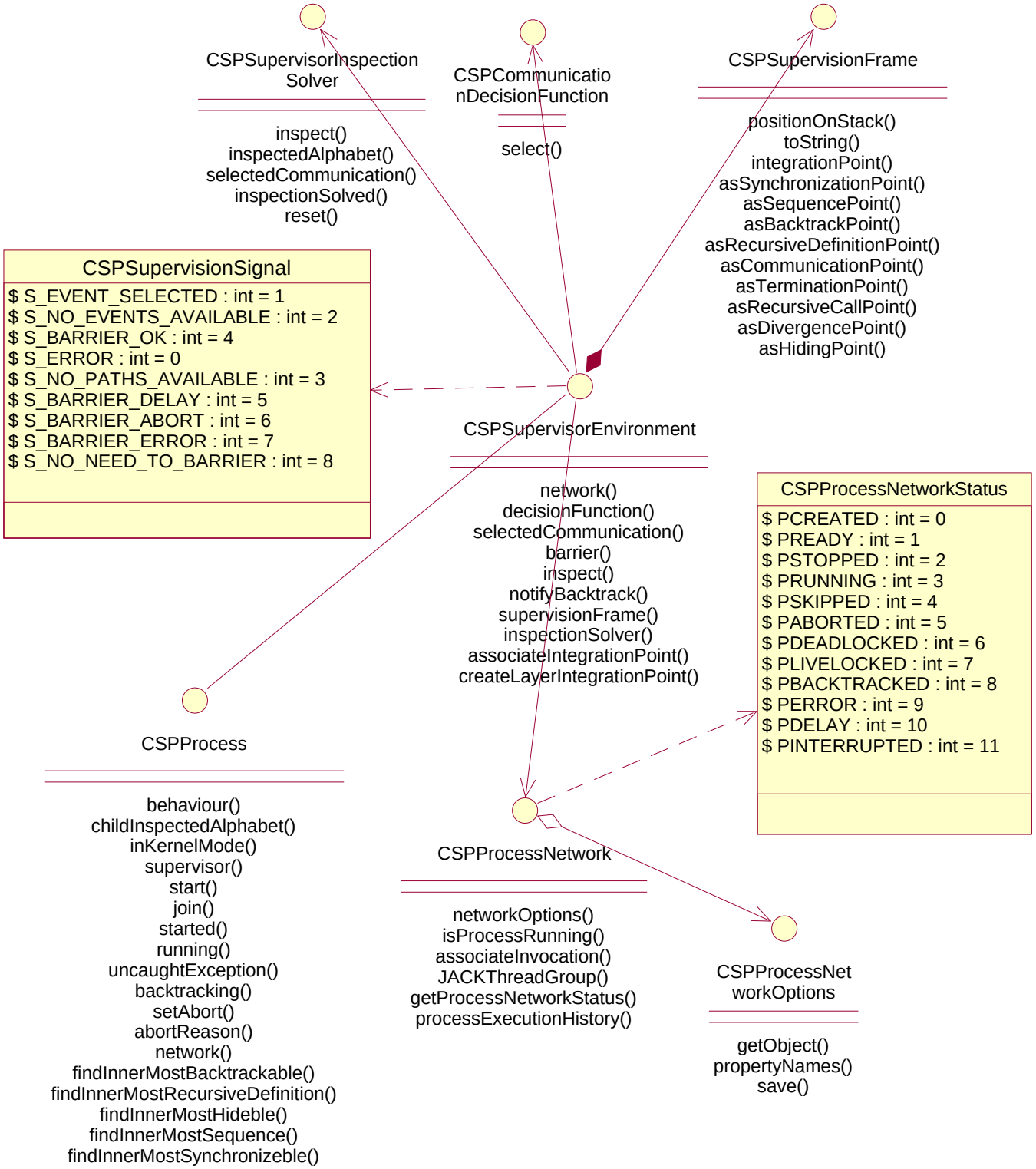

Figure 6.8: JACS Integration Entities Structure

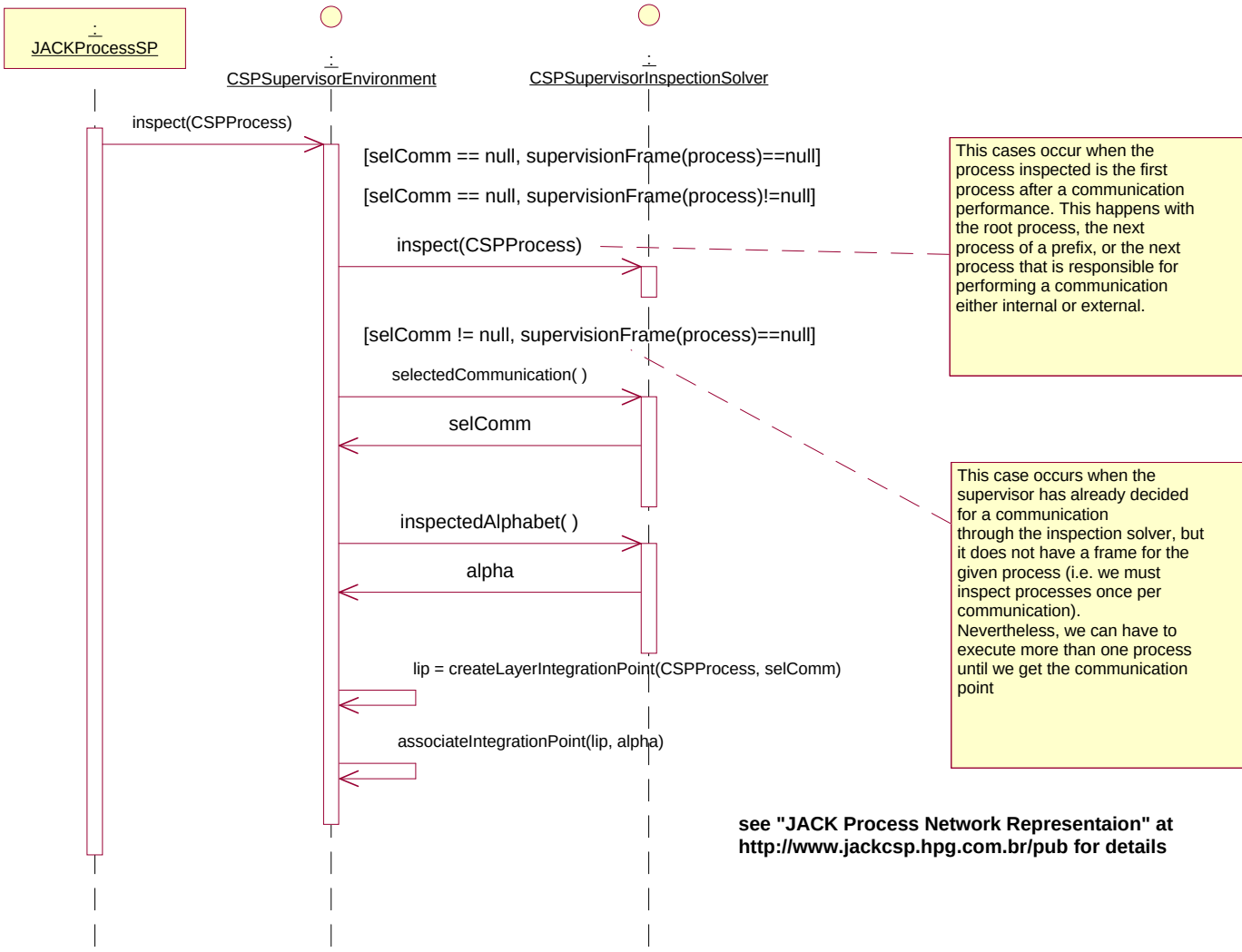

Figure 6.9: Interaction between JACS and JDASCO — an Overview

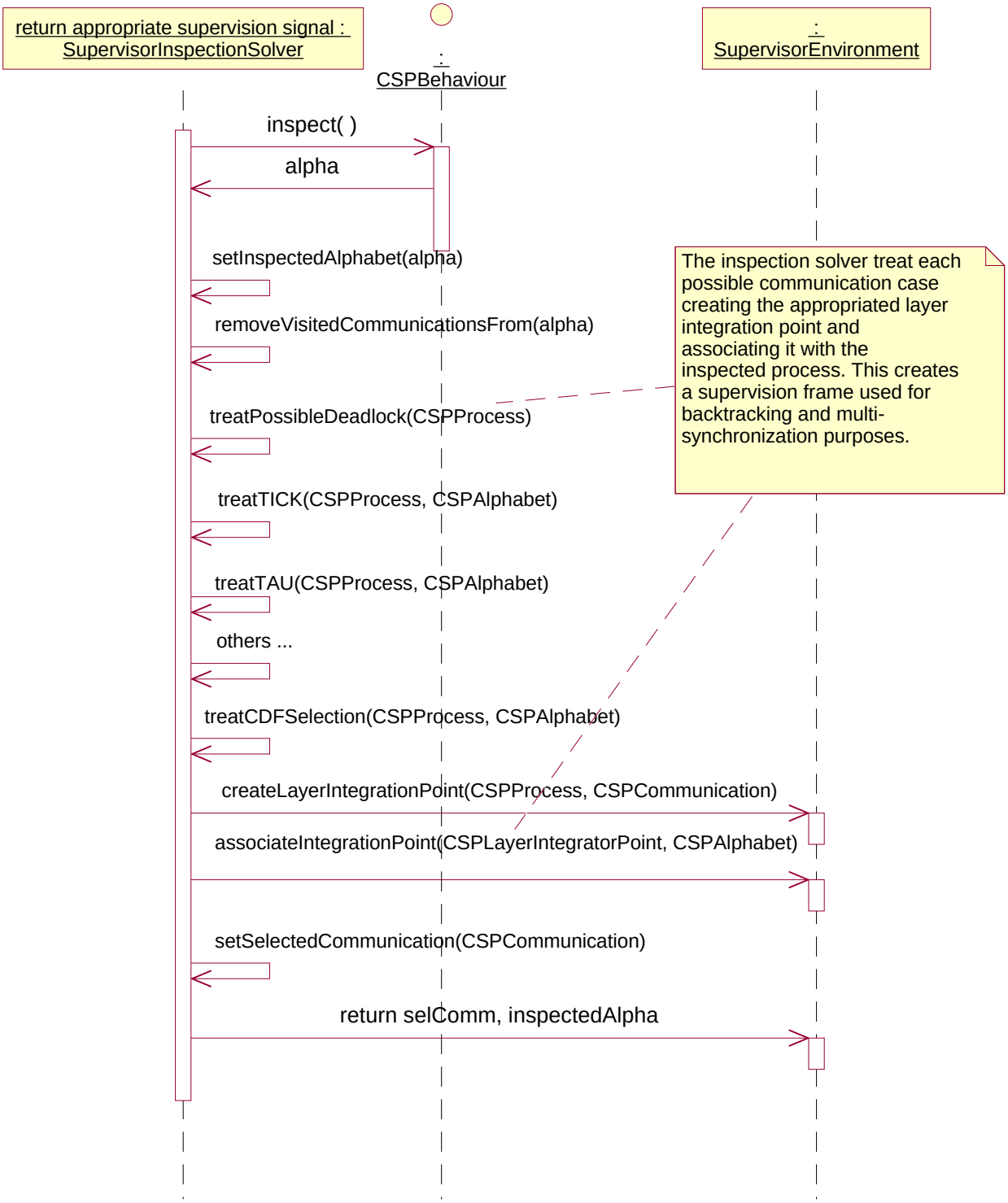

Figure 6.10: Interaction between JACS and JDASCO — an Overview

```
package jack.jacs.japs.csp;

import ...;

public interface CSPSupervisorEnvironment extends HumanReadable {
    // Java related
    public String asString();
    public boolean equals(Object o);

    // Access methods
    public long id();
    public CSPProcessNetwork network();
    public CSPCommunicationDecisionFunction decisionFunction();
    public void setDecisionFunction(CSPCommunicationDecisionFunction cdf)
        throws NullPointerException;

    // Pupil Processes related methods
    public void addPupil(CSPProcess process);
    public void removePupil(CSPProcess process);
    public Iterator pupils();
    // Supervision related methods
    public boolean communicationSelected();
    public CSPCommunication selectedCommunication();
    public void setSelectedCommunication(CSPCommunication comm);
    public void clearSelectedCommunication();
    public CSPAlphabet controllerAlphabet(CSPProcess process);
    public CSPSupervisionSignal barrier(boolean preBarrier, CSPProcess process);
    public CSPSupervisionSignal inspect(CSPProcess process);
    public CSPProcess notifyBacktrack(CSPProcess process);
    public void notifyProcessExecutionFinished(CSPProcess process, CSPBehaviourSignal executionSignal);
    public void notifyJDASCOConcurrentExecutionFinished(CSPProcess process, CSPBehaviourSignal
        executionSignal, ExecutionStatus xStatus);

    // Debugging related methods
    public void dump() throws IOException;//to STDOUT!
    public void dumpTo(Writer w) throws IOException;

    // Layer integration methods
    public Iterator supervisionHistory(CSPProcess process);
    public CSPSupervisionFrame supervisionFrame(CSPProcess process);
    public boolean hasSupervisionFrame(CSPProcess process);
    public CSPSupervisorInspectionSolver inspectionSolver();
    public void associateIntegrationPoint(CSPLayerIntegratorPoint lip, CSPAlphabet normalizedAlpha);
    public CSPLayerIntegratorPoint createLayerIntegrationPoint(CSPProcess process,
        CSPCommunication selComm);
}
```

Figure 6.11: JACK `CSPSupervisorEnvironment` Interface Signature

way as a "normal" backtracking; they are distinguished only by some internal signals. For details on backtracking solution and JDASCO low-level signaling with respect to integration with JACS see [40].

The default supervisor implementation provides a Strategy [28, pp. 315] design pattern to make the final event selection, in order to increase modularity and generalize the implementation. The pupil process that has requested execution permission is inspected and the returned communication alphabet is submitted to a (strategy) communication decision function interface (`CSPCommunicationDecisionFunction`). This interface is responsible for selecting a communication from the resultant inspected alphabet (i.e. select an event from the process `inspect()`ed alphabet). The default supervisor implementation already provides a default random communication selection. In this way, if the JACK client needs to deal with a special kind of event selection it just needs to install a new decision function for the given supervisor environment.

Backtrack Path Decision — performs the combinatorial choice of possible backtrack paths when a process has been aborted. After that, it can inform its pupil process about another paths to follow or notify the deadlock situation to the JDASCO composition layer to take the appropriated recovery decision.

Since the supervisor environment controls the JDASCO composition layer instance, it can inspect the shared process execution history stack, in order to properly select the possible backtrack paths to follow. This stack is filled at the recovery concern `PREPARE` phase. The filling procedure is detailed in [43, Section 1.3.4] and in Section 6.2.3.3.

The supervisor marks all selected paths in order to properly make the combinatorial selection. Depending on the returned result, it informs the JDASCO composition layer, through a supervision signal, to perform the appropriate reaction due to a backtrack condition.

Multisynchronization Barrier — funnels the pupil processes involved in a multisynchronization session into a multiway rendezvous [13, 131].

At the synchronization concern of JDASCO composition layer, the management classes that encapsulate a pupil process of a supervisor environment must inform it about its execution status. With this notification procedure and the supervisor knowledge of its pupils, it is possible to properly implement the multiway rendezvous and thus the multisynchronization between JACK processes.

The supervised pupil queue is filled just before a JACK process enters in *kernel mode* (see Section 6.2.3.5 and [38] for details).

The JACK client is free to build and provide its own supervisor environment implementation, but we do not recommend this decision unless it was really imperative. The JACK process subsystem already provides a default supervisor environment implementation, in order to make it easy to use for the JACK client. If one needs to implement its own supervisor version, be advised about their complexity inherited from the set of operations that the environment process must properly perform.

Most of these operations results are encapsulated in a `CSPSupervisionSignal`. These signals are used at the JDASCO composition layer by the concern management classes, in order to take the appropriate decision with respect to the given signal. The default JACK management classes for both CSP operators and user defined processes already take care of this signalling protocol, thus the JACK client ought to never deal with these signals. On the other hand, JACK clients that provide their own JDASCO management classes versions must be aware of the supervisor signalling protocol, that is completely described in the companion JavaDoc documentation available in [38] and in [40].

In this way, there is a cooperative relationship between the process supervisor environment and the JDASCO management classes. Therefore, the supervisor environment is the ultimate one responsible for integrating the JACS semantic layer and the JDASCO execution layer.

Special Supervision Discussion

The default process supervisor environment implementation performs all necessary operations to properly combine JACK layers. Nevertheless, there is a special supervision case with respect to the `Hide` ($P \setminus X$) CSP operator.

The semantics of this operator specifies that the events under execution that are inside the hidden alphabet (X) must be hidden, thus the external environment can never see them. In this sense, operators that are under it in the process network must be supervised by a special environment that deals with this special hidden event selection semantics. This is easily done through the use of a decorator [28, pp. 175] supervisor that controls hidden events only forwarding any other actions to the default supervisor environment.

Despite this fact, a network can have as many supervisors as necessary. This is normally restricted to be only a single default supervisor environment with any special hide supervisor decorator when that operator appears in the network. However, processes with multiple supervisors are allowed. For instance, this scheme can be used for a layered event selection, where each layer provides a default supervisor environment implementation with a different communication decision function strategy; or some more sophisticated schemes with different supervisor implementations.

```
package jack.jacs.japs.csp;

import ...;

public interface CSPProcessNetwork extends HumanReadable {

    // Network User Configuration Options
    public CSPProcessNetworkOptions networkOptions();

    // JDASCO integration
    public void interrupt();
    public boolean isInterrupted();
    public void clearInterrupted();
    public boolean tryToInterruptOnDeadlock();
    public boolean setTryToInterruptOnDeadlock(boolean value);
    public boolean isInitialized();
    public boolean isProcessRunning(CSPProcess process);
    public boolean isProcessRunning(String processName);
    public boolean anyProcessRunning();
    public void associateInvocation(CSPProcess userProcess, RecoverableMethodObject rmo,
        SynchronizationPredicate sp, RecoveryObject ro, Object userDefined);
    public Object networkLock();
    public ThreadGroup JACKThreadGroup();
    public CSPProcessNetworkStatus getProcessNetworkStatus();

    // Environment information - views
    public CSPCommunicationEnvironment tracer();
    public CSPProcessExecutionHistory processExecutionHistory();
    public void setProcessNetworkStatus(CSPProcessNetworkStatus status);
    public void removeProcess(CSPProcess process) throws CSPProcessException;
}
```

Figure 6.12: JACK `CSPProcessNetwork` Interface Signature

6.2.2.2 Process Network — `CSPProcessNetwork` Interface

The process network acts as the bridge between the JDASCO composition layer and the JACS.JDASCO integration sub-layer. It plays the role of the concern mediator; or more accurately, it holds and is responsible for controlling a reference of concurrent synchronized recoverable concern mediator implementation provided by the JDASCO composition layer. In Figure 6.12, we present the interface signature of the process network.

The process network can also act as an information center for executing processes. It does not control processes, they execute under the supervision of the supervisor environment. During execution, the supervisor environment can request information to the process network, in order to make the appropriate decision about which

communication path to follow or which backtrack decision to take, based on the network information. In this sense, it is an important pool of information. The network provides information like: execution history, communication traces, current running processes, aborted processes, committed processes, controlled access to the underlying JDASCO composition interface for low-level operations, and so forth.

The process network also has a set of configuration properties, called process network options, that may be set by the JACK client through the process network options file. These properties are JDASCO policy adjustment parameters, default CSP operator settings, debugging facilities, and so on.

Each process network can have a different process network configuration option. Configuration files can also be defined to be loaded by the process network configuration options, which makes the execution environment very flexible. However, both the process network and the process network configuration options have a default implementation automatically incorporated by the supervisor environment. In this way, the JACK client does not need to know nor to deal with these classes. Nevertheless, these classes can be instantiated and specialized to be used by JACK processes, like CSP operators (i.e. the `Hiding` — $P \setminus X$ — operator) and possible (very specialized and advanced) user processes. For instance, on the `jack.jcase.csp` case studies package (see class `GeneralTestCSP.java`), one can found a specialized version of process network configuration options usage; with this example, the execution environment shows to be very productive and flexible.

6.2.2.3 Process — `CSPProcess` Interface

The available implementation of the `CSPProcess` interface is the basic class called `JACKProcess`. An already noted important feature (see Section 4.1), is the ability of the JACK client to build its own processes via inheritance, deriving their classes from the `JACKProcess` class; or via decoration [28, pp. 175], implementing the behaviour pattern interface and submitting this interface to a `JACKProcess` class instance to run. This goal is achieved using the same implementation pattern of the Java `Thread` class and `Runnable` interface [63, Chapter 1]. Figure 6.13 shows some parts of the `CSPProcess` interface signature.

JACK processes already implement the behaviour pattern interface (`CSPBehaviour`) providing a no-op implementation (see Figure 5.4), and also has a behaviour pattern mandatory attribute given at process construction. With this, if the user decides for the inheritance approach, he just needs to override the behaviour pattern no-op methods, which is natural. If the user decides for the decoration approach, he just needs to implement the behaviour pattern interface methods and submit this implementation as the behaviour pattern mandatory parameter of the `JACKProcess` instance, which is again natural. These two approaches can be found on case studies in packages `jack.jcase.csp` and `jack.jcase.sawp`.

```
public interface CSPProcess extends HumanReadable, Cloneable, Comparable {
    // Process Graph/Representation related methods
    public long level();
    public CSPProcess parent();
    public CSPBehaviour behaviour();
    public Iterator children(boolean recursive);
    public CSPAlphabet childInspectedAlphabet(CSPProcess child);
    public void setChildInspectedAlphabet(CSPProcess child, CSPAlphabet alpha);
    public boolean hasChild(CSPProcess child, boolean recursive);
    public boolean hasChildren();

    // Process state methods (see Figure 4.1)
    public boolean inKernelMode();
    public boolean started();
    public boolean running();
    public boolean finished();
    public boolean deadlocked();
    public boolean isAnActiveObject();
    public boolean aborted();
    public boolean backtracking();

    // Process Supervision related methods
    public CSPProcessNetwork network();
    public CSPSupervisorEnvironment supervisor();
    public Reason abortReason();
    public void setAbort(Reason reason);
    public void uncaughtException(Thread t, Throwable e);

    // Methods that the user ought to call
    public void start(CSPSupervisorEnvironment sve) throws IllegalProcessStateException;
    public void join();

    // Process Network graph searching facilities
    public CSPProcess findInnerMostBacktrackable();
    public CSPProcess findInnerMostRecursiveDefinition();
    public CSPProcess findInnerMostHideble(CSPCommunication forComm);
    public CSPProcess findInnerMostSequence(CSPCommunication seqOn);
    public CSPProcess findInnerMostSynchronizeble(CSPCommunication forComm);

    // Denotational Semantics Information that might be calculated
    public List after(CSPCommunication comm);
    public CSPTraces tracesModel();
    public CSPFailures failuresModel();
    public CSPDivergencies divergenciesModel();
    public CSPFailuresDivergencies failuresDivergenciesModel();
}
```

Figure 6.13: JACK `CSPProcess` Interface Signature

The process interface also provides some useful operations to be used by JACK clients. Some of them are briefly described below.

Cloning Support — Processes that were not running can be cloned. The cloning procedure is deep in the process network graph; that is, process children are also cloned, in order to properly represent the links between processes in the cloned graph. More details about process network graph and cloning behaviour can be found in [40].

Low-level Tracing — Low-level tracing is a special implementation of the diagnostic logger [123, Chapter 16] pattern language. It allows users to track all kinds of process signals, from CSP event communications to supervisor signals or hidden events.

Searching Support — A process network can be searched either upward or downward with some filtering or configuration options.

For a complete list, see the `CSPProcess` interface companion JavaDoc [38].

Users can define their own `CSPProcess` implementation, but they ought to not do so. This might be done only in the case of a major implementation improvement, since the process implementation must deal with many pieces of functionality like JDASCO integration, supervision relationship, etc. For instance, a distributed version of processes execution implementation can be provided as a possible future work.

6.2.2.4 Process Behaviour Pattern — `CSPBehaviour` Interfaces

The normal behaviour pattern interface of processes has two methods to be implemented, they are shown in Figure 6.7; they are described below.

`CSPAlphabet inspect()`

This method represents the domain D of possible immediately initial events to engage, mentioned by Hoare in [61, pp. 38]. Users that just make a composition of CSP operators as their own process behaviour pattern can easily define this method calling the related operators respective `inspect()` methods and use the `CSPAlphabet` set operations to properly combine the results. If one needs to represent specialized processes, a `CSPAlphabet` domain (D) must be properly built in order to correctly implement the intended semantics. For instance, at an early stage of the development, we build alphabets that represent some process network immediately available events on-the-fly to test JDASCO integration without using semantically defined operators. This allows the incremental development (see Section 5.1) of processes. Some examples of the method usage are given in Chapter 4 and [41, 38].

`CSPBehaviourSignal execute(CSPSupervisorEnvironment superVisor)`

The method represents the function f that describes the process behaviour, mentioned by Hoare in [61, pp. 38]. For instance, a user can make a **switch()** over some condition and follow a specific path according to that condition; or he can provide a specialized routine that uses some complex data structure that represents the process intended semantics.

The process related to this behaviour is under the given supervisor parameter. This parameter is needed in order to decouple the behaviour implementation about the awareness of a supervisor instance. If the JACK client inherits its implementation class from `JACKProcess` to override this method, the given `superVisor` argument must be equal to the one returned by the `CSPProcess.superVisor()` method.

The JACK client must be aware that this method must conform with the returned alphabet of the `inspect()` method, in order to allow the supervisor to proper decide the correct paths to follow. If a user provides an `execute()` method implementation that is not uniform (i.e. the function f must executes only under the domain D) with respect to an `inspect()` call, the final process behaviour is unpredictable.

The method must return a signal informing how does its execution occurred. Currently, there are six possible signals. This set of signals can grow as necessary but the first two are the most common and the only ones that the JACK client must be aware of. The behaviour signal can have an attached `Reason` object. JDASCO composition layer uses this signalling protocol in order to properly inform the process supervisor about the process execution status. These behaviour signals are briefly explained below. For more details about behaviour signalling protocol, see companion JavaDoc [38].

1. `B_EXECUTE_SUCCESSFULLY` — JACK clients must return this signal when its execution has finished successfully. For instance, all CSP operators behaviour implementation returns this signal when their execution finishes successfully.
2. `B_EXECUTION_ERROR` — JACK clients must return this signal when its behaviour execution has finished unsuccessfully. They should normally attach a `Reason` object defining the reason of the error. For instance, all CSP operators behaviour implementation returns this signal when their execution were aborted due to some `Reason`.
3. `B_SUPERVISOR_EVENT_SELECTION_ERROR` — JACK clients ought to return this signal if the supervisor has selected an invalid event. It may occur only if the `inspect()` method implementation was not properly defined

```
package jack.jacs.japs.csp;

public interface CSPExtendedBehaviour extends CSPBehaviour {

    public void setUp(CSPSupervisorEnvironment sve);
    public void finish(CSPSupervisorEnvironment sve, CSPBehaviourSignal signal);
}
```

Figure 6.14: JACK `CSPExtendedBehaviour` Interface Signature

or if the communication decision function strategy used by the supervisor was selecting an invalid event (i.e. an event outside the selection domain). For instance, the CSP `ExternalChoice` (□) operator returns this signal if the `desiredCommunication()` of the supervisor was neither on its left nor on its right side communication alphabets.

4. `B_IO_ERROR` — JACK clients ought to return this signal when its behaviour execution has performed some I/O error. I/O in this context means either a channel data link or value serializer I/O errors (see Chapter 4 for details about data links and serializers). Currently, only CSP prefix operators returns this signal.
5. `B_BACKTRACKING` — JACK clients ought to never need to return this signal, unless it is defining a very specialized user process that needs backtracking. Users must be aware of the backtracking algorithm implemented by the given supervisor environment in order to properly uses this signal. Currently only CSP operators that deals with backtrack return it. This is an advanced topic related to supervision of processes; it is mentioned in [40].
6. `B_NOT_YET_EXECUTED` — Represents a no-op signal. It acts like a unknown initial value. The JDSACO method object starts its underlying behaviour with this value. Currently any JACK process returns it.

The framework provides more specialized extended behaviour interfaces. For instance, there is the `CSPComparableBehaviour` interface, that can be used to compare different process behaviours following some user defined comparison pattern; or the `CSPExtendedBehaviour` interface, that can be used to "listen" process communications during its execution, which can be very useful for debugging purposes or for visual tools development. The extended behaviour pattern interface signature is given in Figure 6.14, and its main methods are described below.

void `setUp(CSPSupervisorEnvironment sve)`

This method gets called by the `CSPProcess` implementation just before the process enters in *kernel* mode. It is called every time a process is started. This is a topic detailed in Section 6.2.3.5.

This set up procedure is interesting in order to avoid the dynamic process growing problem mentioned in Section 5.2.2.3. Therefore, all processes involved in a running session should be known before they start running. This method sounds reasonable, in order to force the user process implementor to do not make mistakes creating unnecessary or undesired processes during the process execution in *kernel* mode at the behaviour `execute()` method, which can make the implementation of `inspect()` more complex or probably wrong.

`finish(CSPSupervisorEnvironment sve, CSPBehaviourSignal signal)`

This method gets called by the `CSPProcess` implementation just after the process has finished and before it leaves from *kernel* mode. This is a topic detailed in Section 6.2.3.5.

This finish procedure can be used by a user process that needs to take some clean up action or an action based on the returned behaviour signal and `Reason` object. For instance, the prefix implementations use this method in order to execute or not their following processes according to the given behaviour signal (i.e. the prefix process is started if, and only if, the event communication has been successfully performed).

For more details on behaviour patterns, see Sections 4.1 and 4.3, and companion JavaDoc [38].

6.2.3 JACS.JDASCO — Integration Sub-Layer

The integration of semantic processes (JAPS process sub-layer) and JDASCO is done at the JACS.JDASCO sub-layer represented by the `jack.jacs.jdasco` Java package. It acts as the JDASCO application layer, implementing all necessary JDASCO management classes like synchronization predicates, method objects, and recovery object points. Figure 6.6 shows a view of the JACK layers.

This section defines, for each JDASCO role, a corresponding JACS.JDASCO participant. As stated in Section 6.1, JDASCO has three main entities: a JDASCO client requesting for a service; a concern mediator coordinating concern addition and combination; and a functional object providing the requested service. In this sense, the service that a JACS functional object must provide is the execution facility for processes. A JACK client desires that a process runs, according to its needs.

The JDASCO functional object is represented by CSP operators and user processes, since they define the desired execution behaviour to be exposed for JDASCO clients. A JACK process (represented by the `CSPProcess`) in this view, acts as the

service provider. In this sense, it needs to expose some functionality to the JDASCO client be able to make the service request. The only service of some JDASCO client interest is the ability to make process definitions to execute. This is achieved by the `execute()` method of the behaviour pattern interface. Therefore, the exposed JDASCO functional object functionality is the `execute()` method, which runs the implemented process semantics.

The JDASCO client is also represented by a JACK process (`CSPProcess` interface). It asks the concern mediator (`CSPProcessNetwork`) to submit a process specification to run at JDASCO, which means that JDASCO must call the `execute()` method of the behaviour pattern interface at the appropriate time (`EXECUTE` phase of concurrency concern). In this way, a JACK process is the actor which requests the service or provides the service, it depends on the calling context.

Therefore, there is a very special situation of the configuration of JDASCO roles, in order to properly implement JACK processes. A JACK process acts both as a JDASCO client and as a JDASCO functional object, depending on the execution context. After a thorough observation of JDASCO code, original DASCO author, and valuable discussions with the pattern community and research fellows, the decision to make the JACK process to plays the role of the JDASCO client and the JDASCO functional object, sounds reasonable due to the intrinsic compositional property of CSP processes. This decision needs some more clarifications.

There is an `execute()` behavioural operation defined at the behaviour pattern interface for both user process and CSP operators. If a JACK process plays only the JDASCO functional object role, JDASCO clients will need to know which `execute()` method to call (i.e. make some operator or process type cast in order to correctly infer the appropriated method to call). Doing so, the framework will not execute the operator semantics naturally, since the operators will be passive not knowing how to proceed, contrasting the idea of an active object[81, pp. 367] mentioned in Section 4.1.

This passive behaviour is not the case of a CSP framework, since a CSP process must act as an active object [81, pp.367], in the sense that it knows by itself, which operation to proceed (i.e. which process `execute()` method to call). Therefore, the JACK process also acts as a JDASCO client because it knows which `execute()` method it must calls due to its semantics knowledgement.

In this way, the JACK client interacts with a JDASCO client that is always a process. This means that a JACK client never interacts with JDASCO (as shown in Figure 6.6), which sounds very reasonable in the sense of organization of layer service providing. Despite this fact, the JACK client (at the level of the JACK user layer) also wants to run a process. In order to achieve this goal, a JACK process must expose a uniform way of process execution.

This goal is achieved through the `CSPProcess` interface `start()` method. That method is slightly different from the `CSPBehaviour` interface `execute()` method, in

the sense that it performs necessary internal tasks to properly start JDASCO and executes the root process semantics. For instance, the generalized parallel operator, override the default implementation of the `JACKProcess.start()` method in order to correctly implement its concurrent expected behaviour, that is two concurrent threads running each operand process at the same time synchronizing on the operator alphabet. Despite this, the `start()` method also provides a way allow JACK clients to inform the process supervisor environment under consideration for this process execution session.

As already mentioned, the JDASCO concern mediator is managed by the process network (`CSPProcessNetwork` interface). This means that the process network implementation holds a JDASCO composition layer concern mediator instance reference. The network uses that reference to submit the JDASCO client calls to JDASCO composition layer for execution, through the `associateInvocation()` method (see Figure 6.12). In this sense, the process network acts as a decorator [28, pp. 175] of the JDASCO concern mediator. Therefore, every JACK process, acting as a JDASCO client, must request the process network to submit their request to JDASCO functional object, which in the case is another or possible the same JACK process. Another interesting detail, is that the supervisor environment is responsible for holding the instance of a process network, this way a JACK process has only access to the process network after it has been started and is in *kernel* mode.

This seems very reasonable since the process network needs to be aware of all information related to process execution in order to properly achieve its goal, that is to provide an information gathering service to all JACK processes. With this detailed composition control of the process supervisor environment, the process network, and JACK processes at JAPS process sub-layer; and execution control at JACS.JDASCO integration sub-layer, the process network can clearly behave as the mediator of process execution submission.

After this exposition of the main JDASCO participants, there is the need to clarify some of the selected policies and management classes used for the integration of JACS.JDASCO sub-layer with JDASCO. For each JDASCO concern, a detailed explanation of each entity are provided below in next Sections.

6.2.3.1 Concurrency Concern Integration

The concurrency concern has one entity that must be defined for each JACK process, that is a method object. Since processes need to deal with backtrack, the framework must use a composed version of method objects that deals with recovery, called recoverable method object and represented in JDASCO as the `RecoverableMethodObject` interface. This special kind of method object is a direct extension interface of the original `MethodObject` interface and it is signaled by the recovery concern `PREPARE` phase in order to adjust the underlying functional object, that the method

object interacts with. The recoverer first level recovery policy manager returns a prepared for recovery version of the functional object, which means a functional object decorated at the process execution stack, to be used as the current functional object of the considered recoverable method object instance. This task is necessary in order to properly make the composition of recovery and concurrency concerns [133, Chapter 7].

There is only one kind of method object, since there is only one functional object method to expose, that is the `execute()` method of the behaviour pattern interface. Meanwhile, for each submitted JACK process, the framework must provide an instance of this method object. Fortunately, this task is simply achieved by the `JACKProcess` implementation class for all sort of processes, since all of them need to have a `CSPBehaviour` interface instance for the method object be able to call its `execute()` method. An implementation guided this way, gives a very dynamic execution environment. For instance, if some operator semantics changes, or by some reason its semantics needs to change, the changing process becomes easier and well-localized. One just needs to alter the `execute()` method of the behaviour interface and reuse the whole already provided generic infrastructure.

Concurrency Policies Configuration

The first level policy manager is called the concurrency generator; it is represented by the JDASCO `Generator` interface. The selected policy for JACK process execution is the `CONCURRENT` one, due to its adequacy related to the CSP semantics. They execute independently of each other following restrictions guided by their execution semantics and supervisor environment.

The second level policy manager is coordinated by future instances; they are represented by the `Future` and `TemplateFuture` interfaces. An important aspect of futures is their use for the synchronous execution of CSP operators. Since a JACK process acts both as a JDASCO client and as a functional object, the JACK client needs not to deal with futures directly, and so the modularity problem of the JDASCO concurrency concern [133, Chapter 4] is completely hidden (see [43, Section 1.1] for some detailed discussion about this problem).

In this sense, the proper use of futures is the responsibility of the JDASCO client, in our case a JACK process. This JDASCO client synchronization is mentioned in Section 6.2.3.5.

The process network configuration options holds the policies selection for each concern. Specialized users or extension developers, can make use of them to temporally change the expected configurable concern policy in order achieve some specific goal, like incremental development. For instance, during the initial development process of JACK integration sub-layer, we use the `NONCONCURRENT` policy (as suggested in [133, Chapter 8]) in order to debug the JDASCO composition of concerns

with respect to JACK processes management class implementation.

JACK Process Method Object Implementation

Method objects have two important jobs:

1. The execution of the concurrent part of the processes operational semantics.
2. To inform the process network through future notifications with success or failure, depending on the synchronization concern execution status, that the process execution has finished. At this point, the JDASCO client activity can be resumed to proceed its execution.

   The recovery concern also takes advantage of this notification scheme, since it can prematurely abort unsuccessfully executed method objects.

The first job corresponds to a direct call to the `execute()` method of the `CSPBehaviour` interface. It represents the behaviour pattern of the process received at the method object construction. The second job is achieved in two steps:

1. After the return of the `CSPBehaviour.execute()` method call, the future is notified about the process execution status at the `MethodObject.execute()` method. This returned value is represented by a `CSPBehaviourSignal` instance and it acts as the method object (process execution) returned value. The available behaviour signals mentioned in Section 6.2.2.4.

   Since the process method object has a return value, we need to use a special kind of future to hold that value, it is called a template future and it is represented by the JDACO `TemplateFuture` interface. This is a minor JDASCO detail, for more information about different available futures, see `TemplateFuture` and `CSPFutureImpl` JavaDoc [38].

2. Following the collaborations of concurrency concern mentioned in [43, Section 1.1.2, Figure 2], the `MethodObject.finish()` method gets called. This method implementation notifies the process supervisor environment that its pupil has finished its execution.

Currently, JACK provides only one method object implementation called `JACKProcessRMO`. Despite this, a specialized user processes that wants to implement its own method object version must implement the `RecoverableMethodObject` JDASCO interface.

6.2.3.2 Synchronization Concern Integration

The synchronization concern has two synchronization entities to define[1] for each exposed functional object method, that is a synchronization predicate and a synchronization data. Each JACK process defines its synchronization predicate to control the execution flow of its semantics. Every process also has a specific synchronization data to register, inspect, and deal with synchronization information, in order to properly follow the JDASCO rule for not breaking modularity due to direct access of functional object information. Therefore, for each JACK process there is a synchronization predicate and a synchronization data implementation class pair to respectively control execution flow and store synchronization information. Since there is only one method to expose, there is only one pair for each CSP operator.

Due to some complexities inside the synchronization finite state machine when combined with other concerns, an extended version of the synchronization predicate and synchronization data interfaces are used. These extended versions are necessary in order to properly implement the history sensitive synchronization second level policy mentioned in [43, Section 1.2]. They are represented by the `HistorySynchronizationPredicate` and `HistorySynchronizationData` interfaces.

Each JACK process should specialize a synchronization predicate and synchronization data instances according to its needs. For each submitted JACK process, the framework must provide an instance of these interfaces. This task is generically achieved by the `JACKProcess` implementation class for user processes and specialized by each CSP operator implementation. Again, implementing processes this way we achieve a very dynamic synchronization environment. User defined processes can trust the already generically defined synchronization predicate and data pair, so JACK users never need to deal with JDASCO predicates unless some very specialized or advanced control was necessary. On the other hand, each CSP operator ought to define its own synchronization semantics. For instance, the *STOP* operator must deadlock and the generalized parallel operator must synchronize on its synchronization alphabet.

Synchronization Policies Configuration

The first level policy manager is called the synchronizer; it is represented by the JDASCO `Synchronizer` interface. It is configured at the level of the process network. The policy selected to be used by JACK processes is the `GENERIC` one, since some operators need to use the `PESSIMISTIC` policy (i.e. prefixes) and some of them need to be `OPTIMISTIC` policy (i.e. generalized parallel), depending on the execution context. Although some operators like the generalized parallel could either be `PESSIMISTIC` waiting for some execution condition at the `PRE-CONTROL` synchroni-

[1] More details on this topic can be found in [43].

zation phase or be `OPTIMISTIC` aborting some operand due to a backtrack situation that has occurred at the `POST-CONTROL` synchronization phase, or due to a deadlock that has been reached.

The second level policy is configured at the level of synchronization predicates and synchronization data pair. The second level synchronization policy selected to be used by JACK was an extended combination of the dynamic priority readers/writers and the producer/consumer enriched with history sensitive synchronization predicate information. The readers/writers policy is applied to situations where the operator needs to inspect other executing operators, like with the generalized parallel. The priorities are necessary in order to avoid reading starvation of reading prefixes, (see details about predicate priorities in [133, Chapter 5]). The producer/consumer policy is applied to situations where the operator needs to inspect the state of other operators, like occur between prefixes and generalized parallel. The history sensitiveness is important in order to correctly inform the supervisor environment about the possible backtrack paths. This is an extended policy not found in original DASCO.

JACK Process Synchronization Data Implementation

The defined synchronization data is responsible for dealing with synchronization information about the functional object during its execution. There is a generically defined synchronization data used for all sort of JACK processes called `JACKProcessSD`. For user defined processes it acts as a default implementation. It provides default support for history sensitiveness, in order to proper implement the selected second level policy. This way, the synchronization data follows the Bloom's [9] rules referenced in [43, Section 1.2].

For each CSP operator there is a different specialized implementation of the synchronization data. This is needed in order to properly know the execution situation and information of each operator independently. For instance, the prefix synchronization data must have access to the prefix operator guard, channel interface, and value set constraint for read prefixes, or a copy of the value to be written by write prefixes. For details about JACK CSP operators, values, and value sets, see Chapter 4 and [38].

An important fact must be observed about a process that enters in *kernel mode* of execution at JDASCO, a topic mentioned in Section 6.2.2. All read/write structures must be copied or sealed (see Section 4.4 for a explanation of these concepts) before enter in *kernel mode* (i.e. be submitted for execution at JDASCO), in order to avoid side effects execution during physical execution. For instance, when a write prefix enters in *kernel mode* its value to be written is copied. Any change on that value after this submission will be observed only in the next execution of the write prefix, if one occurs at all.

A specialized user process that wants to implement its own synchronization data version must be aware of this fact. They also ought to inherit from the synchronization data base class (`JACKProcessSD`), in order to properly inherit the history sensitive implementation mechanism.

JACK Process Synchronization Predicate Implementation

Synchronization predicates are responsible for dealing with synchronization control of functional object execution control flow. In this sense, each JACK process has a synchronization predicate to implement its *dynamic* semantics specifying when, how, and why some process can or cannot proceed with its execution. There is a generically defined synchronization predicate used for all sort of JACK processes called `JACKProcessSP`, also used as default by user processes. It provides default support for dynamic priorities, in order to proper implement the selected second level policy and properly linking with the process supervisor environment. As occurred with the synchronization data, the synchronization predicate follows the Bloom's [9] rules.

The linking with the process supervisor environment is very important, in order to properly implement and inform the process network which operand can or cannot follow, based on the underlying supervision information (i.e. already visited backtracked paths). This path selection is based on the inspection of possible events that the pupil processes can engage immediately (i.e. the resultant alphabet of a call to the `CSPBahaviour` interface `inspect()` method). This alphabet of possible immediately available events is used by the supervisor to guide the possible backtrack paths and is called its desired communication alphabet. The recovery concern also have a role about backtracking possibilities or deadlock situations for abortion, due to synchronization errors, as expected when composing synchronization with recovery [133, Chapter 7]. The supervisor is also responsible for signalling the synchronization concern through the synchronization predicate using a `CSPSupervisorSignal` instance, that the correspondent executing concurrency concern recoverable method object instance needs to finish its execution. The method object in turn, notifies the calling JDASCO client activity.

For each CSP operator there is a different specialized version of predicates that must implement the correspondent operational rule of each operator for synchronization. JACK documentation provides a brief tabular description of some operators [38]. For instance, prefix predicates need to check the guard condition before execution. Read prefixes must also check their value set constraint with respect to possible constrained values written by related write prefixes (i.e. prefixes sharing the same channel). This minimizes the possible backtrack cases, since it avoids data dependency executions (without expansion) that will deadlock due to some value set restriction.

A specialized user process that wants to implements its own synchronization

predicate version must inherits from the synchronization predicate base class (`JACK-ProcessSP`), in order to properly inherit the linking with the supervisor environment.

6.2.3.3 Recovery Concern Integration

The recovery concern has only one entity to be defined. It makes the backtrack process operational. As expected by the JDASCO definition for recovery, there is only one class to be implemented for all JACK processes. It never needs to be extended or implemented by any derived class. This is in contrast with other concerns in which, for each operator, there is the need to define a different implementation entity.

The recovery object point defined keeps a shared process execution history stack represented by the `CSPProcessExecutionHistory` interface. There is only one instance of this interface shared among all recovery object point instances and it is controlled by the process network.

The process execution history stack is adjusted at the recovery `PREPARE` phase. When the execution finishes, it can either commit or abort. If it commits, the committed process is removed from the stack; if it aborts, the aborted process remains inside the process execution stack. In both cases, the process network is indirectly notified by the recovery object point, since they share the same stack instance. There is no safety penalty with the use of a shared instance, since the execution phases where the stack can be altered already execute under a safe region.

For instance, the process execution history stack information and the notification procedure can be used by the generalized parallel on concurrent concern at the `FINISH` recovery phase. With this, the operational part of the backtrack solution procedure, like process path searching, can be achieved inspecting the process stack stored in the process network. This search procedure could define which path to visit based on not yet visited paths and already visited backtracked ones with respect to the supervisor desired communication alphabet. To carry out the search, the supervisor of the process under consideration uses the `CSPBehaviour` interface `inspect()` method that represents the Hoare's domain D of possible initial events.

In this way, the supervisor can decide which paths can be followed. If there is no more paths to follow, a deadlock situation is detected. A combinatorial search of all possible paths must be done, in order to correctly detect the deadlock, considering any possible backtrack path.

Recovery Policies Configuration

The first level policy manager is called the recoverer; it is represented by the JDASCO `Recoverer` interface. The adopted policy in JACK is the `DEFERRED--UPDATE` one, since it is simple to confirm (commit), and also simple to abort; both

operations are very important to JACK. The abortion simplicity is the most important aspect, since it is directly related to backtrack.

The second level policy adopted is the history sensitive compensation operations, an optional not available in original DASCO. DASCO warns that the compensation operations choice, instead of object copying, makes the functional object and the recovery concern mediator to loose modularity. This occurs since the recovery mediator needs to rely on the exposed functional object interface. Nevertheless, this new special kind of compensation operations does not rely on functional object exposed methods, but it does rely on a shared process execution history stack.

Some discussion is needed at this point. The other policy available is object copying, which can be easily achieved in Java, due to its built-in cloning support for object instances. However, because of the compositional property of processes, the whole (deep) copy procedure has a heavy resource cost when the network becomes big. The interface definition of JACK processes combined with this shared stack instance, can directly be used as compensating operations. In this way, modularity is not lost, since process interfaces do not get directly (i.e. there is no need to call one of its interface methods) involved on the compensation procedure.

JACK Process Recoverable Object Implementation

As already mentioned, only one recoverable object implementation is necessary for all sort of JACK processes, that is, a generically defined backtrackable recoverable object called `JACKProcessRO`. For each JDASCO session (i.e. a JACK process execution submission), a new instance of it gets instantiated in order to properly manage the shared process execution history stack. In this way, the whole backtrack procedure is facilitated, since the process supervisor environment needs only to inspect that stack.

A specialized user process that wants to implement its own recoverable object version must inherits from the recoverable object base class (`JACKProcessRO`), in order to properly inherit the link with the mentioned shared stack.

6.2.3.4 Composition of Concerns Integration

The composition of concerns represents the intermediate layer of JDASCO (see Figure 6.6). It composes the concerns of the concern layer in order to allow JDASCO application layer to use them. In original DASCO this composition is done using multiple inheritance. Recent framework modeling techniques [113, Chapter 4] state that multiple inheritance must be avoided.

When composing (or integrating) frameworks, we have to be very careful to avoid problematic situations for the future. The DASCO implementation decision to use multiple inheritance is inadequate for a general purpose framework. The given

implementation acts as an example and not as a real world implementation. In this sense, a different approach to compose the concern frameworks is adopted. The adopted solution follows the role modeling of JDASCO: the use of Java interfaces to represent concern protocols (role types of each concern) [113, Chapter 5 Session 5.4.3]. At the composition layer, a single class implements all the necessary concern interfaces and makes use of their services by aggregation, constituting a special kind of decoration. This means that the composition implementation class has as one of its attributes a concrete instance of each concern interface provided by the specific concern layer. In this sense, the implementing class acts as an adapter [28, pp. 139] pattern that composes multiple concerns.

Composition of Concerns Selection

The composition of concerns used is the concurrent synchronized recoverable one. This does not add any implementation class to be defined, but just some policy adjustment and configuration requirements. The decision for this composition was made based on implementation requirements discussed in this Chapter.

The concurrency concern deals with activities creation, coordination, and management. For instance, they are used when a JACK process enters in *kernel* mode for physical execution.

The synchronization concern deals with execution control flow of JACK processes semantic implementation through lock monitoring and administration. For instance, the prefix needs to wait for synchronization when it was inside a synchronization alphabet of a parent generalized parallel; that parent must notify the shared process supervisor environment about this fact. In this way, the supervisor environment can notify the prefix that it can follow, through predicate queue inspection. Thus, the generalized parallel synchronization predicate is using the reader/writers second level synchronization policy.

The recovery concern deals with JACK processes recovery due to possible backtrack situations. For instance, when an external choice notifies its supervisor about possible selectable paths, the supervisor then choose one of the paths based on some selection decision function. If the selected path leads the process to a deadlock situation, an alternative path must be tried until all possible paths had been visited.

Policy Configurations when Composing Concerns

Some attention must be given to the consequences of composing these concerns. The composition of concurrency and synchronization adds the activity association decision with synchronization predicates to be done before or after the lock acquisition. The selected composition layer association policy is the `IMMEDIATE` association, which associates an activity independently of any lock, in order to increase concur-

rency between JACK processes with a possible resource waste penalty. The other possibility, the `LAZY` association policy, may not be efficiently used, since it needs all locks to be acquired before activity creation. Due to the compositional property of CSP processes lock acquisition may be scattered.

The composition of synchronization and recovery concerns has some restrictions, like composing `OPTIMISTIC` producer/consumer synchronization policies with `DEFFERED-UPDATE` recovery policy [133, Chapter 7]. This restriction does not bring any problem, since the `OPTIMISTIC` synchronization policy, when used as a `GENERIC` synchronization first level policy, is composed with the readers/writers second level synchronization policy. Since it does not inspect the functional object state, it does not break the restriction of composing synchronization with recovery [146, 133].

The composition of concurrency and recovery concerns has neither restrictions nor decisions to be made. The only detail that is important to mention is the fact that the recovery `PREPARE` phase defines the functional object that participates on the concurrency `EXECUTE` phase. This set up procedure is carried out through the recoverable method object extended interface that allows this kind of notification. With this, the recovery process has the control of the boundaries of the JDASCO execution (see Figure 6.5). The functional object that runs is defined at the beginning of the recovery `PREPARE` phase and the committed or aborted functional object is defined at the end of the recovery `FINISH` phase. This means that the process execution history stack can be used at those points in order to properly define which process to run, that in turns perform a very smoothly backtrack procedure. In this way, the complete execution path for composition semantic behaviour definition was described.

6.2.3.5 JACK Process Acting as a JDASCO Client

Previous sections explain how JACK processes are configured to be integrated to use JDASCO services. Thus, we have shown how to integrate a JACK process with JDASCO when it plays the role of a functional object. This section provides the same integration procedure, but with a JACK process acting as a JDASCO client.

In contrast with functional object integration, there are no stated rules or guidelines to be used for JDASCO client specification. The only additional information available is the need of the JDASCO client to be aware of the concern models, a topic already mentioned. Thus, for each concern, a JDASCO client must be aware of the following.

- Future object synchronization at the concurrency concern.
- Possible synchronization execution status errors at the synchronization concern.

- JDASCO clients can explicitly interact with `RecoveryMediator` interface commitment and abortion procedures at the recovery concern.

JACK must take care about the first two topics, since it does not make use of the last one. In this way, a JACK process acting as a JDASCO client must interacts with futures instances, in order to properly observes and notifies the JACK client about any process execution problem. With respect to the concurrency concern, a JACK process must wait (block) on the available future, in order to properly implement the synchronous second level concurrent policy selected; it can also inspect the future for the resultant `CSPBehaviourSignal` of the process execution signalled by the recoverable method object. With respect to the synchronization concern, a JACK process can inspect the future for occurrence of any synchronization error or premature abortion through the `Future` interface `isError()` method. For instance, the abortion signal must be informed to the process supervisor environment under consideration, in order to allow it to proper proceed with the backtrack protocol.

JDACO Client Implementation Details

A JACK process acting as a JDACO client provides to its own clients (a JACK client) the `start()` method to allow them to interact with the framework. This method is the single point where a JACK client instructs the whole JACK framework to starts its execution. The JACK `CSPProcess` interface `start()` method is divided in three parts: notification that the start procedure has beginning and has finished following the before/after [81, Chapter 2] pattern, and the proper process execution handled by the **protected** `JACKProcess` class `doStart()` method. For instance, the process before start action broadcasts to all its children processes the supervisor environment to be used for this execution session. Thread creation at this point does not break the JDASCO rule that a JDASCO client never need to create threads. This thread instantiation abstraction is related only to access to exposed functional object methods and not related to JDASCO client activities.

As already mentioned in Section 3.2.2, a CSP process is an active object with a private thread of control. A process is an active object when it runs under an independent thread of control and has not yet been returned (i.e. successfully terminated). A process is a passive object when it runs under the thread of control of its caller. Therefore, a JACK process can be either in a passive or in an active state (see `CSPProcess` interface `isAnActiveObject()` method documentation [38]). Only root processes and processes created by the parallel operators act as an active object (i.e. create and manage threads). This means that the JACK client thread is always free to proceed its execution.

Since a `JACKProcess` class can act as a running thread under some circumstances, it implements the `java.lang.Runnable` interface. The implementation of the `run()`

method of that interface is used for either situation where a process can be a passive or an active object, which elegantly generalizes the final implementation. Controlling flags ensure that this **public** method can never be called directly by JACK clients or any other class.

Each JACK process must know which JDASCO management class to create for each JDASCO concern. As already mentioned, a default implementation of them are given for all processes (specially user defined ones). A specific implementation is provided for CSP operators when necessary. This instantiation selection is carried out by `JACKProcess` class using the factory method [28, pp. 107] design pattern.

`JACKProcess run()` Method Implementation

The `run()` method implementation is the heart of the process execution. It is partitioned in three specific parts. They are explained below.

1. **Process behaviour setup** — simple step responsible for CSPBehaviour proper initialization of process (see Section 6.2.2.4 for a discussion about this topic).

2. **Process behaviour execution** — step responsible for starting the *kernel* mode of process execution (i.e. submit the process to execute at JDASCO).

   This is the most important part of the process execution procedure. At this point, the factory methods used to instantiate the JDASCO management classes of each concern gets called, in order to dynamically configure each JDASCO concern according to the appropriated CSP operator or user process semantics. A process enters in *kernel* mode when it request the process network to associate the recently created management classes with the JDASCO composition layer through a composition `Associator` interface.

   This step also uses the before/after pattern in order to allow CSP operators or specialized user processes to interfere at these specific execution points. For instance, the read and write prefixes CSP operators uses the before execute action to seal their underlying value set constraint and value to be written, in order to avoid side effects during *kernel* mode execution, as mentioned in Section 6.2.3.2. This is simply achieved through the use of the event notifications available at the level of the JACK type system. These events are mentioned in Chapter 4.

3. **Process behaviour clean up** — step responsible for cleaning up the process execution and restore it to *user* mode again, removing the finished process from the process network process queue.

This is also an important step in the process execution protocol. It is responsible for ensuring the expected synchronous behaviour of the concurrency concern (i.e. blocking on the currently executing future instance).

Furthermore, it notifies the behaviour that execution has finished through the `CSPBehaviour.finish()` method. This is very important because it informs CSP operators about possible execution synchronization errors (`Future.isError()`) and the returned process execution signal (`CSPBehaviourSignal`), when the *kernel* mode is just finishing. The operators in turn, use this notification information to take the appropriated execution decision related to its operational semantics. For instance, the prefixes operators must start the execution of its related process P only if its communication has finished successfully.

This step also implements the before/after pattern. For instance, the default process implementation uses the after clean up action to clear the supervisor environment reference of its children, indicating that the process has leave the *kernel* mode. Another example, is the read and write prefixes CSP operators that use the same action to restore the original states of their underlying data structures altered on the before run action of the same session.

An important detail should be noted about the nature of these mentioned before/after actions (i.e. before/after start, execute, and clean up). Since the JACK processes can run either as a passive or as an active object, these methods might be called under different execution threads. In the following, we present the possible execution contexts of each action.

Before/After start — always executes under the calling thread context. This can either be the JACK client thread starting the root process or some other active object process been started.

Before/After execute or clean up — executes under the calling thread context, when the process acts as a passive object. On the other hand, it executes under the newly created process thread, when the process acts as an active object.

This completes the description of the JACK process acting as a JDASCO client.

6.3 Final Considerations

In this chapter, we present a detailed description of each layer role and the integration between them. It is important to note how these roles communicate with each other to make a complete view of the whole framework model. For a more detailed description of the JACK framework see [38].

JACK documents in [38], test case classes (see packages under `jack.jcase`), and Chapter 4 provide examples and usage guidelines of JACK. With these information, JACK clients can clearly observe how to use the framework to implement their process specifications using JACK framework as a Java extension package.

In the next Chapter, we present the dissertation conclusion and some important improvements and future work for next JACK releases.

Chapter 7

Conclusion

The JACK framework implements an environment that can represent concurrent languages. It instantiates this environment implementing a new version of the CSP [124] process algebra. The implementation is built using separation of concerns in a way that is highly beneficial to class-based design of frameworks. This work empathizes the use of design patterns and pattern languages to properly build frameworks, achieve desired software engineering properties and software quality requirements (see Section 1.3).

The user of the JACK framework is able to:

- Describe its process specification in Java, either in CSPm [33] or in a combined algebra one, like in CSP-OZ [31] or in Circus [148, 149].
- Represent infinite data types without starvation of the process network execution, since the framework provides a symbolic approach for dealing with this sort of types. This is in contrast with the expansion approach used by tools like ProBE [34] and FDR [33].

The extension developer and the user of the JACK framework is benefited by:

- Expressive documentation of the whole framework, modeling decisions, and implementation details.
- The use of design patterns makes it easier to be extended, since it shares the same idiom with other framework developers, an inherited benefit of the use of design patterns.

The JACK framework combines the strengths of role modeling and design patterns with those of class-based and layered modeling while leaving out their weaknesses. This increases the framework robustness. Another major aspect is the possibility to directly use the framework as the heart of a set of import tools to be used in the formal methods field, like formal translation tools. For instance, the

same work done to translate a CSP-OZ specification to Java using CTJ [12] can be done to translate Circus to Java using JACK.

In Chapter 3, we state some goals and objectives that a framework may have, and what we expect that JACK ought to provide. Here, we provide a link between the established and achieved goals.

1. Design Patterns — JACK is designed, implemented and documented using design patterns and pattern languages. Since JACK follows strictly the applicability of adequate design patterns and pattern languages to solve each problem domain, it avoids obscure problems in the rail road of the design of concurrent frameworks, as observed in [36] and in comparison with other related libraries (see Section 3.6). With the use of design patterns, desired software engineering properties like expressive power, reusability, extensibility, modularity, etc, are achieved.

2. Framework Modeling — JACK follows up-to-date framework modeling techniques, like Role Modelling [113], and UML [56, 74]. In this sense, it is prepared for both extension (i.e. white-box framework reuse) and user needs for defining its specifications (i.e. black-box framework reuse).

3. Processes Support — JACK provides processes (i.e. CSP operators and user defined processes) embedded in the Java Language constructors as an extension package. Even more, the framework provides a set of constructs that allow the framework to be generalized to become an environment for the implementation of concurrent language. This can be useful for instance, to teach and study the physical execution of a formal process algebra (i.e the framework acting as an animation tool), that reflects the semantics of the language constructs.

4. CSP [124] Operators Implementation — JACK provides the most important CSP operators.

Since JACK support both user processes and CSP operators, it also allows the user to describe combined algebras like CSP-OZ and Circus directly, which makes the framework to be used as a tool in the formal methods field.

7.1 Contributions and Results

We have presented an approach to implement process algebra [124, 61, 130, 126] with separation and composition of concerns, integrating concurrent and object-oriented programming [133]; framework role modeling [113, 22]; design pattern and pattern languages [28, 76, 123, 102, 81, 121, 23]. In contrast with other available similar

libraries [107, 59], JACK is modeled using object-oriented framework techniques, which leads to a more robust, adaptative, and appropriated for evolution (white-box framework) [113]. The proposed approach for JACK has generic features which allow its use in contexts other than CSP. In other words, JACK is a generic framework for describing process algebras that is specialized for CSP. In this sense, CSP can be regarded as a huge case study.

The implementation consider the most recent version of CSP [124, 126, 130] and has been carried out in Java. The user of the JACK framework can also independently build its own user defined processes, which introduces a sharp raise on the expressive power of the framework, because it opens the possibility to the JACK client to use the framework to implement combined specifications [62], like CSP-OZ [31] and Circus [148, 149]. In this direction, there is a work that provides formal translation rules from CSP-OZ to CTJ [59]. Another possible work is to do the same formal translation rules, but from Circus specifications to JACK.

An important aspect to single out is our strategy to solve two important aspects of an implementation of CSP: multisynchronization (see Section 2.3.3) and backtrack (see Section 2.3.2). The former is directly related to the new version of the parallel operator of CSP (i.e. generalized parallel), that accepts synchronization on more than a pair of processes. The latter is directly related to the data dependent execution conditions that cannot be inferred without expansion.

Our work serves also as a realistic use of the DASCO framework described in [133], which validates the results mentioned there and realizes some of the mentioned future work, like remodeling DASCO to use framework role modeling [113]. The solution using this framework has shown to be a very attractive solution for complex concurrent systems. It is neither easy nor complete, but after a thorough research on the concurrency field, it shows to be adequate for our purposes.

An implementation of a symbolic approach to execute processes without expansion, which leads the specification to be able to deal with infinite data types. This is a contribution when contrasted with tools like FDR [33] which cannot deal with infinite data types because it uses an expansion strategy to analyse processes. For a physical implementation of CSP it is important to accept infinite data types like `Object` and numeric types like integer.

This approach is achieved through the implementation of a robust type system that allows users to define many sort of types, values, sets and logical predicates. JACK provides a normal form reduction algorithm of a subset of the predicate calculus based on [68, 67, 21, 66, 69], in order to provide the symbolic approach to deal with infinite data types.

Due to the layering organization of the framework, important software engineering aspects like incremental development, composition of concerns, modularity, expressiveness, abstraction, anticipation of changes, and so on, are achieved (see Section 1.3). JACK provides implementation of well-known design patterns in Java.

Some of them are mentioned below.

1. Variants of concurrent design patterns defined in [81, 128, 138] like: `Latch`, `Mutex`, and `ReentrantMutext`.

2. The robust and powerful Event Notification [110] design pattern used in all layers of the framework for many different notification purposes.

3. Acyclic Visitor [123] for the implementation of an initial version of a interpreter for JACK (see JCAST layer at Chapter 5).

4. Diagnostic Context Logging [123] for debugging and logging purposes.

5. Double Checking Locking [23] with Thread Local Storage support. This new version of the pattern is better suited for concurrent environments [140], and so on.

The framework also defines and implements new design patterns. Some of them are enumerated below.

1. Flyweight Constant/Factory — Allows one to define enumerated constants in Java with support for methods like `equals()` or `toString()`.

2. Benchmarker — Perform single benchmarking facility; it is very useful to debug and test code efficiency.

3. Extended Iterator [28, pp. 257] — Provides a set of new Iterator with extended functionality like: controlled modification support of the underlying collection, read-only iterators, string representation of them, etc.

4. Template Collection — Decorates [28] all available Java collections to behave like template classes (i.e. classes with template types).

Documentation of the object-oriented framework following up-to-date modeling and documentation techniques [113, 116, 115] (see Chapters 5 and 6), is provided. Solutions are described as design and composition of patterns, using only basic concepts from the object-oriented paradigm. JACK provides a site with on-line JavaDoc documentation following well-defined guidelines [38], a Rose [135] UML [74, 56, 109] model with many illustrative diagrams (i.e. finite state machines, sequence diagrams, collaboration, etc.), draft versions of users and developers guides [41, 42], and some role modeling [113] artifacts.

The framework also defines of test cases following the guidelines stated in [48, 47] and the use of a tool [105] for software metrics and testing analysis. These test cases helped us to properly maintain the code accurate while it evolves. This is a very important aspect of the framework implementation due to the compositional property of CSP, that changes in one place affects many other places.

7.2 Future Work

There are some areas we envision as object of future work following the results achieved in this dissertation. These are related to architectural improvements, CSP extensions, tests, tool support, and documentation.

7.2.1 Architectural Improvements

After some tests and analysis of the JACK code, we identified possibilities for improvements of some architectural aspects. They are mentioned below.

- JACK Type System

    1. Multidimentional types must be normalized. Actually, we accept only a limited set of multidimentional types. This set is sufficiently expressive but it lacks the possibility to describe recursive multidimentional types. For details on types related to this topic, see `jack.jacs.japs-.typesystem.CSPType` interface companion JavaDoc in [38].
    2. The normal form reduction algoritjm, used to implement the symbolic approach to deal with infinite data types, ought to support the predicate calculus predicates with $\forall$ and $\exists$ quantifiers.
    3. The type system should be extended to fully implement the Type Object [123, pp. 47—65] design pattern. The pattern itself is quite complex, but its applicability and companion positive consequences sounds to be very relevant to our work. It will be necessary to adapt the pattern for our scope. Actually, the type system is a partial implementation of this pattern based on the already available implementation of it coming from the JValue framework [117, 112, 111, 44].
    4. Implementation of other versions of value serializers [123, Chapter 17 pp.293] and data links like TCP/IP, or Swing [141] ones, as found in other related libraries [107, 59] should be considered.

- JDASCO Execution Layer

    1. JACK defines some new concern policies for JDASCO not available in the original work of DASCO [133]. They are mentioned in Chapter 6. These new policies extends JDASCO to be more modular. Some of the extended policies were not used due to time and space limits. Nevertheless, it can be an interesting work to better explore these policies like the history sensitive synchronization and recovery policies.
    2. Normalization of the exception handling of the whole layer, to avoid extensibility penalties for future extension on layer integration.

- JACS Semantic Layer

    1. Definition of other `CSPProcess` interface implementations. This might be done in the case of a distributed version of processes execution, using. for instance, Java RMI [63], CORBA or DCOM [26]. Actually, JACK does not support a distributed version of a process network.

- Layer Integration — Generalization of the layer integration packages.

  As already mentioned in this Chapter, we observe that JACK can be used to implement any process algebra, just configuring its policies and adopting some minor points.

  Nevertheless, the original motivation of the framework is to build a CSP [124] implementation in Java. Therefore, JACK, at few layer integration points, is specialized for dealing with CSP (i.e. recursion, fixed points, sequential composition, hiding, and so on). It sounds to be interesting and easy to generalize this to capture more abstract aspects of process algebras.

The mentioned improvements do not cause any direct penalty, except for the exception handling organization between layers. Therefore, this may be the first improvement to be achieved in order to allow full extensibility between the execution and the semantic layers.

7.2.2 CSP Extensions

Here, we present some interesting improvements to be done in the framework directly related to the implementation technique, or to possible new interesting operators and features.

- Coding Techniques

    1. Primitive processes can be implemented using the flyweight [28, pp. 195] pattern to avoid unnecessary explicitly multiple instances.
    2. The user behaviour interface (`CSPBehaviour`) `execute()` method may allow user defined parameters.

- The following extended operators and constructs (see [124, 130]) had not been implemented.

    1. Labelling — $l : P$.
    2. Interrupt — $P \triangle_e Q$ and $P \triangle Q$.
    3. Piping — $P \gg Q$.

4. Event Renaming — $f(P)$, and $f^{-1}(P)$.
5. Multidimentional prefixes — $c?x!y?z \rightarrow P$.
6. Replicated versions of CSP operators.
7. Mutual recursive equations.
8. Recursive equations with parameters.
9. Timed CSP operators described in [130, Chapter 9—13].

Some of them can be implemented straightforward with the available constructs, like Interrupt, recursive equations with parameters, and replicated versions. Operators related to reanaming like event renaming, labelling, and piping, should be more difficult to implement, since one needs to define extended versions of the supervisor environment, like hiding and parallel operators do. We believe that JACK can be extended to implement timed CSP operators, but this should be considered a major improvement of the framework.

- Extended Features

1. Functional expression language as an extension of the JACK type system. With this extended feature the user will be able to describe expressions in its specifications like in value set predicates of type constraints of read prefix, or in the "value to write" of the write prefix.
2. Implementation of a Z [137] toolkit based on already available implementations, like Jakarta Commons project [71]. With the functional expression language and the Z toolkit, it becomes more attractive and easier for the user to express specifications in combined algebras, like CSP-OZ and Circus.
3. Finishing the implementation of the JCAST user layer. This layer, as mentioned in Chapter 5, is responsible to represent a CSPm specification as an abstract syntax tree following the guidelines stated in [143]. This layer implementation combined with available CSPm parsers [126, 1], would make it possible to interpret a CSPm code and formally translate it directly to JACK. This topic is very interesting in the way to build a translation tool using formal translation rules and JACK. This is mentioned below in the Tools section.

7.2.3 Tests

JACK already uses a test framework called JUnit [47]. Despite this, no considerable performance tests has been developed for JACK. To achieve this goal, other tools

like JUnitPerf, JDepend, and JMeter can be used (see JUnit site for references of these tools [47]).

On the other hand, there are other kind of tests related to software metrics that can also be done for JACK using tools like Parasoft JTest [105]. An initial version of the framework uses some of the results of this tool to normalize the library code and create our own set of code conventions based on these results and other sources [114, 80, 91].

Nevertheless, to use JACK in a production environment for industrial scale specifications, a more accurate performance metric analysis must be done. Some performance and bottleneck analysis techniques [70] can be used to increase the power of the framework, raising its execution performance.

7.2.4 Tools

As already mentioned, JACK is a process-oriented framework. This means that it provides process functionality to the framework client that can uses it for any desired purpose. In principle, it should be is possible to use JACK to describe some process specification, possibly in pure CSPm [33], in CSP-OZ [31], or in Circus [148, 149].

In this way, an interesting set of tools can be built using JACK, in order to provide tool support for analysis and execution of specifications. In what follows, we mention some of them.

- Formal Translation — There is a work [12] that provides formal translation rules from CSP-OZ to CTJ [59]. Another possible work is to do the same formal translation rules, but from Circus specifications to JACK. This work is under development and should appear in the near future.

  In [1], a tool that translates CSP-OZ specifications to FDR CSPm input following the guidelines in [100] is presented. An extension of this tool to translate CSPm or CSP-OZ specifications directly to JACK could be considered.

- Model Checking — There is a working plan under consideration to check the feasibility of using JACK to build a model checker for Circus.

This close relation with tools is an important contribution of the JACK framework to the formal methods field, since it is impossible to have a productive application of a formal approach without tool support.

7.2.5 Documentation

As already mentioned in Chapter 3, the better documented a framework is, the more efficient and practical it becomes. In this way, we identify some points that might be documented in more detailed. This documentation vary from code and design

documentation to formalization of some JACK subsystems. They are mentioned below.

- Formalization

    1. Z [137] description of the JACK data type system and its operations to allow us to prove some implementation properties.
    2. Operational semantics [108] laws describing the backtrack and multisynchronization algorithm.
    3. Formal description of the finite state machine that controls the supervision and behaviour signalling protocol, with respect to JDASCO expected roles.
    4. Finite state machine of the normal form algorithm of the subset of the predicate calculus used.
    5. Definition of all role models of JDASCO and JACS following the syntax available in [113]. Actually, only the JDASCO concurrent concern has this description.
    6. Extension of the already available [82, 35] Action Semantics [97] description of an execution environment for CSP that deals with backtrack and multisynchronization.
    7. Description of the annotated labelling transition system, that represents the static aspects of the processes network; and the supervised process execution frame stack, that represents the dynamic aspects of the processes network.

- Design Patterns

    1. Documentation of the newly created design patterns following the guidelines stated in [28, 81, 133, 19], depending on the kind of the pattern.
    2. Generalization of some implementation classes to become design patterns. For instance, the process network configuration options implementation can be abstracted to become a generic service or package configurator.

- Tutorials

    1. Finish the draft version of the JACK Tutorials [41, 42].
    2. Create a framework user manual.
    3. JACK code and JavaDoc conventions.

A JavaDoc Doclet [92] extension to capture the new JavaDoc tags created for documenting the JACK framework is described in [38], for any target documentation format (i.e. HTML) is also an interesting work for the future.

Appendix A

JACK Links — Additional Material

This appendix provides pointers to information referred to JACK. In what follows, some of the most important references related to this JACK release are given.

- JACK Home Page — www.jackcsp.hpg.com.br
- JACK UML Model — www.jackcsp.hpg.com.br/model
- JACK Support E-Mail — jackcsp@ieg.com.br
- JACK News Groups — jackcsp@yahoogroups.com
- JACK Additional Documents — www.jackcsp.hpg.com.br/pub
  - JDSACO Detailed Description
  - JACS.JDASCO Integration: Supervision, Backtracking and Multisynchronization (Draft Version)
  - JACK Tutorial 1: Users Guide (Draft Version)
  - JACK Tutorial 2: Developers Guide (Draft Version)

Bibliography

[1] Alexandre Motta Adalberto Farias and Augusto Sampaio. De CSPz para CSPm - Uma Ferramenta transformacional Java (in Portuguese). In *Proceedings of V Formal Methods Workshop of XVII SBES in Rio de Janeiro*, October 2001.

[2] Ellen Agerbo and Aino Cornils. How to preserve the benefits of design patterns. *OOPSLA*, pages 134–143, oct 1998.

[3] Krzysztof R. Apt and Nissim Francez. Modeling the distributed termination convention of csp. *ACM Transactions on Programming Languages and Systems*, 6(3):370–379, July 1983.

[4] Ken Arnold and James Gosling. *The Java Programming Language*. The Java Series. Addison Wesley, 2 edition, 1997.

[5] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: a systematic approach*. Spring-Verlang, 1998.

[6] Rajive Bagrodia. Synchronization of asynchronous processes. *ACM Transactions on Programming Languages and Systems*, 11(4):585–597, October 1989.

[7] G. Berry. Preemption in concurrent systems. In *Proceeding of FSTTCS*, page 781. Springer-Verlag LNCS, Arpil 1993.

[8] Jim Beveridge and Robert Wiener. *Multithreading Application in Win32*. Addison Wesley Developers Series. Addison Wesley, 1 edition, 1997.

[9] Toby Bloom. Evaluating synchronization mechanisms. In *Proceedings of the Seventh Symposium on Operating System Principles*, pages 24–32, December 1979.

[10] T. Bolognesi and Ed Brinksman. Introduction to the iso specification language lotos. *Computer Networks and ISDN systems*, 14(1):January, 1987.

[11] S. M. Brien and J.E. Nicholls. Z base standard. Prg technical report monograph, Oxford University Computing Laboratory, 1992.

[12] Ana Cavalcanti and Augusto Sampaio. From CSP-OZ to Java with Processes. In *Proceddings of the Seventh Workshop on Formal Methods: Theory and Applications.* IEEE CS Press, April 2002.

[13] Arthur Charlesworth. The multiway rendezvous. *ACM Transactions on Programming Languages and Systems*, 9(2):350–366, July 1987.

[14] S. Christensen and M. H. Olsen. Action Semantics of Calculus of Communicating Systems and Communicating Sequential Processes. Technical report, Aarhus University, September 1988.

[15] Marshall P. Cline. The pros and cons of adopting and applying design patterns in the real world. *Communications of ACM*, 39, No. 10:47–49, oct 1996.

[16] Christopher Colby. Design and implementation of triveni: a process-algebraic api for threads + events. Technical report, Lucent Technologies, Bell Labs and Loyola University of Chicago., 1998.

[17] Christopher Colby. Object and concurrency in triveni: A telecommunication case study in java. In *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS).* USENIX, April 1998.

[18] Christopher Colby. The semantics of triveni: A process-algebraic api for threads + events. Technical report, Lucent Technologies, Bell Labs and Loyola University of Chicago., 1998.

[19] James Coplein. Software patterns. In *Tutorial: Foundations of Pattern Concepts and Pattern Writing.* SIGS Books & Multimedia, October 2001. Tutorial of the XV SBES - Brazilian Symposium of Software Engineering.

[20] Model Checking CSP and Z. Clemens fischer. Technical report, University of Oldenburg, 1997.

[21] Ian Schechter Daniel Jackson and Ilya Shlyakhter. Alcoa: The alloy constraint analyzer. Technical report, Massachusetts Institute of Technology, 1998.

[22] Alan Cameron Wills Desmond Francis D'Souza. *Objects, Components, and Frameworks with UML: The Catalysis Approach.* Object Technology Series. Addison Wesley, December 1998.

[23] Douglas Schimidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Network Object*, volume 2 of *Willey Series in Software Design Patterns.* John Willey & Sons, 2000.

[24] Sophia Drossopoulou and Susan Eisenbach. Machine-checking the java specification: Proving type safety. Technical report, Technische Universitat Munchen, 1996.

[25] Sophia Drossopoulou and Susan Eisenbach. Towards an operational semantics and prof ot type soundness for java. Technical report, Imperial College of Science, Departament of Computing, 1997.

[26] Guy Eddon and Henry Eddon. *Inside Distributed COM*. Microsoft Programming Series. Microsoft Press, 1 edition, 1998.

[27] Wofram Schulte Egon Borger. A programmer friendly modular definition of the semantics of java. Technical report, Universita di Pisa I-56125 Pisa, Italy; Universitat Ulm D-89069, Ulm, Germany, 1997.

[28] Ralph Jonhson Erich Gamma, Richard Helm and John Vlissides. *Design Pattern Elements of Reusable Object-Oriented Software*. Adison Wesley, 1 edition, 1995.

[29] C. Fischer. How to combine z with a process algebra. *ZUM'98: The Z formal Specification Notation, Spring-Verlang*, 1998.

[30] Clemens Fischer. Combining csp and z. Technical report, University of Oldenburg, 1997.

[31] Clemens Fischer. *Combination and Implementation of Process and Data: from CSP-OZ to Java*. PhD thesis, University of Oldenburg, January 2000.

[32] Clemens Fischer. JASS - Java with Assertions Framework, 2000. http://theoretica.informatik.uni-oldenburg.de/~jass/.

[33] Formal Methods (Europe) Ltd. *FDR User's Manual version 2.28*, 1997.

[34] Formal Methods (Europe) Ltd. *PROBE Users Manual version 1.25*, 1998.

[35] Leonardo Freitas. Action Semantics of CSPm. Technical report, UFPE, Semantics of Programming Languages Discipline Project, available at http://www.cin.ufpe.br/~ljsf/pub/ascsp.ps, November 2000. Unpublished.

[36] Leonardo Freitas. The Backtrack Problem — a study of the internals of JCSP and CTJ CSP libraries. Technical report, UFPE, individual project in software engineering, November 2000. Unpublished.

[37] Leonardo Freitas. DASCO Prototype (in Java), April 2001. http://www.cin.ufpe.br/~ljsf/java-csp/dascoJ.

[38] Leonardo Freitas. Jack - a process algebra implementation for java home page, December 2001. http://www.jackcsp.hpg.com.br.

[39] Leonardo Freitas. *A Vida é uma Ordem? A Ordem é a Vida!* not yet published, dec 2001.

[40] Leonardo Freitas. Jack process network representaion — how jack solves backtracking and multisynchronization. Technical report, Universidade Federal de Pernambuco, http://www.jackcsp.hpg.com.br/pub, March 2002. Draft version.

[41] Leonardo Freitas. Jack tutorial part 1 — user guide. Technical report, UFPE, February 2002. http://www.cin.ufpe.br/~ljsf/java-csp/jack-tutorial.ps, not yet published, draft version.

[42] Leonardo Freitas. Jack tutorial part 2 — developers guide. Technical report, UFPE, February 2002. http://www.cin.ufpe.br/~ljsf/java-csp/jack-tutorial2.ps, not yet published, draft initial version.

[43] Leonardo Freitas. Jdasco detailed description. Technical report, Universidade Federal de Pernambuco, http://www.jackcsp.hpg.com.br/pub, January 2002. Draft version.

[44] Leonardo Freitas. Study of JValue version 0.6.5 design and source code, January 2002. http://www.cin.ufpe.br/~ljsf/pub.

[45] Leonardo Freitas and Adolfo Duran. CSP-Z Specification of Telemetry and Telecommand modules of SACI-I: the Brazilian Satellite project (in Portuguese). Technical report, Universidade Federal de Pernambuco, December 2000.

[46] Sigmund Freud. *O Futuro de uma Ilusão*. Imago, 1995.

[47] Erich Gamma. Junit - a software test library version 3.7. http://www.junit.org, 2001.

[48] Erich Gamma. Junit v.3.7 cookbook. http://www.junit.org, 2001.

[49] Wiek Vervoort Gerald Hilderink, Jan Broenink and Andre Bakkers. Communicating java threads. Technical report, University of Twente, The Netherlands, http://www.rt.el.utwente.nl/javapp, 1999.

[50] G.H.Hilderink. Communicating threads for java tutorial for the csp package version 0.9, revision 10. Technical report, University of Twente, The Netherlands, http://www.rt.el.utwente.nl/javapp, September 1998.

[51] G.H.Hilderink. A Distributed Real-Time Java System Based on CSP. *ISORC 2000 - The Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 400–407, March 2000.

[52] G.N.Buckley and A.Silberschatz. An effective implementation for the generalized input-output of csp. *ACM Transactions on Programming Languages and Systems*, 5(2):223–235, April 1983.

[53] Michael Goldsmith. *Programming in OCCAM 2*. Prentice Hall, 1 edition, 1988.

[54] James Gosling and Frank Yellin. *The Java Application Programming Interface: Core Packages*, volume 1 of *The Java Series*. Addison Wesley, 1996.

[55] James Gosling and Frank Yellin. *The Java Application Programming Interface: Window Toolkit and Applets*, volume 2 of *The Java Series*. Addison Wesley, 1997.

[56] Ivar Jacobson Grady Booch, James Rumbaugh. *The Unified Modeling Language User Guide*. Object Technology Series. Adisson Wesley, 1998.

[57] Mark Grand. *Patterns in Java*, volume 1. John Wiley & Sons, 1998.

[58] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, January 1991.

[59] G.H. Hilderink and E.A.R. Hendriks. Concurrent Threads in Java - CTJ v0.9, r17, September 2000. http://www.rt.el.utwente.nl/javapp.

[60] C.A.R. Hoare. Communicating Sequential Process. *Communications of ACM*, 21(8):666–677, August 1978. Seminal Paper.

[61] C.A.R. Hoare. *Communicating Sequential Process*. Prentice Hall, 1985.

[62] C.A.R Hoare and Hi Jfeng. *Unifying Theories of Programming*. Prentice Hall, 1998.

[63] Cay S. Horstmann and Gary Cornell. *Core Java 2 - Advanced Features*, volume 2 of *The Sun Microsystems Press Java Series*. Sun Microsystems Press, 1 edition, 1999.

[64] Cay S. Horstmann and Gary Cornell. *Core Java 2 - Fundamentals*, volume 1 of *The Sun Microsystems Press Java Series*. Sun Microsystems Press, 1 edition, 1999.

[65] Marjorie Russo Isabelle Attali, Denis Caromel. A formal executable semantics for java. Technical report, Univ. Nice Sophia Antipolis, France, http://www.inria.fr/croap/java, 1997.

[66] Daniel Jackson. An intermediate design language and its analysis. *ACM SIGSOFT*, November 1998.

[67] Daniel Jackson. Alloy language and alloy constraint analyzer. http://sdg.lcs.mit.edu/alloy, 2000.

[68] Daniel Jackson. Automating first-order relational logic. In *Proceedings of the ACM Foundations on Software Engineer*, 2000.

[69] Daniel Jackson. Alloy: A lightweight object modelling notation. Technical report, Massachusetts Institute of Technology, November 2001.

[70] Raj Jain. *The Art of Computer Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Willey Professional Computing. John Willey & Sons, 1992.

[71] Apache Inc. Jakarta Project Management Committee. The Jakarta Project - Jakarta Commons Framework, March 2001. http://jakarta.apache.org/commons.

[72] Guy Steele James Gosling, Bill Joy and Gilad Bracha. *The Java Language Specification: Online Draft Version*. Java Series. Adisson Wesley, 2 edition, 2000.

[73] Douglas Schmidt James O. Coplein, editor. *Pattern Language of Program Design*, volume 1 of *Software Pattern Series*. Addison Wesley, 1995.

[74] Grady Booch James Rumbaugh, Ivar Jacobson. *The Unified Modeling Language Reference Manual*. Object Technology Series. Addison Wesley, 1998.

[75] Mark Bull Jan Obdrzalek. Jomp application program interface version 0.1 draft. Technical report, Faculty of Informatics, Masaryk University and EPCC, University of Edinburgh, August 2000.

[76] Norman L. Kerth John M. Vlissides, James O. Coplein, editor. *Pattern Language of Program Design*, volume 2 of *Software Pattern Series*. Addison Wesley, August 1996.

[77] Mark Kambites. Java openmp. Technical report, The University of Edinburgh, EPCC-SS99-05, May 1999.

[78] R.Greg Lavender and Douglas Schimidt. *Active Object - An Object Behavioural pattern for Concurrent Programming*, chapter 2, pages 483–500. Adisson Wesley, 1996.

[79] Doug Lea. CPJ - Concurrent Programming in Java Home Page. http://gee.cs.oswego.edu/dl/concurrency.

[80] Doug Lea. Java coding conventions. http://gee.cs.oswego.edu/dl/, 1999.

[81] Doug Lea. *Concurrent Programming in Java*. The Java Series. Addison Wesley, 2 edition, February 2000.

[82] Hermano Moura Leonardo Freitas, Ana Cavalcanti. Action Semantics of CSPm. In *Proceedings of V Formal Methods Workshop of XVII SBES in Rio de Janeiro*, October 2001.

[83] Fuyau Lin. A formalism for specifying communicating processes. *ACM Transactions on Programming Languages and Systems*, 1993.

[84] Jeff Magee and Jeff Krammer. *Concurrency: State Models & Java Programs*. Worldwide Series in Computer Science. Addison Wesley, April 1999.

[85] Jeff Magee and Jeff Krammer. Fsp - finite state process and ltsa analyzer home page, 1999. http://www-dse.doc.ic.ac.uk/concurrency.

[86] Satoshi Matsuoka and Akinori Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press, 1993.

[87] Ciaran McHale. *Synchronisation in Concurrent, Object-oriented Languages: Expressive Power, Genericity and Inheritance*. PhD thesis, Department of Computer Science, Trinity College, Dublin, 1994.

[88] Paul E. McKenny. Selecting locking primitives for parallel programming. *Communication of ACM*, 39(10):75–82, October 1996.

[89] Bertrand Meyer. Applying design by contract. *Computer (IEEE)*, 25(10):40–51, October 1992. also available on line at http://www.eiffel.com/services/training/seminars.

[90] Bertrand Meyer. *Object-Oriented Software Construction*. Series in Computer Science. Prentice Hall, 2 edition, 1997.

[91] Sun Microsystems. Java coding conventions. http://java.sun.com/products/jdk, 2000.

[92] Sun Microsystems. Java doc homepage. http://java.sun.com/products/jdk/javadoc, 2000.

[93] R. Milner. A Calculus for Communicating Systems. *LNCS 92*, 1980.

[94] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[95] C. Morgan and T. Vickers. On the refinement calculus. *Spring-Verlang*, 1992.

[96] Carol Morgan. *Programming from Specifications*. Prentice Hall, 2 edition, 1994.

[97] P. Mosses. *Action Semantics*. Number 26 in Cambridge Tracts of Theorical Computer Science. Cambridge University Press, 1 edition, 1992.

[98] Alexandre Mota. Modeling CSP as LTSs using Action Semantics. Technical report, Semantics of Programming Language, available at http://www.cin.ufpe.br/~lmf, November 1996.

[99] Alexandre Mota. Formalization and Analysis of the SACI-1 micro satelite in CSP-Z. Master's thesis, UFPE, 1997. in Portuguese.

[100] Alexandre Mota and Augusto Sampaio. Model checking csp-z. *Science of Computer Programming, Elsivier*, 40:59–96, 2001. www.elsevier.nl/locate/scico.

[101] Brian Foote Neil Harrison and Hans Rohnert, editors. *Pattern Languages of Program Design*, volume 4, chapter 8, pages 111–132. Adisson Wesley, 1999.

[102] Hans Rohnert Neil Harrison, Brian Foote, editor. *Pattern Language of Program Design*, volume 4 of *Software Pattern Series*. Addison Wesley, December 1999.

[103] Friederich Nietzsche. *Genealogia da Moral: Uma Polêmica*. Companhia das Letras, 1998. Tradução: Paulo César Souza.

[104] Jan Obdrzalek. Openmp for java. Technical report, The University of Edinburgh, EPCC-SS-2000-08, August 2000.

[105] Parasoft Corporation. *Parasoft JTest v. 3.3 User's Guide*, April 1999. http://www.parasoft.com.

[106] L.C. Paulson. *ML for the working Programmer*. Cambridge University Press, 1 edition, 1991.

[107] P.D.Austin and P.H.Welch. Java Communicating Sequential Process - JCSP, August 2000. http://www.cs.ukc.ac.uk/projects/ofa/jcsp/.

[108] G.D. Plotkin. A structural approach to operational semantics. *Lecture Notes DAIMI FN-19, Aarhus University, Denmark*, 1991.

[109] Terry Quatrani. *Visual Modeling with Rational Rose 200 and UML*. Object Technology Series. Addison Wesley, February 2000.

[110] Dirk Riehle. The event notification - integrating implicitly invocation with object orientation. In *Theory and Practice of Object Systems*, volume 2(1), pages 43–52. UBILAB, Union Bank of Switzerland, 1996. http://www.riehle.org.

[111] Dirk Riehle. JValue History PDF Slides, 1997. http://www.jvalue.org.

[112] Dirk Riehle. JValue Design PDF Slides, 1998. http://www.jvalue.org.

[113] Dirk Riehle. *Framework Design: A Role Modeling Approach*. PhD thesis, Swiss Federal Institute of Technology Zurich, Universität Hamburg, 2000.

[114] Dirk Riehle. Java programming guidelines. http://www.riehle.org/java/programming/index.html, 2000.

[115] Dirk Riehle. Method properties in java. Technical Report 4, SKYVA International, Java Report, May 2000. http://www.skyva.com.

[116] Dirk Riehle. Method types in java. Technical report, SKYVA International, Java Report, February 2000. http://www.skyva.com.

[117] Dirk Riehle. JValue Framework version 0.6.5 - a library for Value objects in Java, 2001. http://www.jvalue.org.

[118] Dirk Riehle and Erica Dubach. Working with Java Interfaces and Classes: How to maximize design and code reuse in the face of inheritance. *Java Report*, 10(4):34, October 1999.

[119] Dirk Riehle and Erica Dubach. Working with Java Interfaces and Classes: How to separate interfaces from implementations. *Java Report*, 7(4):35–46, July 1999.

[120] Dirk Riehle and Thomas Gross. Role model based framework design and integration. In *Conference on Object-Oriented Programming Systems and Applications (OOPLSA)*, pages 117–133. ACM Press, 1998.

[121] Linda Rising. *The Pattern Almanac 2000*. Software Pattern Series. Addison Wesley, May 2000.

[122] Dirk Riehle Robert Martin and Frank Buschmann, editors. *Pattern Languages of Program Design*, volume 3, chapter 15, pages 261–276. Adisson Wesley, 1998.

[123] Frank Buschmann Robert Martin, Dirk Riehle, editor. *Pattern Language of Program Design*, volume 3 of *Software Pattern Series*. Addison Wesley, June 1998.

[124] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1 edition, 1997.

[125] Bryan Scatergood. A parser csp. Technical report, Oxford University, December 1992.

[126] B. Scattergood. *The Semantics and Implementation of Machine Readable CSP*. PhD thesis, University of Oxford, The Queen's College, 1998.

[127] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Inc., 1 edition, 1986.

[128] Douglas Schmidt. ACE - The Adaptative Communication Environment Project. http://www.cs.wustl.edu/~schmidt/patterns-ace.html.

[129] Fred B. Schneider. Synchronization in distributed programs. *ACM Transactions on Programming Languages and Systems*, 4(2):179–195, April 1982.

[130] Steve Schneider. *Concurrent and Real-Time Systems, The CSP approach*. World Wide Series in Computer Science. John Wiley & Sons, 1 edition, 2000.

[131] Ricardo Jimenez Peris Sergio Arevalo and Marta Patino Martinez. Multithreaded Rendezvous: A design Pattern for Distributed Rendezvous. In *Proceedings of the 1999 ACM symposium on Applied computing*, pages 571–579, February 1999.

[132] António Manuel Ferreira Rito Silva. DASCO Project (in C++), June 1997. http://www.esw.inesc.pt/~ars/dasco.

[133] António Manuel Ferreira Rito Silva. *Concurrent Object-Oriented Programming: Separation and Composition of Concerns using Design Patterns, Pattern Languages, and Object-Oriented Frameworks*. PhD thesis, Instituto Superior Técnico, Universidade Técnica de Lisboa, March 1999.

[134] Graeme Smith. *The Object-Z Specification Language*. Kluwer Academic Publisher, 2000.

[135] Rational Software. Rational rose version 2000. http://www.rational.com, 2000.

[136] J. M. Spivey. *Understanding Z: a specification language and its formal semantics*. Cambridge University Press, 1988.

[137] J.M. Spivey. *The Z Notation: a Reference Manual*. Prentice Hall, 2 edition, 1992.

[138] Umar Syyid. The adaptative communication environment tutorial. Technical report, Hughes Network Systems, January 2000.

[139] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 1992.

[140] Alexander Terekhov. Double check lock pattern with tls, 2001. http://jcp.org/jsr/detail/133.jsp or http://www.cs.umd.edu/~pugh/java/memoryModel/.

[141] Kathy Walrath and Mary Campione. *The JFC Swing Tutorial: A guiding to constructing GUIs*. The Java Series. Addison Wesley, 1998.

[142] D. Watt. *Programming Languages Syntax and Semantics*. Prentice Hall, 1991.

[143] David Watt and Deryck F. Brown. *Programming Language Processors in Java: Compilers and Interpreters*. Prentice Hall, 1 edition, 2000.

[144] P.H. Welch. Java Threads in the Light of occam/CSP. In P.H. Welch and A.W.P.Bakkers, editors, *Architectures, Languages and Patterns for Parallel and Distributed Applications (Proceedings of WoTUG-21)*, volume 52 of *Computer Systems Engineering Series*. IOS Press (Amsterdam), April 1998. ISBN 90 5199 391 9.

[145] P.H. Welch. Kyoto conference talk. Power point slides at http://www.cs.ukc.ac.uk/projects/ofa/jcsp/, 2000.

[146] William Wheil. The impact of recovery in concurrency control. *Journal of Computer and System Sciences*, 47(1):157–184, 1993.

[147] J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice Hall, 1996.

[148] Jim Woodcock and Ana Cavalcanti. A Concurrent Language for Refinement. *5th Irish Workshop on Formal Methods*, 2001.

[149] Jim Woodcock and Ana Cavalcanti. The semantics of circus: a concurrent language for refinement. Technical report, Oxford University and UFPE, 2001.