João Carlos Tanganho de Sousa

Mestrado em Engenharia Informática

Parallel Run-Time for CO-OPN

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

Orientadores : João Manuel Santos Lourenço, Prof. Auxiliar,
Universidade Nova de Lisboa
Vasco Amaral, Prof. Auxiliar,
Universidade Nova de Lisboa

Júri:

Presidente:	Manuel Próspero dos Santos
Arguente:	Salvador Luís Bethencourt Pinto Abreu
Vogal:	João Lourenço

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Setembro, 2009

Parallel Run-Time for CO-OPN

Acknowledgements

For everything they provide me, i would like to thank my family. For every moment spent laughing, i would like to thank my friends. For the guidance they gave me, i would like to thank my university advisors.

Abstract

Domain Specific Modeling (DSM) is a methodology to provide programs or system's specification at higher level of abstraction, making use of domain concepts instead of low level programming details. To support this approach, we need to have enough expressive power in terms of those domain concepts, which means that we need to develop new languages , usually termed Domain Specific Languages (DSLs).

An approach to execute specifications developed using DSLs goes by applying a model transformation technique to produce a specification in another language. These transformation techniques are applied sucessively until the specification reaches a language with an implemented run-time. The language named Concurrent Object-Oriented Petri Nets (CO-OPN) is being used successfully as a target language for such model transformation techniques.

CO-OPN is an object-oriented formal language for specifying concurrent systems, that separates coordination from computational tasks. CO-OPN offers mechanisms to define the system structure and behavior, and like DSLs, relieves the developer from stipulate how that structure and behavior are attained by the underlying system.

The currently available code generator for CO-OPN only produces sequential code, despite of this language potential of expressing specifications rich in concurrent behavior. The generated sequential code can be executed either in a Sequential Run-Time or in the step simulator, which is part of CO-OPN Builder IDE. The generation of sequential code turns out to be an adversity to CO-OPN application since concurrent specifications cannot be executed in parallel and therefore this languages potential is not fully exploited. This dissertation aims at filling this CO-OPN's execution gap, through the development of a Parallel Run-Time. The new Run-Time is achieved through the adaptation of the sequential code generator and actual execution support mechanisms. In this manner, all concurrent specifications that target CO-OPN benefit from thread safe code, ready for execution in parallel and distributed environments, relieving the developer from delving into parallel programming details.

By guaranteeing a safe execution environment, CO-OPN becomes an alternative to the way parallel software is nowadays developed.

Keywords: Domain Specific Modeling; Domain Specific Languages; Concurrent Object-Oriented Petri Nets (CO-OPN); Thread Safe Parallel Run-time.

Resumo

A Modelação Específica do Domínio (MED) é uma metodologia que abstrai o processo de desenvolvimento dos detalhes da implementação, utilizando os conceitos do domínio em vez de terminologia ligada à implementação. Para suportar este conceito, é necessário existir expressividade suficiente em termos dos conceitos do domínio, o que implica que é necessário desenvolver e criar novas linguagens de programação, geralmente referidas como Linguagens de Domínio Específico (LDE).

Uma abordagem para executar as especificações desenvolvidas através das LDE passa pela aplicação de uma técnica de transformação de modelos que produz uma especificação numa outra linguagem. As técnicas de transformação de modelos são aplicadas recursivamente até ser atingida uma especificação definida numa linguagem com um ambiente de execução devidamente implementado. A linguagem conhecida como Concurrent Object-Oriented Petri Nets (CO-OPN), está a ser actualmente utilizada como linguagem alvo destas técnicas de transformação de modelos.

A CO-OPN é uma linguagem formal orientada para objectos e que modela sistemas concorrentes. A linguagem separa as tarefas de coordenação das tarefas de computação e oferece mecanismos para definir a estrutura do sistema e o seu comportamento. Contudo, tal como as LDE, afasta o programador dos detalhes relativos à implementação da estrutura e ao comportamento do sistema.

O actual gerador de código para a CO-OPN apenas produz código sequencial, apesar das especificações poderem expressar comportamentos concorrentes. O código gerado pode ser executado num ambiente de execução sequencial ou num simulador passo-a-passo, oferecido pelo ambiente de desenvolvimento designado por *CO-OPN Builder*. A geração de código apenas sequencial é uma desvantagem para o CO-OPN pois as especificações concorrentes não podem ser executadas em paralelo, pelo que o potencial desta linguagem não é totalmente aproveitado.

O objectivo desta dissertação é solucionar este problema através do desenvolvimento de um ambiente de execução paralelo. A solução proposta é baseada na adaptação do

gerador de código existente e alteração dos mecanismos de suporte à execução actuais. Desta forma, todas as especificaões concorrentes que fazem da CO-OPN a linguagem alvo, beneficiariam de um código seguro para uma execução concorrente, libertando o programador dos detalhes da programação paralela.

Ao garantir um ambiente de execução seguro, a CO-OPN torna-se uma alternativa à forma como o desenvolvimento de software paralelo é feito hoje em dia.

Palavras-chave: Modelação Específica do Domínio; Linguagens Específicas de Domínio; Concurrent Object-Oriented Petri Nets (CO-OPN); Ambiente de execução seguro para execuções concorrentes.

Contents

List of Figures

List of Tables

1

Introduction

1.1 Overview and Motivation

Computational problems may have their solution specified in conventional Turing complete programming languages, designed to be general and expressive enough. Typically, one of the first tasks after the solution design is to choose a programming paradigm and language to implement the desired solution. While some programmers turn the solution implementation into a straight-forward task, others do not. For instance, there are occasions where the staff responsible for the implementation of the solution is not proficient in computer programming, e.g., physicists. Even in the case of experienced programmers, it can happen that the semantic gap between the concepts in the mind of the domain expert and the language used to implement the solution is too wide [KT08].

Domain Specific Modeling (DSM) resorts to Domain Specific Languages (DSLs) [vDKV00], in order to tackle specific software problems which occur frequently. This approach focuses the software development process on the use of domain concepts, abstracting the process from implementation details.

The DSM approach can be specially successful when is used by domain specialists with low or no skills in programming. The specialists would use a language with specific terminology to produce models which would be translated to a target programming language, e.g., Java, or to some other form of high level specification.

An example of an effective application of such modeling techniques is the project entitled Building Adaptive Three-dimensional Interfaces for Critical Complex Control Systems (BATIC3S) [RA07]. Its goal is the creation of a comprehensive methodology for semiautomatic generation of 3D user diagnostic interfaces for critical control systems.

BATIC3S deployed a DSL called (H)ALL [BA07]. This language has its own syntax, related

to domain concepts, however its semantics depends on another language known as Concurrent Object-Oriented Petri Nets (CO-OPN) [BBG97]. CO-OPN was carefully preferred for its characteristics, which makes it suitable to represent the structure and the behavior of a dynamic control system.

CO-OPN is an object-oriented formal language for the specification of concurrent systems. It is based on three distinctive paradigms: abstract data types, algebraic nets and coordination. Each paradigm is characterized by a different source module: Abstract Data Type, Class and Context, respectively.

CO-OPN Builder [CBU] is an IDE comprised of a set of tools which supports software development based in the CO-OPN language. Besides including a graphical environment for developing specifications, CO-OPN Builder contains a step simulator used for testing and analysis of CO-OPN specifications.

DSLs using CO-OPN as the target language may benefit from a well-formed formal semantic along with the possibility to simulate the behavior of developed specifications, in order to assure correctness of their programs [vDKV00].

1.2 Problem Statement and Goals

Figure 1.2 illustrates how DSLs are used to express specific problems. Modeled problems form what is termed a *Specification* and this specification may be transformed into another specification which follows the CO-OPN syntax and semantics. The CO-OPN code is then compiled to sequential Java code, which can be executed either in the Step Simulator or in the Sequential Run-Time.

For instance, (H)ALL modeled interfaces comprised of multiple components, each designed for receiving critical sensor data would fit nicely in an architecture where there is a thread per such component, since in that way the interface could independently receive, information from all sensors, even if they are simultaneously signaling. This ability to receive data with no delays is a major issue for Critical Control Systems.

Nowadays, CO-OPN's generates sequential thread unsafe code. Thus, in order to avoid concurrency problems, its parallel specifications are limited to behavioral analysis through a simulator. This severe execution limitation would prevent such (H)ALL interfaces from receiving simultaneous data from different sources, creating execution delays on a Critical Control System.

CO-OPN employs a modular code generation approach. This code generation approach is based in allowing the use of multiple generators in order to provide multiple code generation techniques for each CO-OPN source module type.

The definition of a Parallel Run-Time that would manage all execution details, along with CO-OPN's expressivity to deal with intricate implementation details of the interactions between system entities (like concurrency control), would extend CO-OPN's applications and offer a safe platform to develop and execute parallel software.

Besides the lack of a parallel run-time, CO-OPN also does not possess enough expressivity

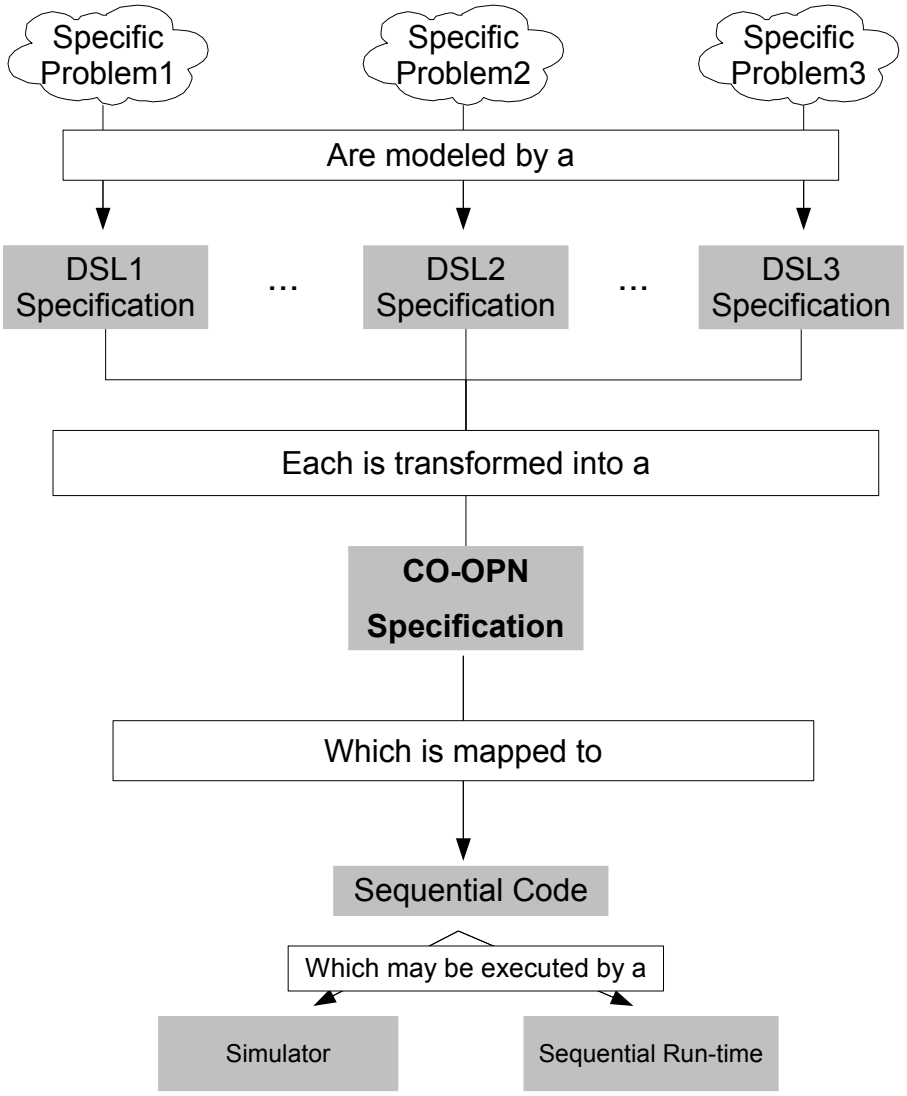


Figure 1.1: CO-OPN used as a target language for DSLs

to allow the developer to physical distribute CO-OPN specifications. In order to tackle this problem a mechanism to handle distribution must be designed and implemented.

1.3 Contributions of this Dissertation

The purpose of this work is to provide CO-OPN with a Parallel Run-Time. The new run-time must assure that interactions between concurrent entities are done correctly, and that it prevents contention issues arising from simultaneous access to the same resource.

Given that CO-OPN is a higher level language and its specifications are not focused on the details of the implementation, the generated code can target any architecture. A good balance is using concurrent components which interact through remote procedure calls. This mapping does not only allow for the execution of concurrent specifications in distinct physical machines but also in tightly coupled architectures, assuming that they have a TCP/IP stack.

Clarifying and summarizing, the main contributions of this dissertation are:

- *Definition of a concurrent execution policy which respects CO-OPN semantics* — The new policy combines multi-threading with synchronization mechanisms in order to insure CO-OPN semantics;
- *Definition of a architectural language named CO-OPNArch* — This language allows the developer to specify the physical location of CO-OPN entities and therefore a true distributed system can be deployed, developed from CO-OPN sources;
- *The transformation of CO-OPN specifications into Java sources that respect the proposed concurrent execution policy* — The modular code generation approach allowed the development of an efficient and modular Parallel code generator which transforms CO-OPN sources into Java sources ready to be executed by the above policy rules;
- *Development of a Parallel Run-Time to support concurrent execution of CO-OPN specifications* — The new Parallel Run-Time possesses mechanisms to assure that the generated code for CO-OPN specifications is successfully executed and respects the execution policy;
- *Parallel Run-Time performance evaluation and comparison with the Sequential Run-Time* — This dissertation also throughly tested the Parallel Run-Time performance and compared it to the Sequential Run-Time execution results.

Figure 1.2 highlights the focus of this work and what are its contributions. The sequential code generator might still be used for other purposes, such as producing code for simulation.

1.4 Document Outline

This document is organized in the following chapters:

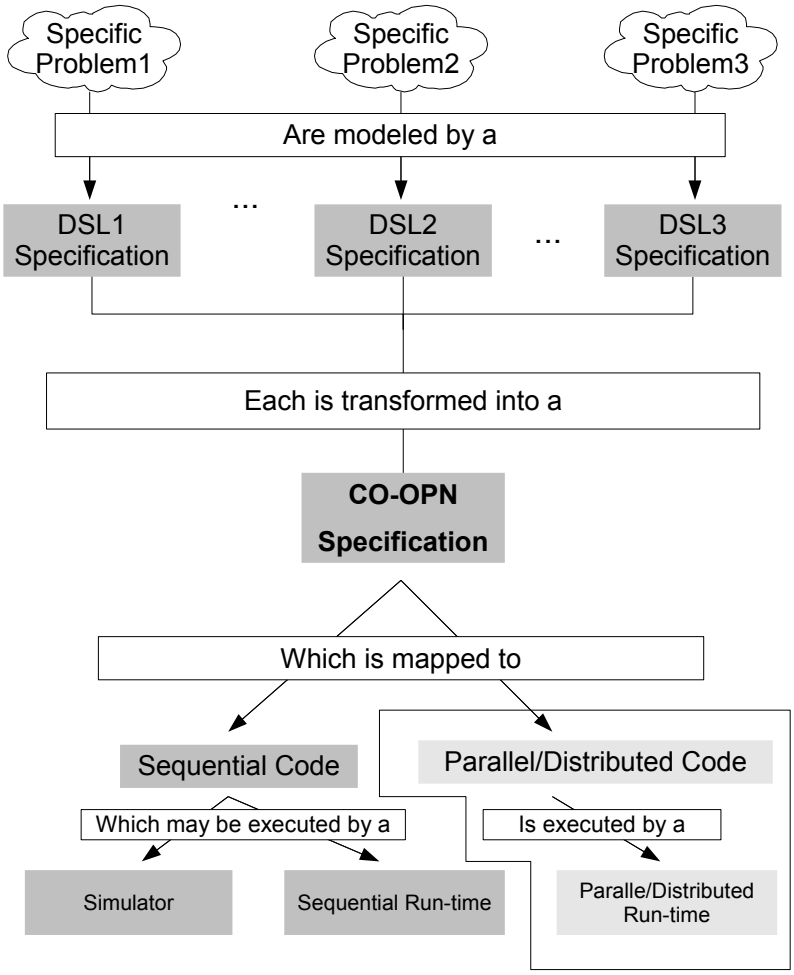


Figure 1.2: Expected Contributions of this dissertation

- **Chapter One** introduced the context for the problem under study and explained lightly how it will be solved;
- **Chapter Two** introduces the CO-OPN language;
- **Chapter Three** depicts all the related work needed to tackle the presented problem;
- **Chapter Four** describes with more detail the proposed solution;
- **Chapter Five** reports one implementation that follows the proposed solution;
- **Chapter Six** presents a use case example to evaluate the implementation described in Chapter 5. The implementation performance is compared to the Sequential Run-Time performance.
- **Chapter Seven** summarizes the results of this dissertation and describes lightly some points for future work.

2

Concurrent Object-Oriented Petri Nets (CO-OPN)

Concurrent Object-Oriented Petri Nets [BBG97] is a formal language whose purpose is to model large concurrent systems. It is based on three formalisms, that are combined in order to define CO-OPN syntax and semantic. Besides the underlying formalisms, CO-OPN also adopted the object-oriented design.

There are three components in this language: (1) Abstract Data Types (ADTs); (2) Classes; and, (3) Contexts. Each reflect one of the underlying formalisms.

2.1 CO-OPN Underlying Formalisms

CO-OPN is based on three underlying formalisms [CB01]: (1) Petri-Nets;(2) Algebraic specifications; and, (3) *Idealized Workers, Idealized Managers* coordination model. The first and second paradigm are combined in a way similar to algebraic nets [Rei91].

2.1.1 Petri-Nets

Petri-Nets [APR08] are graphical and mathematical modeling tools, used to describe discrete distributed systems. They are comprised of places, transitions and arcs.

There are two types of arcs: input and output. The former connect places with transitions and the latter transitions with places [Zim].

Places can contain tokens, and their number and type define the current state of a modeled system, also called marking.

Transitions are active components that model activities which can occur, thus changing the state of the system. Transitions are only allowed to fire if all their preconditions are fulfilled.

When the transition fires, it removes tokens from some of its input places and adds some tokens to some of its output places. The number of tokens removed/added depends on the cardinality of each arc. Preconditions are satisfied if the required tokens are available in the input places.

Places, tokens and arcs are used to represent a Petri-Net as a bipartite oriented graph, however as a mathematical tool it is possible to set up state equations, algebraic equations, and other mathematical models governing the behavior of systems.

Figure 2.1 is a Petri-Net model of a traffic light.

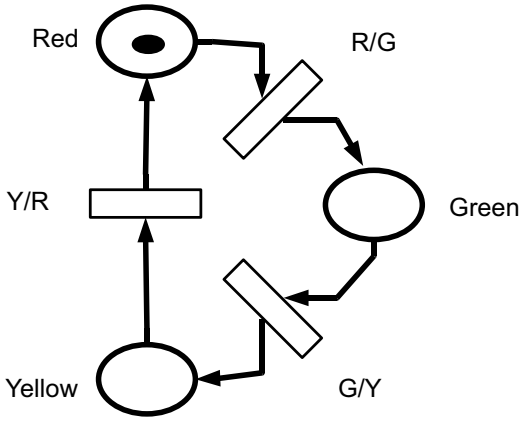


Figure 2.1: An example of a Petri-Net

In this example there are three *Places*, one for each light, and three transitions. *Places* are represented by hollow elipses, transitions by hollow rectangles. As the only token of the system is in the *Red Place*, the active traffic light is the red. Events that trigger transitions are not represented, but as regular traffic lights, a timer would be the most likely trigger.

2.1.2 Algebraic Specifications — Order Sorted Algebra

Order Sorted Algebra (OSA) [Gog92] is a generalization of *Many Sorted Algebra*, obtained by having a partially ordered set of sorts rather than merely a set . There are several variants of order sorted algebras, and some relationships between these are known [GD94].

OSA main motivation is to provide a better way of treating errors in abstract data types and the use of subsorts [Gog92] can greatly speedup certain theorem proving problems [GD94].

2.1.3 Idealized Workers Idealized Managers Model (IWIMM)

IWIM is a generic model of communication to deal with the concurrency of cooperation among any number of entities that comprise a parallel application [Arb96].

The properties that characterize this model are:

- **Compositionality**. Which it inherits from data-flow model;
- **Anonymous communication**. Receivers do not specify the source of the messages they wish to receive;

- **Separation of concerns**. Separates the communication from the computational tasks.

2.2 CO-OPN Components

Each of the paradigms, on which CO-OPN specification language is based, is represented by a different kind of source module: Abstract Data Types, Classes and Contexts, respectively.

2.2.1 Abstract Data Types (ADTs)

Abstract Data Types define data structures and their operations [CB01]. Each ADT may define zero or more of the following elements:

- **Sorts**, or names, of types;
- **Generators**, used to build values of a given sort;
- **Functions** that manipulate values;
- **Axioms** that define functions and generators.

Listing 2.2.1 exhibits an example of an ADT specified in CO-OPN.

Listing 2.1: Specification of an ADT

```
1 Adt Message;
2     Interface
3         Sort message;
4     Generators
5         ack -> message;
6         fail -> message;
7 End Message;
```

Line one declares the name of the ADT (i.e.,"Message"). The data type itself is declared in line three. It is present in the *Interface* field in order to be used by outside entities. Lines five and six define which values are allowed for the *message* data type (i.e., "ack" and "fail"). This ADT does not have any *functions* over its values.

2.2.2 Classes

A Class module describes a collection of objects with the same structure, by means of an encapsulated algebraic net. CO-OPN objects—instantiation of Classes—own an internal state and provide a set of services to the exterior environment.The only way to interact with an object is by requesting its services, therefore its internal state is never accessed directly by other entities [CB01].

A Class specification consists of:

- **An interface**, which contain services offered by the Class instances;

- **Services**, that may be internal, when they are not accessible from the exterior, or otherwise public;
- **A set of Places**, that define the state of the Class instances. Places are multi-sets of algebraic values;
- **The initial values for Places**, also called the initial marking;
- **A set of behavioral formulas** which describe the properties of public and internal services.

Both the external and internal services may assume two forms: (1) methods; or, (2) gates. The first kind is an input service while the second is an output service [CHA04].

Class Example

Figure 2.2 is a graphical representation for a simple entity named *BasicTnode*, that can be seen as a node of a network. This entity receives messages and holds them. When a request arrives, the entity forwards one of the hold messages.

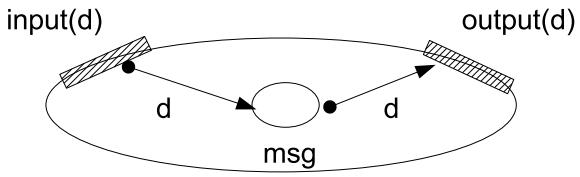


Figure 2.2: Outline of the *BasicTnode*

Figure 2.2.2 is a possible mapping to CO-OPN of the entity presented in figure 2.2. The entity is translated into a *Class* that uses the Abstract Data Type defined in Figure 2.2.1 in order to represent the messages that are saved and forwarded.

Listing 2.2: Specification of Class *BasicTnode*

```
1  Class BasicTnode;
2     Interface
3        Use Message;
4        Type basictnode;
5        Method
6           input _ : message,
7        Gate
8           output _ : message;
9    Body
10       Place msg _ : message;
11   Axioms
12       input(d) :: -> msg d;
13       output(d) :: msg d -> ;
14   Where d:message;
15 End BasicTnode;
```

In the *Interface* field, *BasicTNode*'s both public services are declared, just like it is done in a Java Interface. External module dependencies are requested in the *Use* node.

The presented *Class* has one *Place*, named *msg*, whose purpose is to save tokens typed *message*. This *Place* is the repository used by the Object to store incoming messages and to retrieve messages whenever some other entity prompt the Object to do it.

Service Definition

The *Axioms* field of Listing 2.2.2 is where the Classes' services are defined.

CO-OPN captures the abstract concurrent behavior of each modeled entity with the concurrency granularity associated to service invocations rather than to objects. Hence, a set of service calls may be concurrently performed on the same object.

The services may possess a set of operations over any number of object's internal *Places*. These operations are classified according to the type of access to *Places*:

- **Pre conditions**. Retrieval tokens from any number of *Places*. The operations may specify exactly which tokens are required;
- **Test conditions**. These operations check for the existence of tokens in the designated *Places*. Again, the operations may define which tokens it requires;
- **Post conditions**. The purpose of these operations is to modify the object marking by adding tokens to *Places*.

Pre and Test conditions are evaluated before Post conditions and may fail if:

- The needed tokens are not present in the designated *Places*;or,
- One of the target *Places* does not contain any tokens.

Post conditions never fail. Therefore, from *Places*' point of view, a successfully service execution is purely dependent on its pre and test conditions success.

Some of these operations are visible in the *Class* presented in Listing 2.2.2. The *input* method has one post condition, *msg d*, which adds one object *d* of type *message* to the place *msg*. While *output* gate has one pre condition, *msg d*, whose purpose is to remove a *message* from place *msg*. Despite test conditions are not present in the example, their syntax is as follows: *place token –> place token*. Test conditions syntax is similar to merge the syntax of a pre and a post condition.

At last, as the *Axioms* field define the behavior of the *Class*, besides containing operations over *Places*, the services may also contain synchronizations with other accessible services. A service may define complex synchronization expressions which are composed by the following synchronization operators:

Relating to the examples present in the table above, the semantic of the operators is defined as:

- **Sequence**. *Service1* must be executed before *service2*. Thus, if:

Name	Syntax	Example
Sequence	..	serviceAxiom :: service1 .. service2
Simultaneity	/ /	serviceAxiom :: service1 // service2
Alternative	+	serviceAxiom :: service1 + service2
With	None	serviceAxiom :: service1

Table 2.1: Synchronization Operators

  - *service1* fails, the synchronization expression fails without calling *service2*;
  - *service2* fails, *service1* post conditions must be undone and the synchronization expression fails.

- **Simultaneity**. *Service1* and *service2* must be executed simultaneously. CO-OPN's notion of simultaneity refers that both services have access to the same system state, and therefore their pre/test conditions must not race for the same set of tokens;

- **Alternative**. At least *service1* or *service2* must be executed successfully in order for the synchronization expression to be considered a success.

- **With**. *Service1* is the only service to be requested and it must have success.

Despite of being presented one by one, all but the *With* operator, may be combined to form complex synchronization expressions, e.g., axiomService :: (service1 // service2) + (service1 .. service2).

At last, besides having pre,test, synchronizations with other services and post conditions, a service may also have conditions or guards over any number of its arguments.

Summarizing, a service is said to be executed successfully when:

- Conditions or guards over arguments of the object's service are satisfied;

- All the service's pre and test conditions succeed;

- All synchronizations occur as specified by the synchronization expression;

- Every post condition is completed.

2.2.3 Contexts

Context form what is called the *Coordination Layer* of CO-OPN, which is, as introduced before, based in the IWIN model, suited for the formal coordination of object-oriented systems. Recalling that IWIN stands for *Idealized Workers Idealized Managers*, *Contexts* define the *Managers*, while Classes define the *Workers* [BB97].

Therefore, Contexts separate the computation from coordination tasks. They are high level entities which encapsulate Classes or other Contexts, and provide an environment to specify the inter connections between the encapsulated entities.

In addition to object inclusion, Contexts may have, likewise Classes, *Methods* and *Gates*, but not *Places*. All the encapsulated entities become invisible to the *Context's* outside environment and may only be accessed through public services that the *Context* offers.

Object Mobility

Mobility is not available in the current implementation of CO-OPN, however in [CHA04] there are some ideas on how these concept should be implemented. These ideas are depicted bellow.

A CO-OPN object and its identifier can be considered separately. An object is always encapsulated by a context, however its identifier may exist in several.

Object mobility is attained by allowing *Contexts* to manage references to objects, either local or external to a given *Context*. The classical Proxy mechanism was suggested in order to obtain an homogeneous access to objects. This mechanism, along with a *Known Objects Table*, would forward synchronizations requests to the real objects.

Context Example

To shed some light on *Context* usage, the example of Figure 2.2.2 will be continued in Figure 2.3.

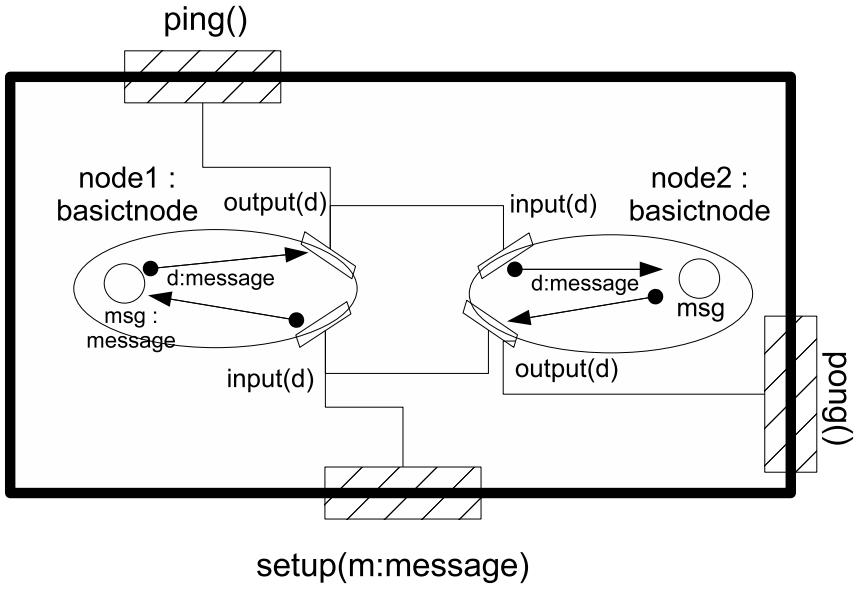


Figure 2.3: A CO-OPN Context

This *Context* has three public methods and encapsulates two objects, *node1* and *node2*. The interactions between these two objects are also defined by the *Context*.

The public method, named *ping*, which is synchronized with the *node1*'s *output* Gate. This synchronization triggers the synchronization between *node1*'s *output* Gate and *node2*'s *input* method.

```
1  Context PingPong;
2  Interface
3      Use
4          Message;
5      Methods
6          ping;
7          pong;
8          setup _ : message
9  Body
10     Use
11         BasicTNode;
12     Objects
13         node1 : basictnode;
14         node2 : basictnode;
15     Axioms
16         ping With node1.output m;
17         pong With node2.output m;
18
19         node1.output m With node2.input m;
20         node2.output m With node1.input m;
21
22         setup m With node1.input m + node2.input m;
23     Where
24         m : message;
25 End PingPong;
```

Figure 2.4: Specification of Context *PingPong*

The *pong* method, does the opposite by triggering the *node2*'s *output* Gate, which is linked with the *node1*'s *input* method.

Context's third and last public method, *setup*, supplies the "ping-pong" entities with a message to be moved around.

This *Context* translation to CO-OPN is present in Listing 2.4. The *Context* was named *Ping-Pong* since the interactions between the two encapsulated objects are similar to *Pong*, one of the earliest arcade video games.

All three methods are declared in the *Context's* interface so they can be called by an outside entity. *Ping* and *Pong* methods translation is straight forward and makes use of the *With* operator.

On the other hand, *setup* resorts to the alternative operator, which states that at least one of the nodes will get the message. The axioms present in lines 19 and 20 define the interactions between both nodes *output* and *input* services.

2.3 From CO-OPN to Java

The code generator takes CO-OPN specifications as parameters and translates their components to a set of Java classes [CB01]. These Java classes are suitable for sequential execution or to be used in the step simulator.

- **ADTs**. The process of converting ADTs to Java can be divided in two related sub-problems [CHA04]: how to represent values and how to implement functions.

  Each ADT sort is transformed into a Java class, while functions are firstly transformed in rewrite systems [**?**] and then to Java methods;

- **Classes**. The object-oriented structure of CO-OPN is maintained in the generated code. Classes and their methods are represented by Java classes and methods, which contain private members and whose purpose is to encapsulate objects' state. Gates do not have an immediate equivalence in Java, therefore they were implemented using the notion of a Java event: each is represented by an event with the same name.

  Axioms that define a method are generated inside the corresponding Java method, while axioms which define connections between gates and methods are generated in event handlers of those gates, represented as inner classes;

- **Contexts**. Likewise classes, each CO-OPN Context produces a Java class and methods. As these coordinator entities may encapsulate other entities of the specification, some of their private members are pointers to the encapsulated entities.

2.4 Simulator and Sequential Run-Time

Both CO-OPN's run-times, are described as "sequential ordered pseudo-resolution", because all the synchronizations between objects are executed sequentially and with the same order—left side followed by the right side of the statement—, it supports backtracking and partial unification of variables, common in logical programming languages like Prolog [CHA04].

2.4.1 Logical Instants

In order to sequentially execute a concurrent system, CO-OPN introduces the concept of *logical instants*. Every object or token in CO-OPN is timestamped with a *creation* and *last use* logical instant. Even the synchronization requests have a logical time, providing means to track exactly at what "time" they should be executed [CHA04].

Logical instants are consistently represented in a tree like structure, where nodes are logical instants and the root is the initial instant. Each instant can only be target of a subdivision at most one time, thus the tree of instant is binary.

Instants are internally represented by a *CoopnTransaction* class instance, which, besides identifying an instant, it can also abort the division of instants or freeze it, by resorting to transaction related methods *commit* and *abort*.

An instant is divided to represent synchronization policies, so each synchronization operator has its own corresponding instant division. Correspondences can be seen in Table 2.2.

As an example, Figure 2.5 illustrates an acceptable instant division tree representation.

Actions with time stamp *Seq1* are supposed to be performed logically before *Seq2*. *Seq2Par1* and *Seq2Par2* represent simultaneous instants. The leaf nodes, *Seq2Par1Alt1* and *Seq2Par1Alt2*,

Synchronization Operator	Type of Instant Subdivision
Alternative	Alternative
Sequence	Sequential
Simultaneity	Simultaneous
With	Included sub-instant

Table 2.2: Instant Division vs Synchronization Policies

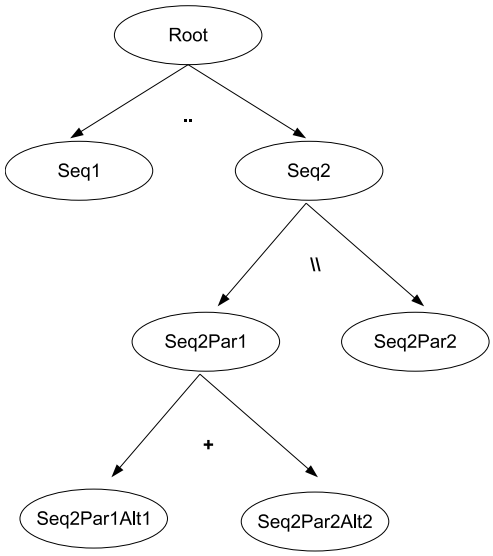


Figure 2.5: Instant Representation Tree

identify logical times which may be used by two distinct alternative operations.

CO-OPN's Sequential Run-Time serializes concurrent behaviors. Thus, there is a part of system state which is produced by sequential execution that can not be consumed, because the producer synchronization was executing logically in parallel with the consumer, e.g., service :: produce // consume. The *produce* service will be requested before *consume*, although *produce's* generated tokens may not be used by the *consume* service). Even if the state already exists, sometimes it cannot be accessed too due to simultaneous access of another operation.

By resorting to logical timestamps of conflicting operations, previous access conflicts are solved, since a token belongs to earlier instant if it has a creation time stamp that logically precedes the instant at which an operation tries to access that token. Parallel accesses follow the same line of thought, however when a synchronization has to access a token, it must also check the last use time of that token and if the token is being accessed at a parallel logical time, then only one of the accesses can succeed. In the next section some ideas are presented to clarified this issue.

2.4.2 Representing Synchronization Policies

The synchronizations between objects obey to the transactional semantics, in particular they are atomic. Hence, if an operation cannot succeed at the specified logical time then the transaction, in which that operation is included, fails. Otherwise the transaction can commit.

Complex synchronizations are grouped like nested transactions. As an example, every leaf node in the tree illustrated in Figure 2.5, would correspond to an instant at which an operation should be executed. Therefore, For every operation a transaction will be created, resulting in seven distinct transactions, which are visible in Figure 2.6

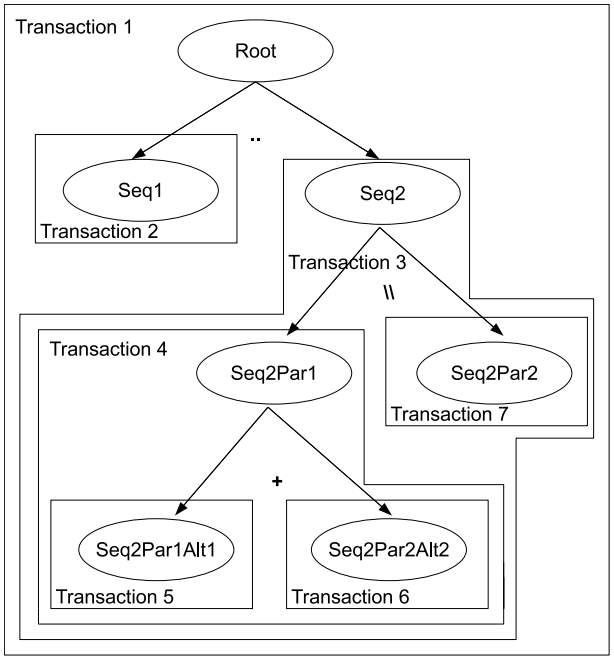


Figure 2.6: Instant Subdivision Tree and Nested Transactions

The root of the synchronization tree is encapsulated in one transaction and for each root's sub branch a transaction is also created (transaction 2 and transaction 3). This transaction creation scheme is repeated for each sub-branch.

Concurrent Accesses to Places

Places of objects are likely to be accessed at the same logical time. Therefore, CO-OPN must enforce policies to disallow inconsistent accesses to stored tokens. Access policies are represented in Tables 2.3 and 2.4, where cells are identified using a (row,column) fashion.

Table 2.3 shows CO-OPN's simultaneous access policy. Both column and row operations try to use the same token.

	Type of Access		
Type of Access	Pre condition	Post condition	Test operation
Pre condition	No	Yes	No
Post condition	No	Yes	Yes
Test operation	No	Yes	Yes

Table 2.3: Simultaneous Accesses to Places

Simultaneous operations will succeed most of the times, except when:

- Two distinct operations try to remove the same token, e.g., cell (1,1);

- One tries to check for the existence of the token while the other removes it, e.g., cells (1,3) and (3,1);
- One pre condition over a token created by a post condition at a simultaneous time, e.g., cell (2,1).

Table 2.4 is similar to table 2.3, however row operations precede column operations: i.e., cell (2,3) should be read as "Post condition" *sequence* "Test operation".

	Type of Access		
Type of Access	Pre condition	Post condition	Test operation
Pre condition	No	Yes	No
Post condition	Yes	Yes	Yes
Test operation	Yes	Yes	Yes

Table 2.4: Sequential Accesses to Places

Through a careful analysis of Table 2.4, the sequential policies are the following:

- A token cannot be removed twice, e.g. cell (1,1);
- A token cannot be removed and later be used in a test operation, e.g. cell (1,3);
- Multiple sequential test operations over the same token are valid, e.g. cell (3,3);

2.4.3 CO-OPN Non-Determinism and Backtracking

There are two sources of non-determinism to consider [CB01]: non-deterministic choice between possible matching values in Places; and, between alternative transitions.

Handling Non-Determinism in Places

This source of non-determinism was implemented by applying the Prolog execution model to Java [CHA04], but again, *CoopnTransaction* holds a very important role. Prolog execution model is present in Figure 2.7

Prolog provides methods with two entry points: "Enter" and "Redo"; and, two exit points: "Exit" and "Fail". "Enter" followed by "Exit" shows the standard Java method behavior. The method is called and returns regularly. "Redo" re-executes the method demanding a new solution — a set of tokens that satisfy the pre-, test-conditions —, which is different from the ones already returned, and finally "Fail" signals that there are no more solutions available.

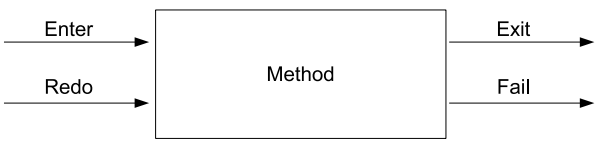


Figure 2.7: Prolog Execution Model

In both Run-Times, distinction between "Enter" and "Redo" modes is done using a *Coopn-Transaction* object that acts as the transaction identifier. If the method is called more than once with the same instant identifier then it enters the second mode otherwise it enters the first. The "Fail" mode is signaled through the Java exception *CoopnMethodNotFirableException*.

Handling Non-determinism in Alternative Transitions

The supported synchronization operators were introduced and explained in Section 2.2.2. The *alternative* operator allows the developer to define alternative synchronization branches. The choice of which branch should be tested is non-deterministic.

Although, CO-OPN has another way to define alternative transitions. By defining multiple axioms for the same service, the developer also creates alternative transitions. Recalling the *Ping-Pong* example of Listing 2.4, the *setup* service of the *Context* could go from

```
setup m With node1.input m + node2.input m;
```

to

```
setup m With node1.input m;
setup m With node2.input m;
```

The Prolog execution model is also applied to alternative transitions. Whenever a service is called in "Redo" mode, besides searching for more solutions in the already tested transition, CO-OPN also tests alternative transitions for the same service. But before doing that, the Run-Time must undo previously made modifications to object's state.

To achieve such transaction-like behavior, a *State* object is used to save local variables and the *CoopnTransaction* object, which holds the instant identifier, is used so the Run-Time may identify exactly which variables were modified and undo modifications.

3

Related Work

This chapter introduces and discusses related work that is relevant for this dissertation.

Section 3.1 briefly introduces Domain Specific Modeling and Languages so the reader may acquire some knowledge in this field.

Section 3.2 studies the Message Passing model, and more specifically the Remote Procedure Call (RPC). The focus is on the Message Passing mode, since as was discussed in Section 2, CO-OPN entities interact through service synchronization.

At last, Section 3.3 addresses concurrency control mechanisms. This section is relevant as it introduces ways to prevent inconsistent results due to concurrent accesses to the same data.

3.1 Domain Specific Modelling (DSM)

Raising the level of abstraction was always a way to improve productivity [KT08]. Advances in tradicional programming languages and modeling languages are contributing little to that goal, at least if compared to the productivity boost accomplished by changing from Assembly to third generation languages (3GLs), like FORTRAN and C. Using 3GLs, one can express the same program semantic by writing just one line instead of the several Assembly lines required otherwise [KT08].

Listings 3.1 and 3.2 illustrate how difficult it can be to write a *While* loop in Assembly comparing to write an equivalent loop in the C programming language.

3.1.1 DSM Goals

DSM makes a difference by means of focusing the software development on problem domain concepts, and therefore abstracting this process from implementation details.

Listing 3.1: *While* loop in C

```
1  int x = 5;
2  while ( x > 0 )
3      x--;
```

Listing 3.2: *While* loop in Assembly

```
1                ...     ...
2                mov     COUNTER, 5
3  WHILE         cmp     COUNTER, 0
4                jle     END_WHILE
5                mov     eax, COUNTER
6                sub     eax, 1
7                mov     COUNTER, eax
8                jmp     WHILE
9  COUNTER       dw      0
10 END_WHILE     ...     ...
```

DSM has two main purposes [KT08]. First, by specifying the solution in a language that uses problem domain terminology and rules, it raises the abstraction level. Second, it offers tools for rapid prototyping which automatically generate code in a chosen programming language.

An extensive automation of development may be achieved since the modeling language, code generator and framework code must uniquely fit the requirements of a narrow application domain.

3.1.2 Domain Specific Languages (DSLs)

A DSL is a programming language whose scope is a particular domain. They are classified as forth generation languages, for they describe what *needs* to be done, rather than *how* it should be done, like 3GLs do [DSL]. A well known example is SQL, used for database queries.

Such languages reduce the conceptual distance between the problem space and the syntax used to express it. This abstraction leads to simpler, easier and more reliable programs and programming techniques. Reliability comes from the ability to validate and optimize specifications at domain level, rather than using debuggers and tests sets. By using domain terminology, DSLs' specifications become concise and self-documented to a wide extent, partially relieving the developer from producing documentation and therefore, allowing him/her to focus in code production tasks.

DSLs can be transformed directly to *bytecode* or machine code, which is ready to be executed by the processor. Nevertheless, to aid the development of an automatic code generator, they can also be transformed to some other high level language with an already implemented machine code or *bytecode* compiler.

Despite of their advantages, DSLs do have some drawbacks [vDKV00], such as: costs of design, implementation and maintenance; possible loss of efficiency if code generation is not done properly; and, necessity to balance domain-specificity with general-purpose programming languages constructs.

3.2 Communication in Concurrent Systems

Communication in concurrent systems is used mainly to synchronized the different system entities and to share data amongst those system entities. This section will introduce the Message Passing model and a communication pattern named Remote Procedure Call that is supported by that model.

3.2.1 Message Passing

In the Message Passing model, processes communicate via messages over communication channels, and each channel provides a bidirectional connection. Therefore, it is defined as a set of processes, which only have local memory, and the transfer of data requires cooperative operations to be performed by each process [CD01]. Figure 2.10 illustrates a simple message passing system, comprised of two processes which interact through the cooperative operations *send(data)* and *receive(data)*.

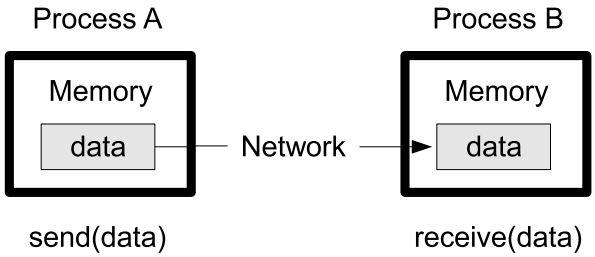


Figure 3.1: A simple message passing example

The topology of this model is defined by the pattern of connections provided by channels. It can be represented by an undirected graph in which each node describes a process and an edge is present whenever a channel exists between two nodes. These connections can the statically created or dynamically during the system progress.

Messages can be sent or received obeying to two different disciplines, either blocking or non-blocking. The first discipline requires that both the sender and receiver process waits until the message is successfully sent or received. The latter states that the routines for sending and receiving may return right after the call, having the calling process to check if the message has been correctly handled.

Examples of message passing libraries include public domain packages that do not target a specific architecture (e.g., PVM [PVM], PARMACS [PAR], MPI [MPI], etc) as well as machine dependent vendor implementations (e.g., ACL [JPH95], HP-MPI [HP-], etc). The common components of message passing libraries include:

1. **Process management routines** e.g., initialize and finalize processes, determine number of processes and process identifiers;

2. **Point-to-point communication routines** e.g., basic sends and receives between any two processes;

3. **Process group/collective communication routines** e.g., broadcast/gather/scatter operations amongst a set of processes, synchronization of processes.

3.2.2 Remote Procedure Call (RPC)

The Remote Procedure Call concept was first suggested by Birrel and Nelson in [BN83], as an inter-process communication technology that allows processes to execute a method in another address space. All the details for the remote synchronization are hidden from the programmer.

The description of the RPC model is provided by Figure 3.2.

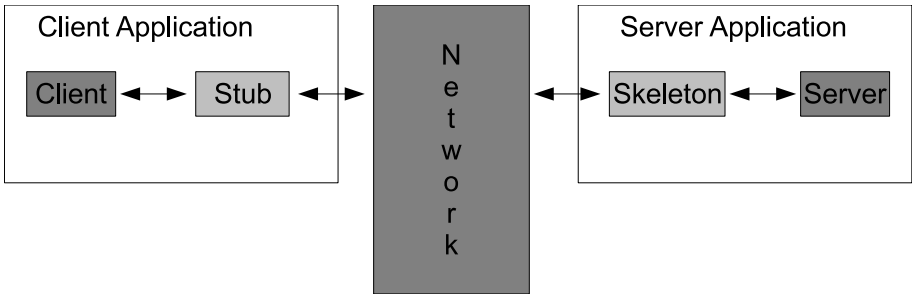


Figure 3.2: Remote Procedure Call model

There are three main components: the client application, the network and the server application. The client application is comprised by the *client* and by the client *stub*, while the server application is composed by the *server* and by the *Skeleton*, which is the server application's stub. The *Client* and the *Server* components are where the procedure is required and provided, respectively. The *Network* handles communication between both stubs.

The existence of a visible distinction between the local code and the stubs, which deal with the remote interaction details, turns RPC into a transparent mechanism [TAN95].

The client stub, packs the parameters of the procedure, sends them in a message to the server and blocks itself until the reply from the server is received. The server stub is bound with the server, typically will be blocked waiting for incoming messages. Upon receiving a message, unpacks the parameters from it and the calls the server procedure. From the server's perspective, the method was called directly by the client. The server executes the procedure and returns control to the server stub, which packs the results in a message and sends it to the client.

The client stub receives the message containing the result of the computation, unpacks it, copies the result to the client local buffer and the control is given to the client.

Figure 3.3 represents a simple example, which omits both stubs. "P1" calls "a" method on "P2", whom executes "a" and replies the result back to "P1". As referred above, none of the processes are aware of how synchronizations are handled.

Java Remote Method Invocation (JMRI)

Java Remote Method Invocation is based on the concept of RPC and extends the Java object model to provide support for distributed objects [CD01]. JRMI resorts to object serialization

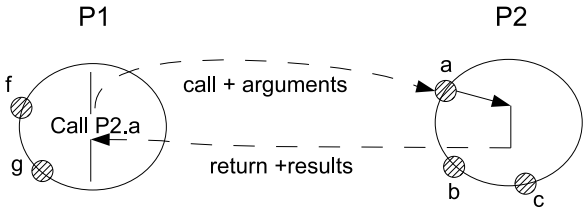


Figure 3.3: Remote Procedure Call example

to marshal — write and send — and unmarshal — read — parameters and does not truncate types, supporting true object-oriented polymorphism.

The homogeneous environment of the Java virtual machine (JVM) is assumed, therefore the system can take advantage of the Java platform's object model whenever possible.

Since remote method invocation on the same remote object may execute concurrently, the specification using this technology needs to ensure thread-safeness.

In JRMI, the network component of the RPC model uses a wire level protocol called Java Remote Method Protocol (JRMP) which runs on top of TCP/IP and is used for looking up and referencing remote objects. By using TCP/IP connections, it provides basic connectivity as well as some firewall penetration strategies. Even if the two JVMs are running on the same physical machine, the TPC/IP stack is used.

Common Object Request Broker Architecture (CORBA)

CORBA [CD01] is a standard defined by the Object Management Group, which describes an architecture for interoperability of distributed objects through remote method invocation.

One of its components is termed *Interface Definition Language (IDL)*, whose purpose is to declare interfaces for the objects, which guarantees interoperability, disregarding the language of the caller or how the callee is implemented. Existing standard mappings translate IDL to Ada, C++, C, Java, etc.

Another important aspect of CORBA is the support for mechanisms to locate objects. These mechanisms are encapsulated in the *Object Request Broker* component which assures correct method invocation and posterior result deliver to the caller object.

Figure 3.4 shows the steps that have to be taken, in order to produce two specifications which communicate with CORBA. It is also present the *Software Bus* that represents the *Network* component of the RPC model.

CORBA is a powerful architecture, on the other hand, it is time consuming and not is straight forward to develop a system using it.

Discussion: JMRI vs. CORBA

A study of these two technologies shows some degree of functional overlapping, though each has it strengths that outshine the other for particular tasks.

Bellow are depicted some factors that may be used to compare these technologies:

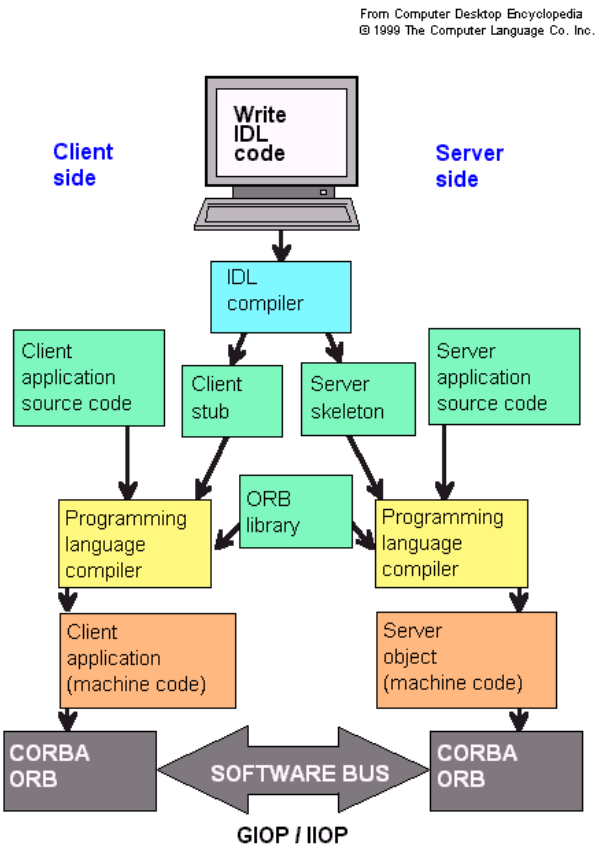


Figure 3.4: Programming with CORBA

- **Cross-Platform**. JRMI is dependent on Java implementations, and therefore, to interact with legacy systems, a Java adapter is required. CORBA specifications may be implemented in different languages and executed on different platforms, including Java.
- **One Interface, Many Implementations**. Both technologies allow for the creation of an interface and multiple implementations of that interface. In CORBA interfaces are written in IDL, whilst in JRMI they are written in Java.
- **Object Mobility**. JRMI supports sharing of objects, using object serialization. CORBA does not.
- **Easy To Master**. CORBA implementations require usually a few more steps than JRMI ones.

CORBA is a rich, extensive family of standards and interfaces, and delving into the details of these interfaces is sometimes excessive for the task at hand. In contrast, JRMI can be relatively easier to master without neglecting its functionality.

JRMI is suitable to meet this work requirements and for that, it was the chosen RPC implementation.

3.3 Concurrency Control for Management of Shared Data Contention

Concurrency Control holds the responsibility for ensuring correct results after concurrent operations. Besides correctness, it is also expected for the results to be achieved as quickly as possible.

This section introduces some mechanisms to control concurrent accesses to shared data, and *Specifications Models* as ways to test design decisions without producing an actual implementation.

3.3.1 Execution Models and Support

This section presents some of the existing mechanisms which allow sharing of resources, without compromising their consistency. Mechanisms are said to be *blocking* whenever a thread is blocked while waiting for a given resource. Motivation for *nonblocking* techniques comes from their immunity to large, unpredictable delays in thread progress often linked with the previous mechanism [MS95].

Execution Models

Execution models may be classified in two main groups: pessimistic or optimistic. In a *pessimistic scheme* resources are locked early in the data-access and are not released until the access is performed. While the *optimistic scheme* allows accesses to shared resources without any synchronization, relying on commit-time validation to ensure serializability [Her90].

Deciding whether or not to use optimistic concurrency depends on the type of system, or operation. Optimistic concurrency control is employed when conflicting accesses are expected to be infrequent, the opposite follows for the pessimistic scheme.

Blocking Mutual Exclusion

Mutual exclusion is perhaps the most prevalent form of coordination [HS08]. This mechanism is used to prevent simultaneous accesses to a common resource and such goal is achieve through Critical Sections. A Critical Section is a block of code which is typically surrounded by *Locks*, however there are several different solutions guarantee exclusive access to the protected code.

A model of a Critical Section is shown in listing 3.3. It is present that the access to the resource is protected by any mutual exclusion mechanism.

Listing 3.3: Mutual Exclusion Region

```
1 GetExclusiveAccess
2     AccessProtectedResource
3 ReleaseExclusiveAccess
```

Locks Locks are synchronization mechanisms to control simultaneous accesses to shared resources. They provide two main methods, *lock* and *unlock*. The first acquires exclusive access to the resource while the latter releases it.

These mechanisms typically require hardware support for efficient implementation, since it allows a single process to test if the lock is free, and if so, acquire the lock in a single atomic operation.

A good locking algorithm should satisfy the following properties [HS08]:

- **Mutual Exclusion**. Critical Sections of different threads do not overlap;
- **Freedom from deadlock**. One of the threads will eventually acquire the lock;
- **Freedom from starvation**. Every thread will ultimately acquire the lock.

Locks are however associated with several disadvantages [DFL02]:

- **Priority Inversion**. Occurs when a higher-priority process requires a lock owned by a lower-priority process;
- **Convoying**. Takes place when a process holding a lock is preempted by exhausting its quantum or by some kind of interrupt. Consequently, running processes requiring the lock are unable to progress;
- **Deadlock**. Happens if different processes attempt acquire the same set of locks in different orders.

Monitors Monitors are modules which encapsulate a shared data structure, its operations and a private lock [HS08]. Each access to the data structure requires previous acquisition of its internal lock.

Since these modules combine data structures and synchronizations in the same package, there is no need to ensure that every access to the data structure follows a cumbersome synchronization protocol. If besides acquiring the lock the operation has other pre conditions that are not satisfied, threads can be suspended and resumed when pre conditions are fulfilled. Suspending threads and releasing the lock while all conditions are not satisfied is important, because otherwise no other thread could access the data structure and change its state.

To make all these ideas clearer, one can think in an application composed of two threads — a producer and a consumer —, that communicate through a shared FIFO queue. The queue and its operations, *queue* and *enqueue*, would be packaged in a *monitor* which acts as a synchronization agent, serializing accesses to it. If the consumer would lock the data structure and would not release it until there were elements to consume, the application would deadlock since the producer would not ever gain access to the queue and therefore no elements would be added.

Figure 3.5 shows a linked list encapsulated by a *monitor*. The data structure methods, *ListAdd(...)* and *ListRem(...)*, are synchronized with the external methods, *Add(...)* and *Rem(...)*, provided by the *monitor*. In order to access the list, the caller process must call the external methods.

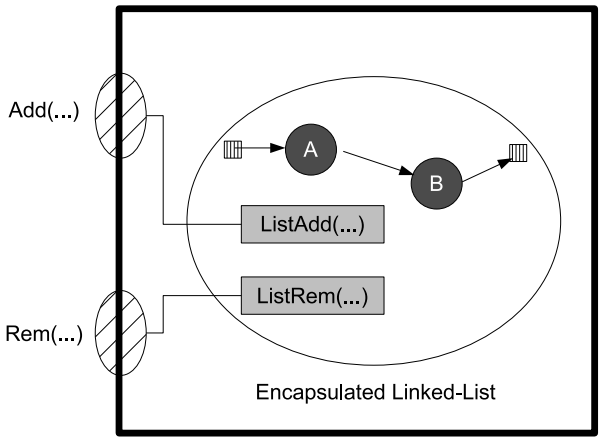


Figure 3.5: An example of a Monitor

This synchronization mechanism suffers from the following drawbacks [Sch00]: (1) Tightly coupling between object functionality and synchronization mechanisms; and, (2) Nested monitor lockout.

Item (1) relates to synchronization logic which is often closely coupled to monitors methods' functionality. Therefore it is hard to change synchronization policies without changing monitors methods' implementation.

Item (2) is a problem similar to deadlock. Having two threads, T1 and T2, and two resources, A and B, it occurs when T1 locks A and B, then releases B and waits for a signal from T2. Thread T2 requires both A and B to send T1 the signal. Therefore, one thread is waiting

for a signal, and another for a lock to be released. Like in a deadlock situation, there are two threads blocked, although, this problem cannot be avoided using lock ordering.

Monitors are present in all Java objects and they allow threads to implicitly serialize their execution through method-call interfaces and coordinate their activities via explicit *wait*, *notify* and *notifyall* operations.

Nonblocking Mutual Exclusion

Nonblocking techniques became a major break through, since they seemed to solve some of the locks' issues. These mechanisms ensure safety while refraining mutual exclusion. However, existing techniques are rarely suitable for practical usage, given that either they are too complex to implement or impose too much memory overhead.

Nonblocking algorithms can be classified according to the kind of progress guarantee that they offer [FH07]:

- **Obstruction-freedom** is the weakest guarantee: A thread is only guaranteed to make progress as long as it does not access any location concurrently with another thread. Requires an out of ban mechanism to prevent live lock;
- **Lock-freedom** adds the requirement that the system as a whole makes progress, even if there is contention. This is sufficient to avoid live lock, although it does not guarantees of per-thread fairness.
- **Wait-freedom** is the strongest guarantee, since it adds the requirement that every thread makes progress, even if it experiences contention.

The underlying core of these techniques are atomic hardware instructions, such as *compare-and-swap*, which atomically updates one or more memory locations from a set of expected values to a set of new values.

Performance considerations Under light to moderate contention, these techniques perform better than their blocking counterparts, since most of the times the required hardware instructions succeed at the first try. However under high contention environments, blocking mechanisms offer better throughput, as threads are suspended and restarted when the lock is released.

Java nonblocking API Java offers a suite of nonblocking data structures with its *java.util.concurrent* package. Just as an example, its nonblocking stack uses the Treiber's algorithm, which uses the *compare-and-swap* hardware instruction to modify the top node of the stack [MS98].

3.3.2 Specification Models

The analysis of the highly complex and dynamical character of concurrent systems may be achieved through the construction of specifications, that may be used to test the design decisions without producing an actual implementation. Some of the existing specification models

for concurrency are: PI-Calculus [MPW89], CSP [Hoa78] and Petri-Nets. The latter will be summarily introduced bellow.

There are three important properties for concurrent programs: *Liveness*, *Safety* and *Fairness* [Sis99,VVK05]. The first asserts that a program execution will eventually reach a desirable state. *Safety* properties assert that something bad never happens. The third property may be seen as a special class of *Liveness* properties, and states that a particular choice is taken sufficiently often provided that it is sufficiently often possible.

The introduced specification models lack the ability of executing the specifications, instead they solely serve the purpose of verifying and analyzing the generated specifications. CO-OPN, besides allowing the programmer to analyze the behavior of the specifications, also has an implemented run-time, which acts as a platform to execute CO-OPN specifications.

3.3.3 Concurrency Control Mechanisms Applicability

The required concurrency control for a parallel implementation of the CO-OPN execution model, goes through protecting data structures of each object from concurrent accesses, while maintaining synchronization atomicity. By using *Monitors*, which are present in all JAVA objects, resources may be shared and inter-object synchronization can be made atomic.

4

A Parallel Run-Time for CO-OPN

A Parallel Run-Time for CO-OPN provides a parallel way to execute CO-OPN specifications, filling the execution gap in which CO-OPN was bounded.

The run-time should be able to suffice all CO-OPN requirements linked with its special semantic that makes the language so interesting. This section depicts the requirements of a parallel run-time and proposes an abstract solution ready to be implemented.

4.1 Requirements for Parallel Execution of CO-OPN Specifications

When a language execution is moved from a Sequential Run-Time into a Parallel one, it is necessary to fully understand the language in order to know which parts of its execution should be parallelized and which parts of the execution run-time are likely to be endangered by contention issues.

This section depicts the core characteristics of CO-OPN which should be tackled to build a good and robust parallel run-time.

4.1.1 Synchronization Operators Semantic

The first requirement concerns the semantic of the operators used in synchronization expressions (e.g., example of a synchronization expression *leftMember SynchronizationOperator rightMember*) :

1. **Sequence**. This operator has a simple translation, since it only requires that the left member is executed successfully before the right member.

2. **Alternative**. Can have a translation similar to the *Sequence* operator, in view of the fact that the semantic of the *Alternative* states that one of the members is executed and the

other is called if the previously member fails. The sequential run-time does not support the use of the alternative operator in synchronization expressions, but the similar behavior may be attained by defining multiple axioms for the same service.

3. **Simultaneity** states that for a synchronization to have success, both its members must succeed at the same logical time;

4. **With** defines an included sub synchronization.

The *Sequence* operator is straight forward to support, the left member is called and if it is successful then the right member is called. When applying the *Sequence* operator one of three cases may happen: (1) If the first member fails, then it is unnecessary to call the second; (2) If the first member succeeds and the second does not, then it is necessary to undo changes to the marking done by the first member; (3) both members succeed in the defined order, thus the synchronizations expression succeeds. Implementing this operator should be straight forward except the transactional semantics imposed by the second case.

The *Alternative* operator may be fully supported through the definition of multiple axioms. It would be possible to support directly the usage of the alternative operator by translating the synchronization expression to multiple axioms. (e.g., `leftMember alternativeOperator rightMember` would be translated into `axiom1 := leftMember`, `axiom2 := rightMember`).

Support for the *Simultaneity* operator requires a way for the two members of a simultaneous synchronization expression to have access to the same system marking in order to execute their pre/test and post conditions in this system state.

At last, the *With* operator must be defined as a simple service request and therefore it should not raise any implementation issues.

In order to jump outside of the scope of the sequential execution, the simultaneity operator may also take advantage of multi threading by encapsulating each member in its own thread. Doing this supposedly increases the level of system performance but it also implies protecting data from concurrent accesses. The next section will shed some light on how to provide such an environment.

4.1.2 Logical Execution Semantics

The use of subdivision operators enrich CO-OPN with the notion of logical time. Some events should occur before others where some occur at the same logical instant (e.g., *Member1 SimultaneousOperator Member2*, *Member1* and *Member2* must succeed at the same logical time).

Although this notion seems rather trivial, the Sequential execution of CO-OPN required two types of management involving logical timestamps:

1. Control the access to tokens in places;

2. Manager by which order should synchronizations, defined through axioms, be executed. This type of management is achieved by following the synchronization operators semantic defined above.

The first protects tokens from simultaneous illegal accesses (e.g., two pre conditions over the same token) and hides tokens created "later´´ in the logical time line perspective from accesses done in a "previous´´ logical time.

To detect which tokens may be accessed, each element of the state keeps two logical timestamps: the creation instant and the last use instant. The creation instant is the time stamp at which the token was inserted, by a post condition, in the given place. By comparing the creation instant with the instant at which the token is trying to be used, the Sequential Run-Time can detect if this is a legal access. Furthermore, by checking the last use instant of a token, the Sequential Run-Time can also detect if the token is already taken. These comparisons were achieved by the Sequential Run-Time through traversals of the *instant tree*, where each of the nodes of this tree is a logical time stamp. This *instant tree* is generated by the subdivision operators.

Traversals of a tree are not efficient (i.e. in the worst case scenario the run-time has to traverse log (n) nodes) and since this dissertation aims at developing a parallel run-time performance issues should be a requirement. To achieve better results in instant comparison a different representation for logical instants is depicted in Section 4.2.

4.1.3 Protecting the Marking from Concurrent Accesses

In a parallel run-time, a method, which has a non empty set of pre-, test- or post-conditions, may be accessed simultaneously by two or more threads. Therefore the consistency of the objects' data structures— *Places*, in CO-OPN terms —must be guaranteed.

The accesses to *Places* may be classified in two types: (1) updates; (2) insertions.

Updates, occurs whenever a method executes its pre or test conditions. The status of the target tokens is updated to indicate that they are locked in the logical instant at which the method is executing. The implementation must also take in consideration that the same token may be used simultaneously if the accesses are performed exclusively by test conditions.

Insertions, *insertions*, never race for the same token in simultaneous accesses since two insertions are always performed in order to insert two distinct tokens. The only problem comes from the fact that two concurrent insertions may modify the status of the internal data structure where tokens are kept, and therefore, they should be serialized.

4.2 Representing Logical Instants

This dissertation proposes a different representation for the logical instants which aims at a more efficient management of Place access while respecting CO-OPN semantics. The proposed representation for logical instants is depicted bellow:

1. The root of the logical instant tree of execution will be represented by the string "1";

2. Handling subdivision operators:

- **Simultaneous** and **Alternative**, both sub nodes will inherit the representation of the father along with the suffix ".1";
- **Sequence**, the left child and the right child will inherit the father's representation plus the suffix ".1" and ".2" respectively;
- **With**, the sub node representation will be the father's plus the suffix ".1".

3. The initial state is represented by the string "0".

Although the representation for sub nodes of an *Alternative* subdivision is the same, they will not execute simultaneously in the proposed version of a Parallel Run-Time. Section 7.1 introduces some ideas on how alternative synchronizations could take more advantage of parallel execution.

The example in Figure 4.1 will serve as an explanation on how this new representation may be applied to a synchronization tree.

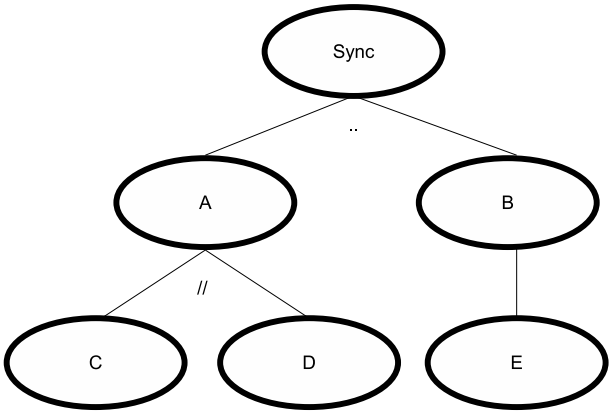


Figure 4.1: Synchronization Tree Example no. 1

The synchronization *Sync* will succeed if *B* succeeds after the synchronization *A* succeeds. While *A* succeeds if *C* and *D* succeed at the same logical time (e.g. both have access to the same state), and *B* succeeds if *E* succeeds.

The proposed representation was applied to the synchronization expression depicted in Figure 4.1 and the tree graph present in Figure 4.2 is the resulting graphical representation.

The root of the tree of the synchronization expression, *Sync*, has the representation "1". Its left child is represented with the string "1.1", where the first "1" is inherited from the root representation and the suffix ".1" is given since it is on the left side of a sequential subdivision. Coming down the tree, nodes C and D, belong to the same logical level and therefore represented by the same string "1.1.1". Both suffixes of nodes C and D are ".1" because they are created through a simultaneous subdivision. Moving to the right side of the tree, node B occurs logically after A, therefore its suffix is ".2". Node D which is the only child of B is represented by the string "1.2.1", where the ".1" comes from the fact that it is an included instant.

Since each instant has its own string representation, time stamp comparison is nothing more than comparing strings of characters. String comparison, following the alphabetic order, may have three possible outcomes:

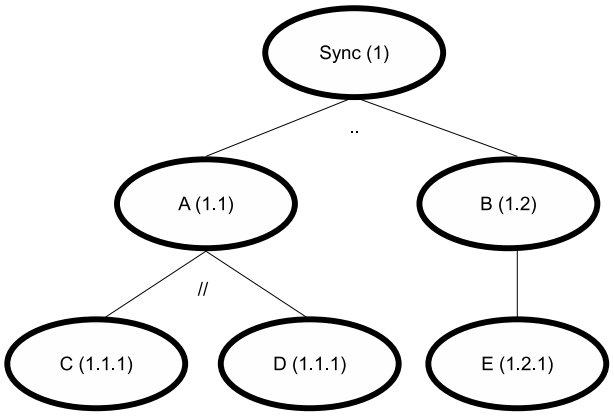


Figure 4.2: Representation for the Logical Execution no. 1

- **representation1** $>$ **representation2**, "representation1" is logically dated after than "representation2";
- **representation1 == representation2**, "representation1" and "representation2" occur at the same logical instant;
- **representation1** $<$ **representation2**, "representation1" is logically dated before than "representation2";

String comparison is linear with the number of characters of the smallest of the two strings that are being compared. Since this representation is fully *cpu bound*, it should perform better than the other representation in small, medium and even large instant trees, since it does not need further memory accesses than the ones required to read the two string representations.

In [CHA04], the author presented an idea of representing logical instants resorting to real numbers. Although the idea is interesting it was not adopted since such implementation does not guarantee correctness when facing a large synchronization tree, exposing the Run-Time to the risk of overflowing the floating point capacity of the hardware.

4.3 Enforcing Logical Execution

Executing a CO-OPN specification in a multi threaded environment without any restrictions could lead to some situations where part of the synchronization tree, that should logically execute later, is already executing. This event may only occur in specifications which contain synchronizations expressions with one or more *Simultaneous* operators. Such cases are problematic since tokens may be consumed by these synchronizations executing in the "future", when they should be available for "present" synchronizations.

To enforce the logical execution in a parallel run-time there are at least two different approaches. The synchronization tree present in Figure 4.3 is used as an example to explain both approaches.

Nodes D and E have pre conditions on the same Place, which has enough tokens for the tree to be correctly executed, but E needs a specific token to succeed (e.g. The *Place* contains a,b, D succeeds with "a" or "b" while E only succeeds with "a").

A static analysis of the Figure 4.3 shows that A and B execute at the same logical time and E should execute before D:

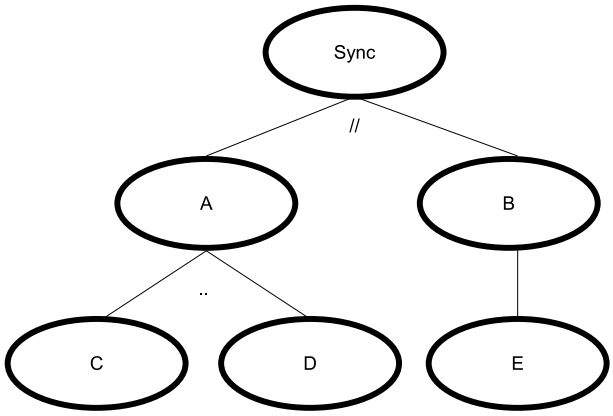


Figure 4.3: Synchronization Tree Example no. 2

4.3.1 Depth-First Execution

This technique was used in the Sequential Run-Time and in some complex synchronizations tree and depends heavily in backtracking.

Likewise a pre order depth-first traversal of a binary tree, the root of the synchronizations tree is visited and then the left subtree is traversed, followed by the right subtree.

Using depth-first execution on the synchronization tree of the Figure 4.1 the trace of the execution could be the one present in Figure 4.4. The dotted lines represent moments in task execution where the task is waiting for child task to complete.

Although, and because D and E need tokens from the same place, D may catch the token needed for E to succeed. If such situation happens the execution trace is visible in Figure 4.4. Again, The dotted lines represent moments in task execution where the task is waiting for child task to complete.

The execution would become "Sync, A, C, D, B, E, A, C, D, B, E", since "E" would fail, its failure would be propagated to "B" and therefore to "Sync". "Sync" would then call "A" again, which would interpret the call as a backtracking call. The process repeats for "A" and its two child synchronizations. After this step, "Sync" calls "B" which would recall "E" and by now, "E" could access to the desired token.

In this simple example there are twice the number of required synchronizations, a number that would grow a lot when executing a more complex synchronization tree. When there are any conflicts and if the one thread per simultaneous call enhancement is being used then the system performance would be outstanding.

Implementing this approach also requires the creation of conditions to prevent tokens pro-

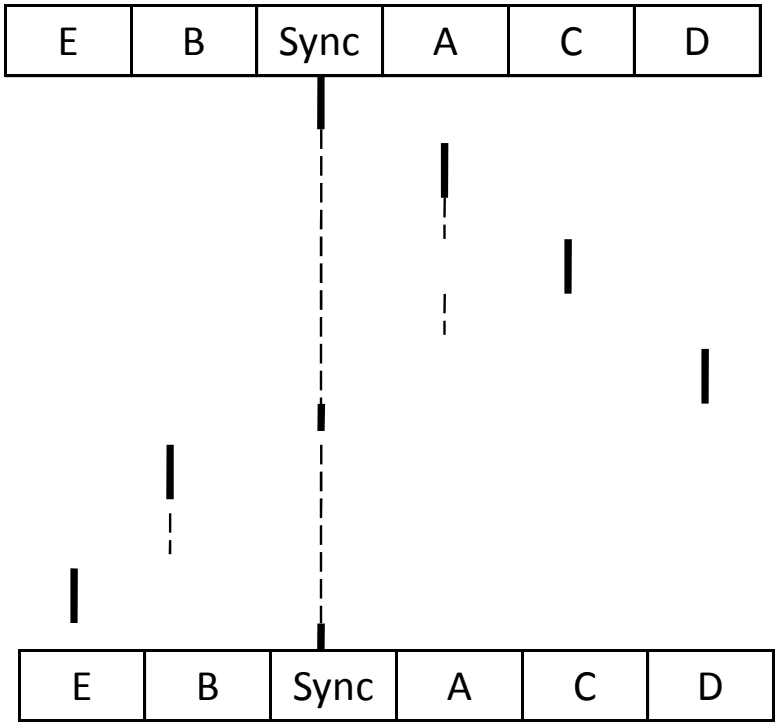


Figure 4.4: Depth-First Execution for Figure 4.3

duced logically later from being consumed by pre/test conditions that should execute logically in a previous time. The Breadth-first execution has good performances when there are no conflicts in the accesses to places, although in the other cases this approach brings out a lot of overhead due to multiple synchronizations redo.

4.3.2 Breadth-First Execution

The idea is to provide the run-time with means to detect failures in execution of CO-OPN specifications earlier by imposing restrictions on the three phases of method execution (e.g. Pre/test conditions, synchronizations, post conditions). The description bellow bounds this type of execution to a multi thread environment, but the Breadth-First execution can also be achieved by using stack manipulations like a Breadth-First tree traversal would do.

A Level by Level Approach

The figure 4.6 shows the result of applying the proposed string representation for logical instants to the synchronization tree present in Figure 4.3.

As visible "A" and "B" belong to the same logical level, "C" precedes "D" which is also preceded by "E". A desirable trace of execution would be "Sync, A // B, C // E, D", where "A // B" state that "A" executes in a parallel thread with "B". Figure 4.7 is a graphical representation of the expected trace of execution. The dotted lines represent moments in task execution where

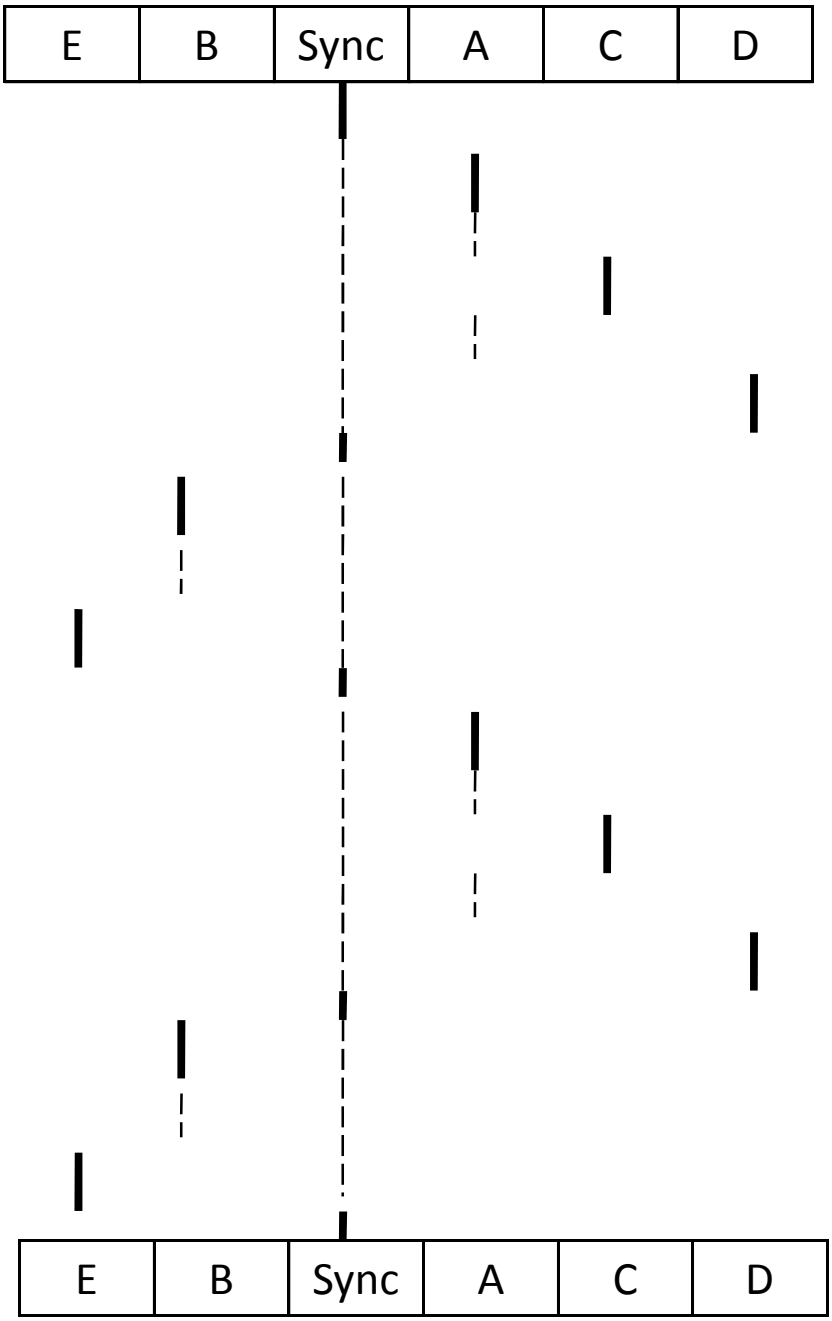


Figure 4.5: Depth-First Execution With Backtracking for Figure 4.3

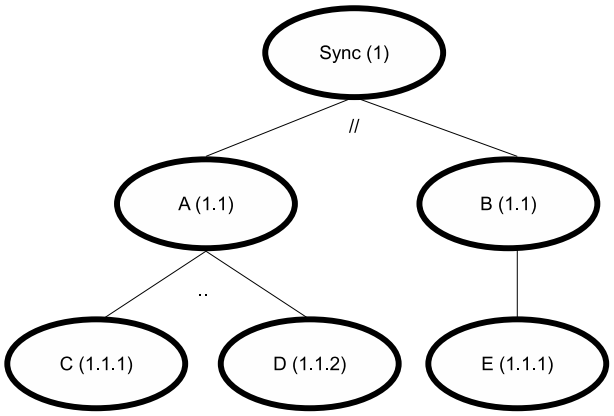


Figure 4.6: Representation for the Logical Execution no. 2

the task is waiting for child task to complete.

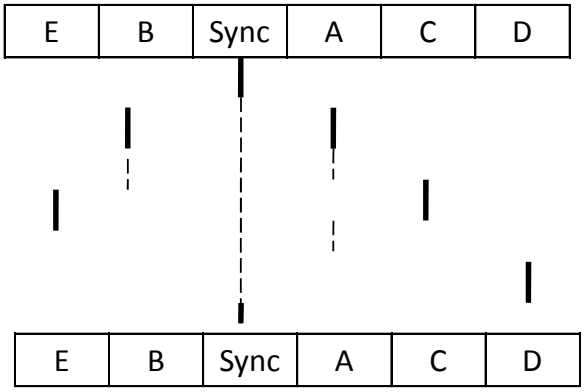


Figure 4.7: Breadth-First Execution for Figure 4.3

To achieve this trace of execution, "Sync" has two start two threads, where "A" and "B" will be executed, and then wait for the result of their execution. The method "B" calls "E" while the method "A" calls "C". Although before calling "D", "A" has to check if the methods "C" and "E" terminated successfully, since these methods logically precede "D".

Therefore, in a breadth-first execution there is the need to check the progress of methods running in the same logical levels. A first approach could involve a local way to check these details, but CO-OPN may be used to produce distributed specifications, so this approach must be generalized into a system wide manager named *Time Stamp Manager* (i.e., this manager has to be visible for all Contexts and Classes of the system).

In the Sequential Run-Time the order by which synchronizations should be handled was defined statically through the generated code (e.g. if "synchronization1" should be executed before "synchronization2", then the generated code would have the call to "synchronization1" before "synchronization2"). The approach previously taken provided good results, but since CO-OPN has the notion of backtracking, the generated code become hard and complex to interpret. This dissertation proposes a different mechanism to allow methods to decide by which

order synchronizations should be handled.

The mechanism to decide the order of execution is named *Synchronization Manager*. This mechanism is local to every method execution and requires that the method's synchronizations are added to the mechanism's "task pool". When all synchronizations are added, the Run-Time can rely on the *Synchronization Manager* to start synchronizations, following their logical order, and gather the results of the started synchronizations (i.e., a synchronization may fail or succeed).

Both the *Time Stamp Manager* and the *Synchronization Manager* will be carefully studied in the next sections.

The Time Stamp Manager

The system wide manager, named *Time Stamp Manager*, has the purpose of enforcing the following restrictions:

1. **Pre/test conditions**. All methods belonging to the same logical level should do their pre/test conditions together. This restriction allows the Run-Time to detect failures driven from not having enough free tokens in the system;

2. **Synchronizations**. Methods can only start their synchronizations if all methods with the same logical time stamp complete their pre/test conditions; This restriction is a natural follow up for the first restriction and prevents the run-time from calling unnecessary methods;

3. **Post conditions**. Before doing its post conditions, a method has to check if the synchronizations started by it were completed successfully, e.g., completed their post conditions. Otherwise the method should signal failure and skip its post conditions.

For each logical level the *Time Stamp Manager* must save a triple with the following information: number of synchronizations of the logical level; pre/test conditions of the logical level that have not yet been done; and, post conditions left.

After doing its pre/test conditions, every method must signal that to the *Time Stamp Manager*. Right before beginning its synchronizations the method must register in the *Time Stamp Manager* how many tasks are going to be performed and in which logical level. After calling every task—respecting the logical levels in which they should be called—the method is blocked until all tasks are completed or have failed.

Example. Breadth-first execution of the synchronization tree present in Figure 4.1 using the *Time Stamp Manager*:

1. *Sync* is registered, by an outside entity, in the *Time Stamp Manager* with the logical time stamp "1". In this logical time stamp the information stored is "1;1;1";

2. *Sync* completes its pre/test conditions and checks with the *Time Stamp Manager* if it may proceed to its synchronizations. Since it is the only registered method the *Time Stamp*

Manager does not block "Sync's" execution and signals that "Sync" can proceed right away. The *Time Stamp Manager* stores the information that a "Sync" finished its pre/test conditions, turning the information about the logical level "1" into "1;0;1";

3. *Sync* informs the *Time Stamp Manager* that two methods with time stamp "1.1" are going to be called; The *Time Stamp Manager* stores the following information about the new methods' logical level: "2;2;2". *Sync* calls *A* and *B* with time stamp "1.1";

4. *A* completes its pre/test conditions and informs the *Time Stamp Manager*. *A* is blocked by the *Time Stamp Manager* until *B* also completes its pre/test conditions. The information about the logical level "1.1" becomes "2;1;2".

   In the meanwhile, *B* reports that it just finished its pre/test conditions. The logical level information is updated to "2;0;2". Both *A* and *B* resume their execution;

5. *A* signals the *Time Stamp Manager* that it is about to call one method with logical time stamp "1.1.1". The *Time Stamp Manager* stores that at the logical level "1.1.1" there is going to be executing one call which translates into triple "1;1;1". *B* registers "1" sub synchronization and thus, the logical level information becomes "2;2;2". *A* and *B* block until their child synchronizations terminate;

6. *C* completes its pre/test conditions and is blocked by the *Time Stamp Manager* until *E* issues the information about completed pre/test conditions. The logical level "1.1.1"'s information is now "2;0;2";

7. Both *C* and *D* signal that their post conditions were successful. The *Time Stamp Manager* records that for level "1.1.1" the information is "2;0;0";

8. The execution of *B* is resumed and after completing its post/conditions informs the *Time Stamp Manager*. The logical level "1.1" information becomes "2;0;1".

   A is unblocked and registers one synchronization with time stamp "1.1.2". After calling *D*, the *Time Stamp Manager* blocks *A* execution until *D* finishes;

9. *D* after completing its pre/test conditions informs the *Time Stamp Manager*. The logical level "1.1.2" information becomes "1;0;1". The same happens for *D* post conditions, updating the logical level information to "1;0;0";

10. *A* regains control of its execution and after checking that its two child methods completed their post conditions successfully, *A* starts its post conditions and reports that information to the *Time Stamp Manager*. The information about the logical level "1.1" becomes "2;0;0";

11. *Sync* is unblocked and check the status of *A* and *B*. Since it was a successful execution, *Sync* does the post conditions and the logical level "1" information is updated to "1;0;0". The execution of the synchronization tree terminates.

The example is an instantiation of an execution, therefore "**A**" could complete its execution before "**B**", or they could both finish at the same time. In any case, the access to the logical level information should be serialized.

The Synchronization Manager

The *Time Stamp Manager* is used with the *Synchronization Manager* to achieve the depth-first execution. While the former manages execution globally, the latter manages synchronization calls locally to each method execution, e.g., method "a" is called twice then there should be two *Synchronization Managers* present in the system.

After the required synchronizations are added to the *Synchronization Manager*, the run-time is able to decide when each of the synchronization should be called. Again, the "when" is defined by the synchronizations' logical time stamps.

Abstracting from implementation details, the *Synchronization Manager* may be seen as an ordered list of synchronizations. The proposed version should be ready to support Breadth-First execution with multi threading. Hence, each logical level can have more than one synchronization.

Besides launching synchronizations while respecting their logical time stamp, the *Synchronization Manager* also has to interpret the synchronizations results (i.e. failure or success). The following interpretations should be considered while processing each logical level:

- **One synchronization**. If the synchronization fails it is not necessary to launch further synchronizations in the current service's axiom. Otherwise, the Run-Time can proceed to the synchronizations belonging to the next logical level;
- **More than one synchronization**. If one of the launched synchronization fail, the logical level must be restarted since the failure may be due to swapped tokens, e.g., synchronization "a" needs token "ta" while synchronization "b" needs a token. The logical level fails when all synchronizations of that level fail. Otherwise, if all synchronizations succeed the logical level also succeeds and the Run-Time can proceed to next logical level.

4.4 CO-OPN Places

The *marking* represents the state of the system, it associates a *Place* of an object with a multi set of logically timestamped values.

There are three basic manipulations of the *marking* that may occur while a *CO-OPN* specification is being executed:

- Defining the initial *marking* for each *Place*;
- Modifying the *marking* when evaluating pre and test conditions. The tokens which are subject to these manipulations are chosen non-deterministically;
- Modifying the *marking* by evaluating post conditions.

The basic manipulations can be represented by the following code-generation functions:

- **Init(o, p, jt[])**. Returns code that inserts the initial state, represented by the array of Java terms "jt", into *Place* "p" of object "o";
- **Pre(o, p, i)**. Returns non-deterministic code that marks as used one token from the *Place* "p" of object "o" at instant "i";
- **Test(o, p, i)**. Returns a similar non-deterministic code as the one returned by *Pre*, although the token is not marked as used but as tested;
- **Post(o, p, i, jt)**. Returns code that inserts new token into the *Place* "p" of object "o" at the instant "i".

Pre and test conditions over *Places* are non deterministic. Thus, pre and test conditions may be implemented as intended by the Parallel Run-Time developer. Although, tokens belonging to *Places* may be accessed if one of the following rules is respected: (1) the token was created logically in the past and it is not marked as used; or, (2) the token is being used at the same logical time.

The first condition is rather trivial and can be implemented "as is". The second rule can appear awkward at the first glance since it states that a token maybe accessed even if it is being used. This is true for test conditions, but CO-OPN semantics say that a token can not be consumed by pre conditions twice. In a Sequential Run-Time the second rule could be discarded, though in a concurrent environment if an active simultaneous iterator exists over the same place there is the possibility that the token locked by the simultaneous iterator may be released and therefore become free to be used.

4.5 Handling Simultaneous Pre/Test Conditions

Even with the rules introduced in section 4.4, handling real time simultaneous pre/test conditions is an intricate problem since it is difficult to decide for how long should a pre/test condition wait for the other simultaneous pre/test conditions to release their locked tokens. In some situations the simultaneous pre/test conditions may never release their locked tokens and therefore the run-time would become live-locked.

A simple solution to avoid live locks would prevent pre/test conditions from waiting for locked tokens. It is simple to implement, although sometimes the run-time may not execute successfully a CO-OPN specification just because two simultaneous pre/test conditions locked tokens in an inversed order, e.g., pre/test condition "p1" caught token "ta" and pre/test condition "p2" caught token "tb". "P1" discarded "ta" because "ta" did not fulfill "p1" condition. Although "p2" has not discarded "tb" yet, thus "p1" would return failure without testing token "tb" first.

To overcome the drawback of the tokens locked in inversed order, whenever a method can complete its pre/test conditions but this method execution is aborted due to other methods,

belonging to the same logical level, being unable to complete their pre/test conditions, the method must release the locked tokens, save its state and signal that the failure was cause by other method.

In this new solution, when a method calls two or more sub methods and if only one of the sub methods fail, they both need to be retried. Since the sub method that completed its pre/test conditions saved its state, its second execution is going to be identified as a backtracking call, and therefore new solutions will be searched.

4.6 Supporting Backtracking

Backtracking is used by the Run-Time to fully explore the solutions space, thus providing different solutions from the ones offered in previous calls to a number of given methods. In this section the word "method" is going to be used to refer a CO-OPN service, either a "method" or a "gate".

As discussed before, CO-OPN execution uses the same execution model than *Prolog*. Therefore, the Run-Time must differentiate between a regular call from a backtracking one. In order to do so, the Parallel Run-Time resorts to transaction identifiers like the Sequential Run-Time did. Whenever a method completes successfully, its transaction identifier, which is given as an argument, must be saved along with the system marking that existed when the method executed.

In the Sequential Run-Time transaction identifiers were stored in a object wide stack data structure (e.g., each object had one stack). This option proved to be wrong because when executing a redo on a simple synchronization as `A .. B`, where "A" and "B≪≪ are methods of the same object, the Run-Time could not identify a redo, because the top of the stack had the transaction identifier of "B" and not of "A".

To overcome the drawback of the solution used by the Sequential Run-Time, the transaction identifiers and the system state should be stored in a data structure which enables the Run-Time to access other elements than the ones on top, like the stack does. To attain a good performance, the chosen data structure should support indexing.

At last, when a redo is identified, the Run-Time must be able to locate where in the method's code the execution should continue (e.g., which axiom was last executed). To provide the run-time with this ability, each axiom should have an identifier within the method and the identifier of the axiom executed must be saved along with the transaction identifier and the system marking.

This way, the previously executed axiom may be resumed and if this axiom or others that were not tested can return other solutions, then the process repeats itself by saving the transaction identifier and all the required information to restored the execution state.

4.7 Handling Distribution

CO-OPN allows us to specify concurrent and distributed systems, although CO-OPN does not have enough expressiveness to specify which elements are distributed. The lack of this information comes from the actual application that CO-OPN specifications have been target of.

Until now this language has been used to fast prototype components for other systems and to study behavioral properties of some systems by modeling them in CO-OPN and using the simulator run-time to analyze their execution. For both activities it is not necessary to have concurrent or distributed components within the CO-OPN specification, and for the second it is even undesirable since it would turn simulation into a more complex task.

The basis to decide what elements should be distributed in the new type of execution was the function of the three CO-OPN components, e.g., Contexts, Classes and Abstract Data Types. Contexts are the system coordinators, they are responsible for inter-object interaction and encapsulate objects, either Classes or other Contexts, with identical concerns in the same environment. Therefore it is natural to make the assumption that Contexts should be the key elements targeting distribution.

However, if the proposed solution turned each Context into a distributed element it may not suit the semantic of every CO-OPN specification (e.g., a user may want to specify that a EXEMPLO).

To fit every CO–OPN specification the proposed solution includes a mechanism which offers a way to choose which Contexts are distributed and where they should be physically executed. This mechanism must respect the hierarchy of Contexts defined by the CO-OPN semantic, e.g., Context A encapsulates Contexts B and C. If the programmer defines A and C as distributed, thus B will execute in the same physical node as A.

5

The Parallel Code Generator

A Parallel Code Generator was developed in order to allow concurrent execution of CO-OPN specifications. The goal of this new code generator is to map CO-OPN sources to Java files ready to be deployed in a distributed system which supports Java and the Remote Method Invocation API.

5.1 Modular Code Generation Approach

The sequential code generator that was developed allows modular code generation, where a different code generator can be defined for each CO-OPN module (e.g., Classes, Contexts and Abstract Data Types). The process of generating executable code from CO-OPN specifications is then comprised of three stages, which are represented in Figure 5.1:

- **Configuration**. This step defines the association between code generators and the type of source module. Through the support of multiple generators the code generator allows the application of different generations strategies to each module type. The result of this step is a list of pairs: (generator, module type);
- **Export**. A CO-OPN specification may define modules with dependencies between them. These dependencies are created by using methods, types or other symbols that other modules define. In a global generation strategy the generator would trivially know how to represent the dependencies since the sources of the specification are handled in the same place. This modular approach deals with module dependencies by forcing each modular generator to export the representation of primitives (e.g. methods, types, ...) offered by its module;
- **Code Generation**. In the third stage, each generator is asked to produce code for its

source module. Dependencies are implemented resorting to the global set of exported primitives.

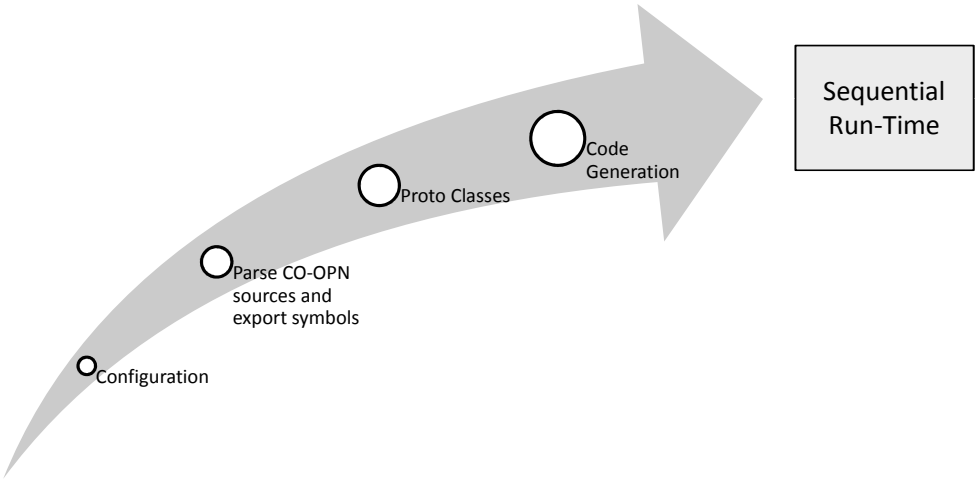


Figure 5.1: The Sequential Code Generator Process

When all module types have a generator assigned, the source modules are parsed and their symbols exported, the so called "Proto Classes" are created. These "Proto Classes" are memory representations of the CO-OPN structures (e.g., services and their axioms, encapsulated objects, ...). The *Code Generation* step is also handled by the "Proto Classes".

A parallel code generation was achieved by deploying new generators for the various CO-OPN module types, which are fresh and modified versions of the ones used by the sequential code generator. This approach was made possible thanks to the modular framework developed before. A graphical representation of the changes applied to the sequential code generator ate visible in Figure 5.2.

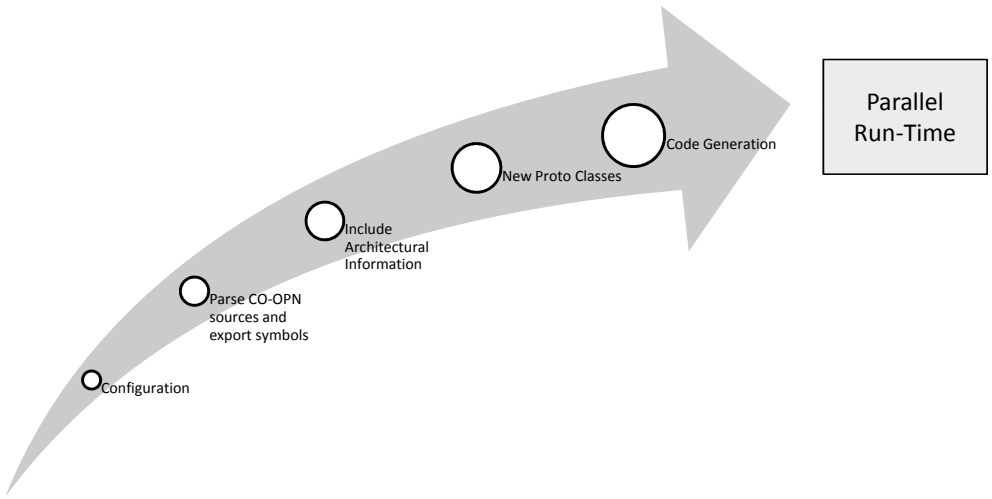


Figure 5.2: The Parallel Code Generator Process

The modifications to the code generation process are the ones highlighted in Figure 5.2 (i.e. "Include architectural information" and "New Proto Classes"). The "Architectural Information" is a fundamental step and allows the new generator to possess details concerning the physical distribution of CO-OPN entities.

The new "Proto Classes" are responsible for the transformation of CO-OPN sources into Java code, which complies with the parallel execution directives studied and explained in Section 4.

5.2 Parallel Run-Time

This section depicts the Java Classes that are present in every parallel execution. These Classes are responsible for enforcing the semantic of the proposed parallel environment.

5.2.1 Support for Logical Instants

The proposed implementation for logical instants is based on the implementation introduced by the sequential code generator, although the tree-like structure is used uniquely for hierarchy storing purposes.

Alike `CoopnTransaction`, the new `CoopnTransaction2`, besides containing logical instants related methods it also has the transaction methods `commit` and `abort`.

`CoopnTransaction2` is almost fully compatible with `CoopnTransaction` and it was not implemented as an extended version of `CoopnTransaction` uniquely because there was the will to modify some of its implementation details which will be discussed bellow.

Since the new representation of Logical Instants is still a tree-like structure, each `CoopnTransaction2` object has one pointer to the super instant and two pointers, each for one sub-instant (i.e., the logical time stamp tree is binary). Particularly, in the case of the root of the tree the super instant is *null*. To differentiate the kind of subdivision each node possesses an integer named *subdivision*. If the node has yet to be subject to any kind of subdivision this integer will be equal to a constant named *OP_NULL*.

Until now `CoopnTransaction2` is just like the previous version, the differences come with the support for the proposed string representation of logical instants and their subdivision. A call to the default constructor will return a new `CoopnTransaction2` object with the subdivision operator set to *OP_NULL* and the string representation set to "1". Calls to the subdivision methods will return new `CoopnTransaction2` objects with different representations. The available subdivision methods are:

- `with()` creates an included sub instant;
- `sim1()` and `sim2()` separate the instant into two simultaneous sub instants;
- `seq1()` and `seq2()` divide the instant into two sequential sub instants.

The subdivision methods which end in "1" define the left child and the ones that end in "2" define the right child. Simultaneous subdivision methods set the subdivision integer to

the constant *OP_SIM*, sequential ones set the integer to *OP_SEQ*, and at last, when the `with` method is used the subdivision integer is set to *OP_WITH* constant. Subsequent calls to the same subdivision method over the same node will not create a new child but instead return the existent child.

There is an extra method *public void isInitial()* which sets the string representation to "0". This method is to be used when creating transaction identifiers for the initial state of the system.

The string representation becomes more efficient since instead of traversing the nodes of a logical tree of instants while trying to figure the relation between two instants, it uses simple string comparisons which are only CPU bound. Thus, the Run-Time can compare the relation between two instants by simple checking the alphabetic relation between their representations, e.g., "1.1.2" comes after "1.1.1".

Example. Using the `CoopnTransaction2` Class:

- The main instant is created by using the default constructor:

```
CoopnTransaction2 i0 = new CoopnTransaction2();
```

- Creating two sub instants, one to represent the "past" and another for the "present":

```
CoopnTransaction2 past = i0.seq1();
CoopnTransactio2 present = i0.seq2();
```

- Structuring the present using different approaches with the same semantic:

```
CoopnTransaction i11 = I0.seq2().seq1().sim1();
CoopnTransaction i11pr = present.seq1().sim1();
```

- Comparing instants is done by comparing object references:

```
i11 == i11pr; // true
i11 == past; //false
```

- Comparing logical levels:

```
CoopnTransaction2 i12 = present.seq1().sim2();
i11.getRepresentation().compareTo(
            i12.getRepresentation()) == 0; //true
```

- Checking the consistency of the set:

```
present.seq2(); //create and store
present.seq2(); //dont create, return stored
present.seq2(); //idem
present.sim2(); //exception is thrown
                //because an instant can not
                //be subdivided in more than
                //one type
```

5.2.2 Support for Concurrency Aware Places

The Marking and its supported manipulations are implemented by the class *CO-OPNParallelPlace*. Like in the Sequential Run-Time, the matching of values by pre and test conditions is done outside of the responsibility of the *Places*. This means that a *Place* only returns tokens and the method must keep asking for tokens until the desired token is found.

Interface for COOPNParallelPlace

CO-OPNParallelPlace provides support for the already presented basic marking manipulations in the following manner:

- **Initialization**. The programmer may use either the default constructor, which creates an empty place, or one that has as arguments an array of initial values:
  - `public CoopnParallelPlace();`
  - `public CoopnParallelPlace(Object[] initialMarking).`
    The current implementation supports another way to add initial values one by one to a given place: `addInitial`. This alternative is not actually used by the new generator, instead it is to be used when a programmer wishes to do modifications to the generated classes.
- **Pre conditions** are evaluated one token at a time. Access to *Places* is provided through iterators:

  ```
  public ParallelStateIterator pre(CoopnTransaction2 t)
  ```

  The first and only argument represents the instant at which the pre condition must be executed. The *ParallelStateIterator* class will be depicted later;
- **Test conditions** likewise pre conditions are evaluated token by token and are supported by calling

  ```
  public ParallelStateIterator test(CoopnTransaction2 t)
  ```

  The only argument has the same purpose that the one used in pre conditions;
- **Post conditions** add values to a given place. Values must be added one at a time, using the following Java method:

  ```
  public void post(CoopnTransaction2 t, Object value)
  ```

  The first argument is the logical time stamp at which "value" is to be inserted.

  The corresponding undo operation, required by backtracking, is supported by the following Java method:

  ```
  public void remove(CoopnTransaction2 t)
  ```

  This method removes from the *Place* all values inserted with the transaction identifier "t";

- **Destruction** is implicit and occurs when the object which encapsulates the *Place* becomes referenceless.

Implementation of CoopnParallelPlace

The *CoopnParallelPlace* was implemented as a concurrent hash table implemented by the *Java Concurrency* package, where the key is the instant representation (i.e., a *String*) and the values are instantiations of the concurrent linked list data structure provided also by the *Java Concurrency* package. Each of these linked lists save tokens created in the same time stamp as the one used as the hash table key for the list. Tokens stored in the linked lists are instantiations of the *Node* Class, which is presented bellow.

By using concurrent versions for the data structures, the *CoopnParallelPlace* becomes thread safe, allowing multiple simultaneous accesses over the same *Place*.

Initialization *CoopnParallelPlace's* default constructor initializes the encapsulated hash table with this data structure default constructor. The other constructor calls the default constructor and then creates a concurrent linked list which contains the initial values. This list is added to the hash table with key "0". By using this value as a key the initial values become visible for every logical time stamp since the root of the logical synchronization tree starts with value "1". More details describing the access to specific tokens will be given later.

Pre, Test and Post conditions As described, both pre and test conditions are done using an instantiation of the *ParallelStateIterator* class.

The constructor for each iterator starts by adding to a private list a reference to every token which is accessible in the given logical time stamp. The constructor also saves a boolean that states the nature of the iterator: test or pre condition iterator.

The *ParallelStateIterator* `public boolean setNext()` method begins by testing the status of the variable that holds the next to return token. If this variable is not null then the iterator was used previously and thus the iterator is being used for a redo, which implies that the locked token must be released and a new token to return must be found. Otherwise, it is a regular pre condition and the method solely finds the a token to return.

To find a new node to return, the iterator's private list, which holds the reference for possible token matches, is analyzed and its nodes are subject to an operation that tries to mark the nodes as used. In case this operation is successful then a new next to return token is found and the reference to the node is removed from the private list. Otherwise, and if the private list has more references, the method continues to iterate the other tokens. Nodes that can not be marked as used by the iterator are not removed from the private list, again this decision was made because the nodes may be released by a redo of a concurrent active iterator.

Post conditions Evaluating a post condition is a synonym for adding one value to a *Place*. As said before, this is implemented by the `post` Java method. First of all, the method searches in the hash table for an already inserted value with the same logical representation, if one is found,

then the new value is inserted in the same linked list, otherwise, a new linked list containing the new value is created and then inserted in the hash table.

The Node Class This class has the purpose of encapsulating the values that are saved in *Places* along with transaction information referring at which logical time the values were created and last used. The *Node* class is based in the *Basic* Class implemented in the Sequential Code Generator, although this version is less complex due to the new structuring of the proposed implementation for the CO-OPN *Place*.

Each of the instantiations of the *Node* Class stores: one Java *Object*, which represents the value; the creation and last use logical instant; and, two booleans, each to express if the value is being used by a pre or by a test condition, respectively.

This Class offer simple methods such as:

- `public Node(CoopnTransaction2 t, Object value)`

  that is the definition of the constructor, where "t" is the transaction identifier and "value" what the run—time desires to store in the *Place*;

  `public boolean lock(CoopnTransaction2 t)`

  which returns true if the node may be used successfully by a pre condition;

- `public boolean test(CoopnTransaction2 t)`

  that returns true if the node may be used successfully by a test condition;

- `public void release()`

  which is used to mark the node as unused, either by pre or test conditions.

Example. Representing *Place p _: sometype* (i.e., a *Place* named p which holds objects of type "sometype").

- The *Place* is declared as a private data member in the corresponding Object class:

  ```
  protected CoopnParallelPlace p;
  ```

- Initialization with *value1, value2* (i.e., *Place* p has "value1" and "value2" in its initial marking):

  ```
  p = new CoopnParallelPlace(new Object[]{value1,value2});
  ```

- Enumerating tokens to be used by a pre condition:

  ```
  ParallelStateIterator itt = p.pre(t);
  while(t.setNext())
    t.next();
  ```

- Post condition of *value3* (i.e., add "value3" to *Place* p) and subsequent "undo":

```
  CoopnTransaction2 t = new CoopnTransaction2();
p.post(t,value3);
p.remove(t);
```

Examples of places and logical instants. In each example a *Place* named p with empty marking is used:

- Parallel execution:

```
p.insert(T.sim1(),``1'');

ParallelStateIterator i = p.pre(T.sim2());
i.setNext();//returns null,
            //cannot consume token created in parallel
i.next();//throws an exception because the iterator
         //does not have a next to return token
```

- Recurrent pre conditions:

```
p.insert(T.seq1(),``1'');

ParallelStateIterator i = p.pre(T.seq2());
i.setNext(); //returns true
i.setNext(); //returns false, no more tokens in p
```

- Simultaneous pre and test conditions:

```
p.insert(T.seq1(),``1'');

ParallelStateIterator i1 = p.test(T.seq2().sim1());
i1.setNext(); //returns true, last usage time stamp
              //set to T.seq2.sim1

ParallelStateIterator i2 = p.pre(T.seq2().sim2());
i2.setNext(); //returns false, because the only
              //existent token
              //is marked as taken

ParallelStateIterator i3 = p.test(T.seq2().sim2());
i3.setNext(); //returns TRUE,
              //time-stamp updated to T.seq2
```

5.2.3 Time Stamp Manager

The `TimeStampManager` Class was created to enforce the logical instant semantic to which CO-OPN's synchronization operators bound the execution.

When using the proposed representation for logical instants is easy to catalog synchronizations referring to their logical instant representation. Therefore, the run—time can track which synchronizations have completed their pre/test and post conditions. This way synchronizations may be halted until the "correct" time for their execution comes.

The only data member of the `TimeStampManager` Class is a Concurrent Hash Map, implemented in the *Java Concurrency* package. This map is indexed by the String representation of the logical instants and in each entry stores one object typed `TimeStampInfo`. By default a `TimeStampInfo` object, with the logical representation "1" is inserted into the map. A concurrent hash map is used because this object is possible subject to simultaneous calls, and a thread safe structure prevents inconsistent states.

This class provides the following public methods:

- `public void addTimeStampInfo(CoopnTransaction2 t, int number)`

  notifies the manager that "number" calls will be executed at instant "t";

- `public void completedPre(CoopnTransaction2 t)`

  signals that a call executing at instant "t" completed its pre/test conditions;

- `public void completedPost(CoopnTransaction2 t)`

  signals that a call executing at instant "t" completed its post conditions;

- `public void signalFail(CoopnTransaction2 t)`

  signals that a call executing at instant "t" failed;

- `public boolean allPostCompleted(CoopnTransaction2 t)`

  checks if all logical child nodes of "t" completed their post conditions or failed. Returns true if children complete their post conditions successfully, otherwise returns false. The caller object will be blocked until all children report their status;

- `public boolean allPreCompleted(CoopnTransaction2 t)`

  checks if all calls executing at same logical instant as "t" completed their pre/test conditions. Returns true if all sibling calls completed, otherwise returns false. The caller object will be blocked until all siblings report their status.

When a method completes its pre/test conditions, it signals that information to the `TimeStampManage`
and it will be suspended until all the calls running in the same logical instant complete their pre/test conditions.

Before starting the synchronization phase, the method must notify the `TimeStamp Manager` with the amount of calls that are going to be launched in the given instant. This notification

leads to an increase on the number of calls that are running in the given instant, and it also increases the number of pre/test and post conditions that must be completed in the given instant.

After launching the synchronizations, the method will be suspended until all the nodes which are logically under it in the tree of synchronizations complete their post conditions. If any of them fails, the failure is spread to the logical upper levels.

The TimeStampInfo Class

The `TimeStampInfo` Class has the purpose of storing information about the progress of calls done in the same logical instant.It is comprised of:

- A String which holds the instant representation for the object;
- An integer for the total number of calls on that given instant;
- The number of method calls that have not completed their pre/test conditions;´
- The number of method calls that have not completed their post conditions;
- A boolean to signal if any of the calls failed;
- A list of `TimeStampInfo` objects for the children calls.

Example. Using a `TimeStampManager` object:

- Instantiating an object typed `TimeStampManager`:

```
TimeStampManager tsManager = new TimeStampManager();
```

- Registering the syncronization tree A .. (B // C), where the root of the tree is at instant "1":

```
CoopnTransaction2 t = new CoopnTransaction2();
CoopnTransaction2 leftBranch = t.seq1();
CoopnTransaction2 rightBranch = t.seq2();
tsManager.addTimeStampInfo(leftBranch, 1);//A
tsManager.addTimeStampInfo(rightBranch.sim1(), 1);//B
tsManager.addTimeStampInfo(rightBranch.sim2(), 1);//C
```

  Since B and C belong to the same logical instant (e.g., "1.2.1"), both will refer to the same `TimeStampInfo` object;

- "B" and "C" notify completed pre conditions:

```
Thread B = new Thread(){
 public void run(){
  tsManager.completedPre(rightBranch.sim1());
 }
}

Thread C = new Thread(){
 public void run(){
  tsManager.completedPre(rightBranch.sim2());
 }
}
B.start();
C.start();
```

When B and C issue the *completedPre* method both will be suspended until the two called the method;

- "A" tests if it can proceed to its post conditions:

```
if( tsManager.allPostCompleted(leftBranch) ){
 //proceed to post conditions
 tsManager.postCompleted(leftBranch);//notify completed
                                     //post conditions
}
//else notify post conditions failed
```

The call *allPostCompleted* will return true immediately since A does not have sub instants. The same is true for B and C.

5.3 Implementation of Methods and Gates

The behavior of CO-OPN entities, Contexts and Class instantiations, is defined by the so called behavioral axioms. These axioms are comprised of: input parameters matching, a set of pre conditions, synchronizations and a set of post conditions. Axiom execution occurs step by step following the presented order.

As seen before the programmer may define multiple axioms for the same method. In this implementation a method call will only fail if and only if all axioms have been tried and none had success. If the call fail then method *Failure* is notified by throwing the special *CoopnMethod-NotFirable* Exception.

The definition of Methods and Gates, more explicitly, their behavior, is defined through axioms. This section explains the implementation of these axioms.

5.3.1 Handling Pre/Test Conditions

Pre conditions consume tokens from places, while test conditions check if there are tokens available in the given places. An axiom may define multiple pre or/and test conditions and both kinds may search for specific tokens.

To handle pre/test conditions, this implementation uses the methods which are offered by the *Place* implementation. Pre and test conditions are done through the *ParallelStateIterator* Class and the axioms have the responsibility to choose which tokens are necessary.

As token matching is done in the axiom body, pre or test conditions are translated into Java code as:

```
1 while(iterator.hasNext())
2   if(iterator.next() == condition)
```

The cycle is used to iterate the available tokens and the condition is for token matching, therefore it would not be required if the pre/test condition does not specify which token it needs. Whenever an axiom has multiple conditions, they are evaluated one by one in the following manner:

```
1  while(iterator.hasNext()){
2    if(iterator.next() == condition)
3      while(iterator1.hasNext()){
4       if(iterator1.next() == condition1)
5             while(iterator2.hasNext()){
6               if(iterator2.next() == condition2)
7                 ...
8             }
9      }
10 }
```

The order by which pre and test conditions are evaluated is irrelevant since the Run-Time must iterate per all pre and test conditions.

5.3.2 Handling Synchronizations

An axiom may define a synchronization expression which follows the logical execution imposed by the synchronization operators used between the synchronization calls, e.g., A .. (B // C), states that after A is successfully executed, B and C must succeed in simultaneously.

To implement efficiently the synchronization expressions for each of the synchronization calls (e.g. in A .. (B // C) there are three synchronization calls: A, B and C) an object typed `SyncInfo` is created.

The SyncInfo Class

The `Syncinfo` Class is responsible for storing the thread in which the synchronization will be executed; the state of the thread execution; and, the `CoopnTransaction2` object that holds

the logical time stamp at which the thread should be executed. The state of the thread execution may take one of the following values:

- **READY**, the thread is ready to be started;
- **RUNNING**, the thread is currently running;
- **FAIL**, the synchronization call failed;
- **SUCCESS**, the synchronization call succeeded.

The state of the thread execution is switched from *READY* to *RUNNING* when the `public void start()` method is called.

Synchronization calls are wrapped around an anonymous class in the following way:

```
new Thread() {
              public void run() {
                 try {
                   synchronization();
                   infoN.setSuccess();
                 } catch (RemoteException e){
                     infoN.setFail();
              }
             }
                     }
```

When the synchronization call succeeds then the `public void setSuccess()` method of the corresponding `SyncInfo` object is called, setting the state of the thread execution to *SUCCESS*. Otherwise, the synchronization throws the *CoopnMethodNotFirableException* which is caught by setting the state of the thread execution to *FAIL*. These anonymous classes are added to the `SyncInfo` object through the `public void setThread(Thread t)` method.

Example. Using the SyncInfo Class:

- Creating a SyncInfo object:

```
CoopnTransaction2 t = new CoopnTransaction2();
final SyncInfo info0 = new SyncInfo(t);
```

- Setting its "task":

```
info0.setThread(new Thread() {
  public void run() {
    try {
     someTask();
     info0.setSuccess();
        } catch (RemoteException e) {
```

```
            info0.setFail();
           }
         }
       });
```

- Starting the synchronization execution:

```
info0.start();
```

The SyncManager Class

By now every synchronization call is wrapped in its own `SyncInfo` object, nevertheless it is missing a way to manage all these ready to execute synchronizations. So the Class `SyncManager` serves the purpose of starting synchronizations when they should be started and gather the results of their execution.

Whenever an axiom has synchronizations its generated code include an instantiation of the `SyncManager` Class, i.e., `SyncManager` is local to each axiom with synchronizations.

`SyncInfo` objects are added to an hash table in the `SyncManager`. This hash table is indexed by the *String* representation of the logical instants and each position has one linked list which stores the objects typed `SyncInfo`.

Example. Using the Sync Manager:

- Instantiating a `SyncManager` object:

```
SyncManager syncManager = new SyncManager();
```

- Adding a synchronization call to the manager:

```
CoopnTransaction2 t = new CoopnTransaction2();
final SyncInfo info0 = new SyncInfo(t);//empty call
                                  //since thread is undefined
info0.setThread( ... );
syncManager.addSync(t.getRepresentation(), info0);
```

- Preparing tasks for execution:

```
syncManager.prepThreads();//returns one, which corresponds
                          //to one task (info0)
                          //ready for execution
```

- Starting prepared tasks:

```
syncManager.startAll();
```

- Setting a redo:

```
syncManager.skipPrep();
```

- Preparing tasks for execution:

```
syncManager.prepThreads();//returns 1.
                          //Since a redo was asked
                          //task belonging to info0
                          //will be prepared
```

When all synchronizations are added to the `SyncManager` object, the so called *synchronization cycle* may occur. This cycle is present in Figure 5.3.

```
1  int nThreads = 0;
2  boolean skip = false;
3  int syncResult;
4  while ( (nThreads = syncManager.prepThreads()) != 0) {
5
6       if (skip) {
7        skip = false;
8       } else {
9        tsManager.addTimeStampInfo(syncManager
10                                 .getCurrentTimeStamp(), nThreads);
11         }
12
13         syncManager.startAll();
14         syncResult = syncManager.levelDone();
15         switch (syncResult) {
16           case SyncManager.FAIL:
17             tsManager.signalFail(t);
18             throw new CoopnMethodNotFirableException();
19
20          case SyncManager.REDO:
21            skip = true;
22            syncManager.skipPrep();
23            break;
24
25          case SyncManager.SUCCESS:
26            break;
27    };
28
29   }
```

Figure 5.3: SynchronizationCycle

A variable named "nThreads", which will hold the number of tasks that are going to be started, is declared and initialized with the value "0". This value is used since the `SyncManager` method `prepThreads()` returns "0" if there are no more tasks. A second variable, typed boolean, is initialized with the "false" value, meaning that `syncManager` may not skip the preparation phase.

The next instruction is the *synchronization cycle* itself. The variable `nThreads` is assigned with the value returned by the method `prepThreads()`. If the returned value is different from "0", execution proceeds into the cycle.

Once inside the cycle the variable `skip` is checked. If it has the value "true", then the `syncManager` will skipped the preparation phase and therefore the `tsManager` (e.g. instantiation of the TimeStampManager Class) does not need to be notified of task launching. Otherwise the `tsManager` will be notified that at instant "syncManager.getCurrentTimeStamp()", "nThreads" tasks will be launched.

The main thread will be suspended inside the method `levelDone()` until all launched tasks terminate. The method will then collect all the termination values for every tasks and return one of the following values:

- **SUCCESS** if all tasks completed successfully. May proceed to the next series of tasks;
- **REDO** if at least one task completed successfully. A redo must be done;
- **FAIL** if any task completed successfully. Notify that a failure occured.

5.3.3 Handling Post Conditions

Post conditions add tokens to places and are issued when an axiom finished successfully its synchronizations. Post conditions are translated to Java code by resorting to the *public void insert(CoopnTransaction2 t, Object value)* method belonging to the *CoopnParallelPlace* Class. Multiple post conditions are transformed into multiple calls to the *insert* method.

Whenever it is necessary to undo post conditions, the generated code includes calls to the `public void remove(CoopnTransaction2 t)` method of the *Place* implementation class.

5.3.4 Handling Method Backtracking

The support for backtracking is done in a similar fashion as implemented by the sequential code generator, but again, it was simplified and improved in order to allow simultaneous backtracking calls. The `CoopnTransaction2` acts as an identifier to provide a way to differentiate a normal method call from a backtracking call.

Each axiom has an unique identifier inside the method in which the axiom belongs. Every time an axiom succeeds, the Places' iterators, used for pre/test conditions, along with the axiom identifier are saved in an instantiation of the class `StateHolder`, which serves solely the purpose of storing values and provide ways to access those values. The `StateHolder` instantiations are stored in a concurrent hash map, implemented in the Java Concurrency package, and their key in this hash table is the `CoopnTransaction2` which was used to call the method. Using directly a concurrent hash map allows the previously states for the objects to be thread safe.

The first instruction present in all methods is to check if the call is a redo by matching the input parameter `CoopnTransaction2` with the key set of the hash map. There are two possible cases:

- **the key is found**, then the corresponding `StateHolder` instantiation is retrieved and the necessary variables are restored. The tokens inserted by the last execu tion post conditions are removed. And the index of the axiom which was stored the `StateHolder` is used to locate where, in the method body, the execution should continue;
- **the key is not found**, proceed to the first axiom without restoring variables since it is a regular call.

5.4 Distributing CO-OPN Entities

CO-OPN allows the specification of distributed systems. However, in the sequential run-time the distribution of entities was not discussed or implemented. To tackle the physical distribution problem, this thesis proposes a language which will be depicted later in this section.

As proposed in the solution model chapter, CO-OPN's *Contexts* are the entities which may be subject to distribution. Java RMI introduced the concept of Remote Objects. These objects, despite of being present in different computer processes and usually in distinct machines, may be accessed like "normal" Java objects, supposing that they are registered in the *RMI Registry*.

To comply with the requirements of *Java RMI*, the generated classes for the distributed contexts also have a corresponding RMI Interface, where the *Context* interface methods and gates are declared. Is through these RMI Interfaces that the *Context* "advertises" its services.

With this mapping which allows the interaction between distributed *Contexts*, there is the need to define where, in terms of node physical location, each of the *Contexts* will be available. For this purpose a simple language named *CO-OPNArch* was introduced. Its grammar is defined in Figure 5.4.

```
1 SpecificationConfiguration := RMIRegistry TimeStampManager
2                               RootContextInfo (ContextInfo)+
3
4 RootContextInfo := 0 ContextName
5
6 ContextInfo := LevelNo ContextName (URI)?
```

Figure 5.4: CO-OPNArch Grammar

A brief look over the grammar shows that a configuration is represented by: the URI of the RMI Registry; the name of the remote object which acts as the *TimeStampManager*; information describing the architectural details for the root context; and, by one or more nodes with architectural details for each of the other Contexts present in the specification.

The architectural details are the hierarchical level at which the *Context* is inserted, being the root *Context* the level 0; the name of the Context, that will be used, in the compilation phase, as

an identifier which locates the corresponding CO-OPN source; and, an optional field that when present states which identifier should be used to retrieve the *Context* Remote Object from the RMI Registry. The root context does not need an URI since no other context of the specification needs to explicitly or implicitly communicate with the root context.

The *CO-OPNArch*'s grammar was defined using the The Java Compiler Compiler (JavaCC). This tool reads the grammar specification and converted it into a Java program capable of recognize matches to the grammar. By employing JavaCC, the task of generating the grammar parser was made automatically.

The code generator gathers all the required architectural details from the configuration file and offers them to the corresponding modular generator, which deals with the aspects of Context code generation. For each Context module that is to be transformed into code in the target language, the required architectural details are:

- **RMIRegistry**. Corresponds to the URI of the RMI Registry;
- **TimeStampManager**. Provides the Context with information about which id is used to register the object that imposes order on the system flow of execution;
- **Children Contexts**. A list of directly encapsulated contexts (e.g. if *A* contains *B* which contains *C*, *A* only has information about *B*) plus information on how should this context be initialized: (1) it is a local context, than it is a normal initialization; (2) it is a remote context, then the corresponding ID is provided and it is this ID which is used to locate the object in the RMI Registry.

Apart from providing architectural information to the Contexts configuration file also plays an important part generating a Java file from which the so called *Time Stamp Manager* is initialized and registered in the RMI Registry.

Nowadays this language does not have a graphical or textual editor, therefore specifications must be done manually by the programmer. There is the need to provide exact matches with the names given in the *CO-OPN* specification, otherwise code generation will not be successful.

Example. Applying Architectural Information to the CO-OPN Specification present in Figure 5.5:

- Aspect of a possible configuration file:

```
//someAddress
theTimeStampManagerID
0 MainContext
1 PingPongOne
1 PingPongTwo pingpongtwoID
```

The RMI Registry is present at the URI "//someAddress" and the object that applies CO-OPN policies to the system flow of execution has the id "theTimeStampManagerID". The

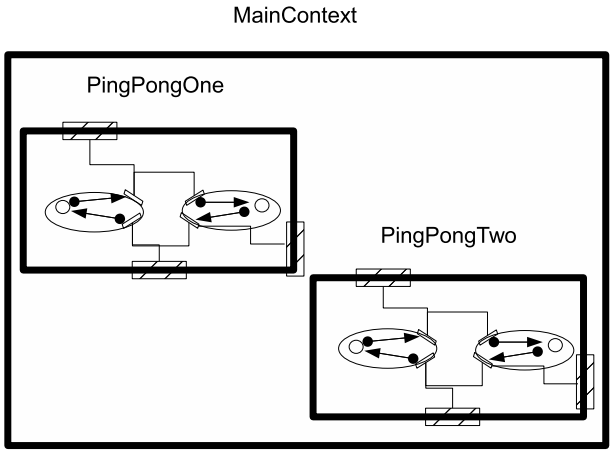


Figure 5.5: Multiple PingPong Contexts

"PingPongOne" Context is local to the "MainContext", but the "PingPongTwo" Context is not and may be retrieved from the RMI Registry using the id "pingpongtwoID".

Another approach would involve modifying the CO-OPN language itself in order to introduce the required architectural details, merging all in one language, although the chosen approach is less intrusive and this way different concerns are separated: *CO-OPN* expresses the interactions and *CO-OPNArch* the physical distribution.

6

Use Case: The Producer—Consumer Problem

6.1 Introduction

The "Producer—Consumer" problem is a classical example to study synchronizations between multiple processes. There are three main components: a producer whose job is to generate data; a consumer which consumes the generated data; and, a buffer that holds the data and acts as an interface between the producer and the consumer. The only two constraints that apply to this problem are connected with the state of the buffer, the producer cannot add data if the buffer is full and the consumer can not consume data if there is not any. This problem may be generalized to hold many producers and many consumers.

Concurrent Object-Oriented Petri Nets (CO-OPN) allows us to specify the interactions between the components of the "Producer—Consumer" problem in a rather simple way. However, firstly the properties of the components have to be defined.

6.1.1 One Producer and One Consumer

Let us start by a simple system as depicted in Figure 6.1, where there are one producer, one consumer and one buffer.

A correct mapping of this system to CO-OPN may create one Context module for each of the three components and a top level Context whose purpose is to encapsulate the other three Contexts and specify their interactions. These interactions are visible in Figure 6.1 and focus the communication between the producer and the buffer, and between the buffer and the consumer.

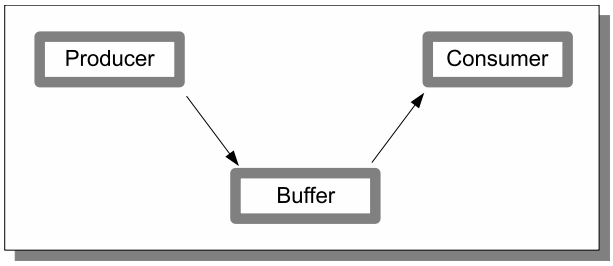


Figure 6.1: A system comprised of one producer and one consumer

Producer Context

The *Producer Context*, present in Figure 6.2, will contain: one object, which is an instantiation of the *Producer Class* (line 14); two *Methods*, one to store data (line 8) and the other to trigger data output (line 9); and, one *Gate* to output the saved data (line 6).

```
1  Context ProducerContext;
2  Interface
3      Use
4          Naturals;
5      Gate
6          offer _ : natural;
7      Methods
8          store _ : natural;
9          produce;
10 Body
11     Use
12         producerObject;
13     Object
14         po : typeProducerObject;
15     Axioms
16         store a With po . store a;
17         po . offer a With offer a;
18         produce With po . produce;
19     Where
20         a : natural;
21 End ProducerContext;
```

Figure 6.2: Producer Context Layout

The *Producer Class* is comprised of one *Place* (line 12); two Methods (lines 8 and 9); and, one Gate (line 6). The *store* method will store provided data in the *Place* and the gate will retrieve data from the place, *produce* makes data available in the *offer* gate. The *Producer Class* described above is present in Figure 6.3.

Both actions of adding (*store* method) and removing (*offer* method) data from the *Place* are achieved through pre and post conditions, respectively.

The *Producer Context*'s gate is the point where the place's data may be accessed. In order to make this data accessible, the *Producer Class*'s gate has to be synchronized with the *Producer Context*'s gate, creating a flow of data from the place to the Class's gate and finally to the Con-

```
1  Class producerObject;
2  Interface
3      Use
4          Naturals;
5      Gate
6          offer _ : natural;
7      Method
8          store _ : natural;
9          produce;
10 Body
11     Place
12         data _ : natural;
13     Axioms
14         store a::
15                 -> data a;
16         (this = Self) =>
17                 produce With this . offer a::
18                 data a ->;
19     Where
20         a : natural;
21         this : typeProducerObject;
22 End producerObject;
```

Figure 6.3: Producer Class Layout

text's gate.

The *store* method is synchronized with the *store* method of the encapsulated object (Producer Context, line 16), passing along its argument named *a*, which will be saved in the encapsulated object's *Place* (Producer Class, lines 14 and 15).

There are several other approaches for defining this Context, such as defining an initial state for the object's *Place*, where the *Producer* data already exist. Or even defining the *Producer's* output equal for every *produce* call, making the presence of the *Producer Class* obsolete.

Consumer Context

The *Consumer Context* is very simple, since this entity will only have to issue requests to the buffer and receive the data sent by it. The requests will only be served if the buffer has available tokens. In order to act as a "Consumer" the context will have two methods, one to issue requests to the buffer and another to receive the data.

To save the received data, the *Consumer Context* must encapsulate one object containing one *Method* for data input, which saves the data in the object's *Place*.

A template for the encapsulated object is visible in Figure 6.4, while the layout for the *Consumer Context* is present in Figure 6.5.

The *consumerObject* Class method named *save* (line 8), has one argument typed *natural* and as was referred it saves the received data by adding it to the *savedData* Place (lines 13 and 14). The add instruction is achieved through the post condition *savedData a*, where *a* is also of type *natural*.

```
Class consumerObject;
Interface
    Use
        Naturals;
    Type
        typeConsumerObject;
    Method
        save _ : natural;
Body
    Place
        savedData _ : natural;
    Axiom
        save a::
                -> savedData a;
    Where
        a : natural;
End consumerObject;
```

Figure 6.4: Consumer Class Layout

```
Context Consumer;
Interface
    Use
        Naturals;
    Gate
        request _ : natural;
    Methods
        get _ : natural;
Body
    Use
        consumerObject;
    Object
        cObject : typeConsumerObject;
    Axioms
        get a With cObject . save a;
    Where
        a : natural;
End Consumer;
```

Figure 6.5: Consumer Context Layout

In the *Consumer Context* layout, the method *get* is synchronized with the method *save* provided by the *cObject* (line 15), which is an instantiation of the Class *consumerObject*. Therefore, when the *get* method is triggered it calls the *save*, adding the received data to the place of the encapsulated object.

Buffer Context

Just as the *Producer Context*, the *Buffer Context* will encapsulate one object, whose purpose is to save the received data until a request from any *Consumer Context* arrives. The Class from which the encapsulated object will be instantiated is named *Buffer Class*.

The *Buffer Class* has one *Place* to hold the data, named *queue* (line 14); a method to act as an input mechanism that adds data to the *Place* (line 11); a method to trigger data output (line 10); and, a gate which will output the save data (line 8). The layout of this *Class* is visible in Figure 6.6.

```
1  Class BufferClass;
2  Interface
3      Use
4          Naturals;
5      Type
6          repObjType;
7      Gate
8          send _ : natural;
9      Methods
10         request;
11         put _ : natural;
12 Body
13     Places
14         queue _ : natural;
15     Axioms
16         put a::
17                 -> queue a;
18         (this = Self) =>
19                 request With this . send a::
20                 queue a->;
21     Where
22         a : natural;
23         this : repObjType;
24         b : natural;
25 End BufferClass;
```

Figure 6.6: Buffer Class

The *request* method is synchronized with the gate *send* (lines 18-20), making data from the place *queue* available for output.

With the *Buffer Class* defined, the *Buffer Context's* CO-OPN specification for this context is present in Figure 6.7.

The *Buffer Context* can be seen as an interface to this *Class* instantiation, providing one method which will be synchronized with the *Class*'s input method (Buffer Context, line 8)

```
1  Context BufferContext;
2  Interface
3      Use
4          Naturals;
5      Gate
6          send _ : natural;
7      Methods
8          put _ : natural;
9          request;
10 Body
11     Use
12         RepObj;
13     Object
14         repobj : repObjType;
15     Axioms
16         put a With repobj . put a;
17         repobj . send a With send a;
18         request With repobj . request;
19     Where
20         a : natural;
21 End BufferContext;
```

Figure 6.7: Buffer Context Layout

and one gate that will output the tokens provided by the *Class*'s gate (Buffer Context, line 6).

When the *request* method (Buffer Context, line 9) is called it is synchronized with the method *request* of the encapsulated object (Buffer Context, line 18). The *send* gate it is also synchronized with the *send* gate of the encapsulated object, making the data available for the *Consumer Context* (Buffer Context, line 17). Saving data is straightforward, after receiving a *put* method call the data is sent to the *put* method of the encapsulated object (Buffer Context, line 16), which stores the data in the correct *Place* (Buffer Class, lines 16 and 17).

System Coordinator Context

The *System Coordinator Context* sets up the interactions between the previous Contexts. The layout for this context is present in figure 6.8.

This context encapsulates the *Producer*, the *Consumer* and the *Buffer Contexts*, which is stated in the *Use Contexts* field. The *Coordinator* also has two methods: (1), *produce* (line 7); and (2), *consume* (line 8).

The first method, calls *store* followed by *produce* (lines 15 and 16), both methods belonging to the *Producer Context*. The second call — *produce In Producer* — triggers a set of transitions in the *Producer Context* which ends up by calling the second axiom *offer In Producer a With put In Buffer a* (line 17). The argument used in the *store* method is an instantiation from the *Naturals ADT* (line 4) and does not have any special meaning to the method.

The second method synchronizes itself with the *request* method of the *Buffer Context* (line 19) which triggers transitions in the *Buffer* that end up by calling the *get* method of the *Consumer* (line 18).

```
Context SystemCoordinator;
Interface
    Use
        Naturals;
        producerObject;
    Methods
        produce;
        consume;
Body
    Use Contexts
        Producer : ProducerContex;
        Consumer : ConsumerContext;
        Buffer : BufferContext;
    Axioms
        produce In Container With
                        store In Producer 1 . . produce In Producer;
        offer In Producer a With put In Buffer a;
        send In Buffer a With get In Consumer a;
        consume In Container With request In Buffer;
    Where
        a : natural;
End SystemCoordinator;
```

Figure 6.8: System Coordinator Layout

6.1.2 Multiple Producers and Consumers

Following the ideas drawn by the single producer and consumer example it is easier to generalize the problem to support multiple producers and consumer.

Focusing on the producer side, the only modification required is the number of *Producer Contexts* that need to be deployed in the *System Coordinator Context*, although since CO-OPN does not support multiple instantiations of a Context module, it is required to create a module for each desired producer. Aside from this, the old *Producer Context* module does not need any further modifications.

As we are dealing with multiple consumers the buffer has the need to know which one is requiring data, therefore the *request* method now needs to include one argument that identifies the consumer. The number of the buffer's output gates, named *send* in the single consumer version, also will vary accordingly with the number of consumers, because CO-OPN only supports one listener per gate.

6.2 Performance and Comparison Tests

In order to acquit if a parallel run-time provides a stable, scalable and high performance platform to execute CO-OPN specifications, the Producer–Consumer problem presented above was used in conjunction with a multitude of test benches. Parallel performance related items were compared with the execution of the same specification in the sequential run-time.

As CO-OPN is a formal and logical language, the execution of the Producer–Consumer

problem must follow this trivial rule "A *Consumer* can not acquire a token that was generated by a *Producer* at the same logical time". Therefore, a *Producer* must execute before a *Consumer*, which is translated into CO-OPN through this axiom *Producer.produce .. Consumer.consume*.

Multiple producers and consumers may be paired using the previous axiom. Although in order to have multiple *Producers* executing simultaneously the axiom have to be transformed into *(Producer1.produce // (...) // ProducerN.produce) .. (Consumer1.consume // (...) //ConsumerN.consumer)*. A careful analysis of the axiom shows that ConsumerN may use the token generated by ProducerM.

Both the single and the multiple version of the Producer—Consumer problem allows the parallel run-time to be tested for synchronization handling details which is also called *Functional Validation*, but performance related indicators are not stressed out uniquely through the variation of the number of *Producers/Consumers*.

To take accurate measures of performance, the generated code for the *Producers* was posteriorly modified to contain a cpu bound operation before sending the data to the *Buffer*. The chosen operation was an approximation of the PI constant that allows the programmer to vary the number of used samples, adapting the work amount of needed computation. This variation shows how the system performance changes when the amount of work increases or decreases.

Summarizing, the test bench will compare the performance of the parallel execution with the sequential one through the variation of two factors: (1) number of *Producers* and *Consumers*; and (2) amount of work done by the *Producers*. The first factor may take four distinct values $\{1; 2; 4; 8\}$, where each of the values is the number of the *Producers* and the *Consumers*, e.g., "1" states that the system is composed of one *Producer* and one *Consumer*. The second factor is a number of this set $\{300 \times 10^5; 100 \times 10^5; 50 \times 10^5; 30 \times 10^5; 20 \times 10^5; 10 \times 10^5; 5 \times 10^5\}$.

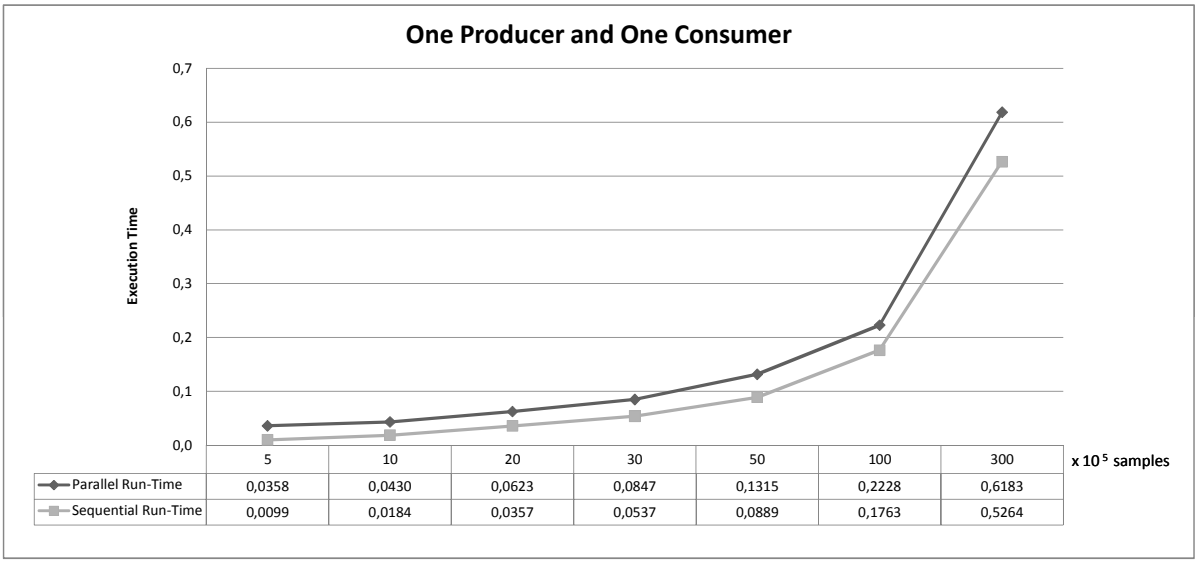
Each batch of tests was a combination of the first factor and the second, e.g., (1) two Producers and two Consumers with (2) 2000000 samples of work. To attain a result for each available combination, its test was comprised of two hundred iterations of the *produce-consume* axiom, and the performance result is the mean time value of all iterations, except the maximum and minimum values.

All tests were performed on a machine with two quad cores AMD Opteron at 2.3Ghz with 512KB of cache, running Linux with the 2.6.16.46-0.12-smp kernel.

6.2.1 Test Results

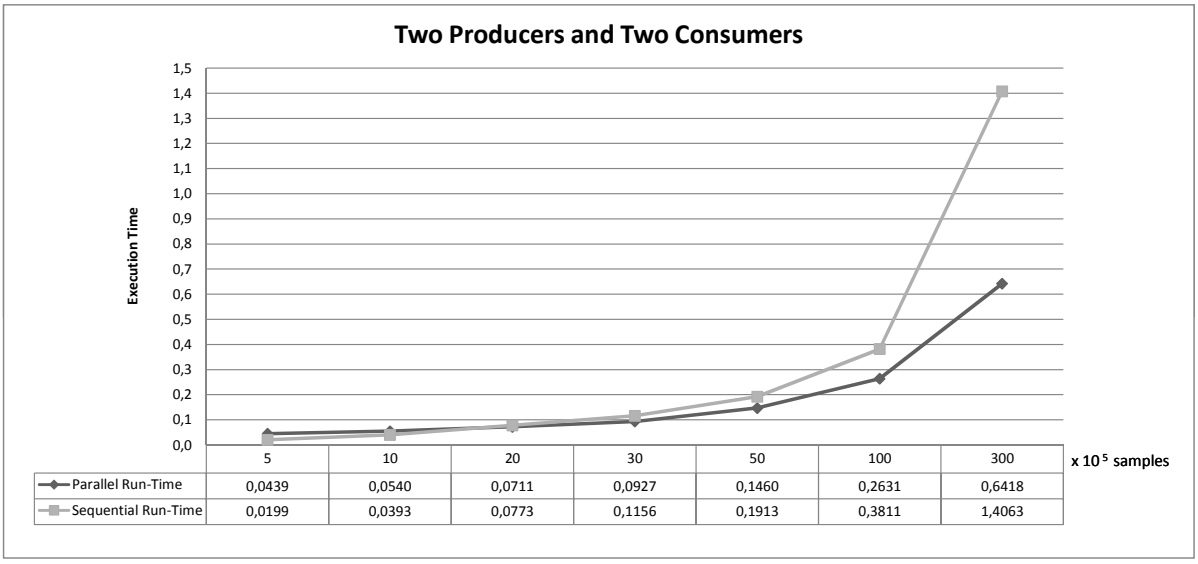
Figures 6.9, 6.10, 6.11 and 6.12 are graphical representations for the execution results of the different configurations. Each graph compares how the system performance varies with the amount of work the *Producers* have to complete before sending data to the *Buffer*. Figure 6.9 uses one *Producer* and one *Consumer*; and Figures 6.10, 6.11 and 6.12 show the system results with two, four and eight *Producers* along with two, four and eight *Consumers*, respectively.

The first graph, one *Producer* and one *Consumer*, illustrates that the sequential execution outperforms the parallel execution, but not by far. The small gap between the two execution results is due to the overhead introduced by the Parallel run-time breadth-first execution mech-



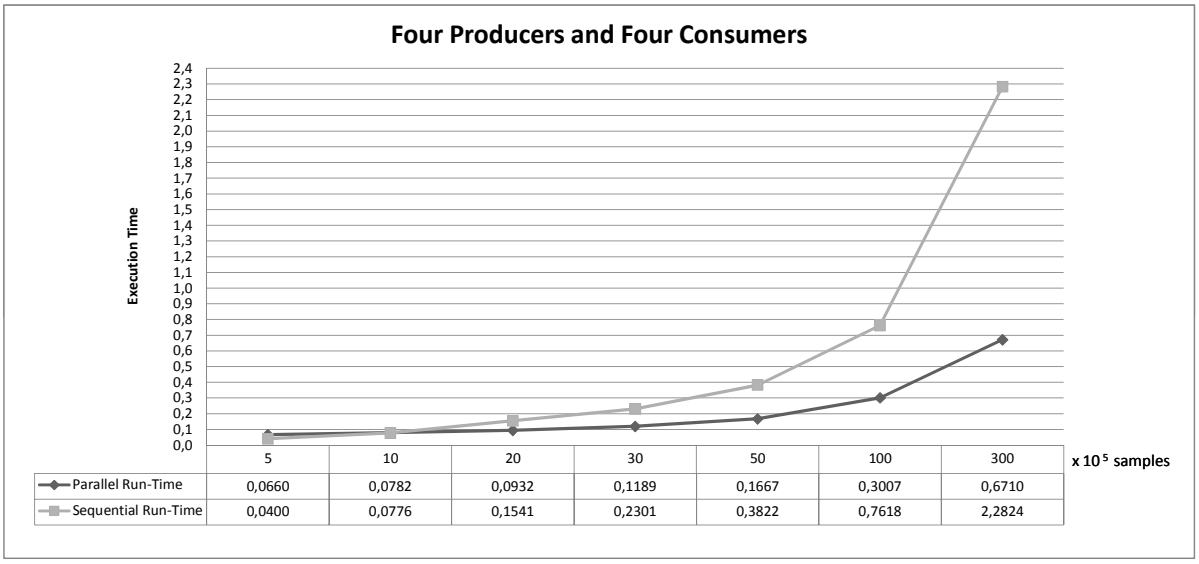
	5	10	20	30	50	100	300
Parallel Run-Time	0,0358	0,0430	0,0623	0,0847	0,1315	0,2228	0,6183
Sequential Run-Time	0,0099	0,0184	0,0357	0,0537	0,0889	0,1763	0,5264

Figure 6.9: One Producer and One Consumer



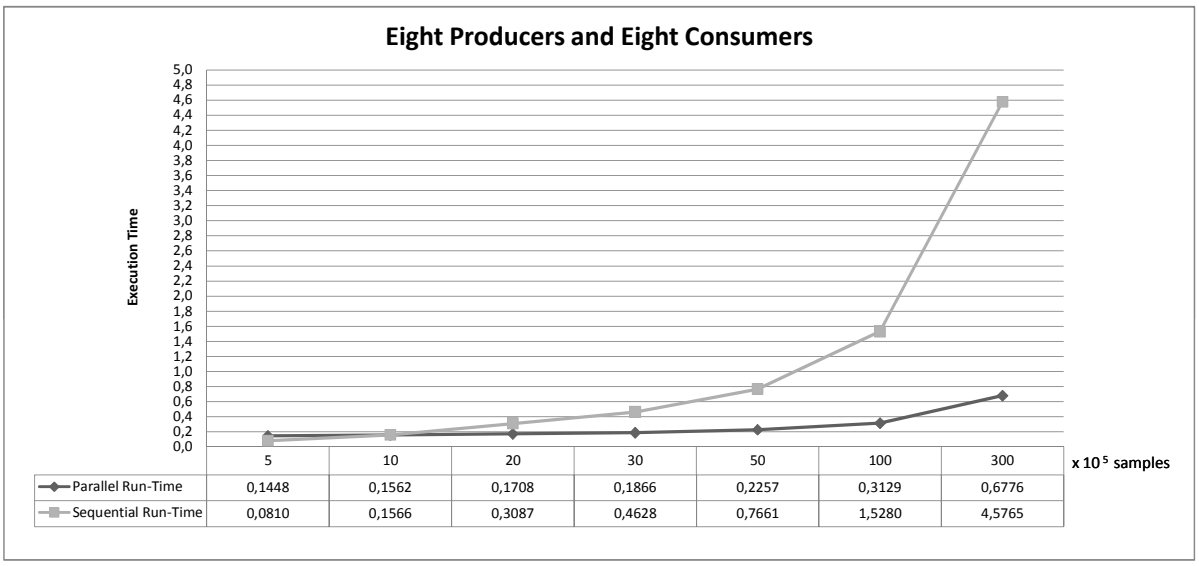
	5	10	20	30	50	100	300
Parallel Run-Time	0,0439	0,0540	0,0711	0,0927	0,1460	0,2631	0,6418
Sequential Run-Time	0,0199	0,0393	0,0773	0,1156	0,1913	0,3811	1,4063

Figure 6.10: Two Producers and Two Consumers



	5	10	20	30	50	100	300
Parallel Run-Time	0,0660	0,0782	0,0932	0,1189	0,1667	0,3007	0,6710
Sequential Run-Time	0,0400	0,0776	0,1541	0,2301	0,3822	0,7618	2,2824

Figure 6.11: Four Producers and Four Consumers



	5	10	20	30	50	100	300
Parallel Run-Time	0,1448	0,1562	0,1708	0,1866	0,2257	0,3129	0,6776
Sequential Run-Time	0,0810	0,1566	0,3087	0,4628	0,7661	1,5280	4,5765

Figure 6.12: Eight Producers and Eight Consumers

anism. As one can see the synchronization overhead is almost constant and less than 0.1ms in every test.

When the Sequential Run-Time has to deal with two *Producers*, the graph 6.10 shows that at 20 x 10^5 samples the average execution time is the same as the time spent by the Parallel Run-Time. From 20 x 10^5 samples to 300 x 10^5, the Parallel Run-Time outperforms the Sequential Run-Time.

The graphs in Figures 6.11 and 6.12 substantiate the previous results, although the difference between the average execution time of both Run-Times become larger when the amount of work and the number of *Producers* increase.

The number of samples at which the execution time of the Parallel Run-Time outperforms the Sequential Run-Time also decreases when the number of *Producers* increase. This occurs since the overall work load increases with the number of *Producers*.

Despite of the increase of communication and synchronization overhead imposed by the Parallel Run-Time, when the *Producers* have a bigger amount of samples to compute it is faster to deploy multiple threads (e.g., one for each producer) to do the computations than to serialize the needed computations.

6.2.2 Results Discussion

Graph 6.13 summarizes the comparison between the average execution time between both Run-Times. The Sequential Run-Time performance stands at level "1.0", so as an example, the Parallel Run-Time is approximately half as fast when two *Producers* compute 5×10^5 samples each.

For each batch, the overall amount of needed computation is defined as

the numbers of producers $\times$ *the number of samples*.

After an analysis of Figure 6.13 one would say that the greater the amount of computations, the greater is the parallel speedup over the sequential execution. Although, if the reader studies the graph carefully, he or she will notice that when the number of producers varies for small amount of computations (5 x 10^5 and 10 x 10^5 samples), the speedup gain is null or even worse when moving from 4 to 8 producers with 5 x 10^5 samples.

This decrease of speedup is caused mainly due to communication and synchronization overhead. The communications with the Time Stamp Manager are a major bottleneck of the system only overran when the overall amount of computation grows.

When the amount of work is maximum, 8 producers $\times$ (300 $\times$2 10^5) samples, the Parallel Run-Time performs 6.75 times better than the Sequential Run-Time, this number translates into a difference of execution time of 4 seconds, which is massive in computer terms.

The used example is purely theoretical, it would be interesting to study the behavior of the Parallel Run-Time when dealing with a real distributed system specification.

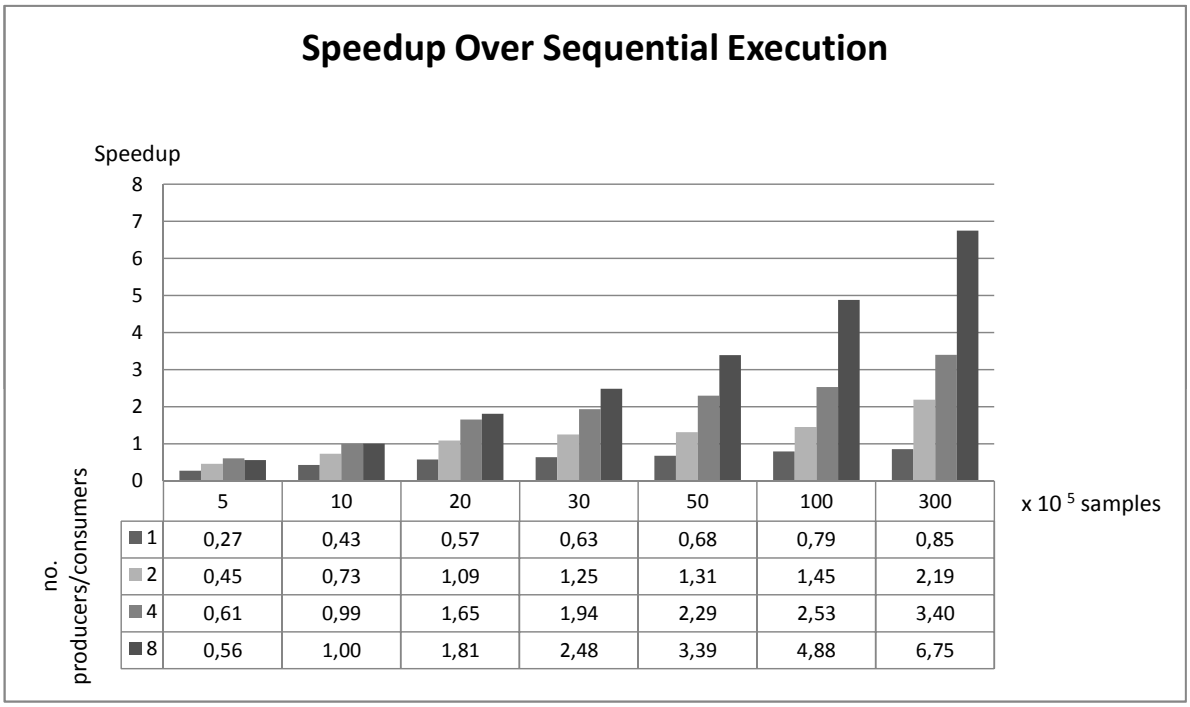


Figure 6.13: Speedup over Sequential Run-Time

7

Conclusions

The main goal of this dissertation was to provide CO-OPN with a stable concurrent execution environment for its specifications. To achieve this goal, the requirements for parallel execution were carefully analyzed and studied, where the biggest concern was the need to respect CO-OPN's synchronization operator semantic.

The requirements could be met either by following a modified version of the execution policy implemented in the Sequential Run-Time architecture or by an alternative Breadth-First execution policy. The new execution policy, is a breadth-first like traversal of the synchronization trees, which combines multi-threading with synchronization mechanisms to ensure an execution conforming to the original CO-OPN's semantics.

This dissertation also found a different type of requirement for the parallel execution. This requirement was not linked with the language itself, but with the execution of CO-OPN's specifications. Since these specifications could be translated into real distributed systems, a language named *CO-OPNArch*, aiming at specifying which and where entities should be distributed was defined and implemented as well. With *CO-OPNArch*, the developer can enrich the CO-OPN language with details about the physical distribution of the specification entities.

To transform CO-OPN specifications into Java programs that comply with the Breadth-First execution policy, this work deployed a new set of modular code generators, one for each type of CO-OPN components.

The parallel run-time is comprised of several components, specifically, the"Sync Manager" for managing synchronization calls in *methods* and *gates*, and the "Time Stamp Manager" to control the execution progress. Together, both components implement the Breadth-First execution policy.

At last, but not of less importance, the Parallel Run-Time was throughly tested using CO-OPN specifications, including the "Producer-Consumer" problem reported in this dissertation.

All testing results were compared with the results achieved by the Sequential Run-Time while executing the same specification. An analysis of the comparison results shows that this dissertation goal was effectively accomplished in producing concurrent code that complies to all CO-OPN semantics and performs better under well defined conditions, namely if the overall amount of needed computation masks the communication and synchronization latency.

7.1 Future Work

Despite of the achievements of this dissertations in the field of parallel execution of CO-OPN specifications, some doors were opened and there is a multitude of improvements that can be studied in future works:

- Develop a graphical editor for *CO-OPNArch*. Nowadays, CO-OPN architectural information is specified in plain text format, what proves to be non-user friendly and time consuming, depending on the hierarchy size of the CO-OPN specification. Thus, a graphical editor with full support for displaying the specification hierarchy would simplify the task;
- Implement the Depth-First execution policy and make performance comparisons with the already implemented Breadth-First policy. As processing power is growing almost daily, maybe the optimistic approach of Depth-First can offer better results;
- The following suggestions step outside the focus of this work, although, in order to take full advantage of the Parallel Run-Time, CO-OPN could be extended in two ways:
  - Offer a mechanism to specify a set of instructions that should be part of a *main* method. This way, the initial behavior of the system could be specified during the CO-OPN specification design;
  - Include *Black-Boxes* in CO-OPN syntax. The *Black-Boxes* could be specified likewise methods and gates, but their axioms besides containing pre, test, post conditions and synchronizations could also encapsulate blocks of Java code. Such an extension would increase CO-OPN expressivity.

Bibliography

[APR08] C. Adam Petri and W. Reisig. Petri net. *Scholarpedia*, 3(4):6477, 2008.

[Arb96] Farhad Arbab. The iwim model for coordination of concurrent activities. In *COORDINATION '96: Proceedings of the First International Conference on Coordination Languages and Models*, pages 34–56, London, UK, 1996. Springer-Verlag.

[BA07] B. Barroca and V. Amaral. (h)all: a dsvl for designing user interfaces for control systems. In *Proceedings of the 5th Nordic Workshop on Model Driven Engineering NW-MoDE 2007 27-29 August 2007*. Blekinge Institute of Technology, 2007. URL=http://www.ituniv.se/ miroslaw/node.htm.

[BB97] Mathieu Buffo and Didier Buchs. A coordination model for distributed object systems. In *COORDINATION '97: Proceedings of the Second International Conference on Coordination Languages and Models*, pages 410–413, London, UK, 1997. Springer-Verlag.

[BBG97] O. Biberstein, D. Buchs, and N. Guelfi. Co-opn/2: A concurrent object-oriented formalism. In *In Proc. Second IFIP Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS*, pages 21–23. Chapman and Hall, 1997.

[BN83] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. In *SOSP '83: Proceedings of the ninth ACM symposium on Operating systems principles*, page 3, New York, NY, USA, 1983. ACM.

[CB01] Stanislav Chachkov and Didier Buchs. From formal specifications to ready-to-use software components: The concurrent object oriented petri net approach. In *Second International Conference on Application of Concurrency to System Design, Los Alamitos*, 2001.

[CBU] Co-opn builder. http://sourceforge.net/projects/coopn/.

[CD01] George F. Coulouris and Jean Dollimore. *Distributed systems: concepts and design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[CHA04] Stanislav CHACHKOV. Generation of object oriented programs from co-opn specifications. Master's thesis, EPFL, Lausanne, Switzerland, 2004.

[DFL02] Yann Orlarey Dominique Fober and Stephane Letz. Lock-free techniques for concurrent access to shared objects, 2002.

[DSL] Domain-specific language. `http://en.wikipedia.org/wiki/Domain-specific_language`.

[FH07] Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2):5, 2007.

[GD94] Joseph Goguen and Razvan Diaconescu. An oxford survey of order sorted algebra. In *Mathematical Structures in Computer Science*, pages 363–392, 1994.

[Gog92] Joseph A. Goguen. Order-sorted algebra i: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105:217–273, 1992.

[Her90] M. Herlihy. Apologizing versus asking permission: optimistic concurrency control for abstract data types. *ACM Trans. Database Syst.*, 15(1):96–124, 1990.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[HP-] Hp-mpi user's guide. `http://docs.hp.com/en/B6060-96022/index.html`.

[HS08] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, March 2008.

[JPH95] Michael Krogh James Painter, Patrick McCormick and Charles Hansen. The acl message passing library. Los Alamos National Laboratory , USA and Guillaume Colin de VerdiÃ¨re,Centre dÉtudes de Limeil-Valenton, F-Villeneuve-Saint-Georges, 1995.

[KT08] Steven Kelly and Juha-Pekka Tolvanen. *Domain Specific Modeling: Enabling Full Code Generation*. John Wiley and Sons, New Jersey, 2008.

[MPI] Mpi manual. `http://www.lam-mpi.org/download/files/7.1.4-user.pdf`.

[MPW89] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, part i. *I and II. Information and Computation*, 100, 1989.

[MS95] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. Technical report, Rochester, NY, USA, 1995.

[MS98] Maged M. Michael and Michael L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *J. Parallel Distrib. Comput.*, 51(1):1–26, 1998.

[PAR] Netlib.org on parmacs. `http://netlib.org/utk/papers/comp-phy7/node4.html`.

[PVM] Pvm tutorial. `http://www.csm.ornl.gov/pvm/`.

[RA07] Matteo Risoldi and Vasco Amaral. Towards a formal, model-based framework for control systems interaction prototyping. In Nicolas Guelfi and Didier Buchs, editors, *Rapid Integration of Software Engineering Techniques, Third International Workshop, RISE 2006, Geneva, Switzerland, September 13-15, 2006. Revised Selected Papers*, volume 4401 of *Lecture Notes in Computer Science*. Springer, 2007.

[Rei91] Wolfgang Reisig. Petri nets and algebraic specifications. *Theor. Comput. Sci.*, 80(1):1–34, 1991.

[Sch00] Douglas C. Schmidt. Object behavior pattern for concurrent programming. *C++ Report*, 12, 2000.

[Sis99] A. Prasad Sistla. Safety, liveness and fairness in temporal logic. In *Formal Aspect of Computing*, pages 495–511, 1999.

[TAN95] Andrew S. TANENBAUM. *Distributed Operating Systems*. Prentice Hall, 1995.

[vDKV00] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices*, 35:26–36, 2000.

[VVK05] Hagen Völzer, Daniele Varacca, and Ekkart Kindler. Defining fairness. pages 458–472, 2005.

[Zim] Armin Zimmermann. Petri nets. `http://pdv.cs.tu-berlin.de/~zimmermann/petri.html`.