Pós-Graduação em Ciência da Computação

“ALGEBRAIC LAWS FOR PROCESS SUBTYPING”

Por

JOSÉ DIHEGO DA SILVA OLIVEIRA

Dissertação de Mestrado

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

RECIFE, AGOSTO/2011

Universidade Federal de Pernambuco
Centro de Informática

José Dihego da Silva Oliveira

ALGEBRAIC LAWS FOR PROCESS SUBTYPING

*Trabalho apresentado ao Programa de Pós-graduação em
Ciência da Computação do Centro de Informática da Uni-
versidade Federal de Pernambuco como requisito parcial
para obtenção do grau de Mestre em Ciência da Com-
putação.*

Orientador: *Augusto Sampaio*

Recife
Agosto, 2011

Catálogo na fonte
Bibliotecária Jane Souto Maior, CRB4-571

Oliveira, José Dihego da Silva
Algebraic laws for process subtyping / José Dihego
da Silva Oliveira - Recife: O Autor, 2011.
ix, 121 folhas

Orientador: Augusto Sampaio.
Dissertação (mestrado) - Universidade Federal de
Pernambuco. CIn, Ciência da Computação, 2011.

Inclui bibliografia e apêndice.

1. Engenharia de Software. 2. Linguagem de
programação. I.Sampaio, Augusto (orientador). II. Título.

005.1

CDD (22. ed.)

MEI2011 – 116

Dissertação de Mestrado apresentada por **José Dihego da Silva Oliveira** à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título "**Algebraic Laws for Process Subtyping**", orientada pelo **Prof. Augusto Cezar Alves Sampaio** e aprovada pela Banca Examinadora formada pelos professores:

Prof. Alexandre Cabral Mota
Centro de Informática / UFPE

Prof. Marcel Vinicius Medeiros Oliveira
Departamento de Informática e Matemática Aplicada / UFRN

Prof. Augusto Cezar Alves Sampaio
Centro de Informática / UFPE

Visto e permitida a impressão.
Recife, 12 de agosto de 2011.

Prof. Nelson Souto Rosa
Coordenador da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco.

*A meus pais Genário e Lúcia, meus irmãos Igo e Luana e
minha noiva Ivana.*

AGRADECIMENTOS

Deitado em um rede no Sítio Serra da Arara III, distante duas léguas de Cajazeiras, cidade do alto Sertão Paraibano, observo duas sabiás a fazer um ninho no tronco de uma palha de coqueiro; minha mãe, Lúcia, esta na cozinha a preparar um capote. Meu pai, Genário, assiste seu programa preferido sobre o mundo rural. É domingo, e meus dois irmãos, Igo e Luana, estão a pescar no barreiro perto de casa. Penso em meu amor, Ivana, que pensa em mim e neste momento eu sinto a brisa passar de leve... Ai amigos, eu agradeço a Deus, autor de todas as coisas, pai eterno e maestro do universo, que me deu o dom de apenas admirar sua criação, entender somente um pouco e amá-la em toda a sua extensão.

Obrigado meu Deus por ter chegado a este ponto de minha vida acadêmica e pessoal, obrigado por todas as pessoas que contribuíram direta ou indiretamente para a concretização deste trabalho, e que gratamente nomeio.

Meu pai, Genário, que da terra proveu o sustento de toda a minha família e de seu entendimento nos mostrou, a mim e meus irmãos, o valor da educação. A minha mãe Lúcia, pela sua cumplicidade, sua total e devota entrega a família, ao lar, ao trabalho com os enfermos. A meus irmãos Igo e Luana, que foram e sempre serão eternos companheiros de aventuras na roça. A minha noiva Ivana, meu amor, minha companheira, incentivadora e ‘gerente’ do andamento deste trabalho.

Agradeço de forma muito sincera ao grande orientador e amigo, Augusto Sampaio, que orquestrou todo o meu trabalho com muita capacidade, honestidade e domínio. Agradeço pela imensurável paciência com minhas limitações técnicas e lingüísticas. Agradeço por cada email respondido, por cada ligação, por cada reunião proveitosa e esclarecedora e por fim, por todo o conhecimento que me passou com generosidade.

Agradeço aos colegas da Embasa-GSAN, pela amizade inesquecível: Lucas, Juci, Márcio, Jeferson, Betinho, Cynthia,... Agradeço aos colegas professores do IFBA, em especial ao general Lúciano, pelos conselhos, braço forte e mão amiga. A Ricardo, pela amizade e apoio.

Agradeço por fim a Deus por ter tantas pessoas a agradecer.

*Faith, certainly, is a gift, a divine grace. But another divine gift is reason.
According to the ancient exhortations of the saints and doctors of the
Church, the Christian believes in order to understand; but he is also
called to understand in order to believe.*

—HIS HOLINESS JOHN PAUL II

RESUMO

Uma abordagem formal é crucial na especificação e desenvolvimento de sistemas complexos. Inspirado pela engenharia, o desenvolvimento de software deve preterir a abordagem empírica e seguir uma abordagem estruturada, formal, passível de repetição e prova face ao advento de sistemas mais complexos, paralelos e concorrentes.

Este trabalho apresenta uma extensão conservativa de *OhCircus*, uma linguagem de especificação concorrente, que integra CSP, Z, orientação a objetos e um cálculo de refinamento. Esta extensão suporta a definição de herança de processo, onde fluxo de controle, operações e componentes de estado em um superprocesso, podem ser reusados por seus subprocessos. Neste trabalho nós apresentamos a gramática estendida de *OhCircus*, acompanhada por um conjunto de regras de tipos que lidam com as novas construções da linguagem.

Nós apresentamos, em termos da Unifying Theories of Programming definida por Hoare e He, a semântica formal de herança de processo e suas construções de suporte. A principal contribuição deste trabalho é um conjunto, formalmente provado, de leis algébricas que lidam com herança de processo. Nós também explanamos informalmente como essas leis podem contribuir para uma teoria de completude para *OhCircus*. Finalmente nossas leis são exercitadas através de um estudo de caso.

Palavras-chave: Herança Comportamental, *OhCircus*, UTP, Leis Algébricas

ABSTRACT

A formal approach is crucial for the specification and development of complex system. Inspired by engineering, the software development must derogate the empirical approach in favor of a structured, repeatable, provable and formal approach faced with the advent of more complex, parallel and concurrent systems.

This work presents a conservative extension of *OhCircus*, a concurrent specification language, which integrates CSP, Z, object-orientation and a refinement calculus. This extension supports the definition of process inheritance, where control flow, operations and state components in a superprocess are eligible for reuse by its subprocesses. In this work we present the extended *OhCircus* grammar accomplished by a set of typing rules dealing with the new constructs.

We give, based on Hoare and He's Unifying Theories of Programming, the formal semantics of process inheritance and its supporting constructs. The main contribution of this work is a set of sound algebraic laws addressing process inheritance. We also informally explain how these laws can contribute to a completeness theory for *OhCircus*. Finally our laws are exercised by a case study.

Keywords: Behavioral Subtyping, *OhCircus*, UTP, Algebraic Laws

CONTENTS

Chapter 1 – Introduction	1
1.1 Process Inheritance and Refinement	2
1.2 Objectives and Structure of this Document	2
Chapter 2 – <i>OhCircus</i>	4
2.1 LRU Algorithm Specification in <i>OhCircus</i>	4
2.1.1 CSP: Process and Channel Declarations	7
2.1.2 Z: Schema Definitions	8
2.1.3 Process Refinement	9
2.1.4 Process Inheritance	11
2.2 Extending the LRU Algorithm	12
Chapter 3 – Redesigning Process Inheritance in <i>OhCircus</i>	14
3.1 Our Approach to Process Inheritance in <i>OhCircus</i>	14
3.2 Grammar Extensions	15
3.3 Revisiting the LRU Specification	16
3.4 The Benefits of this Approach	19
3.5 Final Considerations	24
Chapter 4 – Typing Rules for <i>OhCircus</i>	26
4.1 Typing	26
4.1.1 Data and phrase types	26
4.1.2 Typing Environment	27
4.2 Typing Rules	29
4.2.1 Programs	30
4.2.2 Environment Elaboration	33
4.3 Final Considerations	35
Chapter 5 – A UTP Semantics for Process Inheritance	37
5.1 Semantics of Inheritance	37
5.1.1 Explanatory Example	38
5.2 UTP Semantics for new Parallel Operator, Visibility and super Clause .	43
5.2.1 UTP Semantics for super and protected	45
5.3 Final Considerations	48

Chapter 6 – Algebraic Laws	50
6.1 The semantics of <i>Circus</i> processes	51
6.2 Laws	52
6.2.1 Access Modifications	52
6.2.2 Localized Eliminations	56
6.2.3 Element Interchanges	66
6.2.4 Subprocess Extraction	78
6.3 Final Considerations	81
Chapter 7 – Case Study and Completeness Notion	82
7.1 Case Study: a centralized store	82
7.2 <i>Store</i> Adaptation before Subprocess Extraction	88
7.3 Improvement in <i>Store</i> Design	92
7.4 Completeness	97
7.5 Final Considerations	98
Chapter 8 – Conclusion	99
8.1 Contributions	99
8.2 Related Work	100
8.3 Future Works	102
Appendix A – <i>OhCircus</i> original syntax	103
Appendix B – <i>OhCircus</i> extended syntax	105
Appendix C – UTP framework	107
C.1 <i>OhCircus</i> Processes	107
C.2 Schema Expressions	107
C.3 Command	107
Appendix D – Access Levels Equivalence	108
Appendix E – Algebraic Laws	109
E.1 Access Modifications	109
E.2 Localized Eliminations	110
E.3 Element Interchanges	113
E.4 Subprocess Extraction	116

CHAPTER 1

Introduction

Mathematics well-applied illuminates rather than confuses [40]. This has influenced, in one way or another, the application of formal methods in the software development process. The lack of formalism in earlier times has created a situation where software development became an ‘art’, restricted to few individuals, instead of an orderly and unambiguous process widespread among the software development community.

Some formal languages, which embody a refinement theory, have tried to support a rigorous software development process. Model-based languages like Z[43], VDM (Vienna Development Method) [26] and B[1] focus on system data aspects, without direct constructions for its behavioral aspects. Other languages, like CSP[23] and CCS (Calculus of Communicating Systems) [32] play a complementary role: they are designed to offer elegant and concise behavior modeling without attention to data representation.

This has motivated several attempts to unify these theories in a powerful language which consistently deals with behavioral and data aspects of a system. This is exactly what *Circus* [49] aims to be: a combination of Z and CSP, which includes Dijkstra’s language of guarded commands [16] and constructions in the style of Morgan’s refinement calculus [33]. With the intention to handle object orientation, in a Java [20] style, it was proposed the *OhCircus*[11] language, a conservative extension of *Circus*.

Other combinations have been proposed in the two past decades. In the CSP-Z [18] language, the behavior is defined purely in CSP, where events trigger state transitions specified in Z schemas; there is an implicit linkage between events and Z schemas. Another approach is CSP||B [42]. It is a combination of CSP and B machines. CSP processes control the communication between B machines. After defining the system’s data through B machines, processes in CSP are designed to control the communication flow between them. The main advantage of these languages is that the verification process can be made by different tools, separately designed, for CSP, B and Z.

In *Circus* and *OhCircus* events and state are decoupled, differently from CSP-Z and CSP||B, where there is an explicitly linkage between them. This characteristic of *OhCircus* encourages and simplifies the development of a refinement calculus not available in another languages. This is the main reason to use *OhCircus* in this work for the definition of our algebraic laws addressing process inheritance.

This work is concerned with the evolution of parallel and concurrent system models. In this context, evolution is the capacity to change a model, without changing its behavior, in order to satisfy non-functional requirements. For example, a client-server application can be modeled as a monolithic process aiming to test its communication protocol. The next step could involve the decomposition of the monolithic process into two: client and server, assuming that there is a non-functional requirement that preconizes the application must be able to run the client and server sides in different physical hosts. The most important

thing here is: the initial monolithic process must have the same behavior presented by the composition of the client and server processes.

The restructuring of programs has been a concern since the first programming languages were created [30]. Manually or supported by tools, these improvements are applied even when there is no guarantee of behavior preservation. The current challenge is the construction of transformations that assure, formally, behavior preservation. A refinement is defined as a relation, typically a partial ordering, involving meta variables standing for the program elements. *Circus* and consequently *OhCircus*, since the former is a sub-language of the latter, have a relevant set of refinements [10, 35, 9, 41]. Nevertheless the current refinements do not address process inheritance. The laws developed in Chapter 6 aim to contribute to a more comprehensive set of refinements and algebraic laws for *OhCircus*.

1.1 Process Inheritance and Refinement

Class inheritance, in object-orientated paradigm, is a well-established concept [28]; several works, based on the substitutability principle, have developed theories that recognize a valid inheritance relation between classes [28, 46, 3]. On the other hand, the semantics of process inheritance is not consolidated. Some of the most well known works about this topic [47, 19] have used the failures behavioral model of CSP to define a process inheritance relation.

In [47] active-objects (processes) are described using general labeled transition systems (LTS), which are a common framework used by behavioral languages for defining behavioral semantics. In that work, a process Q extends P , if the former refines (hiding its new events) the latter in the failures model. In our work we use a behavioral model based on failures and Z schema refinement [15] to characterize the subtype relation between processes in *OhCircus*, considering the integration between Z and CSP constructs. Based on this notion we propose sound refinements addressing process inheritance in *OhCircus*.

Process inheritance, as originally supported by *OhCircus*, has a practical disadvantage: there is no code reuse; state component declarations, invariants, initializers and other schemas, defined in a superprocess, are not inherited by its subprocesses. This increases code replication and makes inheritance a simple wrapper for parallel composition, as discussed in Chapter 2.

1.2 Objectives and Structure of this Document

The main goal of this work is the definition of sound refinements to support stepwise introduction or elimination of process inheritance and process elements in presence of this feature. As a consequence of this effort we develop an extended syntax, accomplished by a formal semantics, for *OhCircus*, redesigning process inheritance to allow code reuse; typing rules are developed to validate programs considering the new *OhCircus* syntax. We use the strategy ‘divide to conquer’ reflected in the structure of this work.

In Chapter 2 we explain the constructions of *OhCircus* through an example, where we

highlight its component languages CSP and Z (for simplicity we do not address object-oriented constructs in this introductory example). In the sequel, we briefly discuss the CSP process refinement models [23] and relate the failures behavioral model with the process inheritance semantics defined in [47]. Considering this semantics, adapted for *OhCircus* (by considering functionality extension and state space enlargement), we resume our example adding a subprocess, which motivates the need of changing *OhCircus* to allow code reuse.

In Chapter 3 we present an extended grammar for *OhCircus* which supports code reuse, through the creation of a new level of access for the superprocess elements. We discuss informally the new semantics of *OhCircus* updating the example from Chapter 2 with the new construction. We conclude by explaining the benefits of our approach in relation to the older version of *OhCircus* and other alternatives to process inheritance.

In Chapter 4 we present, based on Hoare and He's Unifying Theories of Programming [24], the typing semantics of the new constructs inserted in *OhCircus*. These typing rules are a conservative extension of those defined in [52] and strongly based on those defined for class inheritance in ROOL [12].

Following the typing semantics, in Chapter 5 we define a UTP semantics for the new constructs. This includes the formal definition of a new parallel operator that allows the branches to share state elements. This operator is used to give the semantics of process inheritance.

The major contribution of this work is presented in Chapter 6, where we define a set of sound algebraic laws addressing process inheritance. These refinements are classified into simple and composed laws. They are expressed in two directions of application, by an equality relation between metavariables standing for program elements. These laws have a set of preconditions in each direction of application considering the inheritance tree of the target process.

Our laws are exercised in a real and relatively elaborate example in Chapter 7, where we also informally explain how our algebraic laws can contribute to a relatively complete set of laws for *OhCircus*. Finally, in Chapter 8, we present our conclusions and future work.

CHAPTER 2

OhCircus

In this chapter we present *OhCircus* using an example. The initial specification of this example does not use process inheritance. We conclude this chapter extending the initial example using process inheritance as originally supported by *OhCircus*. Then, we emphasize why this does not allow elegant, concise and reusable specifications.

Since *OhCircus* is an integration of Z, CSP and object-orientated paradigm, we emphasize the main concepts of each language component, excepting object-orientation that is considered in the case study presented in Chapter 7.

The refinements proposed in Chapter 6 are closely related to the refinement models developed for CSP. Therefore we present some of these models and a proposal that relates them with process inheritance relation. The semantics of process inheritance, presented in Chapter 5 is based on UTP and we use this semantics in the proofs of our laws developed in Chapter 6. It is not our goal provide a test characterization scenario for process inheritance relations, so we do not provide an equivalent theory between behavior CSP models and UTP.

In the next chapter this example is revised to illustrate our proposal of redesigning process inheritance in *OhCircus*.

2.1 LRU Algorithm Specification in *OhCircus*

In multiprogramming systems, a set of process instances (or tasks) shares the same main memory offered by the underlying hardware in a given moment. This memory is limited and might not be enough to hold the active task demands. In an immediate view, this is an issue that application developers must deal with; in this approach the programmer must carefully switch its programs and data between the main and secondary memory (disk). This obligation is unacceptable in terms of security and productivity. This responsibility is commonly assigned to an operating system(OS) [45]; this allows the application developer to focus only on the business logic.

Ideally, the application developer does not need to know about the limited size of the main memory; the OS is responsible for the creation of a virtual main memory bigger than that offered by the underlying hardware. This goal is achieved by the use of a secondary storage as an extension of the main memory in a coordinate switch of data between them. The virtual memory is divided into pages of fixed size; is such a way that a page can be in the main or secondary memory. When the application generates an address, the OS translates it in a page identifier and an offset to the data inside that page. If it stays in main memory, the data is retrieved normally; otherwise the page is copied from the secondary to main memory, and after that the data is delivered.

When a page in the secondary memory is requested and there is not enough space in

the main storage, one of its pages must be chosen to go back to the secondary storage. This choice is the heart of a paging algorithm, because a bad decision implies in a high cost of switching pages between storage levels. A well known algorithm to perform this choice is the LRU (last recent used). The basic idea is to preserve the most recent used pages in the main memory and when space is requested, send the older used page to the secondary storage. We can specify this algorithm in *OhCircus* using an $n \times n$ matrix of bits[45], where n is the maximum number of pages allowed in the virtual memory. When a request for a page i ($1 \leq i \leq n$) arrives, all bits in the line i are set to 1 and those in the column i are set to 0. When a page needs to be chosen for passivation (to go back to the secondary memory), it will be that with the least value given by the bits in its line.

Suppose that we have a limited main memory with space to hold four pages. Each page is identified by a sequential number from 1 to n , in this example $n = 4$. If we access pages 1, 4, 2, 3, 1, in this order, the matrix will change as shown bellow. In the first matrix, since the page 1 is accessed, the bits in the line 1 are set with 1s and those in the collum 1 are set with 0s.

$$\begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

If a program generates an address that points to a page in the secondary memory, one of those in the main memory must be passivated. Following the algorithm given above page 4 will be chosen for passivation, since the bits in line 4 have the lower value.

A possible specification of this algorithm in the original version of *OhCircus* follows. The matrix is linearized as a sequence of bits, it simplifies the LRU specification, explained in the next subsections.

| $B : \{0, 1\}$

$[X]$	
$sum : \text{seq } X \rightarrow X$	
$\forall s : \text{seq } X \bullet$	
$sum \ s = \text{if } \#s = 1 \text{ then } s(1)$	
$\text{else } s(1) + sum(\text{tail } s)$	

channel $size, input, output : \mathbb{N}_1$

process $LRULog \hat{=} \text{begin}$

state St
$matrix : \text{seq } B$
$n : \mathbb{N}_1$
$\#matrix = n * n$

$Init$
St'
$n? : \mathbb{N}_1$
$n' = n?$
$matrix' = (\lambda x : 1 \dots n * n \bullet x \mapsto 0)$

$RegisterAccess$
ΔSt
$i? : \mathbb{N}_1$
$1 \leq i? \leq n$
$matrix' = matrix \oplus \{\lambda x : 1 \dots n \bullet (i? * (n - 1)) + x \mapsto 1\}$
$\oplus \{\lambda x : 1 \dots n \bullet i? + (x * (n - 1)) \mapsto 0\}$
$n' = n$

$ShowLRU$
ΞSt
$i! : \mathbb{N}_1$
let $pairs ==$
$\lambda k : 1 \dots n \bullet k \mapsto$
$sum(\lambda x : 1 \dots n \bullet matrix((k * (n - 1)) + x) * pow(2, n - x - 1))$
$\bullet i! \in \text{dom } pairs \wedge \min \text{ran } pairs = pairs(i!)$

$Input \hat{=} input?i \rightarrow RegisterAccess$

$Output \hat{=} \text{var } i : \mathbb{N} \bullet ShowLRU; output!i \rightarrow Skip$

$\bullet size?n \rightarrow Init;$
 $\mu X \bullet (Input \sqcap Output); X$
end

A program in *OhCircus* is a sequence of paragraphs: process/class definitions (class declarations appear in the study case presented in Chapter 7), Z paragraphs and channel definitions.

2.1.1 CSP: Process and Channel Declarations

In *OhCircus*, CSP is used to define the behavior exhibited by processes. The behavior of a process is determined by all possible sequences of communication events that it can engage with its environment. The communication between processes is established via channels or channel sets. In our example we have three channels: *size*, *input* and *output*. *LRULog* receives through *size* the number of pages that it must track. When a page is requested, its number is sent to the *LRULog* via *input*. Through *output* it sends to the environment the page number that must be removed when the main memory is full.

Each channel has a type. For example, $size : \mathbb{N}_1$ indicates that *size* communicates only natural numbers, except zero. An event represents a communication action via a channel. So when the environment sends the number 5 via *size*, the event *size.5* happens. Therefore since $\mathbb{N}_1$ is an infinite set, we have an infinite set of events associated to the communications through this channel.

A process in *OhCircus* has always a main action that defines its behavior, in other words, all sequences of communication events that it can engage. CSP has an extensive set of powerful operators acting over actions and processes. In the *Input* action, the operator $\rightarrow$ means that after engaging in an event *input.x*, where $x \in \mathbb{N}_1$, the process executes *RegisterAccess*. This highlights the integration between CSP and Z, since *RegisterAccess* is a Z schema, triggered by an CSP event, which act over *LRULog*'s state (we explain the Z part in the next subsection).

In the output action we see again the integration between CSP and Z. A link is made between the output of *ShowLRU* and the variable *i* (CSP allows variable declarations in actions). Whereupon *i* is written, by the execution of *ShowLRU*, its value is communicated to the environment via the *output* channel, then the process behaves like *Skip*, a process that represents a successful termination. A deadlock or design error is represented by *Stop*, a process that never engages in any action.

The process *LRULog*, after engaging in an event through the *size* channel, executes its initializer *Init*; an implicit binding is made between $?n$ and the input of this schema. The operator $;$ indicates that if and only if $size?n \rightarrow Init$ finishes successfully the process behaves like $\mu X \bullet (A)$; *X*, a recursive process that behaves like *A* and if *A* terminates successfully it behaves again like *A*, and so on. In our example, *A* stands for $Input \sqcap Output$.

The operator $\sqcap$ offers a choice between two actions or processes. In our example the environment must choose between the behavior exhibited in a page access (*Input*) or that responsible by the election of the page last recently used (*Output*). When the choice is decided by the environment, the behavior is captured by the external choice operator, as in our example. Otherwise, if the choice is taken internally by the process, the internal choice operator $\sqcap$ is used.

Processes or actions can also be composed in parallel. There are three forms of composing processes in parallel [38]: synchronous, alphabetized and interleaving. Considering that Σ stands for the set of all events that might occur in a specification, the alphabetized parallel composition of two processes $P \hat{=} ?x : A \rightarrow P'$ and $Q \hat{=} ?x : B \rightarrow Q'$ is given by:

$$\begin{aligned}
P_X \parallel_Y Q &\hat{=} ?x : C \rightarrow ((P'_X \parallel_Y Q') \\
&< x \in X \cap Y > \\
&(P'_X \parallel_Y Q) < x \in X > (P_X \parallel_Y Q'))
\end{aligned} \tag{2.1}$$

where P is allowed to communicate in the set X , its alphabet, and similarly Q in its alphabet Y (X and Y are subsets of Σ). P and Q must agree in the events belonging to their alphabets intersection, $X \cap Y$. The set C stands for:

$$C = (A \cap (X \setminus Y)) \cup (B \cap (Y \setminus X)) \cup (A \cap B \cap X \cap Y) \tag{2.2}$$

The first component $A \cap (X \setminus Y)$ represents the set of events that P might engage without the participation of Q , that is, the events in its alphabet X that are not in Q 's alphabet Y , but in A ; similarly the events that Q can perform on its own is given by $B \cap (Y \setminus X)$. The last component of C represents the events where both processes must synchronize.

In (2.1) if an event is in $X \cap Y$, by (2.2) it is also in $A \cap B$ and $P_X \parallel_Y Q$ behaves like $P'_X \parallel_Y Q'$, otherwise it behaves in two possible ways: like $P'_X \parallel_Y Q$ if the event is in X and, again by (2.2), is not in Y , but in A ; otherwise it behaves like $P_X \parallel_Y Q'$.

The synchronous parallel operator is a particular case of the alphabetized parallel composition where we have that the sets X and Y are equal to Σ . In this case:

$$P \parallel Q = P_\Sigma \parallel_\Sigma Q \hat{=} ?x : A \cap B \rightarrow P'_\Sigma \parallel_\Sigma Q' \tag{2.3}$$

The parallel composition by interleaving of P and Q represents the situation where these processes run completely independent of each other.

$$\begin{aligned}
P \parallel Q &\hat{=} ?x : A \cup B \rightarrow (P' \parallel Q) \sqcap (P \parallel Q') \\
&< x \in A \cap B > \\
&(P' \parallel Q) < x \in A > (P \parallel Q')
\end{aligned} \tag{2.4}$$

If an event belonging to $A \cap B$ occurs the choice of which process will be executed is nondeterministic. If an event is outside $A \cap B$, the process behaves like $(P' \parallel Q)$ if the event belongs to A , otherwise like $(P \parallel Q')$. Further details on the CSP notation can be found in [38].

2.1.2 Z: Schema Definitions

The Z language offers support for modeling systems that need elaborate data types and the operations over them. This capability is the main reason for its introduction in *OhCircus*. The *LRULog*'s state components are defined in the schema St , which declares *matrix* as a sequence of bits (B) and n as a natural number greater than zero ($\mathbb{N}_1$). In addition Z allows to define invariants over state components, which are predicates that

must hold in all states the system can achieve. In our example, *matrix* has always the size $n * n$.

Another schema is responsible by the state components initialization, obviously satisfying the invariant over them. *Init* plays this role in our example. It receives an input with the size of the quadratic matrix, initializes n with this value, and *matrix* with size $n * n$, fulfilling it with zeros. Note that such an initialization does not remove the obligation of initializing n . In the semantics of Z an invariant must be explicitly satisfied, otherwise the Z schemas are considered undefined, which means that they can do anything including non-termination (divergence) [31].

A notation is used in Z to differ the initial and final values of state components before/after the operations performed by a schema. The undashed *matrix*, in the schema *RegisterAccess*, indicates its initial value and the dashed *matrix'* its final value. Even if a state component has its value unchanged it is necessary to specify, explicitly, this fact as in *RegisterAccess* where we wrote $n' = n$.

The dashed state components are inserted in a schema by using the state schema name dashed. The same rule is applied to the insertion of undashed components. In our example the state components are declared in the *St* schema, and since *Init* only initializes these components, it includes the schema *St'*, which inserts *matrix'* and n' in the context.

Some schemas update state components based on these current values. This is the case of *RegisterAccess*, where we have both dashed and undashed versions of the *LRULog*'s state components. To create an alternative to the use of $State \wedge State'$, Z allows the use of the equivalent shortcut $\Delta State$. If a schema just reads the state component values we use the notation Ξ . In our example *ShowLRU* retrieves the line number of the *matrix* which has the lowest binary value¹. Therefore we use ΞSt in this schema.

Sometimes an operation performed by a schema needs inputs, generates outputs, or both. In Z, an input variable is declared with the question mark (?); the exclamation (!) indicates an output variable. In *RegisterAccess* the accessed page number is received by the input variable $i?$. $i!$ is an output variable of *ShowLRU*.

When a Z schema appears in a CSP action, generally its inputs come from input channels and its outputs are sent, to the environment, via output channels. The high integration of these languages allows the definition of modular, concise and reusable processes. More details about the Z notation can be found in [43]. In the sequel it is presented the notion of refinement in CSP and how it is applied to *OhCircus* considering the Z constructions. Finally, it is shown the linkage between refinement and process inheritance.

2.1.3 Process Refinement

Three models to define the behavior of a CSP process are, formally, established in [23]: traces, failures and failures-divergences. To understand these models let us present some background concepts.

¹In the case where two or more lines have the same lowest value, the schema has a nondeterministic behavior returning any of these line numbers.

A trace generated by a process execution is a finite sequence of symbols recording the events in which it has engaged up to some moment in time. In our example if the events *input.1*, *input.2*, *input.3* and *output.3* happen sequentially, the maximum trace active of *LRULog* is $\langle \text{input.1}, \text{input.2}, \text{input.3}, \text{output.3} \rangle$. In [23], the semantics is defined in terms of a function from any CSP process to its set of traces.

Another model to describe the process behavior is based on failures. A failure is a pair (s, E) meaning that after the trace $s \in \text{trace}(P)$ is performed, P refuses all events in the set of events E . The set of these pairs, considering P 's traces, is given by $\text{failures}(P)$. Formally it is defined as:

$$\text{failures}(P) = \{(s, X) \mid s \in \text{traces}(P) \wedge X \in \text{refusals}(P/s)\} \quad (2.5)$$

The function *refusals* for a process P gives the set of all events where P cannot engage considering its current state. But we are normally interested, in the above definition, in the refused events after the trace s , so we apply this function to P after s , P/s .

The relation between traces and failures models are given bellow:

$$\text{traces}(P) = \text{dom } \text{failures}(P) \quad (2.6)$$

Another powerful behavior model, failures-divergences, extends the failures model with the addition of the process divergences. A divergence of a process is defined as a trace that makes it behaves like *Chaos*, which is the most nondeterministic CSP process that performs, indefinitely, an infinite number of internal transitions.

$$\text{divergences}(P) = \{s \mid s \in \text{traces}(P) \wedge (P/s) = \text{Chaos}_{\alpha P}\} \quad (2.7)$$

These models of behavior representation are used to define three levels of process refinement in CSP. We present bellow, respectively, what means a process Q refining P considering the traces, failures and failures-divergences models.

$$\begin{aligned} P \sqsubseteq_T Q &\hat{=} \text{traces}(Q) \subseteq \text{traces}(P) \\ P \sqsubseteq_F Q &\hat{=} \text{failures}(Q) \subseteq \text{failures}(P) \\ P \sqsubseteq_{FD} Q &\hat{=} \text{failures}(Q) \subseteq \text{failures}(P) \wedge \text{divergences}(Q) \subseteq \text{divergences}(P) \end{aligned} \quad (2.8)$$

If a process Q refines P , we can replace every occurrence of the latter by the former, considering one of the three models. The weaker refinement relation is defined using the traces model, for example, $P \sqsubseteq_T \text{Stop}$, for any P , since *Stop* stands for a canonical deadlock and its trace semantics is given by the empty set of traces. If we consider, in addition to the traces, the set of refusal events, as is made in the failures model, it is possible to detect deadlocks; a process in deadlock refuses all events and is equivalent to *Stop*. In the failures model, assuming that Q refines P we have $(s, E_1) \in \text{failures}(P)$ and $(s, E_2) \in \text{failures}(Q)$ then $E_2 \subseteq E_1$; this implies that Q refuses after a trace s , the same or less events than P , no extra event must be signaled by the refusals of Q/s . The failures-divergences model is the strongest model, since it considers deadlocks and

livelocks. In this state, a process stays locked while executing an infinite number of internal transitions.

The failures model is used by, perhaps, the most well-known work [47] on process inheritance. It guarantees the substitutability principle, in the process refinement, considering deadlocks. It can be adapted to livelocks by using failures-divergences refinement instead of failures refinement as a basis in testing characterizations scenarios considering that every livelock in a subprocess has to have its counterpart in the superprocess. Based on [47], in our approach we extend the notion of failures refinement in CSP for *OhCircus*, defining that a process Q refines P in *OhCircus* if:

$$P \sqsubseteq Q \hat{=} failures(Q) \subseteq failures(P) \wedge refSchemas(Q, P) \quad (2.9)$$

The function $refSchemas(Q, P)$ is true when each Z schema of Q refines its counterpart in P , and false otherwise. The refinement of schemas is presented in Chapter 3. This addition is necessary because all operations in [47] are considered atomic, which is not the case in *OhCircus*, so we must guarantee in addition the refinement of schemas. The process refinement is a monotonic operation concerning Z schemas refinement.

2.1.4 Process Inheritance

Perhaps the most well-known approach to semantics for process inheritance is the one presented in [47]; it is based on the failures semantics of CSP [23]. The definition takes into consideration the relation between the failure sets of two processes to establish the existence and the type of inheritance. Four types of inheritance are defined: weak, safe, optimal and implementation subtyping. The weak subtyping is based on restriction of new events (forbids them) in a subprocess to establish a refinement relation with its superprocess. Safe subtyping guarantees that each new schema in a subprocess must be explained in terms of the old schemas in its superprocess, it is similar to the extension map idea presented in [28]. A restrict version of safe subtyping is optimal subtyping where the new schemas are mapped in the empty schema. The last type of inheritance relation is implementation subtyping, where the new events found in the subprocess are hidden, and makes them internal, invisible for the outside.

We address only one of the subtyping relations presented in [47]: implementation subtyping [47], because it satisfies the substitutability principle: in all contexts where the user is not able to inspect the internal representation or type of a particular process P , a subprocess Q , having its new events $(\alpha Q \setminus \alpha P)$ hidden, can be used where a (direct or indirect) superprocess is expected. This semantics is crucial in this work, but in its original form it does not address the reuse of code, which is a great motivation to introduce inheritance in a language. Also, it focus only on control behavior, and *OhCircus* involves data aspects as well. We choose implementation subtyping since safe and optimal subtyping are very restrictive on the definition of subprocess and weak subtyping by the opposite.

A process Q is a subprocess of P if $P \sqsubseteq Q \setminus (\alpha Q \setminus \alpha P)$. Considering the failures semantics, if $failures(Q \setminus (\alpha Q.act - \alpha P.act)) \subseteq failures(P)$ as proposed by [47], where $\alpha Q.act - \alpha P.act$ stands for the new events in $Q.act$ concerning $P.act$. CSP focuses on the behavior representation, with low support to data modeling. *OhCircus* addresses behavior and rich data representations, as well as inheritance of classes and processes support. The same concept of inheritance from [47] was brought to *OhCircus*. This is reflected in the obligation that a subprocess main action (its behavior) refines, in failures semantics, the main action (hiding its new events) of its superprocess. Therefore the substitutability principle is satisfied.

In [34] it is presented a complete account of the *Circus* [48] denotational semantics based on Hoare and He's Unifying Theories of Programming [24]. As *OhCircus* is a conservative extension of *Circus* we can use the semantics defined in [34] as a basis to formalise an inheritance notion. So if two process P and Q have, respectively, $P.act$ and $Q.act$ as their main actions, Q **extends** $P \Leftrightarrow P.act \sqsubseteq Q.act \setminus (\alpha Q \setminus \alpha P)$. The object orientation concepts present in *OhCircus* are not addressed in this work since our focus is on process inheritance. Nevertheless, object-oriented constructs are sufficiently independent and do not interfere in the subset of *OhCircus* we are tackling.

In the next section we present a subprocess of *LRULog* defined using the original support of *OhCircus* for process inheritance. The process *LRULogCount* is a proper subprocess of *LRULog* since $LRULog \sqsubseteq LRULogCount \setminus (\alpha LRULogCount \setminus \alpha LRULog)$. From now on, we use $P \sqsubseteq Q$ with the same meaning, unless we specify otherwise, of $P \sqsubseteq Q \setminus (\alpha Q \setminus \alpha P)$, in the context of process inheritance.

2.2 Extending the LRU Algorithm

In order to create a more deterministic (some nondeterminism remains) version of the *LRULog* process we define a subprocess, *LRULogCount*, that holds the number of accesses to each page between two page miss events, and it uses this information to take a better decision of which page must be moved back to the secondary memory.

process *LRULogCount* $\hat{=}$ **extends** *LRULog* **begin**

The process *LRULogCount* does not seize the opportunity of reusing *LRULog*'s state components and schema declarations, instead, it replicates them; this is a consequence of the fact that the original design of *OhCircus* does not provide any facility of reusing the elements of a process.

state St
$matrix : \text{seq } B$
$n : \mathbb{N}_1$
$countList : \mathbb{N}_1 \rightarrow \mathbb{N}$
$\text{dom } countList = 1 \dots n$

<i>Init</i>
<i>St'</i>
$n' = n?$
$matrix' = (\lambda x : 1 \dots n * n \bullet x \mapsto 0)$
$countList' = (\lambda x : 1 \dots n \bullet x \mapsto 0)$

The next schema does in addition to that found in *LRULog* the update of the number of access between two page miss events.

<i>RegisterAccess</i>
ΔSt
$i? : \mathbb{N}_1$
$1 \leq i? \leq n$
$matrix' = matrix \oplus \{\lambda x : 1 \dots n \bullet (i? * (n - 1)) + x \mapsto 1\}$
$\oplus \{\lambda x : 1 \dots n \bullet i? + (x * (n - 1)) \mapsto 0\}$
$n' = n$
$countList' = countList \oplus \{i? \mapsto countList(i?) + 1\}$

This version of *ShowLRU* set with zeros the *countList* since a page miss event happens.

<i>ShowLRU</i>
ΔSt
$i! : \mathbb{N}_1$
let <i>pairs</i> ==
$\lambda k : 1 \dots n \bullet k \mapsto$
$sum(\lambda x : 1 \dots n \bullet matrix((k * (n - 1)) + x) * pow(2, n - x - 1))$
$\bullet i! \in \text{dom } pairs \wedge \text{min ran } pairs = pairs(i!)$
$i! \in \text{dom } countList \wedge \text{min ran } countList = countList(i!)$
$countList' = (\lambda x : 1 \dots n \bullet x \mapsto 0)$

The *LRULog*'s main action is not replicated because, by the semantics of process inheritance, presented in Chapter 5, the behavior of a subprocess is given by its declared main action put in interleaving with that declared by its direct superprocess. Considering *Skip* as a process such that, $P \sqsubseteq Skip \parallel P$ (for any process P), we have that $LRULog \sqsubseteq LRULogCount$. Therefore the main action of *LRULogCount* is *Skip*.

• *Skip*
end

In the next chapter we present our solution to allow code reuse; it increases the conciseness and maintainability of specifications in *OhCircus* offering support for a more natural conceptual system design.

CHAPTER 3

Redesigning Process Inheritance in *OhCircus*

The *OhCircus* language was designed with the clear intent of providing direct syntactic support to process and class inheritance. Class inheritance is well-established. Its semantics was already defined by several works [28, 46, 3] and it is widely accepted both in industry and in academy. Process inheritance is not yet consolidated and what it exactly means is an open question. The proposition of sound algebraic laws that deal with process inheritance is the main goal of this work. Nevertheless, there is no consensual accepted semantics for process inheritance. We address this problem adopting the semantics presented in the works [17, 47]. This chapter explains this semantics and presents our approach to propose and formalise a notion of process inheritance in *OhCircus*. We conclude with a comparison, through examples, between the proposed and previous forms used to define processes in *OhCircus*.

3.1 Our Approach to Process Inheritance in *OhCircus*

In [47] inheritance is addressed from a purely semantic viewpoint. There is no syntactic connection between a subprocess and its superprocess. Therefore, there is no way for a subprocess to reuse code from its superprocesses. In this work we introduce such a link because this is traditional for class inheritance in object orientated programming languages like Java [20]. Following common practice for class inheritance, our design allows the subprocess to inherit state components, as well as to reuse and possibly redefine operations described as Z schemas. We have changed the syntax of *OhCircus* in two central ways: the creation of a new access level to allow visibility of a process element (state and schema operation) by its subprocesses (like the protected mechanism in Java) and the addition of a new form to define Z schemas [43], very similar to the Z schema inclusion feature, with the aim of allowing schema redefinitions. The **super** clause allows schemas in a subprocess to refer protected schemas (it is explained in the next paragraphs) in its superprocesses.

As originally designed, a process, both in *Circus* and *OhCircus*, is a black box with interaction points through channels that exhibits a behavior defined by its main action. The effort of introducing inheritance with this process structure is prohibitive because the benefits of code reuse cannot be reached and the introduction of a type hierarchy, by itself, is not enough to justify inheritance, from a practical perspective. This has motivated the new access level introduced in *OhCircus* to allow code reuse. The keyword **protected** signalises states and schemas belonging to this level.

We emphasize that a superprocess remains a black box to all other processes in its environment that are not in its inheritance hierarchy — even considering it is granted to a subprocess the permission to reuse the superprocess code in compilation time. It

should be clear that when we say that a subprocess inherits a superprocess component (state or Z schema), it means that this subprocess has a **copy** of each state or schema in the protected access level; it does not mean **sharing** of these elements. Therefore the subprocesses can take advantage of code reuse and process encapsulation continues as before. This is relatively simple to achieve in *OhCircus* because processes are not types; therefore no dynamic instances are created and, as a consequence, there is no need for a heap at runtime. In Chapter 5 we detail the semantics of this level of access in *OhCircus*.

The public access level, found in object orientation languages like Java, is not applicable to process elements (state components and schemas) since we maintain, as originally proposed for *OhCircus*, processes as black boxes exhibiting only their interaction points. The private access level, in the manner proposed for object orientation, is equivalent to the default access level in *OhCircus*. Further, as *OhCircus* has not the package concept, the default access level in object orientation has not a counterpart in our extension.

The other relevant language feature we propose is a notation to define schema overriding. If a subprocess has access to the superprocess schemas (for example, a protected schema s_τ) it is natural that it can override s_τ (s_σ overrides s_τ). This overriding must satisfy some syntactic conditions: s_τ and s_σ names must be equal and the contravariance of inputs and covariance of outputs must be satisfied [28]. Semantic restrictions are also necessary: the pre-condition of s_σ should be weaker than or equivalent to s_τ 's pre-condition; the post-condition of s_σ must be stronger than or equivalent to s_τ 's post-condition. It is also possible to include, using the keyword **super** (explained later), the original schema in the overriding version. In the next section we give the extensions to the *OhCircus* grammar that support the new features discussed.

3.2 Grammar Extensions

Appendix A presents the original syntax of *OhCircus* [11]. The changes made in the original version are presented in Appendix B and highlighted below.

```

OhProcessDefinition ::= process N  $\hat{=}$  [extends N] Process
Process              ::= begin
                        PParagraph*
                        [state N Schema-Exp | Constraint]
                        PParagraph*
                        • Action
                        end
                        | ...

```

$$\begin{aligned}
\text{PParagraph} &::= \text{SchemaText} \mid N \hat{=} \text{Action} \\
&\quad \mid \underline{[\text{PQualifier}]} N \text{SchemaText} \\
\text{SchemaText} &::= (\oslash N)^+ [\text{Declaration}^+] [\underline{\text{super.N}^+}] [\text{Predicate}] \\
\text{Schema-Exp} &::= (\underline{[\text{PQualifier}]} \text{Declaration})^* \\
\text{PQualifier} &::= \underline{\text{protected}}
\end{aligned}$$

A process paragraph (**PParagraph**) is now allowed to refer to one or more *Z* schemas defined in the process itself or inherited from its superprocesses, in any level of inheritance. Process inheritance is a transitive relation. It means that if $X \text{extends} Y \wedge Y \text{extends} Z$ then we have that $X \text{extends} Z$ in semantic terms. Syntactically a process must extend only one process, multiple inheritance is not allowed, mainly due to the possible duplication and ambiguity that arise from this feature. The inheritance relation is not reflexive, therefore, a subprocess has one or more superprocesses, each of them, necessarily, in a different level of the inheritance hierarchy (which is a tree).

A *Z* schema can be defined using an explicitly access modifier, **protected**, or if nothing is said, the default level (not inherited by subprocesses) is adopted. Only *Z* schemas in the protected level are eligible for use in a **super** clause. The others are hidden from subprocesses. The overriding of protected schemas is also supported and it allows a subprocess to refine a protected schema introduced in or inherited by the closest superprocess up in the inheritance tree.

Similar to schemas it is allowed to define an access level for each state component. It generates some restrictions in the subprocess state components declaration. This new syntax, and its restrictions, are exemplified in the sequel.

3.3 Revisiting the LRU Specification

Using the new syntax of *OhCircus* for process inheritance presented above we rewrote the example presented in Chapter 2 to reuse specification code from *LRULog* by its subprocess *LRULogCount*.

process *LRULog* $\hat{=}$ **begin**

state <i>St</i>
protected <i>matrix</i> : seq <i>B</i>
protected <i>n</i> : $\mathbb{N}_1$
$\#matrix = n * n$

<i>Init</i>	_____
<i>St'</i>	
$n? : \mathbb{N}_1$	
$n' = n?$	
$matrix' = (\lambda x : 1 \dots n * n \bullet x \mapsto 0)$	

protected <i>RegisterAccess</i>	_____
ΔSt	
$i? : \mathbb{N}_1$	
$1 \leq i? \leq n$	
$matrix' = matrix \oplus \{\lambda x : 1 \dots n \bullet (i? * (n - 1)) + x \mapsto 1\}$	
$\oplus \{\lambda x : 1 \dots n \bullet i? + (x * (n - 1)) \mapsto 0\}$	
$n' = n$	

protected <i>ShowLRU</i>	_____
ΞSt	
$i! : \mathbb{N}_1$	
let <i>pairs</i> ==	
$\lambda k : 1 \dots n \bullet k \mapsto$	
$sum(\lambda x : 1 \dots n \bullet matrix((k * (n - 1)) + x) * pow(2, n - x - 1))$	
$i! \in \text{dom } pairs \wedge \min \text{ran } pairs = pairs(i!)$	

$Input \hat{=} input?i \rightarrow RegisterAccess$

$Output \hat{=} \text{var } i : \mathbb{N} \bullet ShowLRU; output!i \rightarrow Skip$

• $size?n \rightarrow Init;$
 $\mu X \bullet (Input \sqcap Output); X$
end

In order to create a more deterministic (some nondeterminism remains) version of the *LRULog* process we define a subprocess, *LRULogCount*, that holds the number of accesses to each page between two page missing events, and it uses this information to take a better decision of which page must be moved back to the secondary memory.

process *LRULogCount* $\hat{=}$ **extends** *LRULog* **begin**

state St	
$countList : \mathbb{N}_1 \rightarrow \mathbb{N}$	
$\# \text{dom } countList = n$	
$Init$	
St'	
$countList' = (\lambda x : 1 \dots n \bullet x \mapsto 0)$	
$RegisterAccess$	
ΔSt	
super $RegisterAccess$	
$countList' = countList \oplus \{i? \mapsto countList(i?) + 1\}$	
$ShowLRU$	
ΔSt	
super $ShowLRU$	
$i! \in \text{dom } countList \wedge \min \text{ran } countList = countList(i!)$	
$countList' = (\lambda x : 1 \dots n \bullet x \mapsto 0)$	

The state component $countList$ must be reseted after each page missing events. This guarantees the temporal locality of the information used to move a page from the main memory to disk.

• *Skip*
end

The subprocess initialization schema sets the range of the relation $countList$, which maps a page index to the number of requests for this page since the last missing event, to zero. When a page is requested, the $RegisterAccess$ schema increases the counter page by one and holds the same behavior of its counterpart found in $LRULog$ by the reference made in the **super** clause.

The schema $ShowLRU$ is a more deterministic version than that found in $LRULog$. In the original version when two or more lines have the same lowest binary value we cannot determine which one will be returned. In the $LRULogCount$ version we use a tiebreaker criteria, based on the number of accesses between two events of the missing page, to take a more deterministic decision about what page must be passivated.

The *LRULogCount*'s main action is *Skip*. This means that the behavior of this process, by the process inheritance semantics, is given by $Skip \parallel LRULog.act_{ref}$, where the references for the schemas *RegisterAccess* and *ShowLRU* in $LRULog.act_{ref}$ actually use the schemas defined in *LRULogCount*, which are more deterministic than those in *LRULog*. We have used *Skip* because, if $LRULog.act_{ref}$ refines $LRULog.act$, it implies that $Skip \parallel LRULog.act_{ref}$ refines $LRULog.act$.

3.4 The Benefits of this Approach

The previous example shows our approach to the introduction of process inheritance in *OhCircus*. Some questions naturally arise: what are the benefits of this approach? Can the definition of *LRULogCount* as a self-contained process or parallel process composition be simpler than the use of inheritance? Is this feature really needed in the language? To help us answer these questions, we write *LRULogCount* as a self-contained process and via parallel composition. Then we compare these forms with that previously developed.

It is possible to define *LRULogCount* as a self-contained process, without inheritance. This process has the same state components found in *LRULog* and must define in its *Z* schemas the behavior exhibited by the schemas referenced by the **super** clause. The main action is syntactically equals to that of *LRULog*.

process *LRULogCount* $\hat{=}$ **begin**

state <i>St</i>	
<i>matrix</i> : seq <i>B</i>	
<i>n</i> : $\mathbb{N}_1$	
<i>countList</i> : $\mathbb{N}_1 \rightarrow \mathbb{N}$	
$\#matrix = n * n$	
$\# \text{dom } countList = n$	

<i>Init</i>	
<i>St'</i>	
<i>n?</i> : $\mathbb{N}_1$	
$n' = n?$	
$matrix' = (\lambda x : 1 \dots n * n \bullet x \mapsto 0)$	
$countList' = (\lambda x : 1 \dots n \bullet x \mapsto 0)$	

$\frac{\text{RegisterAccess} \quad \Delta St \quad i? : \mathbb{N}_1}{\begin{array}{l} 1 \leq i? \leq n \\ \text{matrix}' = \text{matrix} \oplus \{\lambda x : 1 \dots n \bullet (i? * (n - 1)) + x \mapsto 1\} \\ \oplus \{\lambda x : 1 \dots n \bullet i? + (x * (n - 1)) \mapsto 0\} \\ n' = n \\ \text{countList}' = \text{countList} \oplus \{i? \mapsto \text{countList}(i?) + 1\} \end{array}}$
$\frac{\text{ShowLRU} \quad \Delta St \quad i! : \mathbb{N}_1}{\begin{array}{l} \text{let } \text{pairs} == \\ \quad \lambda k : 1 \dots n \bullet k \mapsto \\ \quad \quad \text{sum}(\lambda x : 1 \dots n \bullet \text{matrix}((k * (n - 1)) + x) * \text{pow}(2, n - x - 1)) \\ \quad \bullet i! \in \text{dom } \text{pairs} \wedge \min \text{ran } \text{pairs} = \text{pairs}(i!) \\ i! \in \text{dom } \text{countList} \wedge \min \text{ran } \text{countList} = \text{countList}(i!) \\ \text{countList}' = (\lambda x : 1 \dots n \bullet x \mapsto 0) \end{array}}$

$\text{Input} \triangleq \text{input}?i \rightarrow \text{RegisterAccess}$

$\text{Output} \triangleq \text{var } i : \mathbb{N} \bullet \text{ShowLRU}; \text{output}!i \rightarrow \text{Skip}$

$\bullet \text{size}?n \rightarrow \text{Init};$
 $\mu X \bullet (\text{Input} \sqcap \text{Output}); X$
end

In this version of *LRULogCount*, we have lost the opportunity to reuse the code already defined in *LRULog*. There is the danger of the copy/paste technique, which is one of the great problems in software specification/development. The loss of conciseness is a direct consequence of the lack of code reuse, for example, the schema *ShowLRU* is now more complicated to understand and maintain. Actually, this self-contained version of the process can be obtained by semantic expansion of the previous version. At the semantic level, both are the same, but the first one is clearly more convenient from the software engineering point of view.

The code duplication can be reduced if we define *LRULogCount* as a parallel composition of two auxiliary processes: *LRULogAux* that behaves like *LRULog*, except by the

addition of communication points with *LRUCountAux*, that deals with the changes in the access counter *countList*.

```

channel n, input, output :  $\mathbb{N}_1$ 
channel forward_i, int_output :  $\mathbb{N}_1$ 
channel forward_set :  $\mathbb{P} \mathbb{N}_1$ 
chanset LRUProtocol  $\hat{=}$   $\{\{forward\_i, int\_output, forward\_set\}\}$ 

```

```

process LRULogAux  $\hat{=}$  begin

```

state <i>St</i>
<i>matrix</i> : seq <i>B</i> <i>n</i> : $\mathbb{N}_1$
$\#matrix = n * n$

<i>Init</i>
<i>St'</i> <i>n?</i> : $\mathbb{N}_1$
<i>n'</i> = <i>n?</i> <i>matrix'</i> = $(\lambda x : 1 \dots n * n \bullet x \mapsto 0)$

<i>RegisterAccess</i>
ΔSt <i>i?</i> : $\mathbb{N}_1$
$1 \leq i? \leq n$ <i>matrix'</i> = <i>matrix</i> $\oplus$ $\{\lambda x : 1 \dots n \bullet (i? * (n - 1)) + x \mapsto 1\}$ $\oplus \{\lambda x : 1 \dots n \bullet i? + (x * (n - 1)) \mapsto 0\}$ <i>n'</i> = <i>n</i>

<i>ShowLRU</i>
ΞSt <i>i!</i> : $\mathbb{N}_1$ <i>int_set!</i> : $\mathbb{P} \mathbb{N}_1$
<i>int_set!</i> = let <i>pairs</i> == $\lambda k : 1 \dots n \bullet k \mapsto$ $sum(\lambda x : 1 \dots n \bullet matrix((k * (n - 1)) + x) * pow(2, n - x - 1))$ $\bullet i! \in \text{dom } pairs \wedge \min \text{ran } pairs = pairs(i!)$

$Input \triangleq forward_i?i \rightarrow RegisterAccess$

$Output \triangleq \mathbf{var} \ i : \mathbb{N}, \mathbf{var} \ int_set : \mathbb{P}\mathbb{N} \bullet ShowLRU;$
 $int_output!i \rightarrow forward_set!int_set \rightarrow Skip$

$\bullet size?n \rightarrow Init;$
 $\mu X \bullet (Input \sqcap Output); X$
end

process $LRUCountAux \triangleq \mathbf{begin}$

state St	
$n : \mathbb{N}_1$	
$countList : \mathbb{N}_1 \rightarrow \mathbb{N}$	
$\# \text{dom } countList = n$	
$Init$	
St'	
$n? : \mathbb{N}_1$	
$n' = n?$	
$countList' = (\lambda x : 1 \dots n \bullet x \mapsto 0)$	
$RegisterAccess$	
ΔSt	
$i? : \mathbb{N}_1$	
$int_i! : \mathbb{N}_1$	
$1 \leq i? \leq n$	
$countList' = countList \oplus \{i? \mapsto countList(i?) + 1\}$	
$int_i! = i?$	
$ShowLRU$	
ΔSt	
$i! : \mathbb{N}_1$	
$int_set? : \mathbb{P}\mathbb{N}_1$	
$i! \in int_set?$	
$i! \in \text{dom } countList \wedge \min \text{ran } countList = countList(i!)$	
$countList' = (\lambda x : 1 \dots n \bullet x \mapsto 0)$	

$$Input \triangleq input?i \rightarrow \mathbf{var} \ int_i : \mathbb{N} \bullet RegisterAccess; forward_i!int_i \rightarrow Skip$$

$$Output \triangleq forward_set?int_set \rightarrow \mathbf{var} \ i : \mathbb{N} \bullet ShowLRU; output!i \rightarrow Skip$$

$$\begin{aligned} &\bullet size?n \rightarrow Init; \\ &\mu X \bullet (Input \sqcap Output); X \\ &\mathbf{end} \end{aligned}$$

After definition of the auxiliary processes we are ready to define *LRULog* and *LRULogCount* with the expected behavior originally proposed via inheritance.

$$LRULogCount \triangleq (LRULogAux \parallel [LRUProtocol] LRUCountAux) \setminus LRUProtocol$$

$$LRULog \triangleq LRULogAux[forward_i := input, int_output := output] \setminus LRUProtocol$$

This latter approach is an improvement of the previous one concerning code duplication. We adapt the initial specification of *LRULog*, via an auxiliary process, *LRULogAux*. It exchanges information with another auxiliary process, *LRUCountAux*. The former behaves like *LRULog*, but it provides information via internal channels to the latter. The schema *LRULogAux.ShowLRU* informs, in addition, what page must be removed(*i!*), all possible pages that can be correctly chosen for passivation. This information is sent, via *LRULogAux* main action, to the *LRUCountAux.ShowLRU* and this uses a tiebreaker criteria to increase the determinism of its decision. Another addition was made in *LRUCountAux.RegisterAccess*: it forwards the value received as input to the *LRULogAux.RegisterAccess* via *LRUCountAux* main action.

To intermediate the internal information exchange between the two auxiliary processes, a channel set was defined to group the channels *forward_i*, *int_output*, *forward_set*. The former is used to send the index of accessed page from schema *LRUCountAux.RegisterAccess* to *LRULogAux.RegisterAccess*. The second is used to send the chosen index, without application of the tiebreaker criteria, to the environment. But in fact, the environment will never know about this channel, because this was renamed in *LRULog* definition. This is a trick used to allow sending, via *output* channel, the chosen index, after the tiebreaker criteria application, to the environment. Finally, the latter is used to send the page indexes with the lowest binary value from *LRULogAux.ShowLRU* to *LRUCountAux.ShowLRU*, which uses this set to apply the tiebreaker criteria taking a more deterministic decision.

There are lots of communication code in this approach and many scattered changes were made in the schemas to provide the information exchanged between the two processes. The addition of new channels to provide the internal communication is another

potential disadvantage in comparison with the inheritance based solution. A slight code duplication concerns the state component n , which appears in the two process definitions and it shows that the code duplication cannot be completely avoided. The main actions of the two processes were polluted with internal communication events and exhibits, too, some code duplication.

These two previous approaches show up a fact already known in the object-oriented paradigm: it is possible to construct programs without inheritance, but not using this feature affects the expressive power of the developer. The benefits of code reuse, conciseness and maintainability are strong reasons to adopt inheritance features in the modeling of concurrent systems with process inheritance. Our view is that the modularity provided by parallel composition and inheritance are complementary. Splitting a task between orthogonal and cooperating processes is certainly an opportunity for parallel composition. Although specializing a process behavior can also be achieved with parallel composition, as illustrated, the use of inheritance in such contexts seems to be more appropriate, as it allows code reuse and more concise specifications.

3.5 Final Considerations

In this chapter we have built the basis for the proposition of sound refinements for specifications in *OhCircus* that use process inheritance. A notation, explained through an evolutionary (the non-determinism is reduced) example, for process inheritance in *OhCircus* promotes code reuse and encourages a better conceptual system modeling if compared to the original *OhCircus* support.

The *OhCircus* process inheritance in its original form does not allow a subprocess to access the state and Z schemas of its superprocesses. It creates a hard limitation for code reuse. We change the grammar and semantics of process inheritance in *OhCircus* to allow visibility of process elements by its subprocesses. The main changes were: the creation of a new access level to state components and Z schemas, a new way to declare Z schemas that allows it to refer to superprocess schemas and, finally, the semantics of these new constructs.

The heart of a paging algorithm, used in the kernel of operating systems, is modeled in *OhCircus* exemplifying the new syntax and semantics for process inheritance. A process, which holds information about page accesses and decides which page must be transferred from the main to the secondary memory, when the former is full and a space is requested to a page that comes from the latter, is firstly modeled. When more than one page satisfy the criteria to be moved back to the secondary memory, the process takes an aleatory and nondeterministic decision. A second process, a subprocess of the first, is then defined to apply a second criteria over the page candidates selected by the superprocess criteria. It becomes a more deterministic version of the first.

The benefits of this approach are informally discussed through a comparison with the inheritance approach as originally conceived in *OhCircus*. In the first of these forms, the subprocess is replaced by a self-contained process, and because there is no code reuse, it was detected a high level of code duplication. In the second possibility we use parallel composition to avoid code duplication. In fact we satisfy partially this goal, but it

implies the insertion of internal communication between processes, what motivates some adaptations in the Z schemas to provide the information exchanged between processes. We conclude from these attempts that inheritance, in the manner proposed in this work, seems a promising feature to obtain code reuse, maintainability and conciseness in development of the concurrent systems. It is complementary to the modularity facilities provided by parallelism.

CHAPTER 4

Typing Rules for *OhCircus*

The algebraic laws, presented in Chapter 6, assume that the specifications are well-formed and well-typed. Based on [52, 12] we formulate a set of typing and well-formedness rules addressing process inheritance in *OhCircus*. These rules are the central contribution of this chapter.

4.1 Typing

An *OhCircus* program combines constructions of Z, CSP, object-oriented paradigm and Dijkstra's guarded commands [16]. To be well typed a program must satisfy certain constraints normally formalized in terms of typing rules. In the case of *OhCircus*, this presentation is modular: its typing rules are defined from rules for each component language. The formal type system of Z is completely defined in [25]. Formal Systems Europe Limited¹ has developed a type checker covering all the CSP language syntax. These results are converged in the typing checking developed for *Circus*[52]. We extend this typing checking for *OhCircus* considering process inheritance. Typing rules for object-oriented features is not formalized yet, but its formalization can benefit from that of Java like languages.

As the focus of our algebraic laws, developed in Chapter 6, stays in the evolution of specifications involving process inheritance, we concentrate our efforts in the definition of a type system that deals with this feature. The auxiliary functions used in the additional typing rules are based on those defined for class inheritance in *ROOL* [12].

4.1.1 Data and phrase types

Data types T are the types of channels, constants and variables. They are either primitive (like $\mathbb{N}$ and $\mathbb{Z}$), user defined type names N or a symbol standing for absence of type (wt). The latter is used in the synchronized channel declaration.

$$T \in Type ::= N \mid \mathbb{N} \mid \mathbb{Z} \mid wt \mid \dots \text{ other primitive and composite types}$$

Besides the data types T , other phrase types θ are available for programs, *OhCircus* paragraphs, channels, process declarations and complete programs.

$$\theta ::= T \mid \mathbf{Program} \mid \mathbf{CircusParagraph} \mid \mathbf{CDeclaration} \mid \mathbf{Process} \mid \mathbf{Action} \mid \dots$$

The types of phrases, in the context of a collection of process declarations and a specific process, are given by the typing relation $\triangleright$. For example, $\Gamma, N \triangleright s : \mathbf{SchemaText}$

¹<http://www.fsel.com/software.html>

asserts that s is a schema that can appear inside a process N . Here Γ is a typing environment; it records process/class declarations, their local definitions (schemas and state components) and the inheritance relations between them.

4.1.2 Typing Environment

A typing environment is a record that registers the typing and inheritance information in a collection of process and class declarations. It also registers typing information related to local declarations of schemas, actions and state components. In the same way, global declarations of channels, channel sets, constants and axiomatic declarations. The set $TEnv$ contains all valid type environments Γ .

In [52] a type environment with twenty fields is defined to record the typing information of *Circus*. Some of these fields are designed to hold external information to processes, *channels*, for example, records channel names and their types. Others, to record information about local definitions: *st* records the names and types of all state components of every declared process.

As the original definition of $TEnv$, in [52], does not have inheritance information, we need both to add and redefine some fields to record inheritance information. Actually, *st* records the names and types of all declared and **inherited** state components of every declared process. The same is adopted concerning the field that holds typing of Z schemas.

$$\begin{aligned} st &: PName \multimap LSignature \\ LSignature &= LName \multimap T \end{aligned}$$

The set $PName$ contains the names of all declared processes. $LSignature$ associates a local name, $LName$, with a type. The field *sc* records the names and schema declarations of all declared and inherited schemas of every declared process. $SDecl$ associates a local name with a schema text.

$$\begin{aligned} sc &: PName \multimap SDecl \\ SDecl &= LName \multimap SText \end{aligned}$$

To the original version of *OhCircus*, we have added the access level modifier **protected**. Therefore, we need to know, when validating a program, if a process has the appropriate access to a schema or state component. The field *vis* records the visibility of schemas and state components of the declared processes.

$$\begin{aligned} vis &: PName \multimap (LName \multimap Visibility) \\ Visibility &= \{protected, default\} \end{aligned}$$

If for a schema s of a process P , we have $vis\ P\ s = protected$, then s is a protected schema inherited or declared by P . In addition, a visibility is assigned to each schema and state component of a process. A local name cannot be simultaneously used as a schema name and a state component name declared or inherited by a process.

$$\begin{aligned} \forall P : pnames \bullet (\text{dom}(st\ P) \cup \text{dom}(sc\ P) &= \text{dom}(vis\ P) \wedge \\ \text{dom}(st\ P) \cap \text{dom}(sc\ P) &= \emptyset) \end{aligned}$$

The field *ac* maps a process name to an action definition.

$$ac : PName \multimap (LName \multimap CSPAct)$$

The *pnames* field of a typing environment is a set containing the names of all declared processes. This is the common domain of *vis*, *st*, *sc*, *ac* and other fields in the original *TEnv* definition.

$$\begin{aligned} pnames &: \mathbb{P} \ PName \\ pnames &= \text{dom } st = \text{dom } sc = \text{dom } vis = \text{dom } ac \end{aligned}$$

To the original definition of *TEnv* was added the *suppds* field. It associates a process name to the name of its immediate superprocess.

$$suppds : PName \rightarrow PName$$

A process can have or not a superprocess. Furthermore, a superprocess is a declared process.

$$\begin{aligned} \text{dom } suppds &\subset pnames \\ \text{ran } suppds &\subset pnames \end{aligned}$$

As the inheritance relation does not have circularities the range and domain of *suppds* cannot be equal to *pnames* and the next equality is satisfied, where *id* is the identity relation (*id* *X* on a set *X* relates each member of *X* to itself [44]).

$$suppds^+ \cap \text{id } PName = \emptyset$$

This condition guarantees that the transitive closure of *suppds*, that is a transitive (but not reflexive) relation, whose notation is given by the superscript *+*, does not relate a process name to itself. We say that a process *P* is a subprocess of *S*, *P* < *S*, in Γ if:

$$P < S \Leftrightarrow (P, S) \in (\Gamma.\text{suppds})^+$$

Another field records the types of the local variables, state components, process parameters and inputs/outputs of Z schemas in the context of a particular process. Therefore, the same information in *sc* is kept in *locals*, this redundancy simplifies typing rules. Before each process validation, *locals* is emptied and internally to a process the inputs/outputs of a Z schema are removed when its validation finishes.

$$locals : LSignature$$

The type of *locals* range is a primitive type or a declared class name. Actually, process names cannot be used as a variable type, since processes are not types in *OhCircus*.

$$\begin{aligned} \forall lt : \text{ran } st \cup locals; N : CName \mid N \in \text{ran } lt \bullet N \in cnames \\ \forall sdecl : \text{ran } sc \bullet clrefs \ sdecl \end{aligned}$$

There are three disjoint sets: $PName$, $LName$ and $CName$. The latter contains the names of all declared classes. The $cnames$ is a field of $TEnv$ where:

$$cnames : \mathbb{P} \ CName$$

The function $clrefs$ determines the set of class names referenced by a schema in their inputs/outputs.

$$\begin{aligned} clrefs &: SDecl \rightarrow \mathbb{P} \ CName \\ clrefs \ sdecl &= \{N : CName \mid \exists \text{stext} : \text{ran } sdecl \bullet \text{ref } (ioext \ \text{stext}) \ N\} \end{aligned}$$

The function ref tests if the inputs/outputs of a schema text, given by the function $ioext$, introduce an input/output of type N .

$$\begin{aligned} \neg(\text{ref } \emptyset \ N) \\ \text{ref } (io, io\text{set}) \ N &\Leftrightarrow (\text{ref } io \ N) \vee (\text{ref } io\text{set} \ N) \\ \text{ref } (x! : T) \ N &\Leftrightarrow N = T \\ \text{ref } (x? : T) \ N &\Leftrightarrow N = T \end{aligned}$$

The function $ioext$ returns the declaration inputs and outputs of a schema.

$$\begin{aligned} ioext &: \mathbb{P} \ OslashClause \rightarrow \mathbb{P} \ Declaration \rightarrow \mathbb{P} \ SuperClause \rightarrow Predicate \rightarrow \\ &\quad \mathbb{P} \ Declaration \\ ioext \ (oslashc, decls, superc, preds) &= decls \end{aligned}$$

The typing rules presented in the sequel focus on the validation of process declarations related by inheritance. Processes that do not use inheritance (**super** clause, **protected** access level or **extends** clause) are validated just as before, in the exactly manner presented in [52]. Although we have added $cnames$ to the initial definition of $TEnv$, it is not our intention to validate the class declarations, but uniquely verify if the types of state components, local variables, process parameters and the inputs/outputs of the Z schemas use types that are properly declared classes.

4.2 Typing Rules

Typing rules validate judgements considering a set of premisses (judgements evaluate to true). The general form of a judgement is $\Gamma \triangleright \mathfrak{S}$, where Γ is an environment and $\mathfrak{S}$ is an assertion. If the assertion depends on a process P , it can be written as $\Gamma, P \triangleright \mathfrak{S}$. For example, consider $\Gamma \triangleright pd : \mathbf{OhProcessDefinition}$, it asserts whether pd is a valid *OhCircus* process definition. Likewise, $\Gamma, P \triangleright par : \mathbf{OhCircusParagraph}$ asserts whether par is a valid process paragraph of P .

$$\frac{\Gamma_1 \triangleright \mathfrak{S}_1 \dots \Gamma_n \triangleright \mathfrak{S}_n \quad (\text{premisses})}{\Gamma \triangleright \mathfrak{S}}$$

The validation rules presented in [52] are proposed for *Circus* that has not the concept of process inheritance. Therefore we need to define new validation rules or update some function used by those previously defined, to take into account the inheritance information. For example, we need to consider, when validating a process, not only the declared state components, but those inherited from its superprocesses.

4.2.1 Programs

A program in *OhCircus* is formed by a set of paragraphs. A paragraph can be an axiomatic definition, a channel, a channel set, a class or a process. The functions that extract the typing information of the axiomatic definitions, channels and channel sets are the same found in [52]. For the class declarations it is registered, in the field *cnames*, only their names. For process declarations we have the next typing rule and its supporting functions.

Program	::=	OhCircusParagraph*
OhCircusParagraph	::=	Paragraph
		ChannelDefinition ChanSetDefinition
		OhProcessDefinition ClassDefinition

Consider *pds* as a set of process declarations. The function *VPDecls* extracts information from *pds* and checks that the schema texts in *pds* are well-typed in the environment Γ' . Γ is the environment before the validation of *pds*. It has the type information of the class declaration (only *cnames*) and those found in axiomatic definitions, channels and channel sets. Γ' is obtained from overriding Γ with *VPDecls pds*.

$$\frac{\Gamma' = \Gamma \oplus VPDecls\ pds \quad VSDecls\ \Gamma'\ pds \quad VADecls\ \Gamma'\ pds}{\Gamma \triangleright pds : \mathbf{CircusParagraphList}}$$

The function *VPDecls* is defined in terms of a function *PDecls* that extracts type and visibility information about state components and schemas, and the inheritance relationship from a sequence of process declarations. The action names are also registered.

$$PDecls : Pds \rightarrow (sc : PName \leftrightarrow (LName \leftrightarrow T), \\ st : PName \leftrightarrow (LName \leftrightarrow SText), \\ vis : PName \leftrightarrow (LName \leftrightarrow Visibility), \\ suppds : PName \leftrightarrow PName, \\ ac : PName \leftrightarrow (LName \leftrightarrow CSPAct), \dots)$$

Differently from *TEnv*, the function *PDecls* is modeled using relations instead of functions, because we have no guarantee that there are no errors in *pds*.

$$\begin{aligned}
& PDecls \ \emptyset = (sc = \emptyset, st = \emptyset, suplds = \emptyset, ac = \emptyset) \\
& PDecls \ (\mathbf{process} \ P \hat{=} [\mathbf{extends} \ Q] \ \mathbf{begin} \\
& \quad \mathbf{state} \ State \hat{=} [x_1 : T_1; \mathbf{protected} \ x_2 : T_2 \mid pred] \\
& \quad \mathbf{initial} \ Init \hat{=} sinit \\
& \quad st_1 \hat{=} stext_1 \\
& \quad \mathbf{protected} \ st_2 \hat{=} stext_2 \\
& \quad ac \hat{=} cspact \\
& \quad \bullet \ cspmain \\
& \quad \mathbf{end} \\
& \quad pds) = \\
& \left(\begin{array}{l}
sc = \{P \mapsto \{x_1 \mapsto T_1, x_2 \mapsto T_2\}\} \cup \\
\quad (PDecls \ pds).sc \\
st = \{P \mapsto \{State \mapsto sstate, Init \mapsto sinit, st_1 \mapsto stext_1, st_2 \mapsto stext_2\}\} \cup \\
\quad (PDecls \ pds).st \\
vis = \{P \mapsto \{x_1 \mapsto default, x_2 \mapsto protected, st_1 \mapsto default, st_2 \mapsto protected\}\} \cup \\
\quad (PDecls \ pds).vis \\
suplds = \{P \mapsto Q\} \cup \\
\quad (PDecls \ pds).suplds \\
ac = \{P \mapsto \{ac \mapsto cspact, main \mapsto cspmain\}\} \cup \\
\quad PDecls \ pds).ac, \dots
\end{array} \right)
\end{aligned}$$

The schema text $sstate$ stands for $[x_1 : T_1; \mathbf{protected} \ x_2 : T_2 \mid pred]$. Although we define a process P with a fixed number of state components, schemas and actions, the generalization to a process with an arbitrary number of these elements is obvious, but longer, so it is omitted.

If the sequence of process declarations does not have errors and P is a process in pds , $VPDecls \ pds \ P$ yields a typing environment determined by pds and extracted by $PDecls$. $VPDecls$ is a partial function, only valid pds declarations and names in $pnames$ generate a proper environment.

$$VPDecls \ Pds \rightarrow PName \rightarrow TEnv$$

Some conditions apply to the domain of $VPDecls$:

(hiding of state components)

There is no redeclaration of a protected state component in a subprocess. Otherwise, if a state component has the access level default in a superprocess, it cannot be seen by its subprocesses and these are free to reuse the same name to define any state component. This is not overriding.

$$\begin{aligned}
& \forall P_1 : \text{dom}((PDecls\ pds).sc); P_2 : \text{supers}\ pds\ P_1 \bullet \\
& \text{dom} \left(\begin{array}{c} ((PDecls\ pds).vis\ P_1) \triangleright \{\text{protected}\} \\ \triangleleft \\ \text{dom}((PDecls\ pds).sc\ P_1) \end{array} \right) \\
& \cap \\
& \text{dom} \left(\begin{array}{c} ((PDecls\ pds).vis\ P_2) \triangleright \{\text{protected}\} \\ \triangleleft \\ \text{dom}((PDecls\ pds).sc\ P_2) \end{array} \right) = \emptyset
\end{aligned}$$

First, we obtain all visible protected elements (schemas and state components) from P_1 , applying range restriction $\triangleright \{\text{protected}\}$ over the relation $(PDecls\ pds).vis\ P_1$. Since we are interested only in the protected state components, we apply a domain restriction over the relation $((PDecls\ pds).vis\ P_1) \triangleright \{\text{protected}\}$. The set $\text{dom}((PDecls\ pds).sc\ P_1)$ used in this restriction contains the names of all state components of the process P_1 . We do the same thing to each superprocess P_2 of P_1 . After, we assert whether the domains of both resultant relations are disjoint. We have modeled $(PDecls\ pds).sc$ and $(PDecls\ pds).vis$ as relations, but here we use them as functions, if and only if pds is well-typed and consequently $VPDecls\ pds\ P$ is a proper typing environment.

(circularities)

The function *supers* gives the set of all superprocesses of a process P , considering a set of process declarations. The processes in the domain of *supers* are those in the domain of $(PDecls\ pds).suppds$.

$$\begin{aligned}
& \text{supers} : Pds \rightarrow PName \rightarrow \mathbb{P}\ PName \\
& \text{supers}\ pds\ \emptyset = \emptyset \\
& \text{supers}\ pds\ P = \{(PDecls\ pds).suppds\ P\} \cup \text{supers}\ pds\ ((PDecls\ pds).suppds\ P)
\end{aligned}$$

This function is well defined if there are no circularities in pds , otherwise the typing environment cannot be elaborated and typing checker breaks indicating an error in the target specification. So, if $P, S \subset pnames$ and $P < S$, it implies $\neg (S < P)$.

(method overriding)

A method can be overridden in a subprocess, but the contravariance of inputs and covariance of outputs must be satisfied.

$$\begin{aligned}
& \forall P_1 : \text{dom}((PDecls\ pds).st); P_2 : \text{supers}\ pds\ P_1 \\
& m : \text{dom}((PDecls\ pds).st\ P_1) \cap \text{dom}((PDecls\ pds).st\ P_2) \bullet \\
& \text{vsig}\ ((PDecls\ pds).st\ P_1\ m)\ ((PDecls\ pds).st\ P_2\ m) \wedge \\
& (PDecls\ pds).vis\ P_1\ m = \text{protected}
\end{aligned}$$

The function *vsig* checks whether two schema texts satisfy the contravariance of inputs and covariance of outputs.

$$\begin{aligned}
& \text{vsig}(stx, stx') \Leftrightarrow \#(\text{ioext}\ stx) = \#(\text{ioext}\ stx') \wedge \\
& \forall io_1 : \text{ioext}\ stx; \exists io_2 : \text{ioext}\ stx' \mid \\
& \quad \text{isinput}(io_1) \rightarrow \text{type}\ io_1 \leq \text{type}\ io_2 \\
& \quad \parallel \text{type}\ io_2 \leq \text{type}\ io_1
\end{aligned}$$

The contravariance and covariance principle is the same proposed for class subtyping in [28]. First of all, *vsig* checks if the number of inputs and outputs in the schema texts are equal. Next it verifies if each input/output of the first schema text has a counterpart in the second. If it is the case, the former must be a subtype of (or equal to) the latter, when both are inputs. But, if they are outputs the subtype relation is tested in the opposite direction.

4.2.2 Environment Elaboration

The initial environment used in typing rule is obtained from that constructed by the functions presented in [52], when applied to the elements that we have reused from [52] (axiomatic definitions, channels and channel sets) in addition to the field *cnames*, that holds the declared class names. This environment is enriched by the information extracted from a set of process declarations, *pds*, using the relation *PDecls*. If *pds* is well-formed, *VPDecls pds P* gives a proper environment. We define *VPDecls pds P* by specifying the value of each of its fields considering that the above restrictions are met.

The state components and schemas of a process are those declared in the process and those inherited from its superprocesses.

$$(VPDecls\ pds\ P).sc = \left\{ \begin{array}{l} P : \text{dom}(PDecls\ pds).sc; \text{ } ls : Lsignature \mid \\ ls = (PDecls\ pds).sc \cup \bigcup \\ \left(\begin{array}{l} \text{dom}((PDecls\ pds).vis \mid supers\ pds\ P) \triangleright \{protected\} \\ \triangleleft \\ (PDecls\ pds).sc \mid supers\ pds\ P \end{array} \right) \end{array} \right\}$$

The expression $((PDecls\ pds).vis \mid supers\ pds\ P) \triangleright \{protected\}$ gets the set of all protected elements (state components and schemas) of the superprocesses of *P*. Then, a domain restriction operation gets only the protected state components from $(PDecls\ pds).sc \mid supers\ pds\ P$. The *st* field is populated in the same way.

$$(VPDecls\ pds\ P).st = \left\{ \begin{array}{l} P : \text{dom}(PDecls\ pds).st; \text{ } stext : SText \mid \\ stext = (PDecls\ pds).st \cup \bigcup \\ \left(\begin{array}{l} \text{dom}(((PDecls\ pds).vis \mid supers\ pds\ P) \triangleright \{protected\}) \\ \triangleleft \\ (PDecls\ pds).st \mid supers\ pds\ P \end{array} \right) \end{array} \right\}$$

All the elements in $(VPDecls\ pds\ P).sc$ and in $(VPDecls\ pds\ P).st$ must have their visibility mapped in $(VPDecls\ pds\ P).vis$, so:

$$(VPDecls\ pds\ P).vis = \left\{ \begin{array}{l} P : \text{dom}(PDecls\ pds).vis; \text{ } v : LName \mapsto Visibility \mid \\ v = (PDecls\ pds).vis\ P \cup \bigcup \\ ((PDecls\ pds).vis \mid supers\ pds\ P) \triangleright \{protected\} \end{array} \right\}$$

Because a process inherits only protected elements (state components and schemas), we get from P 's superprocesses in $(VPDecls\ pds\ P).vis$ only the elements that have the protected access level.

The actions are not inherited. In Chapter 3, the semantics presented composes in interleave the main action of a process with that of its superprocess, to give the process behavior. This is part of the semantics of process inheritance, but neither process actions are visible outside a process.

$$(VPDecls\ pds\ P).ac = (PDecls\ pds).ac$$

The process names are those in the domain of $(PDecls\ pds).sc$ or of $(PDecls\ pds).st$; these domains happen to be the same.

$$(VPDecls\ pds\ P).pnames = \text{dom}(PDecls\ pds).sc = \text{dom}(PDecls\ pds).st$$

The *suppds* is the same found in $(PDecls\ pds).suppds$.

$$(VPDecls\ pds\ P).suppds = (PDecls\ pds).suppds$$

The *locals* field contains all visible state components of a process, those declared and those in the protected level declared in its superprocesses.

$$(VPDecls\ pds\ P).sc \subseteq (VPDecls\ pds\ P).locals$$

Another condition verifies whether all schema texts are well typed following Z rules and considering the environment with *pds* declarations, that also includes all declared class names and the elements whose semantics and grammar we maintained unchanged in relation to the *Circus* original specification. For simplicity **super** clauses are removed by the semantics presented in chapter 5. This simplifies *VSDecls*:

$$\begin{aligned} & VSDecls\ \Gamma\ \emptyset \Leftrightarrow \text{true} \\ & VSDecls\ (\text{process } P \hat{=} [\text{extends } P'] \text{ begin} \\ & \quad \text{state } State \hat{=} x : T, \text{protected } x' : T' \mid pred \\ & \quad \text{initial } Init \hat{=} sinit \\ & \quad st \hat{=} stext \\ & \quad \text{protected } st' \hat{=} stext' \\ & \quad ac \hat{=} cspact \\ & \quad \bullet\ cspmain \\ & \quad \text{end} \\ & \quad pds) \Leftrightarrow \\ & \Gamma; \Gamma.sc, P \triangleright stext, stext' : \mathbf{SchemaText} \wedge VSDecls\ \Gamma\ pds \end{aligned}$$

This function verifies whether a schema, protected or not, has its schema text in according to the Z typing rules. In [52] this validation checks, for example, if all state components referenced by a schema text belongs to the process where it is found. The concept of inheritance enlarges the set of possible state components that a schema can

refer to, and instead of changing the validation rule, we simply extend this set with the inherited state components. So our work extends [52] to deal with process inheritance, using the typing concepts for class inheritance developed in [12].

Finally the following typing rule verifies whether all declared actions, including the main action, satisfy the CSP typing rules.

$$\begin{array}{l}
VADecls \ \Gamma \ \emptyset \Leftrightarrow \text{true} \\
VADecls \ (\text{process } P \hat{=} [\text{extends } P'] \text{ begin} \\
\quad \text{state } State \hat{=} x : T, \text{protected } x' : T' \mid pred \\
\quad \text{initial } Init \hat{=} sinit \\
\quad st \hat{=} stext \\
\quad \text{protected } st' \hat{=} stext' \\
\quad ac \hat{=} cspact \\
\quad \bullet \ cspmain \\
\quad \text{end} \\
pds) \Leftrightarrow \\
\Gamma, P \triangleright cspact, cspmain : \mathbf{Action} \wedge VADecls \ \Gamma \ pds
\end{array}$$

The initial environment used in the typing rule presented was previously enriched with the names found in the class declarations cds extracted by $ExtCds$, in the same way found in [12].

$$\begin{array}{l}
ExtCds \ \emptyset = \emptyset \\
ExtCds \ (\text{class } N \hat{=} [\text{extends } N'] \text{ begin} \\
\quad \text{state } x : T, \text{protected } x' : T' \\
\quad \text{initial } Init \hat{=} sinit \\
\quad st \hat{=} stext \\
\quad \text{protected } st' \hat{=} stext' \\
\quad \text{end} \\
\quad cds) = \\
\{N\} \cup ExtCds \ cds
\end{array}$$

It is not our intent to validate the class declarations, but rather verifying whether the composite types of variables are declared as classes. Therefore, we do not validate the state components, neither schemas of a class declaration. This does not impact the laws.

4.3 Final Considerations

The new constructs inserted in the *OhCircus* grammar impose the obligation of defining rules that validate a program in this language. As we have these rules for *Circus* [52], we extend them to address inheritance. A typing rule is an assertion about a program considering an environment that records its typing information. Our strategy enriches the typing environment of *Circus* with the inheritance information found in an *OhCircus* program (the existing typing rules are unchanged). To extract the inheritance information we follow the work for class inheritance in ROOL [12]. The next chapter presents the

formal semantics, based on Hoare and He's Unifying Theories of Programming [24] for the new constructions of *OhCircus*.

CHAPTER 5

A UTP Semantics for Process Inheritance

Although the complete semantics of *OhCircus* has not yet been developed, we propose the semantics of process inheritance, from which we prove algebraic laws that deal with this feature. This is possible because we define a mapping from processes with inheritance, written in *OhCircus* extended, into regular processes in *Circus*, which has its semantics completely defined in [34]. Therefore, it is possible to formally prove, based on the semantics of *Circus*, that the left- and right-hand sides of an algebraic law have the same meaning in this semantics. We give a UTP semantics for a new parallel operator, used in the definition of inheritance, as well as for **super** clause and **protected** mechanisms.

5.1 Semantics of Inheritance

The next definition maps a subprocess written in *OhCircus* in a regular process in *Circus* whose semantics is defined in [34, 35]. Consider the processes *Super* (whose schemas have not the **super** clause) and *Sub* with the following structure.

<pre> process <i>Super</i> state $st \hat{=} st_1 \wedge st_2$ <i>pps</i>₁ <i>pps</i>₂ • <i>act</i> end </pre>	<pre> process <i>Sub</i> $\hat{=} \text{extends } Super$ state <i>st</i> <i>pps</i> • <i>act</i> end </pre>
--	---

The meaning of *Sub* is defined as:

$$Sub \hat{=} \left(\begin{array}{l} \textbf{begin state} \hat{=} Super.st_1 \wedge Super.st_2 \wedge Sub.st \\ \quad Super.pps_1 \wedge \Xi Sub.st \\ \quad Super.pps_2^{ref} \wedge \Xi Sub.st \\ \quad Sub.pps \\ \quad \bullet Super.act \llbracket Super.st \mid Super.st \wedge Sub.st \rrbracket Sub.act \\ \textbf{end} \end{array} \right)$$

The state components $Super.st_2$ and $Super.st_1$ are categorized into protected and default level, respectively. The same categorization is applied to its schemas, $Super.pps_1$ and $Super.pps_2$. $Super.pps_2^{ref}$ is obtained from $Super.pps_2$ by eliminating the paragraphs redefined in $Sub.pps$.

Although all components of *Super* are copied to *Sub*, only its protected components can be accessed by the original declared elements of *Sub*; those in default level cannot be

accessed by *Sub* even semantically they are brought to it. Because *Super.act* can refer to any schema in *Super.pps*, and these to any state in *Super.st*, we need to bring all protected and default elements from *Super* to *Sub*.

This semantics may lead to name conflict. If an element (state or schema) in *Super* has the default access level, its name can conflict with elements in *Sub*. We handle this problem by prefixing the elements of the super process with *Super* (it is guaranteed that each process has a unique name).

Another precondition for the application of this mapping is that none of the schemas in *Sub.pps* use the **super** clause. This guarantees that, before we ‘lost’ a schema in *Super.pps*₂, because it was rewritten in *Sub.pps*, all schemas in this set have not a dependence, through **super**, from the removed schema in *Super.pps*₂. The next section summarizes in an example what we have shown.

5.1.1 Explanatory Example

In order to visualize the application of the mapping previously proposed, we construct a simple example that shows in practice how process inheritance works in the updated *OhCircus*.

Consider initially a buffer of limited size that holds natural numbers sequentially. It receives a number and appends it to the bounded sequence and increase its size by one. We cannot remove elements from the buffer.

channel *input* : $\mathbb{N}$

process *Buffer* $\hat{=}$ **begin**

state <i>St</i>	
protected <i>seq</i> : seq $\mathbb{N}$	
$\#seq \leq maxSize$	
<i>Init</i>	
<i>St'</i>	
$seq' = \langle \rangle$	
protected <i>Append</i>	
ΔSt	
$x? : \mathbb{N}$	
$\#seq < maxSize$	
$seq' = seq \frown \langle x? \rangle$	

$$Input \triangleq (\#seq < maxSize \ \& \ input?x \rightarrow Append) \sqcap Stop$$

• *Init*; μX • *Input*; X
end

In the sequel we present the subprocess *PlusBuffer* that adds new capabilities to its superprocess, namely *Buffer*. The first is the possibility of a double addition to the buffer; The average of all elements stored are also offered, as well as the empty buffer check.

$\frac{[X]}{sum : seq\ X \rightarrow X}$ <hr/> $\forall s : seq\ X \bullet$ $sum\ s = \mathbf{if}\ \#s = 1\ \mathbf{then}\ s(1)$ $\qquad \mathbf{else}\ s(1) + sum(tail\ s)$
--

channel *inputTwo* : $\mathbb{N} \times \mathbb{N}$
channel *av*, *emp*
channel *average* : $\mathbb{R}$
channel *isEmpty* : $\mathbb{B}$

process *PlusBuffer* $\triangleq$ **extends** *Buffer* **begin**

state <i>St</i> <hr/> <i>totalSum</i> : $\mathbb{N}$ <hr/> <i>totalSum</i> = <i>sum(seq)</i>

<i>Init</i> <hr/> <i>St'</i> <hr/> <i>totalSum'</i> = 0

<i>Append</i> <hr/> ΔSt <hr/> super <i>Append</i> <hr/> <i>totalSum'</i> = <i>totalSum</i> + <i>x</i> ?
--

<i>AppendTwo</i>	_____
ΔSt	
$x?, y? : \mathbb{N}$	
$\#seq + 1 < maxSize$	
$seq' = seq \cap \langle x?, y? \rangle$	
$totalSum' = totalSum + x? + y?$	

<i>Average</i>	_____
ΞSt	
$av! : \mathbb{N}$	
$av! = totalSum \text{ div } \#seq$	

<i>IsEmpty</i>	_____
ΞSt	
$emp! : \mathbb{B}$	
$emp! = (seq = \langle \rangle)$	

$$Input \hat{=} (\#seq + 1 < maxSize \ \& \ inputTwo?x?y \rightarrow AppendTwo) \sqcap Stop$$

$$AverageAct \hat{=} av \rightarrow \mathbf{var} \ av : \mathbb{N} \bullet Average; average!av \rightarrow Skip$$

$$EmptyCheck \hat{=} emp \rightarrow \mathbf{var} \ emp : \mathbb{B} \bullet IsEmpty; isEmpty!emp \rightarrow Skip$$

$$\bullet Init; \mu X \bullet (Input \sqcap AverageAct \sqcap EmptyCheck); X \\ \mathbf{end}$$

The process *PlusBuffer* provides the possibility of storing two elements at once with the *AppendTwo* schema. The schema *Average* uses the state component *totalSum* to do the obvious. An empty check operation is also provided. In the sequel we apply our semantics for process inheritance to *PlusBuffer* to obtain a regular process in *Circus*. Before, the **super** clause in *Append* schema is removed through applying the semantics developed in the next subsection.

$$\mathbf{process} \ PlusBuffer \hat{=} \mathbf{begin}$$

state $St \triangleq Buffer_St \wedge PlusBuffer_St$

$Buffer_St$	_____
$seq : seq \mathbb{N}$	_____
$\#seq \leq maxSize$	_____

$PlusBuffer_St$	_____
$totalSum : \mathbb{N}$	_____
$totalSum = sum(seq)$	_____

Our semantics for process inheritance admits that the state components defined or inherited by a subprocess can appear in separated schemas that together represent the subprocess state components or in a unique schema. The same reasoning applies to initializers.

$Init \triangleq Buffer_Init \wedge PlusBuffer_Init$

$Buffer_Init$	_____
$Buffer_St'$	_____
$seq' = \langle \rangle$	_____

$PlusBuffer_Init$	_____
$PlusBuffer_St'$	_____
$totalSum' = 0$	_____

Since $PlusBuffer.Append$ is a redefinition of $Buffer.Append$, only the former (without the **super** clause) will belong in $PlusBuffer$.

$Append$	_____
ΔSt	_____
$x? : \mathbb{N}$	_____
$\#seq < maxSize$	_____
$seq' = seq \frown \langle x? \rangle$	_____
$totalSum' = totalSum + x?$	_____

The *AppendTwo* schema is defined in the subprocess and remains unchanged like as *Average* and *IsEmpty*.

<i>AppendTwo</i>	_____
ΔSt $x?, y? : \mathbb{N}$	
$\#seq + 1 < maxSize$ $seq' = seq \frown \langle x?, y? \rangle$ $totalSum' = totalSum + x? + y?$	

<i>Average</i>	_____
ΞSt $av! : \mathbb{N}$	
$av! = totalSum \text{ div } \#seq$	

<i>IsEmpty</i>	_____
ΞSt $emp! : \mathbb{B}$	
$emp! = (seq = \langle \rangle)$	

$$Buffer_Input \hat{=} (\#seq < maxSize \ \& \ input?x \rightarrow Append) \sqcap Stop$$

$$Input \hat{=} (\#seq + 1 < maxSize \ \& \ inputTwo?x?y \rightarrow AppendTwo) \sqcap Stop$$

$$AverageAct \hat{=} av \rightarrow \mathbf{var} \ av : \mathbb{N} \bullet Average; average!av \rightarrow Skip$$

$$EmptyCheck \hat{=} emp \rightarrow \mathbf{var} \ emp : \mathbb{B} \bullet IsEmpty; isEmpty!emp \rightarrow Skip$$

• *Init*;
 $(\mu X \bullet Buffer_Input; X)$
 $\llbracket Buffer_St \mid St \rrbracket$
 $(\mu X \bullet (Input \sqcap AverageAct \sqcap EmptyCheck); X)$
end

The mapping for process inheritance was used to obtain the previous version of *PlusBuffer* in *Circus*. The preconditions for the mapping application, concerning *Buffer*, are satisfied: it is not a subprocess and consequently its schemas do not have the **super** clause.

The **super** clause is eliminated, before the elimination of inheritance. As the state and initial schemas from both processes have the same name, we have renamed those of *Buffer* with *Buffer_* prefix; this guarantees the uniqueness of process names, so name conflicts are avoided. The same thing is done with the *Input* action. To maintain the same name of *Init* and *St* in *PlusBuffer* we have renamed the original versions with *PlusBuffer_* prefix, so $St \hat{=} Buffer_St \wedge PlusBuffer_St$ and $Init \hat{=} Buffer_Init \wedge PlusBuffer_Init$.

The main action of *PlusBuffer* initializes its state components calling the initializer *Init*, then it recursively offers the input of two elements at once, the stored elements average and the buffer empty check operation; all this is composed in interleaving with the recursive input of an unique element, like in the regular *Buffer* process. It seems a real design choice, among other possibilities, to specify a buffer with the above capabilities.

5.2 UTP Semantics for new Parallel Operator, Visibility and **super** Clause

An important issue is related to the parallel composition semantics given in [34]. If we have $A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2$, the final state of the variables in ns_1 is given by A_1 and those variables in ns_2 by A_2 , such that $ns_1 \cap ns_2 = \emptyset$. It avoids conflicts about what action will determine the final value of a possible shared variable. Our semantics for process inheritance does not respect this principle. This becomes evident from the main action of *Sub*, $Super.act \llbracket Super.st \mid Super.st \wedge Sub.st \rrbracket Sub.act$ presented in 5.1. This apparent inconsistency with the semantics of parallel composition can be resolved, if we consider that the changes made in a state component by a schema sc in a subprocess cannot contradict the changes made by sc in its superprocess, since the former refines the latter; it follows the same principle described in [28].

Formally if we consider $ns_1 \cap ns_2 \neq \emptyset$ in $A_1 \llbracket ns_1 \mid ns_2 \rrbracket A_2$ (cs empty implies that A_1 and A_2 are composed by interleaving), we are using a new operator, and it arises the obligation of defining its semantics. We define the semantics of this new operator in UTP (Unifying Theories of Programming) [24] based on the semantics given in [34] when $ns_1 \cap ns_2 = \emptyset$. The differences between the current and new interleaving operator are highlighted at the end of this section.

A program is called reactive if its behavior can be observed or even altered at intermediate stable states between initialization and termination [24]. This is the precise case of *OhCircus*. There are four observational variables used to define a program behavior: the Boolean variable *ok* is true for a program that has started in a stable state; its decorated version *ok'* is true when a program, subsequently, stabilizes in an observable state. The Boolean variable *wait* is true if a program is prepared to engage but is waiting for some event; *wait'* is true if a program is in a stable intermediate state and false when the program has reached a final state. The variable *tr* records the sequence of events engaged

by a process. The variable ref records the set of events that a process may refuse before starting; ref' records the set of events that a process may refuse in a stable intermediate state of its execution. The non-observational variables v and v' stand, respectively, for the initial and intermediate values of all program variables.

In the UTP a process is defined as a reactive design of the form $\mathbf{R}(pre \vdash post)$. The design $pre \vdash post$ means $ok \wedge pre \Rightarrow ok' \wedge post$: if a program starts in a state satisfying its preconditions it will terminate and satisfy its postconditions [34]. Using the reactive design we present below the formal UTP semantics for $A_1 \llbracket ns_1 \mid ns_2 \rrbracket A_2$, which represents the interleave of the actions A_1 and A_2 considering a possibly non empty intersection between the variables in ns_1 and ns_2 .

$$A_1 \llbracket ns_1 \mid ns_2 \rrbracket A_2 \hat{=} \mathbf{R} \left(\begin{array}{l} \neg A_{1f}^f \wedge \neg A_{2f}^f \\ \vdash \\ ((A_{1f}^t; U1(out\alpha A_1)) \wedge (A_{2f}^t; U2(out\alpha A_2)))_{\{v, tr\}}; M_{\parallel} \end{array} \right)$$

The precondition $A_{1f}^f \wedge \neg A_{2f}^f$ is a shortcut for $A_1[false/ok'][false/wait] \wedge \neg A_2[false/ok'][false/wait]$. $A[false/ok'][false/wait]$ represents an action A that diverges when it is not waiting for its predecessor to finish. Since both A_1 and A_2 may execute independently, the interleaving of these actions diverges if either of them diverge [24, 34]. So the precondition guarantees that A_1 and A_2 do not diverge when they are not waiting for their predecessor to finish.

In the postcondition we follow the parallel by merge principle used in [34] and defined in [24]. It executes both actions independently, merging their results when both finish. The notation A_f^t represents an action A that does not diverge when it is not waiting for its predecessor to finish. Consider that $A_{1f}^t; U1(out\alpha A_1)$. A_{1f}^t indicates the execution of A_1 without divergence. $out\alpha A$ and $in\alpha A$ stand, respectively, for the initial observations of the observational variables in A (undecorated) and for the subsequent observations (eventually the final ones) of A 's observational variables (dashed). As we are defining a postcondition, we are interested only in the final values of the observational variables of A_1 and A_2 . To avoid name conflicts in the predicate we use a renaming function Ui that prefixes with i the variables in these actions, generating a predicate in the form:

$$Ui(\{v'_1, \dots, v'_n\}) = i.v'_1 = v_1 \wedge \dots \wedge i.v'_n = v_n \quad (5.1)$$

For example the application of $U1$ to $\{ok', tr'\}$ will generate the predicate $1.ok' = ok \wedge 1.tr' = tr$. Divergence can only happen if it is possible for either of the actions to reach divergence, so the predicate the predicate $((A_{1f}^t; U1(out\alpha A_1)) \wedge (A_{2f}^t; U2(out\alpha A_2)))$ says that both A_1 and A_2 do not diverge. This predicate is extended with the state components and local variables v and traces tr accompanied by their dashed counterparts. It is expressed using the notation $P_{\{n\}}$, where P is a predicate and n a variable or trace, that means $P \wedge n' = n$.

The last entire predicate is passed to the interleave merge function, which merges the traces of both actions (tr), the state components, local variables (v) and the UTP

observational variables ($out\alpha A_1$ and $out\alpha A_2$), exactly as the parallel merge function found in [34].

$$M_{\parallel} \triangleq tr' - tr \in (1.tr - tr \parallel 2.tr - tr) \\ \wedge \left(\begin{array}{l} (1.wait \vee 2.wait) \wedge \\ ref' \subseteq (1.ref \cup 2.ref) \\ < wait' > \\ \neg 1.wait \wedge \neg 2.wait \wedge MSt \end{array} \right)$$

In $M_{\parallel}$ the sequence of traces generated by the execution of A_1 and A_2 , $(tr' - tr)$ must be a sequence generated by the interleave composition of the traces of A_1 and A_2 ; this operator is defined in [38]. The interleave composition only terminates if both actions do so. So if $wait'$ is true it is because one of the actions has not finished, $1.wait \vee 2.wait$, and the refusals is contained or equals to the refusals of A_1 and A_2 together. Otherwise if $wait'$ is false it means that both actions has terminated $\neg 1.wait \wedge \neg 2.wait$ and the state components and local variables have changed according to the predicate generated by MSt .

$$MSt \triangleq \forall v \bullet (v \in ns_1 \wedge v \notin ns_2 \Rightarrow v' = 1.v) \wedge (v \in ns_2 \wedge v \notin ns_1 \Rightarrow v' = 2.v) \wedge \\ (v \in ns_1 \cap ns_2 \Rightarrow v' = 1.v = 2.v) \wedge \\ (v \notin ns_1 \cup ns_2 \Rightarrow v' = v)$$

This predicate says that each variable in v is changed by A_1 if it belongs uniquely to ns_1 , by A_2 if it belongs uniquely to ns_2 . If $v \in ns_1 \cap ns_2$, A_1 and A_2 must agree in the final value of v . This apparent inconsistency with the semantics of parallel composition is solved, if we consider that the changes made in a state component by schemas in a subprocess cannot contradict the changes made schemas in superprocess.

The new parallel operator is symmetric, associative and distributive. The proof of these properties is similar to the original parallel operator.

The main difference between our interleaving operator and that defined in [34] concerns mainly MSt . The original form to this function considers ns_1 and ns_2 disjuncts.

The semantics of our new parallel operator assures that if ns_1 and ns_2 are disjuncts it behaves exactly as the interleave parallel operator defined in [34]. It guarantees that the laws developed for *OhCircus* and *Circus* still valid.

5.2.1 UTP Semantics for super and protected

The formal meaning of **super** clause, crucial in the proofs of our algebraic laws, must be given, as well as the meaning of the new **protected** access level. We formalize these new constructions in the UTP, as previously done for the new interleave operator.

Schemas in *OhCircus* can be normalized in the same way as schemas in *Z*. The normalization technique moves all restrictions from the declaration part of a schema to its predicate part. This reduces the declaration part to a canonical form [51]. As an illustration of the schema normalization, consider a process with state St and a schema *NextMonth*, as bellow:

$\frac{St}{\begin{array}{l} month : \mathbb{N} \\ year : \mathbb{N} \end{array}} \quad \frac{1 \leq month \leq 12}{year \geq 2011}$	$\frac{NextMonth}{\Delta St} \quad \frac{month' = (month + 1) \bmod 12}{(month + 1) = 13 \Rightarrow years' = years + 1}$
---	---

The normalization of *NextMonth* will generate the equivalent schema:

$\frac{NextMonth}{month, month', year, year' : \mathbb{Z}}$ $month \in \mathbb{N} \wedge month' \in \mathbb{N} \wedge month' = (month + 1) \bmod 12 \wedge$ $(1 \leq month \leq 12) \wedge (1 \leq month' \leq 12) \wedge$ $year \in \mathbb{N} \wedge year' \in \mathbb{N} \wedge ((month + 1) = 13 \Rightarrow years' = years + 1) \wedge$ $year \geq 2011 \wedge year' \geq 2011$
--

The notation ΔSt introduces four state variables (*month*, *month'*, *year* and *year'*) of type $\mathbb{N}$, but in the canonical form they assume the more general type $\mathbb{Z}$. Next, this restriction about the type of the four variables is included in the schema predicate, as well as its original schema predicate and the invariant of *St*.

In [34], the semantics of a Z schema is obtained by transforming it into a statement, whose semantics is given by the following reactive design.

$$w : [pre, post] \hat{=} \mathbf{R} (pre \vdash post \wedge \neg wait' \wedge tr' = tr \wedge u' = u)$$

By this reactive design a statement terminates successfully ($\neg wait'$), satisfying its postcondition, if its precondition holds. The traces are unchanged ($tr' = tr$) as well as the variables outside *w*, represented by *u* ($u' = u$). A transformation of a normalized schema into a statement is given bellow.

$$[udecl; ddecl' \mid pred] \hat{=} ddecl : [\exists ddecl' \bullet pred, pred]$$

In a normalized schema the notations for input (?) and output (!) are replaced by undashed (*udecl*) and dashed (*ddecl'*) variables, respectively. A predicate (*pred*) determines the effect of the schema. In the statement the variables in *ddecl* assume a final state that satisfies the predicate (*pred*), the precondition; the predicate itself is the postcondition. As an example consider the meaning of *NextMonth* in this semantics.

$$\begin{array}{l}
month, year : \\
\left[\begin{array}{l}
\exists month', year' : \mathbb{Z} \bullet \\
\left(\begin{array}{l}
month \in \mathbb{N} \wedge month' \in \mathbb{N} \wedge month' = (month + 1) \bmod 12 \wedge \\
(1 \leq month \leq 12) \wedge (1 \leq month' \leq 12) \wedge \\
year \in \mathbb{N} \wedge year' \in \mathbb{N} \wedge ((month + 1) = 13 \Rightarrow year' = year + 1) \wedge \\
year \geq 2011 \wedge year' \geq 2011
\end{array} \right) , \\
\\
\left(\begin{array}{l}
month \in \mathbb{N} \wedge month' \in \mathbb{N} \wedge month' = (month + 1) \bmod 12 \wedge \\
(1 \leq month \leq 12) \wedge (1 \leq month' \leq 12) \wedge \\
year \in \mathbb{N} \wedge year' \in \mathbb{N} \wedge ((month + 1) = 13 \Rightarrow year' = year + 1) \wedge \\
year \geq 2011 \wedge year' \geq 2011
\end{array} \right)
\end{array} \right]
\end{array}$$

We extend this definition to deal with schemas having a **super** clause; it formalizes the **super** semantics. Consider the processes $P_1, P_2, \dots, P_n$, where $P_1 < P_2 < \dots < P_n$ and the schemas $P_1.sc_1, P_2.sc_2, \dots, P_n.sc_n$ where sc_1 references sc_2 via **super** clause, which itself references sc_3 and so on; sc_n has no **super** clause. The semantics of sc_1 , in the UTP, is given bellow. We consider each sc as a normalized protected schema.

$$sc_1 \hat{=} [sc_1.udecl; \dots; sc_n.udecl; sc_1.ddecl'; \dots; sc_n.ddecl' \mid sc_1.pred; \dots; sc_n.pred]$$

The UTP semantics for **protected** is quite simple since it does not change the original semantics for a schema. Therefore a protected schema sc means exactly what means in default level.

$$\mathbf{protected} \ sc \hat{=} sc$$

The same is true for a protected state component. Nevertheless, a protected schema or state component in a superprocess impacts in the meaning of its subprocesses, in any hierarchy level. Consider two processes P and Q , where $Q < P$.

```

process  $P \hat{=}$ 
  state  $st \wedge [x : T \mid pred]$ 
   $sc$ 
   $pps$ 
   $\bullet act$ 
end
process  $Q \hat{=}$  extends  $P$ 
  state  $st$ 
   $pps$ 
   $\bullet act$ 
end

```

The meaning of Q , in the UTP, is given by:

$$Q \hat{=} \left(\begin{array}{l} \text{begin state} \hat{=} P.st \wedge [x : T \mid pred] \wedge Q.st \\ \quad P.pps_1 \wedge sc_{\wedge} \Xi Q.st \\ \quad P.pps_2^{ref} \wedge \Xi Q.st \\ \quad Q.pps \\ \quad \bullet P.act \llbracket P.st \wedge P.x \mid P.st \wedge P.x \wedge Q.st \rrbracket Q.act \\ \text{end} \end{array} \right)$$

where $P.pps_2^{ref}$ are obtained removing the elements redefined in $Q.pps$. The schemas in $Q.pps_1$ have not the **super** clause and cannot access the default state components $P.st_1 \wedge P.x$. This is possible for schemas in $Q.pps_2$, which have the **super** clause. If we change the above specification to:

```

process P  $\hat{=}$ 
  state st  $\wedge$  [protected x : T | pred]
  protected sc
  pps
  • act
end
process Q  $\hat{=}$  extends P
  state st
  pps
  • act
end

```

The meaning of Q in UTP becomes:

$$Q \hat{=} \left(\begin{array}{l} \text{begin state} \hat{=} P.st \wedge [\text{protected } x : T \mid pred] \wedge Q.st \\ \quad P.pps_1 \wedge \Xi Q.st \\ \quad P.pps_2^{ref} \wedge \Xi Q.st \\ \quad \text{protected } sc \\ \quad Q.pps \\ \quad \bullet P.act \llbracket P.st \wedge P.x \mid P.st \wedge P.x \wedge Q.st \rrbracket Q.act \\ \text{end} \end{array} \right)$$

As $P.sc$ is now a protected schema, it can be overridden by Q , therefore the addition of $\Xi Q.st$ to this schema is removed. Changing $P.x$ to protected allows all schemas, including the main action of Q to access it directly.

5.3 Final Considerations

The original design of *OhCircus* [11] allows a process to extend another, but without code reuse. We develop two new constructs that allow code reuse. First, we define a

new access level to state components and schemas, the **protected** level. Elements in this level are inherited by subprocesses. Another construction, the **super** clause, lets a schema in a subprocess reference any protected schema present (inherited or defined) in its superprocesses.

In the sequel, a process that models a limited buffer was specified. It offers the possibility to add an element at a time, until its limit is reached. A process that extends this initial behavior is modeled. It provides, in addition to the superprocess, the capacity of adding two elements at once, checking if the buffer is empty and the average of the stored elements. We use the algebraic laws to translate both processes to *Circus*.

This chapter presents the semantics, in the UTP, of process inheritance considering the new proposed constructions. This semantics is based on the mapping of any process in *OhCircus* to a regular process in *Circus*, whose semantics are completely defined in [34].

CHAPTER 6

Algebraic Laws

This chapter presents a set of soundness algebraic laws for *OhCircus*. These laws address specifications with a process hierarchy. As far as we are aware, this is an original contribution of this work, as it seems to be the first systematic characterization of a comprehensive set of laws for process inheritance that use rich data types and access control for state components and behavior components (Z schemas).

The proposed laws act on the process hierarchy of a specification, as well as on the state components and schemas of a process. Laws of actions for *Circus*, proposed in [9], are also valid for *OhCircus*, so this is not the focus in this work.

The laws are classified into two groups. Simple laws are justified directly from the semantics, whereas composite laws are proved to be a consequence of other laws, simple or composed laws.

Each law may have a **provided** clause that contains the premisses that must be satisfied before its application. As an algebraic law has always two directions of application, we must define the premisses for each direction.

Consider two processes P and Q . If P is refined by Q ($P \sqsubseteq Q$) the main action of P is refined by that of Q ($P.act \sqsubseteq Q.act$). From the semantics of process inheritance, if $Q < P$, the state components of Q are those declared in Q and those inherited from P . The same rule is applied to Z schemas. As process inheritance is a transitive relation, the elements inherited from P are those declared in P and those inherited from its immediate superprocess, and so on. Differently from state components and schemas, actions cannot be inherited, so any action defined in P is hidden from Q . These restrictions were formalized in the Chapter 4. Although the actions are not eligible for inheritance, the behavior of the main action of a subprocess Q is given by that declared in Q in interleave with that **representing** (not necessarily the declared action) the behavior of its immediate superprocess. The reason is that the superprocess P might itself inherit from another process. These behavior was formalized in the Chapter 5.

The behavior of a process is not given only by its declared main action. For a subprocess Q there is an implicit main action, $Q.\overline{act}$, formed by the interleave of its declared main action $Q.act$ and the implicit main action $P.\overline{act}$ of its immediate superprocess P . If P is at the top of the inheritance hierarchy, then $P.\overline{act} = P.act$, otherwise it follows the same reasoning applied for Q .

If we have $Q < P$, it implies that $P.\overline{act} \sqsubseteq Q.\overline{act}$. This implication is used in the proof of correctness of our laws. As already mentioned, the laws of actions are the subject of [9]. We use such laws to justify some of the laws we propose for processes involved in an arbitrary inheritance hierarchies.

6.1 The semantics of *Circus* processes

In accordance with [34] (see Appendix C) a process meaning is given by a command. Consider the process P as bellow:

$$P \triangleq \left(\begin{array}{l} \text{begin state} \triangleq P.st \\ P.pps \\ \bullet P.act \\ \text{end} \end{array} \right)$$

The meaning of such a process is the command:

$$\mathbf{var} P.st.decl \bullet ER(P.act, P.st.inv, P.pps) \quad (6.1)$$

which, also by [34], is defined by the existential quantification:

$$\exists P.st.decl, P.st.decl' \bullet ER(P.act, P.st.inv, P.pps) \quad (6.2)$$

In the above predicate $P.st.decl$ denotes the declaration part of $P.st$. Its invariants $P.st.inv$ are conjoined with the schemas $P.pps$ and in the main action $P.act$, in the form **if** $P.st.inv \rightarrow \text{Skip} \parallel \text{Stop}$, after each atomic component action.

Finally all occurrences of schemas $P.pps$ (conjoined with $P.st.inv$) in the main action are replaced by semantically equivalent commands (specification statements) following the semantics for schemas presented in [34]. All this work is done by the enforce-replace (ER) function.

$$\begin{aligned} ER &: \text{Action} \rightarrow \mathbb{P} \text{Constraint} \rightarrow \mathbb{P} \text{PParagraph} \rightarrow \text{Action} \\ ER \ a \ \emptyset \ \emptyset &= a \\ ER \ a \ \emptyset \ p &= R(a, p) \\ ER \ a \ inv \ \emptyset &= E(a, inv) \\ ER \ a \ i : invs \ p : pps &= ER(E(R(a, E(p, i)), i), invs, pps) \end{aligned}$$

When this function receives an action and empty sets of constraints and paragraphs returns the action unchanged. If the set of invariants is empty, the function R replace the occurrences of a given schema p in the action a by its equivalent specification statement (see Subsection 5.2.1). When the set of schemas is empty, the function E enforces the action a in order to guarantee the invariant inv . In the general case, the function recursively enforces all paragraphs with the invariants, replaces them in the action, and enforces this resulting action with the invariants.

$$\begin{aligned} E &: \text{Action} \rightarrow \text{Constraint} \rightarrow \text{Action} \\ E \ a \ inv &= a; \text{if } inv \rightarrow \text{Skip} \parallel \text{Stop} \\ E \ (a \ op \ b) \ inv &= E(a, inv) \ op \ E(b, inv) \end{aligned}$$

The function E receives an action and an invariant, if the action a is atomic E returns a ; **if** $inv \rightarrow \text{Skip} \parallel \text{Stop}$. Otherwise if the action can be written as a composition of other actions ($a \text{ op } b$), using a proper operator op , the function enforces each action and composes the result actions with op . We overload this function to schemas. It receives a schema and an invariant and returns a schema whose predicate is conjoined with the invariant.

$$E : \text{Paragraph} \rightarrow \text{Constraint} \rightarrow \text{Paragraph}$$

$$E [udecl; ddecl' \mid pred] inv = [udecl; ddecl' \mid pred \wedge inv]$$

Finally the function R replaces all occurrences of a schema in an action by its equivalent specification statement. If an atomic action a is just a call for a schema sc ($a = sc$ with a small abuse of notation), it can be replaced by sc equivalent specification statement, otherwise a is returned unchanged. If the action can be written as a composition of other actions ($a \text{ op } b$), using a proper operator op , the function replaces in each action all occurrences of sc by its equivalent specification statement and composes the resulting actions with op .

$$R : \text{Action} \rightarrow \text{Paragraph} \rightarrow \text{Action}$$

$$R a \text{ sc} \hat{=} [udecl; ddecl' \mid pred] = \text{if } a = sc \text{ then } [udecl; ddecl' \mid pred] \text{ else } a$$

$$R (a \text{ op } b) \text{ sc} = R(a, \text{sc}) \text{ op } R(b, \text{sc})$$

To prove some of the laws presented in this chapter we need to work with the semantics of a process at the predicate level. These laws, excepting the Law 6.3, are proved from the UTP semantics for process inheritance, based on [34]. Since [34] does not present the semantics for a program, we use in the Law 6.3 the semantics developed in [50] for a *Circus* program. Considering the absent of object-orientation constructs we can apply this semantics to a program in *OhCircus*, where process inheritance was removed by our semantics developed in Chapter 5. It is not our intent to provide a possible theory of equivalence between [50] and [34].

6.2 Laws

We categorize the laws in three groups: localized eliminations, access modifications and element interchanges between processes related by inheritance. The laws in the first group insert or remove elements of a process considering its hierarchy. In the second group we have the laws that change the access modifiers of state components and schemas. The latter groups laws responsible for interchange elements between a process and one of its subprocess.

6.2.1 Access Modifications

To change the access level of a schema sc in P from protected to default we must guarantee that all superprocesses of P do not declare a schema homonymous to sc , or if it happens, this schema must have the default access level. Note that it is not possible to have a default schema in a subprocess with the same name as one in the protected level in any

of its superprocesses; therefore we do not need explicit side conditions to capture this. In addition, all subprocesses of P cannot redefine sc .

The provisos in the reverse direction of the law application guarantee that the superprocesses of P have not a schema homonymous to sc . According to our typing rules a default schema cannot redefine a protected one. Furthermore, the subprocesses of P have not a schema homonymous to sc . The typing rule mentioned motivates this restriction.

Law 6.1 (change schema access level).

$$\begin{array}{|l}
 \text{process } P \hat{=} \text{extends } Q \\
 \quad \text{state } st \\
 \quad \text{protected } sc \\
 \quad pps \\
 \quad \bullet \text{ } act \\
 \text{end}
 \end{array}
 \quad =_{pds} \quad
 \begin{array}{|l}
 \text{process } P \hat{=} \text{extends } Q \\
 \quad \text{state } st \\
 \quad sc \\
 \quad pps \\
 \quad \bullet \text{ } act \\
 \text{end}
 \end{array}$$

provided

- $(\rightarrow) (1) \forall Q \mid P < Q \bullet (\forall s \in Q.pps \mid N(s) = N(sc) \rightarrow \neg PL(s))$
 $(2) \forall R \mid R < P \bullet \neg occurs(sc, R.act) \wedge \forall s \in R.pps \bullet (\neg N(sc) = N(s) \wedge \neg occurs(sc, s))$
 $(\leftarrow) (1) \forall Q \mid P < Q \bullet sc \notin P.pps \quad (2) \forall R \mid R < P \bullet sc \notin R.pps$

proof

The meaning of P on the left-hand side is defined as:

$$P \hat{=} \left(\begin{array}{l}
 \text{begin state } \hat{=} Q.st \wedge P.st \\
 \quad Q.pps_{1 \wedge} \Xi P.st \\
 \quad Q.pps_{2 \text{ } ^{ref}} \wedge \Xi P.st \\
 \quad P.pps \\
 \quad \text{protected } P.sc \\
 \quad \bullet Q.act \llbracket Q.st \mid Q.st \wedge P.st \rrbracket P.act \\
 \text{end}
 \end{array} \right)$$

$= [\text{By Lemma D.1}]$

$$P \hat{=} \left(\begin{array}{l}
 \text{begin state } \hat{=} Q.st \wedge P.st \\
 \quad Q.pps_{1 \wedge} \Xi P.st \\
 \quad Q.pps_{2 \text{ } ^{ref}} \wedge \Xi P.st \\
 \quad P.pps \\
 \quad P.sc \\
 \quad \bullet Q.act \llbracket Q.st \mid Q.st \wedge P.st \rrbracket P.act \\
 \text{end}
 \end{array} \right)$$

That is the exact meaning of P , in the UTP, on the right-hand template.

A subprocess of P , lets say R , in pds means:

$$R \hat{=} \left(\begin{array}{l} \mathbf{begin\ state} \hat{=} Q.st \wedge P.st \wedge R.st \\ Q.pps_1 \wedge \Xi P.st \wedge R.st \\ Q.pps_2^{ref} \wedge \Xi P.st \wedge R.st \\ P.pps_1 \wedge \Xi R.st \\ P.pps_2^{ref} \Xi R.st \\ \mathbf{protected\ } P.sc \\ R.pps \\ \bullet Q.act \llbracket Q.st \mid Q.st \wedge P.st \rrbracket P.act \\ \llbracket Q.st \wedge P.st \mid Q.st \wedge P.st \wedge R.st \rrbracket \\ R.act \\ \mathbf{end} \end{array} \right)$$

= [By Lemma D.1]

$$R \hat{=} \left(\begin{array}{l} \mathbf{begin\ state} \hat{=} Q.st \wedge P.st \wedge R.st \\ Q.pps_1 \wedge \Xi P.st \wedge R.st \\ Q.pps_2^{ref} \wedge \Xi P.st \wedge R.st \\ P.pps_1 \wedge \Xi R.st \\ P.pps_2^{ref} \Xi R.st \\ P.sc \\ R.pps \\ \bullet Q.act \llbracket Q.st \mid Q.st \wedge P.st \rrbracket P.act \\ \llbracket Q.st \wedge P.st \mid Q.st \wedge P.st \wedge R.st \rrbracket \\ R.act \\ \mathbf{end} \end{array} \right)$$

The above process is the exact meaning of R , in the UTP, on the right-hand template. This can be generalized for each subprocess of P , in any hierarchy level. This concludes our proof. The application of this law in the reverse direction is similar and we omit it here.

□

To reduce the access level of a state component st_1 in P , from protected to default level, it cannot be used by schemas nor actions of P 's subprocesses. As the superprocesses of P do not even know that P exists and the overriding of state components is not allowed, no restrictions are applied to them.

A protected state component is unique in a process hierarchy, so if a process declares a protected state component st_1 neither of its super/subprocesses can have one homonymous to st_1 . The proviso for the right application of the law guarantees that the state component st_1 is unique in the P 's hierarchy. The uniqueness for protected state components does not apply for those with default access level. Therefore it is perfectly possible

to have a default state component $A.st_1$ and a protected $B.st_1$, where $B < A$, as $A.st_1$ is not visible in B . Otherwise if $A.st_1$ is protected, it is visible in B and we have that $st_1 \notin B.st$.

Law 6.2 (change state component access level).

<pre> process $P \hat{=}$ extends Q state $st \wedge$ protected st_1 pps $\bullet$ act end </pre>	$=_{pds}$	<pre> process $P \hat{=}$ extends Q state $st \wedge st_1$ pps $\bullet$ act end </pre>
---	-----------	---

provided

$$\begin{aligned}
 (\rightarrow) \quad & \forall R \mid R < P \bullet \neg occurs(R.st_1, R.act) \wedge \neg occurs(R.st_1, R.pps) \\
 (\leftarrow) \quad & \forall R \mid R < P \bullet st_1 \notin PS(R.st) \wedge \forall Q \mid P < Q \bullet st_1 \notin PS(Q.st)
 \end{aligned}$$

proof

The meaning of P in the left-hand side is defined as:

$$P \hat{=} \left(\begin{array}{l} \textbf{begin state} \hat{=} Q.st \wedge P.st \wedge \textbf{protected } st_1 \\ \quad Q.pps_{s1} \wedge \Xi P.st \wedge P.st_1 \\ \quad Q.pps_2^{ref} \wedge \Xi P.st \wedge P.st_1 \\ \quad P.pps \\ \quad \bullet Q.act \llbracket Q.st \mid Q.st \wedge P.st \wedge P.st_1 \rrbracket P.act \\ \textbf{end} \end{array} \right)$$

= [By Lemma D.2]

$$P \hat{=} \left(\begin{array}{l} \textbf{begin state} \hat{=} Q.st \wedge P.st \wedge st_1 \\ \quad Q.pps_{s1} \wedge \Xi P.st \wedge P.st_1 \\ \quad Q.pps_2^{ref} \wedge \Xi P.st \wedge P.st_1 \\ \quad P.pps \\ \quad \bullet Q.act \llbracket Q.st \mid Q.st \wedge P.st \wedge P.st_1 \rrbracket P.act \\ \textbf{end} \end{array} \right)$$

The above process is the exact meaning of P , in the UTP, on the right-hand template. A subprocess of P , lets say R , in pds means:

$$R \hat{=} \left(\begin{array}{l} \text{begin state} \hat{=} Q.st \wedge P.st \wedge \text{protected } P.st_1 \wedge R.st \\ Q.pps_1 \wedge \Xi P.st \wedge P.st_1 \wedge R.st \\ Q.pps_2^{ref} \wedge \Xi P.st \wedge P.st_1 \wedge R.st \\ P.pps_1 \wedge \Xi R.st \\ P.pps_2^{ref} \Xi R.st \\ R.pps \\ \bullet Q.act \llbracket Q.st \mid Q.st \wedge P.st \wedge P.st_1 \rrbracket P.act \\ \llbracket Q.st \wedge P.st \wedge P.st_1 \mid Q.st \wedge P.st \wedge P.st_1 \wedge R.st \rrbracket \\ R.act \\ \text{end} \end{array} \right)$$

= [By Lemma D.2]

$$R \hat{=} \left(\begin{array}{l} \text{begin state} \hat{=} Q.st \wedge P.st \wedge P.st_1 \wedge R.st \\ Q.pps_1 \wedge \Xi P.st \wedge P.st_1 \wedge R.st \\ Q.pps_2^{ref} \wedge \Xi P.st \wedge P.st_1 \wedge R.st \\ P.pps_1 \wedge \Xi R.st \\ P.pps_2^{ref} \Xi R.st \\ R.pps \\ \bullet Q.act \llbracket Q.st \mid Q.st \wedge P.st \wedge P.st_1 \rrbracket P.act \\ \llbracket Q.st \wedge P.st \wedge P.st_1 \mid Q.st \wedge P.st \wedge P.st_1 \wedge R.st \rrbracket \\ R.act \\ \text{end} \end{array} \right)$$

The above process is the exact meaning of R , in the UTP, on the right-hand template. This can be generalized for each subprocess of P , in any hierarchy level. This concludes our proof. The application of the this law in the reverse direction is similar. $\square$

6.2.2 Localized Eliminations

The meaning of a program in *OhCircus* is given by the meaning of each process and class declaration. Furthermore, global Z schema, constant, channel, and channel set definitions [50] are also taken into account.

A process having its main action as *Skip* does not affect the meaning of a program in *OhCircus*. In addition it does not have a superprocess, but can be one, since the meaning of its subprocesses remains unchanged with or without the inheritance relation.

We use the notation $occurs(P, pds)$ to represent the fact that the process P is used (as superprocess or in a process composition) by at least one process in pds . The function N defines the set of process names of a set of process declarations.

On the right-hand side of this law, the proviso guarantees that the process P is not used in pds . On the left-hand side the process declared in pd_1 has a fresh name in pds .

Law 6.3 (process elimination).

$pds \text{ } pd_1 = pds$

where

$pd_1 = \text{process } P \hat{=} \text{begin} \bullet \text{Skip end}$

provided

$(\leftrightarrow) \neg \text{occurs}(P, pds)$

proof

This law can be directly justified from the semantics of programs given in [50], where a program is defined as a conjunction of the semantics of each component process. Therefore in the semantics of $pds \text{ } pd_1$ below, $\llbracket prog \rrbracket^{PROG}$ stands for the semantics of a program $prog$ and $\llbracket p \rrbracket^P$ for the semantics of a process p . The semantics is defined as a relation where *Skip* is mapped into an empty relation.

$$\begin{aligned}
 & \llbracket pds \text{ } pd_1 \rrbracket^{PROG} \\
 &= [\text{From the semantics of programs}] \\
 & \llbracket pds \rrbracket^{PROG} \llbracket pd_1 \rrbracket^P \\
 &= [\text{From the assumption}] \\
 & \llbracket pds \rrbracket^{PROG} \emptyset \\
 &= [\text{Skip is the empty relation}] \\
 & \llbracket pds \rrbracket^{PROG}
 \end{aligned}$$

□

A default schema $P.sc$ can be eliminated from P when it is no longer referenced by P ' schemas or used by its actions. No restrictions are applied to its superprocesses or subprocesses. This is justified by the fact that a default schema cannot be inherited and, consequently, it cannot be redefined. $PL(sc)$ represents the fact that sc is a protected schema.

The addition of a default schema sc in P is allowed if sc is distinct from those schemas declared in P and if in all superprocesses of P there is not a protected schema with the same name as sc . The latter condition must hold because a default schema cannot refine a protected one. There are no restrictions about the subprocesses of P since the default elements of P are hidden.

Law 6.4 (default schema elimination).

<pre> process $P \hat{=}$ extends Q state st sc pps $\bullet \text{ } act$ end </pre>	$=_{pds}$	<pre> process $P \hat{=}$ extends Q state st pps $\bullet \text{ } act$ end </pre>
---	-----------	---

provided

$(\rightarrow) \neg \text{occurs}(P.sc, P.act) \wedge \neg \text{occurs}(P.sc, P.pps)$

$(\leftarrow) sc \notin P.pps \wedge (\forall Q \mid P < Q \bullet sc \notin Q.pps, \text{if } PL(sc))$

proof

The semantics of P on the left-hand side is defined as:

$$P \hat{=} \left(\begin{array}{l} \text{begin state} \hat{=} Q.st \wedge P.st \\ Q.pps_1 \wedge \Xi P.st \\ Q.pps_2 \overset{ref}{\wedge} \Xi P.st \\ P.pps \\ P.sc \\ \bullet Q.act \llbracket Q.st \mid Q.st \wedge P.st \rrbracket P.act \\ \text{end} \end{array} \right)$$

= [By C.1]

$$\begin{aligned} & \text{var } Q.st.decl \wedge P.st.decl \bullet ER(Q.act \llbracket Q.st \mid Q.st \wedge P.st \rrbracket \\ & \quad P.act, P.st.inv \wedge Q.st.inv, Q.pps \wedge P.pps \wedge P.sc) \end{aligned} \quad (6.3)$$

= [By C.3]

$$\begin{aligned} & \exists Q.st.decl \wedge P.st.decl, Q.st.decl' \wedge P.st.decl' \bullet ER(Q.act \llbracket Q.st \mid Q.st \wedge P.st \rrbracket \\ & \quad P.act, P.st.inv \wedge Q.st.inv, Q.pps \wedge P.pps \wedge P.sc) \end{aligned} \quad (6.4)$$

In the above predicate $Q.st.decl$ denotes the declaration part of $Q.st$; the same applies to $P.st.decl$ (their dashed values represents variables final values). Their invariants, respectively $Q.st.inv$ and $P.st.inv$ are conjoined with the schemas $Q.pps \wedge P.pps \wedge P.sc$ and enforced in the main action $Q.act \llbracket Q.st \mid Q.st \wedge P.st \rrbracket P.act$ by the ER function.

Finally all occurrences of schemas $Q.pps \wedge P.pps \wedge P.sc$ (conjoined with $Q.st.inv \wedge P.st.inv$) in the main action are replaced by semantically equivalent commands (specification statements) following the semantics for schemas presented in [34]. All this work is done by the enforce-replace (ER) function.

Considering an action A , $ER(A, inv, pps \wedge sc) = ER(A, inv, pps)$ if $\neg \text{occurs}(sc, A)$ and $\neg \text{occurs}(sc, pps)$. Considering this and the provisos we have that the semantics of P is:

$$\begin{aligned} & \exists Q.st.decl \wedge P.st.decl, Q.st.decl' \wedge P.st.decl' \bullet ER(Q.act \llbracket Q.st \mid Q.st \wedge P.st \rrbracket \\ & \quad P.act, P.st.inv \wedge Q.st.inv, Q.pps \wedge P.pps) \end{aligned} \quad (6.5)$$

= [By C.3]

$$\begin{aligned} & \mathbf{var} Q.st.decl \wedge P.st.decl \bullet ER(Q.act \llbracket Q.st \mid Q.st \wedge P.st \rrbracket \\ & \quad P.act, P.st.inv \wedge Q.st.inv, Q.pps \wedge P.pps) \end{aligned} \quad (6.6)$$

= [By C.1]

$$P \hat{=} \left(\begin{array}{l} \mathbf{begin\ state} \hat{=} Q.st \wedge P.st \\ \quad Q.pps_{s_1 \wedge} \Xi P.st \\ \quad Q.pps_{s_2}^{ref} \wedge \Xi P.st \\ \quad P.pps \\ \quad \bullet Q.act \llbracket Q.st \mid Q.st \wedge P.st \rrbracket P.act \\ \mathbf{end} \end{array} \right)$$

The above process is the exact meaning of P , in the UTP, on the right-hand template. This concludes our proof. The application of the this law in the reverse direction is similar and we omit it here. $\square$

To remove a default state component st_1 from P it is necessary that st_1 be not used by actions nor schemas of P . No restrictions are applied to the subprocesses of P since st_1 is a default schema and cannot be inherited.

The insertion of default state component st_1 in P is conditional to the absence of a state component in P homonymous to st_1 . No restrictions are applied to super/subprocesses of P since default state components can appear many times in a process hierarchy, whereas protected state components are unique.

Law 6.5 (default state component elimination).

$\begin{array}{l} \mathbf{process\ } P \hat{=} \mathbf{extends\ } Q \\ \quad \mathbf{state\ } st \wedge st_1 \\ \quad pps \\ \quad \bullet \textit{act} \\ \mathbf{end} \end{array}$	$=_{pds}$	$\begin{array}{l} \mathbf{process\ } P \hat{=} \mathbf{extends\ } Q \\ \quad \mathbf{state\ } st \\ \quad pps \\ \quad \bullet \textit{act} \\ \mathbf{end} \end{array}$
--	-----------	--

provided

$(\rightarrow) \neg occurs(P.st_1, P.act) \wedge \neg occurs(P.st_1, P.pps)$

$(\leftarrow) st_1 \notin P.st$

proof

The semantics of P in the left-hand side is defined as:

$$P \triangleq \left(\begin{array}{l} \mathbf{begin\ state} \triangleq Q.st \wedge P.st \wedge P.st_1 \\ \quad Q.pps_1 \wedge \Xi P.st \wedge P.st_1 \\ \quad Q.pps_2^{ref} \wedge \Xi P.st \wedge P.st_1 \\ \quad P.pps \\ \quad \bullet Q.act \llbracket Q.st \mid Q.st \wedge P.st \wedge P.st_1 \rrbracket P.act \\ \mathbf{end} \end{array} \right)$$

= [By C.1]

$$\begin{aligned} \mathbf{var} Q.st.decl \wedge P.st.decl \wedge P.st_1.decl \bullet ER(Q.act \llbracket Q.st \mid Q.st \wedge P.st \wedge P.st_1 \rrbracket P.act, \\ P.st.inv \wedge P.st_1.inv \wedge Q.st.inv, Q.pps \wedge P.pps) \end{aligned} \quad (6.7)$$

= [By C.3]

$$\begin{aligned} \exists Q.st.decl \wedge P.st.decl \wedge P.st_1.decl \wedge Q.st.decl' \wedge P.st.decl' \wedge P.st_1.decl' \\ \bullet ER(Q.act \llbracket Q.st \mid Q.st \wedge P.st \wedge P.st_1 \rrbracket P.act, \\ P.st.inv \wedge P.st_1.inv \wedge Q.st.inv, Q.pps \wedge P.pps) \end{aligned} \quad (6.8)$$

Considering that $\exists x; x', y, y' : T \bullet A = \exists x; x' : T \bullet A$ if y, y' do not occur in A ; and for an atomic action act , $act \rightarrow \mathbf{if} inv \rightarrow Skip \llbracket Stop = act \text{ if } inv \text{ is a predicate about a variable not used in } act \text{ (this implies that before and after } act \text{ execution, } inv \text{ has the same evaluation)}, \text{ the existential quantification, considering also the provisos, becomes:}$

$$\begin{aligned} \exists Q.st.decl \wedge P.st.decl \wedge Q.st.decl' \wedge P.st.decl' \\ \bullet ER(Q.act \llbracket Q.st \mid Q.st \wedge P.st \rrbracket P.act, \\ P.st.inv \wedge Q.st.inv, Q.pps \wedge P.pps) \end{aligned} \quad (6.9)$$

= [By C.3]

$$\begin{aligned} \mathbf{var} Q.st.decl \wedge P.st.decl \bullet ER(Q.act \llbracket Q.st \mid Q.st \wedge P.st \rrbracket P.act, \\ P.st.inv \wedge Q.st.inv, Q.pps \wedge P.pps) \end{aligned} \quad (6.10)$$

= [By C.1]

$$P \hat{=} \left(\begin{array}{l} \text{begin state} \hat{=} Q.st \wedge P.st \\ \quad Q.pps_1 \wedge \Xi P.st \\ \quad Q.pps_2^{ref} \wedge \Xi P.st \\ \quad P.pps \\ \quad \bullet Q.act \llbracket Q.st \mid Q.st \wedge P.st \rrbracket P.act \\ \text{end} \end{array} \right)$$

The above process is the exact meaning of P , in the UTP, on the right-hand template. This concludes our proof. The proof of the this law in the reverse direction is similar and we omit it here.

□

The insertion of a protected state component st_1 in P is possible if its superprocesses and subprocesses, including P itself, do not declare a protected state component homonymous to st_1 . The function PS determines, from a set of state components, those in the protected level.

The provisos for the right-hand side application of the law guarantee that the protected state component st_1 is not used by P nor by its subprocesses.

Law 6.6 (protected state component elimination).

<pre> process $P \hat{=} \text{extends } Q$ state $st \wedge \text{protected } st_1$ pps • act end </pre>	$=_{pds}$	<pre> process $P \hat{=} \text{extends } Q$ state st pps • act end </pre>
--	-----------	--

provided

- ($\rightarrow$) $\forall R \mid R \leq P \bullet \neg \text{occurs}(R.st_1, R.act) \wedge \neg \text{occurs}(R.st_1, R.pps)$
 ($\leftarrow$) $\forall R \mid R \leq P \bullet st_1 \notin PS(R.st) \wedge \forall Q \mid P < Q \bullet st_1 \notin PS(Q.st)$

proof

The proof of this law is a consequence of Law 6.2 and Law 6.5. We present the left-hand side proof; the right-hand side is similar.

```

process  $P \hat{=}$  extends  $Q$ 
  state  $st \wedge$ 
    protected  $st_1$ 
   $pps$ 
  •  $act$ 
end

```

= [By Law 6.2 application]

```

process  $P \hat{=}$  extends  $Q$ 
  state  $st \wedge st_1$ 
   $pps$ 
  •  $act$ 
end

```

= [By Law 6.5 application]

```

process  $P \hat{=}$  extends  $Q$ 
  state  $st$ 
   $pps$ 
  •  $act$ 
end

```

□

The insertion of a protected schema sc in P is possible if its superprocesses and subprocesses, including P itself, do not have a schema with the same name as sc . We abuse the notation when we write $sc \in pps$; in fact, it is true if the name of schema sc is used by at least one schema in pps .

In the case where a subprocess of P has a schema with the same name as that of sc , this must refine sc . Likewise, if a superprocess of P has an homonymous schema to sc , this schema must be refined by sc . Therefore, there is a restriction about schema overriding: if a schema is overridden in a subprocess the specialized version must maintain the same level of access as the original version, obviously, the protected level.

We overload the function *occurs* in $occurs(sc, R.act)$, $occurs(sc, R.pps)$ and $occurs(sc, R.sc)$. The former represents the fact that the schema sc is used in $R.act$; the second, the fact that sc is used in $P.pps$; the latter the fact that sc is referenced via the **super** clause in $R.sc$.

The proviso, on the right-hand side of the law, guarantees that the schema sc is not

used by any subprocess of P , neither by P itself. If it is the case, sc can be removed from P . This condition is needed since sc is in the protected level and all subprocesses of P inherits it. If a subprocess of P redefines sc without including it using the **super** clause, the removal of sc from P is still valid.

Law 6.7 (protected schema elimination).

$$\begin{array}{|l}
 \text{process } P \hat{=} \text{extends } Q \\
 \quad \text{state } st \\
 \quad \text{protected } sc \\
 \quad pps \\
 \quad \bullet \text{ } act \\
 \text{end}
 \end{array}
 \quad =_{pds} \quad
 \begin{array}{|l}
 \text{process } P \hat{=} \text{extends } Q \\
 \quad \text{state } st \\
 \quad pps \\
 \quad \bullet \text{ } act \\
 \text{end}
 \end{array}$$

provided

$$\begin{aligned}
 & (\rightarrow) \forall R \mid R < P \bullet \neg \text{occurs}(sc, R.act) \wedge \neg \text{occurs}(sc, R.pps) \vee \\
 & N(sc) \in N(R.pps) \Rightarrow P.sc \sqsubseteq R.sc \wedge \neg \text{occurs}(sc, R.sc) \\
 & (\leftarrow) sc \notin P.pps \wedge (\forall R \mid R < P \bullet sc \notin R.pps \vee (sc \in R.pps \wedge P.sc \sqsubseteq R.sc)) \wedge \\
 & (\forall Q \mid P < Q \bullet sc \notin Q.pps \vee (sc \in Q.pps \wedge Q.sc \sqsubseteq P.sc))
 \end{aligned}$$

proof

The proof of this law is a consequence of Law 6.1 and Law 6.4. We present the left-hand side proof; the right-hand side is similar.

$$\begin{array}{|l}
 \text{process } P \hat{=} \text{extends } Q \\
 \quad \text{state } st \\
 \quad \text{protected } sc \\
 \quad pps \\
 \quad \bullet \text{ } act \\
 \text{end}
 \end{array}
 = [\text{By Law 6.1 application and its provisos}]$$

$$\begin{array}{|l}
 \text{process } P \hat{=} \text{extends } Q \\
 \quad \text{state } st \\
 \quad sc \\
 \quad pps \\
 \quad \bullet \text{ } act \\
 \text{end}
 \end{array}$$

= [By Law 6.4 application and its provisos]

```

process  $P \hat{=}$  extends  $Q$ 
  state  $st$ 
   $pps$ 
  •  $act$ 
end

```

□

To remove '**super** sc ' from a schema sc in R , it is necessary that there exists a protected schema sc , in a superprocess of R , as in the bellow definition. This superprocess must be the closest process to R in its hierarchy. If $P.sc$ has the **super** clause, this is first resolved; as a process hierarchy is a finite structure, it is always possible to find a schema without **super**. The symbol $\oslash$ stands for the Z notation Ξ or Δ .

Law 6.8 (**super** elimination).

proof

This law is a direct consequence of the semantics of **super**, which is given in the process level. See Section 5.1. □

Whenever there is a protected schema sc in a superprocess of P , it is possible to define in P a protected schema $P.sc$, whose body is composed uniquely by a **super** clause referring to sc (see **super** semantics in subsection 5.2.1). $P.sc$ is a trivial redefinition of sc . This trivial redefinition can be removed, no matter the context.

Law 6.9 (eliminating a trivial schema redefinition).

$$\begin{array}{c}
 \boxed{
 \begin{array}{l}
 \text{process } Q \hat{=} \text{extends } M \\
 \text{state } st \\
 \text{protected } sc \\
 \text{pps} \\
 \bullet act \\
 \text{end} \\
 \\
 \text{process } P \hat{=} \text{extends } Q \\
 \text{state } st \\
 \text{protected } sc \hat{=} [\text{super } sc] \\
 \text{pps} \\
 \bullet act \\
 \text{end}
 \end{array}
 }
 \end{array}
 =
 \begin{array}{c}
 \boxed{
 \begin{array}{l}
 \text{process } Q \hat{=} \text{extends } M \\
 \text{state } st \\
 \text{protected } sc \\
 \text{pps} \\
 \bullet act \\
 \text{end} \\
 \\
 \text{process } P \hat{=} \text{extends } Q \\
 \text{state } st \\
 \text{pps} \\
 \bullet act \\
 \text{end}
 \end{array}
 }
 \end{array}$$

proof

The meaning of P on the left-hand side is defined as:

$$P \hat{=} \left(\begin{array}{l}
 \text{begin state } \hat{=} Q.st \wedge P.st \\
 Q.pps_1 \wedge \Xi P.st \\
 Q.pps_2^{ref} \wedge \Xi P.st \\
 P.pps \\
 \text{protected } P.sc \\
 \bullet Q.act \llbracket Q.st \mid Q.st \wedge P.st \rrbracket P.act \\
 \text{end}
 \end{array} \right)$$

According to the semantics of **super** clause, $P.sc$ has the same meaning as $Q.sc$; therefore, P becomes:

$$P \hat{=} \left(\begin{array}{l} \text{begin state} \hat{=} Q.st \wedge P.st \\ \quad Q.pps_1 \wedge \hat{=} P.st \\ \quad Q.pps_2^{ref} \wedge \hat{=} P.st \\ \quad P.pps \\ \quad \text{protected } Q.sc \\ \quad \bullet Q.act \llbracket Q.st \mid Q.st \wedge P.st \rrbracket P.act \\ \text{end} \end{array} \right)$$

The above process is the exact meaning of P , in the UTP, on the right-hand template. This concludes our proof. The application of the this law in the reverse direction is similar and we omit it here. $\square$

6.2.3 Element Interchanges

A state component st_2 of a process P can be moved to one of its subprocesses, say R , if st_2 is not used by P neither by its subprocesses, except those that are also subprocesses of R , including itself. For these, the state component st_2 will be inherited from R instead of P , and no restriction must be applied to them. It must be clear that st_2 is unique through P process hierarchy, so we do not need to check, for example, that $st_2 \notin R.st$, since according to our typing rules it must be satisfied by a valid specification in *OhCircus*. The provisos consider $P.st_2$ as a protected element.

If st_2 is a protected state component it can be moved to P if its subprocesses, excepting those that are also subprocesses of R , do not declare a protected state component equals to st_2 .

Law 6.10 (moving state component to subprocess).

provided

- $(\rightarrow) \forall S \mid S \leq P \wedge \neg (S \leq R) \bullet \neg occurs(st_2, S.pps) \wedge \neg occurs(st_2, S.act)$
 $(\leftarrow) \forall S \mid S \leq P \wedge \neg (S \leq R) \bullet st_2 \notin PS(S.st)$

proof

We observe that this law is not a compositional application (to remove and after insert st_2 from P to R) of the Law 6.4 nor Law 6.7. Move is an atomic operation, there is no intermediate states.

The meaning of P on the left-hand side is defined as:

$$\begin{aligned}
 P &\hat{=} \left(\begin{array}{l} \text{begin state} \hat{=} Q.st \wedge P.st_1 \wedge P.st_2 \\ Q.pps_1 \wedge \Xi \wedge P.st_1 \wedge P.st_2 \\ Q.pps_2^{ref} \wedge \Xi \wedge P.st_1 \wedge P.st_2 \\ P.pps \\ \bullet Q.act \llbracket Q.st \mid Q.st \wedge P.st_1 \wedge P.st_2 \rrbracket P.act \\ \text{end} \end{array} \right) \\
 &= [\text{By C.1}] \\
 &\quad \text{var } Q.st.decl \wedge P.st_1.decl \wedge P.st_2.decl \bullet ER(Q.act \llbracket Q.st \mid Q.st \wedge P.st_1 \wedge P.st_2 \rrbracket P.act, \\
 &\quad \quad P.act, Q.st.inv \wedge P.st_1.inv \wedge P.st_2.inv, Q.pps \wedge P.pps) \quad (6.11)
 \end{aligned}$$

$$\begin{aligned}
 &= [\text{By C.3}] \\
 &\quad \exists Q.st.decl \wedge P.st_1.decl \wedge P.st_2.decl \wedge Q.st.decl' \wedge P.st_1.decl' \wedge \quad (6.12) \\
 &\quad \quad P.st_2.decl' \bullet ER(Q.act \llbracket Q.st \mid Q.st \wedge P.st_1 \wedge P.st_2 \rrbracket P.act, \\
 &\quad \quad Q.st.inv \wedge P.st_1.inv \wedge P.st_2.inv, Q.pps \wedge P.pps)
 \end{aligned}$$

Considering that $\exists x, x', y, y' : T \bullet pred(x) \wedge pred(y) \Rightarrow \exists x, x' : T \bullet pred(x)$ and st_2 does not occurs in $P.pps$ nor $P.act$ the above predicate becomes:

$$\begin{aligned}
 &\quad \exists Q.st.decl \wedge P.st_1.decl \wedge Q.st.decl' \wedge P.st_1.decl' \bullet \quad (6.13) \\
 &\quad ER(Q.act \llbracket Q.st \mid Q.st \wedge P.st_1 \rrbracket P.act, Q.st.inv \wedge P.st_1.inv, Q.pps \wedge P.pps) \\
 &= [\text{By C.3}]
 \end{aligned}$$

$$\begin{aligned}
 &\quad \text{var } Q.st.decl \wedge P.st_1.decl \bullet ER(Q.act \llbracket Q.st \mid Q.st \wedge P.st_1 \rrbracket P.act, \quad (6.14) \\
 &\quad \quad Q.st.inv \wedge P.st_1.inv, Q.pps \wedge P.pps)
 \end{aligned}$$

= [By C.1]

$$P \hat{=} \left(\begin{array}{l} \mathbf{begin\ state} \hat{=} Q.st \wedge P.st_1 \\ Q.pps_1 \wedge \Xi \wedge P.st_1 \\ Q.pps_2^{ref} \wedge \Xi \wedge P.st_1 \\ P.pps \\ \bullet Q.act \llbracket Q.st \mid Q.st \wedge P.st_1 \rrbracket P.act \\ \mathbf{end} \end{array} \right)$$

The above process is the exact meaning of P, in the UTP, on the right-hand template. The meaning of R on the left-hand side template is:

$$R \hat{=} \left(\begin{array}{l} \mathbf{begin\ state} \hat{=} Q.st \wedge P.st_1 \wedge P.st_2 \wedge R.st \\ Q.pps_1 \wedge \Xi P.st_1 \wedge P.st_2 \wedge R.st \\ Q.pps_2^{ref} \wedge \Xi P.st_1 \wedge P.st_2 \wedge R.st \\ P.pps_1 \wedge \Xi R.st \\ P.pps_2^{ref} \Xi R.st \\ R.pps \\ \bullet Q.act \llbracket Q.st \mid Q.st \wedge P.st_1 \wedge P.st_2 \rrbracket P.act \\ \llbracket Q.st \wedge P.st_1 \wedge P.st_2 \mid Q.st \wedge P.st_1 \wedge P.st_2 \wedge R.st \rrbracket \\ R.act \\ \mathbf{end} \end{array} \right)$$

= [P.st₂ = R.st₂]

$$R \hat{=} \left(\begin{array}{l} \mathbf{begin\ state} \hat{=} Q.st \wedge P.st_1 \wedge R.st_2 \wedge R.st \\ Q.pps_1 \wedge \Xi P.st_1 \wedge R.st_2 \wedge R.st \\ Q.pps_2^{ref} \wedge \Xi P.st_1 \wedge R.st_2 \wedge R.st \\ P.pps_1 \wedge \Xi R.st \\ P.pps_2^{ref} \Xi R.st \\ R.pps \\ \bullet Q.act \llbracket Q.st \mid Q.st \wedge P.st_1 \wedge R.st_2 \rrbracket P.act \\ \llbracket Q.st \wedge P.st_1 \wedge R.st_2 \mid Q.st \wedge P.st_1 \wedge R.st_2 \wedge R.st \rrbracket \\ R.act \\ \mathbf{end} \end{array} \right)$$

$$= [P.pps \text{ and } P.act \text{ does not access } R.st_2]$$

$$R \hat{=} \left(\begin{array}{l} \text{begin state} \hat{=} Q.st \wedge P.st_1 \wedge R.st_2 \wedge R.st \\ Q.pps_1 \wedge \Xi P.st_1 \wedge R.st_2 \wedge R.st \\ Q.pps_2^{ref} \wedge \Xi P.st_1 \wedge R.st_2 \wedge R.st \\ P.pps_1 \wedge \Xi R.st \wedge R.st_2 \\ P.pps_2^{ref} \Xi R.st \wedge R.st_2 \\ R.pps \\ \bullet Q.act \llbracket Q.st \mid Q.st \wedge P.st_1 \rrbracket P.act \\ \llbracket Q.st \wedge P.st_1 \mid Q.st \wedge P.st_1 \wedge R.st_2 \wedge R.st \rrbracket \\ R.act \\ \text{end} \end{array} \right)$$

The above process is the exact meaning of R , in the UTP, on the right-hand template. This concludes our proof. The application of the this law in the reverse direction is similar.

□

If the main action of a process P can be written as a parallel composition of two actions act_1 and act_2 , that access exclusively st_1 and st_2 , respectively, we can move one of these actions (in this case, act_2) to a subprocess of P , say R . The state components in st_2 must be protected, so it is possible to refer to them in the R 's main action. This law changes the behavior of P , so it cannot be extended by any process in pds excepting R and its subprocesses (indirectly). Finally, P cannot be used by any of processes declared in pds , excepting by inheritance as already mentioned.

The semantics of R is unchanged as a consequence of the meaning of inheritance. The implicit main action of R , $R.\overline{act}$ is given by its declared main action $R.act$ in parallel composition with the implicit main action of P , $P.\overline{act}$, so $R.\overline{act} = P.\overline{act} \llbracket P.st \mid R.st \rrbracket R.act$. Finally, note that act_2 is restricted to change st_2 . It cannot know about $R.st$ state components, excepting those that are inherited, including st_2 .

The application of this law in the opposite direction will also change the behavior of P , so it cannot be used by any process in pds except through inheritance, directly by R , and indirectly by its subprocesses. The semantics of R and its subprocesses is unchanged since act_2 will be part of the implicit main action of R , no matter if it is in P or in R .

Law 6.11 (move action to subprocess).

$$\begin{array}{ccc}
 \begin{array}{l}
 \text{process } P \hat{=} \text{extends } Q \\
 \text{state } st_1 \wedge st_2 \\
 pps \\
 \bullet \text{ act}_1 \llbracket st_1 \mid st_2 \rrbracket \text{act}_2 \\
 \text{end} \\
 \\
 \text{process } R \hat{=} \text{extends } P \\
 \text{state } st \\
 pps \\
 \bullet \text{ act}
 \end{array}
 & \stackrel{=_{pds}}{=} &
 \begin{array}{l}
 \text{process } P \hat{=} \text{extends } Q \\
 \text{state } st_1 \wedge st_2 \\
 pps \\
 \bullet \text{ act}_1 \\
 \text{end} \\
 \\
 \text{process } R \hat{=} \text{extends } P \\
 \text{state } st \\
 pps \\
 \bullet \text{ act} \llbracket st \mid st_2 \rrbracket \text{act}_2
 \end{array}
 \end{array}$$

provided

$(\leftrightarrow) \forall S \mid S \in pds \wedge S \neq R \bullet \neg \text{occurs}(P, S)$

$(\rightarrow) PL(st_2)$

proof

The meaning of P is changed, otherwise the meaning of R is unchanged just as the meaning of the whole program, since no matter the localization of act_2 if we consider the law provisos.

The meaning of R in the left-hand side template is:

$$R \hat{=} \left(\begin{array}{l}
 \text{begin state } \hat{=} \quad Q.st \wedge P.st_1 \wedge P.st_2 \wedge R.st \\
 Q.pps_{s_1 \wedge} \Xi P.st_1 \wedge P.st_2 \wedge R.st \\
 Q.pps_{s_2}^{ref} \wedge \Xi P.st_1 \wedge P.st_2 \wedge R.st \\
 P.pps_{s_1 \wedge} \Xi R.st \\
 P.pps_{s_2}^{ref} \Xi R.st \\
 R.pps \\
 \bullet Q.act \llbracket Q.st \mid Q.st \wedge P.st_1 \wedge P.st_2 \rrbracket (P.act_1 \llbracket P.st_1 \mid P.st_2 \rrbracket P.act_2) \\
 \llbracket Q.st \wedge P.st_1 \wedge P.st_2 \mid Q.st \wedge P.st_1 \wedge P.st_2 \wedge R.st \rrbracket \\
 R.act \\
 \text{end}
 \end{array} \right)$$

$= [R.act \text{ is able to access } Q.st \wedge P.st_1 \wedge P.st_2 \wedge R.st;$
associativity of interleaving operator]

$$R \hat{=} \left(\begin{array}{l} \mathbf{begin\ state} \hat{=} Q.st \wedge P.st_1 \wedge P.st_2 \wedge R.st \\ Q.pps_1 \wedge \Xi P.st_1 \wedge P.st_2 \wedge R.st \\ Q.pps_2^{ref} \wedge \Xi P.st_1 \wedge P.st_2 \wedge R.st \\ P.pps_1 \wedge \Xi R.st \\ P.pps_2^{ref} \Xi R.st \\ R.pps \\ \bullet Q.act \llbracket Q.st \mid Q.st \wedge P.st_1 \wedge P.st_2 \rrbracket P.act_1 \\ \quad \llbracket Q.st \wedge P.st_1 \wedge P.st_2 \mid Q.st \wedge P.st_1 \wedge P.st_2 \wedge R.st \rrbracket \\ R.act \llbracket R.st \mid P.st_2 \rrbracket P.act_2 \\ \mathbf{end} \end{array} \right)$$

$= [P.act_2 = R.act_2]$

$$R \hat{=} \left(\begin{array}{l} \mathbf{begin\ state} \hat{=} Q.st \wedge P.st_1 \wedge P.st_2 \wedge R.st \\ Q.pps_1 \wedge \Xi P.st_1 \wedge P.st_2 \wedge R.st \\ Q.pps_2^{ref} \wedge \Xi P.st_1 \wedge P.st_2 \wedge R.st \\ P.pps_1 \wedge \Xi R.st \\ P.pps_2^{ref} \Xi R.st \\ R.pps \\ \bullet Q.act \llbracket Q.st \mid Q.st \wedge P.st_1 \wedge P.st_2 \rrbracket P.act_1 \\ \quad \llbracket Q.st \wedge P.st_1 \wedge P.st_2 \mid Q.st \wedge P.st_1 \wedge P.st_2 \wedge R.st \rrbracket \\ R.act \llbracket R.st \mid P.st_2 \rrbracket R.act_2 \\ \mathbf{end} \end{array} \right)$$

That is the exact meaning of R , in the UTP, on the right-hand template. This concludes our proof. The application of this law in the reverse direction is similar. $\square$

If part of the behavior of a schema in a superprocess (including a subset of the state components, related declarations and a predicate) are relevant only for one of its subprocesses, we can introduce a redefinition of this schema in the subprocess and move this part of the original schema to its redefinition.

The state components of P are partitioned in two sets st_1 and st_2 . $P.sc$, on the right-hand side, changes only st_1 , but st_2 is left undefined. $R.sc$ includes $P.sc$ and explicitly constrains the values of the st_2 components according to the predicate $pred_2$; This requires that the state components in this set have the protected access level. Finally there must be no redefinitions of $P.sc$ except in the subprocesses of R .

Law 6.12 (splitting a schema among processes).

provided

$$\begin{aligned}
 (\leftrightarrow) \quad & \forall S \mid S \leq P \wedge \neg (S \leq R) \bullet \neg occurs(st_2, S.pps) \wedge \neg occurs(st_2, S.act) \wedge \\
 & \neg impact(st_1, st_2)^1 \\
 (\rightarrow) \quad & PL(st_2) \wedge N(sc) \notin N(R.pps)
 \end{aligned}$$

proof

The meaning of P on the left-hand side is defined as:

$$P \hat{=} \left(\begin{array}{l} \mathbf{begin \ state} \hat{=} Q.st \wedge P.st_1 \wedge P.st_2 \\ Q.pps_{1 \wedge} \Xi \wedge P.st_1 \wedge P.st_2 \\ Q.pps_2^{ref} \wedge \Xi \wedge P.st_1 \wedge P.st_2 \\ P.pps \\ \mathbf{protected} \ P.sc \\ \bullet \ Q.act \llbracket Q.st \mid Q.st \wedge P.st_1 \wedge P.st_2 \rrbracket P.act \\ \mathbf{end} \end{array} \right)$$

¹ $impact(st_1, st_2)$ is true iff the value of a state component st_1 is affected by the value of st_2

$$= [\text{By C.1}]$$

$$\begin{aligned} \mathbf{var} \quad & Q.st.decl \wedge P.st_1.decl \wedge P.st_2.decl \bullet ER(Q.act \llbracket Q.st \mid Q.st \wedge P.st_1 \wedge P.st_2 \rrbracket P.act, \\ & P.act, Q.st.inv \wedge P.st_1.inv \wedge P.st_2.inv, Q.pps \wedge P.pps \wedge P.sc) \end{aligned} \quad (6.15)$$

$$= [\text{By C.3}]$$

$$\begin{aligned} \exists \quad & Q.st.decl \wedge P.st_1.decl \wedge P.st_2.decl \wedge Q.st.decl' \wedge P.st_1.decl' \wedge P.st_2.decl' \bullet ER(Q.act \llbracket Q.st \mid Q.st \wedge P.st_1 \wedge P.st_2 \rrbracket P.act, \\ & Q.st.inv \wedge P.st_1.inv \wedge P.st_2.inv, Q.pps \wedge P.pps \wedge \\ & sc \hat{=} [\odot st_1 \text{ decls}_1 \mid pred_1] \wedge [\odot st_2 \text{ decls}_2 \mid pred_2]) \end{aligned} \quad (6.16)$$

Considering that $\exists x, x', y, y' : T \bullet pred(x) \wedge pred(y) \Rightarrow \exists x, x', y, y' : T \bullet pred(x)$ and the st_1 values are independent of those of st_2 and st_2 does not occur in $P.pps$ nor in $P.act$ we can remove $[\odot st_2 \text{ decls}_2 \mid pred_2]$ from sc .

$$\begin{aligned} \exists \quad & Q.st.decl \wedge P.st_1.decl \wedge P.st_2.decl \wedge Q.st.decl' \wedge P.st_1.decl' \wedge P.st_2.decl' \bullet ER(Q.act \llbracket Q.st \mid Q.st \wedge P.st_1 \wedge P.st_2 \rrbracket P.act, \\ & Q.st.inv \wedge P.st_1.inv \wedge P.st_2.inv, Q.pps \wedge P.pps \wedge sc \hat{=} [\odot st_1 \text{ decls}_1 \mid pred_1]) \end{aligned} \quad (6.17)$$

$$= [\text{By C.3}]$$

$$\begin{aligned} \mathbf{var} \quad & Q.st.decl \wedge P.st_1.decl \wedge P.st_2.decl \bullet ER(Q.act \llbracket Q.st \mid Q.st \wedge P.st_1 \wedge P.st_2 \rrbracket P.act, \\ & Q.st.inv \wedge P.st_1.inv \wedge P.st_2.inv, Q.pps \wedge P.pps \wedge sc \hat{=} [\odot st_1 \text{ decls}_1 \mid pred_1]) \end{aligned} \quad (6.18)$$

$$= [\text{By C.1}]$$

$$P \hat{=} \left(\begin{array}{l} \mathbf{begin\ state} \hat{=} Q.st \wedge P.st_1 \wedge P.st_2 \\ \quad Q.pps_{s_1 \wedge} \Xi \wedge P.st_1 \wedge P.st_2 \\ \quad Q.pps_{s_2}^{ref} \wedge \Xi \wedge P.st_1 \wedge P.st_2 \\ \quad P.pps \\ \mathbf{protected} \ sc \hat{=} [\odot st_1 \text{ decls}_1 \mid pred_1] \\ \quad \bullet Q.act \llbracket Q.st \mid Q.st \wedge P.st_1 \wedge P.st_2 \rrbracket P.act \\ \mathbf{end} \end{array} \right)$$

The above process is the exact meaning of P , in the UTP, on the right-hand template. The meaning of R on the left-hand side template is:

$$R \triangleq \left(\begin{array}{l} \text{begin state} \triangleq Q.st \wedge P.st_1 \wedge P.st_2 \wedge R.st \\ Q.pps_1 \wedge \triangleq P.st_1 \wedge P.st_2 \wedge R.st \\ Q.pps_2^{ref} \wedge \triangleq P.st_1 \wedge P.st_2 \wedge R.st \\ P.pps_1 \wedge \triangleq R.st \\ P.pps_2^{ref} \triangleq R.st \\ \text{protected } sc \triangleq [\odot st_1 \text{ decls}_1 \mid pred_1] \wedge \\ \quad [\odot st_2 \text{ decls}_2 \mid pred_2] \\ R.pps \\ \bullet Q.act \llbracket Q.st \mid Q.st \wedge P.st_1 \wedge P.st_2 \rrbracket P.act \\ \quad \llbracket Q.st \wedge P.st_1 \wedge P.st_2 \mid Q.st \wedge P.st_1 \wedge P.st_2 \wedge R.st \rrbracket \\ R.act \\ \text{end} \end{array} \right)$$

From the semantics of **super**, the above process is equivalent to:

$$R \triangleq \left(\begin{array}{l} \text{begin state} \triangleq Q.st \wedge P.st_1 \wedge P.st_2 \wedge R.st \\ Q.pps_1 \wedge \triangleq P.st_1 \wedge P.st_2 \wedge R.st \\ Q.pps_2^{ref} \wedge \triangleq P.st_1 \wedge P.st_2 \wedge R.st \\ P.pps_1 \wedge \triangleq R.st \\ P.pps_2^{ref} \triangleq R.st \\ \text{protected } sc \triangleq [\text{super } sc \odot st_2 \text{ decls}_2 \mid pred_2] \\ R.pps \\ \bullet Q.act \llbracket Q.st \mid Q.st \wedge P.st_1 \wedge P.st_2 \rrbracket P.act \\ \quad \llbracket Q.st \wedge P.st_1 \wedge P.st_2 \mid Q.st \wedge P.st_1 \wedge P.st_2 \wedge R.st \rrbracket \\ R.act \\ \text{end} \end{array} \right)$$

The above process is the exact meaning of R , in the UTP, on the right-hand template. This concludes our proof. The application of the this law in the reverse direction is similar. $\square$

To move a schema sc from P to R , where $R < P$, it is necessary, if sc is protected, that it is not being used by P , neither by its subprocesses, excepting those that are also subprocesses of R . Note that we can apply this law, even if a subprocess of P (excepting R) has a redefinition of sc .

To bring a protected schema $R.sc$ to P , where $R < P$, we must guarantee that P neither its subprocesses, excepting those that are also subprocesses of R , have schema equals to sc .

Law 6.13 (move a protected schema to subprocess).

provided

$$(\leftrightarrow) \forall S \mid S < P \wedge \neg (S < R) \bullet \neg occurs(sc, S.pps) \wedge \neg occurs(sc, S.act)$$

proof

The meaning of P in the left-hand side is defined as:

$$P \hat{=} \left(\begin{array}{l} \textbf{begin state} \hat{=} Q.st \wedge P.st \\ Q.pps_1 \wedge \Xi \wedge P.st \\ Q.pps_2^{ref} \wedge \Xi \wedge P.st \\ P.pps \\ \textbf{protected } P.sc \\ \bullet Q.act \llbracket Q.st \mid Q.st \wedge P.st \rrbracket P.act \\ \textbf{end} \end{array} \right)$$

$$= [\text{By C.1}]$$

$$\begin{aligned} & \textbf{var } Q.st.decl \wedge P.st.decl \bullet ER(Q.act \llbracket Q.st \mid Q.st \wedge P.st \rrbracket \\ & \quad P.act, P.st.inv \wedge Q.st.inv, Q.pps \wedge P.pps \wedge P.sc) \end{aligned} \quad (6.19)$$

$$= [\text{By C.3}]$$

$$\begin{aligned} & \exists Q.st.decl \wedge P.st.decl, Q.st.decl' \wedge P.st.decl' \bullet ER(Q.act \llbracket Q.st \mid Q.st \wedge P.st \rrbracket \\ & \quad P.act, P.st.inv \wedge Q.st.inv, Q.pps \wedge P.pps \wedge P.sc) \end{aligned} \quad (6.20)$$

Considering an action A , $ER(A, inv, pps \wedge sc) = ER(A, inv, pps)$ if $\neg occurs(sc, A)$ and $\neg occurs(sc, pps)$. Considering this and the provisos we have that the semantics of P is:

$$\begin{aligned} \exists Q.st.decl \wedge P.st.decl, Q.st.decl' \wedge P.st.decl' \bullet ER(Q.act \llbracket Q.st \mid Q.st \wedge P.st \rrbracket \quad (6.21) \\ P.act, P.st.inv \wedge Q.st.inv, Q.pps \wedge P.pps) \end{aligned}$$

= [By C.3]

$$\begin{aligned} \mathbf{var} Q.st.decl \wedge P.st.decl \bullet ER(Q.act \llbracket Q.st \mid Q.st \wedge P.st \rrbracket \quad (6.22) \\ P.act, P.st.inv \wedge Q.st.inv, Q.pps \wedge P.pps) \end{aligned}$$

= [By C.1]

$$P \hat{=} \left(\begin{array}{l} \mathbf{begin\ state} \hat{=} Q.st \wedge P.st \\ Q.pps_{1\wedge} \Xi \wedge P.st \\ Q.pps_{2\wedge}^{ref} \Xi \wedge P.st \\ P.pps \\ \bullet Q.act \llbracket Q.st \mid Q.st \wedge P.st \rrbracket P.act \\ \mathbf{end} \end{array} \right)$$

This is the meaning of P , in the UTP, on the right-hand side template. The meaning of R in the left-hand side template is:

$$R \hat{=} \left(\begin{array}{l} \mathbf{begin\ state} \hat{=} Q.st \wedge P.st \wedge R.st \\ Q.pps_{1\wedge} \Xi P.st \wedge R.st \\ Q.pps_{2\wedge}^{ref} \Xi P.st \wedge R.st \\ P.pps_{1\wedge} \Xi R.st \\ P.pps_{2\wedge}^{ref} \Xi R.st \\ \mathbf{protected} P.sc \\ R.pps \\ \bullet Q.act \llbracket Q.st \mid Q.st \wedge P.st \rrbracket P.act \\ \llbracket Q.st \wedge P.st \mid Q.st \wedge P.st \wedge R.st \rrbracket \\ R.act \\ \mathbf{end} \end{array} \right)$$

$$= [P.sc = R.sc]$$

$$R \hat{=} \left(\begin{array}{l} \text{begin state} \hat{=} Q.st \wedge P.st \wedge R.st \\ \quad Q.pps_1 \wedge \Xi P.st \wedge R.st \\ \quad Q.pps_2^{ref} \wedge \Xi P.st \wedge R.st \\ \quad P.pps_1 \wedge \Xi R.st \\ \quad P.pps_2^{ref} \Xi R.st \\ \quad \text{protected } R.sc \\ \quad R.pps \\ \quad \bullet Q.act \llbracket Q.st \mid Q.st \wedge P.st \rrbracket P.act \\ \quad \llbracket Q.st \wedge P.st \mid Q.st \wedge P.st \wedge R.st \rrbracket \\ \quad R.act \\ \text{end} \end{array} \right)$$

The above process is the exact meaning of R , in the UTP, on the right-hand template. This concludes our proof. The application of the this law in the reverse direction is similar and we omit it here. $\square$

A default schema sc can be moved from P to R , where $R < P$, if sc is not being used by P and is not defined in R . As sc is default, R is not aware about it, and we must check if R does not define a schema homonymous to sc . It must be clear that the schema $R.sc$, if exists, is not a overriding of $P.sc$ since it is default.

To move the default schema $R.sc$ to P the proviso must guarantee that R doesn't use sc , since R cannot access, directly, a default schema in P . About P , it cannot has a schema equals to sc and neither of its superprocesses define a protected schema homonymous to sc , because it would create an invalid redefinition of this schema. The function PS selects the protected schemas from a set.

Law 6.14 (move a default schema to subprocess).

<pre> process $P \hat{=} \text{extends } Q$ state st sc pps $\bullet act$ end process $R \hat{=} \text{extends } P$ state st pps $\bullet act$ </pre>	$=_{pds}$	<pre> process $P \hat{=} \text{extends } Q$ state st pps $\bullet act$ end process $R \hat{=} \text{extends } P$ state st sc pps $\bullet act$ </pre>
--	-----------	--

provided

$$\begin{aligned}
 &(\rightarrow) \neg \text{occurs}(sc, P.pps) \wedge \neg \text{occurs}(sc, P.act) \wedge N(sc) \notin N(R.pps) \\
 &(\leftarrow) \neg \text{occurs}(sc, R.pps) \wedge \neg \text{occurs}(sc, R.act) \wedge \\
 &\forall T \mid P < T \bullet N(sc) \notin N(PS(T.pps))
 \end{aligned}$$

proof

This proof is similar to Law 6.13 and we omit it here. □

6.2.4 Subprocess Extraction

In the initial specification of a system it is common to model processes with a very specific behavior that hides a generic behavior specialized in face of a particular situation. Therefore we develop a composite law that extracts from a process this generic behavior as a subprocess specializing it with a subprocess. This promotes code reuse and favor a better conceptual representation of the system.

Law 6.15 (subprocess extraction).

This law is composite because its is obtained from the successive application of some of the simple/composite laws presented before.

provided

$$(\leftrightarrow) R \notin N(pds)$$

proof

As already mentioned, this law is derived from previous laws. Particularly, it can be proved from : Law 6.3 (process elimination), Law 6.11 (move action to subprocess),

Law 6.12 (splitting a schema among processes), Law 6.13 (move a protected schema to subprocess), Law 6.14 (move a default schema to subprocess) and Law 6.10 (moving state component to subprocess).

```

process  $P \hat{=}$  extends  $Q$ 
  state  $st_1 \wedge st_2$ 
   $pps_{1 \wedge} \Xi st_2$ 
   $pps_2$ 
  •  $act_1 \llbracket st_1 \mid st_1 \wedge st_2 \rrbracket act_2$ 
end

```

$=$ [By Law 6.3 and renaming]

```

process  $R \hat{=}$  extends  $Q$ 
  state  $st_1 \wedge st_2$ 
   $pps_{1 \wedge} \Xi st_2$ 
   $pps_2$ 
  •  $act_1 \llbracket st_1 \mid st_1 \wedge st_2 \rrbracket act_2$ 
end

```

```

process  $P \hat{=}$  extends  $R$ 
  •  $Skip$ 
end

```

$=$ [By Law 6.11]

```

process  $R \hat{=}$  extends  $Q$ 
  state  $st_1 \wedge st_2$ 
   $pps_{1 \wedge} \Xi st_2$ 
   $pps_2$ 
  •  $act_1$ 
end

```

```

process  $P \hat{=}$  extends  $R$ 
  •  $act_2$ 
end

```

The Law 6.12 (splitting a schema among processes) is then applied for each schema in $P.pps_2$. The set $P.pps'_2$ stands for the schemas in $P.pps_2$ affected by the law and $S.pps''_2$ for those created in S .

```

process  $R \hat{=}$  extends  $Q$ 
  state  $st_1 \wedge st_2$ 
   $pps_{1 \wedge} \Xi st_2$ 
   $pps'_2$ 
  •  $act_1$ 
end

process  $P \hat{=}$  extends  $R$ 
   $pps''_2$ 
  •  $act_2$ 
end

```

The Laws 6.13 (move a protected schema to subprocess) and 6.14 (move a default schema to subprocess) can be or not applied at this point, the application of this step is optional and for the sake of conciseness we omit it here.

= [By Law 6.10]

```

process  $R \hat{=}$  extends  $Q$ 
  state  $st_1$ 
   $pps_1$ 
   $pps'_2$ 
  •  $act_1$ 
end

process  $P \hat{=}$  extends  $R$ 
  state  $st_2$ 
   $pps''_2$ 
  •  $act_2$ 
end

```

This concludes the derivation.

□

This law can be applied in every context where we have a logical division of schemas and actions concerning state components elements. The unique proviso, in both application directions, must guarantee that R is not used in pds .

6.3 Final Considerations

In this chapter we have proposed a set of algebraic laws for *OhCircus*. These address, mainly, specifications based on process inheritance.

The laws are expressed using a metalanguage that represents process elements: state components, paragraphs (for simplicity only Z schemas) and its main action. Each law can be applied in two directions, so we define the provisos needed for each one.

The provisos are expressed as predicates over the meta-terms that must be satisfied previously to the law application. The laws can be categorized in three groups: localized eliminations, access modifications and element interchanges between processes related by inheritance.

In the first group we have: Law 6.3 (process elimination), Law 6.4 (default schema elimination), Law 6.5 (default state component elimination), Law 6.8 (**super** elimination), Law 6.9 (eliminating a trivial schema redefinition), Law 6.7 (protected schema elimination), Law 6.6 (protected state component elimination). These laws insert/remove protected/default elements of a process considering in the provisos the super/subprocesses in its hierarchy.

The second group is formed of Law 6.1 (change schema access level) and Law 6.2 (change state component access level) that only change from protected to default (or vice versa) the access modifiers of schemas and state components, taking care about the super/ subprocesses of the target process. Finally the third group is composed by laws that interchange elements between a process and one of its subprocess: Law 6.12 (splitting a schema among processes), Law 6.10 (moving state component to subprocess), Law 6.11 (move action to subprocess), Law 6.13 (move a protected schema to subprocess) and Law 6.14 (move a default schema to subprocess). The composite Law 6.15 (subprocess extraction) is out of these groups and summarises the overall transformation involved in introducing/eliminating process inheritance.

The intuition of the completeness of our laws is captured by the inverse application of our last law. With our laws it is possible to eliminate process inheritance from an arbitrary *OhCircus* process; this is our notion of relative completeness further discussed in the next chapter.

CHAPTER 7

Case Study and Completeness Notion

In this chapter we develop a case study to illustrate how the composite Law 6.15 (sub-process extraction) can be effectively applied in a specification of a practical situation.

This chapter finishes with a discussion about the completeness of our laws, specially concerning process inheritance structures.

7.1 Case Study: a centralized store

Consider a generic store that remunerates its sellers with a rate over their sales, besides a fixed salary. Rich data manipulated by the process, which represents the store behavior, are modeled as classes. The class *Seller* encompasses the sellers with their identification and salary information. Next, *Seller* is extended by the *CSeller* class, which adds a field to hold the commission obtained by the sellers that opt for this remuneration plan.

$[ID, SID, DATE, PERIOD]$

$minimumSalary : \mathbb{N}_1$
$minimumSalary = 1000$

class *Seller* $\hat{=}$ **begin**

state <i>SState</i>
protected <i>id</i> : <i>ID</i>
protected <i>salary</i> : $\mathbb{N}_1$
$salary \geq minimumSalary$

end

initial <i>SInit</i>
<i>SState'</i>
<i>id?</i> : <i>ID</i>
<i>salary?</i> : $\mathbb{N}_1$
$salary? \geq minimumSalary$
$id' = id?$
$salary' = salary?$

$rate : \mathbb{N}_1$
$rate > 0 \wedge rate \leq 100$

class *CSeller* $\hat{=}$ **extends** *Seller* **begin**

state <i>CSState</i> _____ <i>commission</i> : $\mathbb{N}$

end

initial *CSInit* $\hat{=}$

val *id* : *ID*; *salary* : $\mathbb{N}_1$;
initComm : $\mathbb{N} \bullet SInit$;
commission := *initComm*

The schema *CSInit* uses a parameterized command. It declares three inputs: *id* and *salary* are sent to the schema *SInit* and *initComm* are assigned to the state component *commission*.

The class *Sale* encompasses the information of a sale made by a store vendor. It is a 4-tuple of values: a unique sale identifier, the vendor identifier owning the commission over it, date when it happened, and its price. The **initial** clause introduces the behavior of object creation from instances of this class. This schema receives four appropriate inputs and uses them to initialize the state components of a sale.

class *Sale* $\hat{=}$ **begin**

state <i>SState</i> _____ <i>id</i> : <i>SID</i> <i>owner</i> : <i>ID</i> <i>date</i> : <i>DATE</i> <i>price</i> : $\mathbb{N}_1$
--

initial *SInit* _____

<i>SState'</i> <i>id?</i> : <i>SID</i> , <i>owner?</i> : <i>ID</i> <i>date?</i> : <i>DATE</i> , <i>price?</i> : $\mathbb{N}_1$
<i>id'</i> = <i>id?</i> $\wedge$ <i>owner'</i> = <i>owner?</i> $\wedge$ <i>date'</i> = <i>date?</i> $\wedge$ <i>price'</i> = <i>price?</i>

end

In the sequel, we present the channels used by the process, which represents the operations of a store, to interact with its environment. Note that, because *Seller* is a type it can be used as a channel data type. For example, the channel *contracted* communicates *Seller* objects between the *Store* process and its environment.

channel *id*, *dismissed* : *ID*
channel *contracted* : *Seller*
channel *sale* : *Sale*
channel *period* : *PERIOD*
channel *salary*, *updateSalary* : $\mathbb{N}_1$
channel *sum*, *com* : $\mathbb{N}$
channel *findSalary*, *calcCom*, *resetCom*

Through the channels *id* and *dismiss* the environment sends to the *Store* process the seller id to find (the result is sent via *salary* channel)/update (the new value is received via *updateSalary*) its salary or dismiss it from the store, respectively. Via the *contract* channel, the environment sends the data of a recent contracted seller to the store. *Store* receives via *sale* the data of a sale made in the store. Through the channel *sum* the environment receives the sum of the sales made in a given period, this period is sent to *Store* via *period* channel. Via the channel *com*, *Store* sends to the environment the total commission paid to its commission sellers. The channel *findSalary*, *calcCom* and *resetCom* do not communicate any data, but *Store* and its environment synchronize in these channels to start, respectively, the behaviors responsible for: to find the salary of a seller, to calc the total commission and to reset this value when appropriate.

The *Store* process manages the operation of a store. It registers sellers, sales and handles the commission calculation.

process *Store* $\hat{=}$ **begin**

The process state is composed by a set of sellers, a set of sales and the total commission paid to the commission sellers. When a seller is admitted, it must choose between two plans of remuneration: a fixed monthly salary or a less fixed amount increased by a commission over its sales. Self-confident sellers naturally will choose the latter schema. The former schema is more suitable for sellers whose household income must be stable through the time.

state *SState*

sales : $\mathbb{P}$ *Sale*

sellers : $\mathbb{P}$ *Seller*

cSellers : $\mathbb{P}$ *CSeller*

totalCommision : $\mathbb{N}$

$\forall sa : sales \bullet (\exists se : sellers \bullet se.id = sa.owner)$

cSellers = $\{s : sellers \mid s \text{ instanceof } CSeller\}$

Any *CSeller* object is also a *Seller* instance by inheritance. Therefore the set *cSellers* is contained or equals to *sellers*. The first constraint in *SState* guarantees that each value of *Sale.owner* is equal to the identification code of a seller. The latter guarantees that *cSeller* is contained in *sellers*.

The initialization schema *SInit* makes, initially, the sets of sales and sellers empty; the total commission is set to zero.

initial *SInit*

SState'

sales' = $\emptyset$

sellers' = $\emptyset$

cSellers' = $\emptyset$

totalCommision' = 0

The schema *SumSales* returns the gross profit obtained with the store sales in a given period of time. We use in this schema the *sum* function that receives a set of *Sale* objects returning the sum of their prices. The data type *PERIOD* is treated as a set of dates, this simplification makes it easy to verify if a date happens in a given period.

<i>SumSales</i>	
$\Xi SState$	
$p? : PERIOD$	
$sum! : \mathbb{N}$	
$sum! = sum \{s : sales \mid s.date \in p?\}$	

The function *sum* is defined as:

$sum : \mathbb{P} Sale \rightarrow \mathbb{N}$
$sum =$
$\lambda s : \mathbb{P} Sale \mid s \neq \emptyset \bullet$
$\lambda x : s \bullet x.price + sum(s \setminus \{x\})$

The schema *FindSalary* returns the fixed monthly payment obtained by the seller whose unique identification is equal to that informed in the schema input. Like *SumSales*, *FindSalary* does not change, just read, the *Store* state components.

<i>FindSalary</i>	
$\Xi SState$	
$id? : ID$	
$sal! : \mathbb{N}_1$	
$\exists s \in sellers \bullet s.id = id? \wedge sal! = s.salary$	

The *Lookup* schema provides a frame for operations that act over an existing seller *s*, and produces a modified seller *s'*. It identifies *s* as the seller with the identifier given as input. It also updates *sellers* by removing *s* and inserting the update *s'*. If *s* is also a *CSeller* object we must also update it in the *cSellers* set. We follow, in this schema, the same structure described in [11]. The restrictions over *s'* are defined by the schemas where the *Lookup* operation frame is used.

<i>Lookup</i>	
$\Delta SState$	
$id? : ID$	
$s, s' : Seller$	
$s \in sellers$	
$s.id = id?$	
$sellers' = (sellers \setminus \{s\}) \cup \{s'\}$	
if $s \in cSellers \wedge s.id = id?$	
then $cSellers' = (cSellers \setminus \{s\}) \cup \{s'\}$	

The schema *UpdateSalary* updates the fixed salary of a seller. It is a wrapper for *PUpdateSalary* hiding its internal *Seller* objects s and s' , introduced in *PUpdateSalary* by *Lookup*. The inputs of *PUpdateSalary* are a proper seller identifier and its new fixed salary. It is important to notice that if s is a *CSeller* object it is updated in *sellers* and *cSellers* sets.

<i>PUpdateSalary</i>	_____
$\Delta SState$	
<i>Lookup</i>	
$sal? : \mathbb{N}_1$	
$s.salary = sal?$	

$$UpdateSalary \hat{=} PUpdateSalary \setminus \{s, s'\}$$

The schema *TotalCommision* returns the value stored in the state component *totalCommision*. The schema *ResetCom* sets *totalCommision* to zero.

<i>TotalCommision</i>	_____
$\Xi SState$	
$com! : \mathbb{N}$	
$com! = totalCommision$	

<i>ResetCom</i>	_____
$\Delta SState$	
$totalCommision' = 0$	

The *InsertSale* schema inserts a *Sale* object in the *sales* set. It prevents the insertion of a sale with code already assigned to one of the pre-existent sales. A commission must be computed increasing the *totalCommision* and the seller commission when the vendor has chosen the commission remuneration plan. The schema *Lookup* is included in *InsertSale* by linking the commission seller id with *Lookup* input id ($id? = cs.id$), it allows the update of the commission of this vendor ($s'.commission = s.commission + com$, where s and s' are declared in the *Lookup* schema).

<i>InsertSale</i>
$\Delta SState$
$s? : Sale$
$\neg \exists s : sales \bullet s.id = s?.id$ $\forall s : sales \bullet s.date \leq s?.date$ $sales' = sales \cup \{s?\}$ if $\exists cs \in cSellers \mid cs.id = s?.owner$ then let $com == (s?.price * rate) \text{ div } 100 \bullet$ $totalCommision' = totalCommision + com \wedge$ $Lookup[id? = cs.id] \wedge$ $s'.commission = s.commission + com$

The *Contract* schema adds a new *Seller* object to *sellers*, if it is also a *CSeller* instance a parallel addition must be made in *cSellers*. The same reasoning applies to the removal of a seller in the *Dismiss* schema.

<i>Contract</i>
$\Delta SState$
$s? : Seller$
$\neg \exists s \in sellers \bullet s.id = s?.id$ $sellers' = sellers \cup \{s?\}$ if $s? \text{ instanceof } CSeller$ then $cSellers' = cSellers \cup \{s?\}$

<i>Dismiss</i>
$\Delta SState$
$id? : ID$
$\exists s \in sellers \mid s.id = id? \bullet$ $sellers' = sellers \setminus \{s\} \wedge$ $cSellers' = cSellers \setminus \{s\}$

After initialization, the *Store*'s main action recursively offers insertion of a new sale (*InsertSale*); admission/dismissal of a seller (*Contract/Dismiss*); query/update of the fixed payment received by a seller (*FindSalary/UpdateSalary*) and query the gross profit from the sales in a period (*SumSales*). These partial behaviors are combined as an external choice and this composition is put in parallel with the external choice of: query/reset the total commission paid to the commissioned sellers (*TotalCommision/ResetCom*).

```

• SInit;
   $\mu X \bullet ($ 
    (sale?s  $\rightarrow$  InsertSale
     $\square$  contracted?s  $\rightarrow$  Contract
     $\square$  dismissed?id  $\rightarrow$  Dismiss
     $\square$  id?id  $\rightarrow$ 
      findSalary  $\rightarrow$  var sal :  $\mathbb{N}_1 \bullet$  FindSalary; salary!sal  $\rightarrow$  Skip
     $\square$ 
      updateSalary?sal  $\rightarrow$  UpdateSalary
     $\square$  period?p  $\rightarrow$  var sum :  $\mathbb{N} \bullet$  SumSales; sum!sum  $\rightarrow$  Skip)
     $\llbracket \{sales, sellers, cSellers\} \mid \{totalCommission\} \rrbracket$ 
    (calcCom  $\rightarrow$  var com :  $\mathbb{N} \bullet$  TotalCommision; com!com  $\rightarrow$  Skip
     $\square$  resetCom  $\rightarrow$  ResetCom)
  ); X
end

```

7.2 Store Adaptation before Subprocess Extraction

The initial version of *Store* is so coupled that we cannot apply directly our composite law to extract the generic and specific behaviors from it. We need first to rewrite *Store* to allow a precise matching with the left-hand side of the law. It is not our intent to prove the possible equivalence between the two versions of the *Store* process. This is not a condition for the correctness of our case study.

process *Store* $\hat{=}$ **begin**

Actually the new *Store* state is formed of the composition of two schemas: *SState* and *CState*. The former will be the state schema of the future superprocess *SStore*, which will be generated by the law. The state components in *SState* are qualified as **protected** in order to be viewed by the future subprocess *CStore* whose state schema will be *CState*.

<i>SState</i>	
protected <i>sales</i> : $\mathbb{P} Sale$	
protected <i>sellers</i> : $\mathbb{P} Seller$	
$\forall sa : sales \bullet (\exists se : sellers \bullet se.id = sa.owner)$	
<i>CState</i>	
<i>cSellers</i> : $\mathbb{P} CSeller$	
<i>totalCommision</i> : $\mathbb{N}$	
$cSellers = \{s : sellers \mid s \text{ instanceof } CSeller\}$	

state $SState \wedge CState$

The initializer accomplishes the new arrangement of the *Store*'s state. It is composed by two schemas: *SInit* and *CInit*. These initialize respectively the state components in *SState* and *CState*.

<i>SInit</i>	
<i>SState'</i>	
$sales = \emptyset$	
$sellers = \emptyset$	

<i>CInit</i>	
<i>CState'</i>	
$cSellers' = \emptyset$	
$totalCommision' = 0$	

initial $SInit \wedge CInit$

Some of the Z schemas in *Store* have a mix of generic and specific behaviors. Each of them generates two schemas: a protected one in *SStore* extended with a specific behavior in *CStore*. We have four schemas in this situation: *Lookup*, *InsertSale*, *Contract*, *Dismiss*. They must be marked as **protected** to allow overloading in *CStore*.

protected <i>Lookup</i>	
$\Delta SState \wedge CState$	
$id? : ID$	
$s, s' : Seller$	
$s \in sellers$	
$s.id = id?$	
$sellers' = (sellers \setminus \{s\}) \cup \{s'\}$	
if $s \in cSellers \wedge s.id = id?$	
then $cSellers' = (cSellers \setminus \{s\}) \cup \{s'\}$	

protected <i>InsertSale</i> _____ $\Delta SState \wedge CState$ $s? : Sale$
$\neg \exists s : sales \bullet s.id = s?.id$ $\forall s : sales \bullet s.date \leq s?.date$ $sales' = sales \cup \{s?\}$ if $\exists cs \in cSellers \mid cs.id = s?.owner$ then let $com == (s?.price * rate) \text{ div } 100 \bullet$ $totalCommision' = totalCommision + com \wedge$ $Lookup[id? = cs.id] \wedge$ $s'.commission = s.commission + com$

protected <i>Contract</i> _____ $\Delta SState \wedge CState$ $s? : Seller$
$\neg \exists s \in sellers \bullet s.id = s?.id$ $sellers' = sellers \cup \{s?\}$ if $s? \text{ instanceof } CSeller$ then $cSellers' = cSellers \cup \{s?\}$

protected <i>Dismiss</i> _____ $\Delta SState \wedge CState$ $id? : ID$
$\exists s \in sellers \mid s.id = id? \bullet$ $sellers' = sellers \setminus \{s\} \wedge$ $cSellers' = cSellers \setminus \{s\}$

Some schemas in *Store* read/update only state components in *SState*, and will be placed in *SStore*. Others are exclusively related to the state components in *CState* and will be moved to the *CStore* superprocess. We do not change either of these schemas. Those related to *SState* are:

<i>SumSales</i> _____ $\Xi SState$ $p? : PERIOD$ $sum! : \mathbb{N}$
$sum! = sum \{s : sales \mid s.date \in p?\}$

<i>FindSalary</i>	_____
$\Xi SState$	
$id? : ID$	
$sal! : \mathbb{N}_1$	
$\exists s \in sellers \bullet s.id = id? \wedge sal! = s.salary$	

Those related exclusively to the state components in *CState* are:

<i>TotalCommision</i>	_____
$\Xi CState$	
$com! : \mathbb{N}$	
$com! = totalCommision$	

<i>ResetCom</i>	_____
$\Delta CState$	
$totalCommision' = 0$	

However, notice that we need to consider the schemas *PUpdateSalary* and *UpdateSalary*. We have not categorized these schemas yet. Apparently they are related only with *SState*, but since *PUpdateSalary* includes the schema *Lookup*, consequently, it changes *CState*. As *UpdateSalary* is just a wrapper for *PUpdateSalary* the same reasoning applies. These schemas will be **implicitly** overloaded, in semantic terms, by the inheritance meaning. The overloaded *Lookup* schema in *CStore* will be used in *P/UpdateSalary* schemas when they have been referenced in *CStore*'s main action. This happens because, by the inheritance semantics, *SStore*'s main action is put in parallel composition with that of *CStore*.

<i>PUpdateSalary</i>	_____
$\Delta SState \wedge CState$	
<i>Lookup</i>	
$sal? : \mathbb{N}_1$	
$s.salary = sal?$	

$$UpdateSalary \hat{=} PUpdateSalary \setminus \{s, s'\}$$

We finalize by structuring the main action of *Store* as a parallel composition of two actions *SAct* and *CAct*. *SAct* in *SStore* will view/update only elements in *SState*. In *CStore* the generic schemas will be overridden and *SAct* will have the same behavior presented below. It is important to notice that syntactically there is no *SAct* in *CStore*. Finally, the *CAct* is related only with *CState* and will be moved to *CStore*.

$$\begin{aligned}
SAct &\hat{=} SInit; \\
&\mu X \bullet (\\
&\quad sale?s \rightarrow InsertSale \\
&\quad \square contracted?s \rightarrow Contract \\
&\quad \square dismissed?id \rightarrow Dismiss \\
&\quad \square id?id \rightarrow \\
&\quad \quad findSalary \rightarrow \mathbf{var} sal : \mathbb{N}_1 \bullet FindSalary; salary!sal \rightarrow Skip \\
&\quad \square \\
&\quad \quad updateSalary?sal \rightarrow UpdateSalary \\
&\quad \square period?p \rightarrow \mathbf{var} sum : \mathbb{N} \bullet SumSales; sum!sum \rightarrow Skip \\
&); X
\end{aligned}$$

$$\begin{aligned}
CAct &\hat{=} CInit; \\
&\mu X \bullet (\\
&\quad calcCom \rightarrow \mathbf{var} com : \mathbb{N} \bullet TotalCommision; com!com \rightarrow Skip \\
&\quad \square resetCom \rightarrow ResetCom) \\
&); X
\end{aligned}$$

$$\begin{aligned}
&\bullet SAct \quad \llbracket \{sales, sellers, cSellers\} \mid \{totalCommission\} \rrbracket CAct \\
&\mathbf{end}
\end{aligned}$$

The next section presents the resulting of the Law 6.15 (subprocess extraction) application to this last version of *Store*.

7.3 Improvement in *Store* Design

At this point it is clear that *Store* encompasses two abstractions: a regular store with a simple remuneration plan that is extended by a more elaborate payment schema for its sellers. Law 6.15 (subprocess extraction) not only separates these abstractions, but creates an inheritance relation between them. We present in the sequel the process *SStore*, a simple, conservative store, where the sellers receive a monthly fixed salary.

process *SStore* $\hat{=}$ **begin**

The state of *SStore* registers only sales and sellers. We can still put *CSeller* objects in the seller set *sellers*, but all schemas in *SStore* and its main action do not mention the *CSeller* class.

$ \begin{aligned} &\mathbf{state} \ SState \\ &\quad \mathbf{protected} \ sales : \mathbb{P} \ Sale \\ &\quad \mathbf{protected} \ sellers : \mathbb{P} \ Seller \\ &\quad \forall sa : sales \bullet (\exists se : sellers \bullet se.id = sa.owner) \end{aligned} $

initial <i>SInit</i>	_____
$SState'$	
$sales = \emptyset$	
$sellers = \emptyset$	

As *CState* actually belongs to the *CStore* subprocess the clause $\Delta SState \wedge CState$ is updated to $\Delta SState$.

protected <i>Lookup</i>	_____
$\Delta SState$	
$id? : ID$	
$s, s' : Seller$	
$s \in sellers$	
$s.id = id?$	
$sellers' = (sellers \setminus \{s\}) \cup \{s'\}$	

protected <i>InsertSale</i>	_____
$\Delta SState$	
$s? : Sale$	
$\neg \exists s : sales \bullet s.id = s?.id$	
$\forall s : sales \bullet s.date \leq s?.date$	
$sales' = sales \cup \{s?\}$	

protected <i>Contract</i>	_____
$\Delta SState$	
$s? : Seller$	
$\neg \exists s \in sellers \bullet s.id = s?.id$	
$sellers' = sellers \cup \{s?\}$	

protected <i>Dismiss</i>	_____
$\Delta SState$	
$id? : ID$	
$\exists s \in sellers \mid s.id = id? \bullet$	
$sellers' = sellers \setminus \{s\} \wedge$	

The two schemas below are kept unchanged from *Store*.

<i>SumSales</i>	_____
$\exists SState$	
$p? : PERIOD$	
$sum! : \mathbb{N}$	
$sum! = sum \{s : sales \mid s.date \in p?\}$	

<i>FindSalary</i>	_____
$\exists SState$	
$id? : ID$	
$sal! : \mathbb{N}_1$	
$\exists s \in sellers \bullet s.id = id? \wedge sal! = s.salary$	

Though the schemas *PUpdateSalary* and *UpdateSalary* have been brought from *Store* unchanged, they present now a different behavior, since *Lookup* is overridden here.

<i>PUpdateSalary</i>	_____
$\Delta SState \wedge CState$	
<i>Lookup</i>	
$sal? : \mathbb{N}_1$	
$s.salary = sal?$	

$$UpdateSalary \hat{=} PUpdateSalary \setminus \{s, s'\}$$

As the schemas used in *Store.SAct* are changed in *SStore*, its main action, though syntactically equal to *Store.SAct* has a different meaning, since the schemas *Lookup*, *InsertSale*, *Contract* and *Dismiss* have changed.

```

• SInit;
  μ X • (
    sale?s → InsertSale
    □ contracted?s → Contract
    □ dismissed?id → Dismiss
    □ id?id →
      findSalary → var sal : N1 • FindSalary; salary!sal → Skip
    □
    updateSalary?sal → UpdateSalary
    □ period?p → var sum : N • SumSales; sum!sum → Skip
  ); X
end

```

The process *SStore* offers a conceptual generic basis for a store that decides to adopt a more audacious remuneration plan for its sellers, extending this generic frame behavior. *CStore* is a conservative extension of *SStore*, since it continues offering the simplified remuneration plan. It is up to the seller to be a *Seller* or *CSeller* employee.

process *CStore* $\hat{=}$ **extends** *SStore* **begin**

The state components of *CStore* is formed of those declared in *CState* in addition to those declared, in the protected level, in *SStore*. As a consequence the predicate in *CState* refers to *sellers*, which is a protected state component of *SStore*.

state <i>CState</i>
<i>cSellers</i> : $\mathbb{P} \text{ } CSeller$ <i>totalCommision</i> : $\mathbb{N}$
<i>cSellers</i> = $\{s : \textit{sellers} \mid s \text{ \textbf{instanceof} } CSeller\}$

The initializer of *CStore* is defined uniquely by *CInit* schema. Note that *SInit* is unknown here, since it is a default component of *SStore*.

initial <i>CInit</i>
<i>CState'</i>
<i>cSellers'</i> = $\emptyset$ <i>totalCommision'</i> = 0

The schemas *Lookup*, *InsertSale*, *Contract* and *Dismiss* override their correspondent versions in *SStore*. The typing rules, presented in Chapter 4, impose the continuity of the protected access level for its schemas. The **super** clause is used to include the counterpart schemas in *SStore*. As $\Delta SState$ comes from the superprocess schemas we just write $\Delta CState$ in the following schemas.

protected <i>Lookup</i>
$\Delta CState$ super <i>Lookup</i>
if $s \in cSellers \wedge s.id = id?$ then $cSellers' = (cSellers \setminus \{s\}) \cup \{s'\}$

protected <i>InsertSale</i>	_____
$\Delta CState$	
super <i>InsertSale</i>	
if $\exists cs \in cSellers \mid cs.id = s?.owner$ then let $com == (s?.price * rate) \div 100$ • $totalCommision' = totalCommision + com \wedge$ $Lookup[id? = cs.id] \wedge$ $s'.commission = s.commission + com$	

protected <i>Contract</i>	_____
$\Delta CState$	
super <i>Contract</i>	
if $s? \text{ instanceof } CSeller$ then $cSellers' = cSellers \cup \{s?\}$	

protected <i>Dismiss</i>	_____
$\Delta CState$	
super <i>Dismiss</i>	
$cSellers' = cSellers \setminus \{s\}$	

The schemas *SumSales* and *FindSalary* are not inherited by *CStore*. Therefore when one of them is referenced in its main action a definition for it comes from *SStore*. *CStore.act* does not refer to these schemas, that are not even inherited, but it is said that *CStore.act* ‘uses’ these schemas. In fact, syntactically, this process does not use these schemas, but its implicitly main action is $CStore.\overline{act} = CStore.act \llbracket SState \mid SState \wedge CState \rrbracket SStore.\overline{act}$, and *SStore.act* uses *SumSales* and *FindSalary*.

Those schemas related exclusively to the state components in *CState* are brought unchanged from *Store*.

<i>TotalCommision</i>	_____
$\Xi CState$	
$com! : \mathbb{N}$	
$com! = totalCommision$	

<i>ResetCom</i>	_____
$\Delta CState$	
$totalCommision' = 0$	

The schemas *PUpdateSalary* and *UpdateSalary* are not inherited, but their meaning in *SStore.act* when put in parallel with *CStore.act* is changed, since we redefine the schema *Lookup* in *CStore*, that is used by the schemas *UpdateSalary* and *PUpdateSalary*.

The behavior of *CStore* is given by its main action, presented in the sequel, put in parallel with that of *SStore*. The definitions of schemas referenced in *SStore.act* must be searched first in *CStore*; if it is not found, the search continues in *SStore*. If a schema is defined only in *SStore* but references another, the search for its definition must start in *CStore*, following to *SStore*, in the occasion where it is not found in the former.

```

• CInit;
  μ X • (
    calcCom → var com : ℕ • TotalCommision; com!com
    □ resetCom → ResetCom)
  ); X
end

```

From the semantics of process inheritance presented in the section 5.1, it follows directly that *CStore* has the same behavior as the original version of *Store*.

$$Store \hat{=} CStore$$

7.4 Completeness

The algebraic laws developed in this work focus on the improvement of the process inheritance relations in an *OhCircus* specification. They address state components, schemas and main actions of processes engaged in a hierarchy. Our laws are not concerned with transformations for CSP actions or classes; the works [39] and [6] have already dealt with these topics. Although our work can be part of a completeness theory for *OhCircus*, it is not our intent to establish it, so we just present how process inheritance relations can be removed from specification.

An important issue is a notion of completeness for the proposed set of laws, particularly with respect to inheritance.

Our measure for the completeness of the proposed laws is whether their exhaustive application is capable to remove all subprocess from the target specification; this is exactly what our laws provide if guided by a strategy. In a reduction strategy we apply the laws in a opposite than that applied in the development phase.

Our strategy for subprocess removal can be summarized as the application, in opposite direction, of the laws used in our composite Law 6.15 (subprocess extraction). Therefore, in an high level view, we must apply the laws: Law 6.10 (moving state component to subprocess), Law 6.14 (move a default schema to subprocess), Law 6.13 (move a protected schema to subprocess), Law 6.12 (splitting a schema among processes), Law 6.11 (move action to subprocess) and Law 6.3 (process elimination) in this order.

Some other simple laws might be necessary in the reduction process. For example, we change the visibility of all schemas and state components of the subprocess S , which we want to move to P (S 's superprocess), to **protected**. This transformation might generate name conflicts that can be solved with a simple renaming. After this, Law 6.10 (moving state component to subprocess) can be successively applied to move up each state component from S to P .

7.5 Final Considerations

In this chapter we have developed a case study in *OhCircus* to illustrate the extraction of process inheritance. We start with a process that models the behavior of a store with two remuneration plans for its sellers. It becomes evident that a plan is a specialization of the other. This is exactly the situation for what we have created our composite Law 6.15 (subprocess extraction).

It is rare to have a specification that is suitable for a direct refinement application. One usually needs to rewrite the specification for it to match the left-hand side of the law, as we needed to do in the case of the store process. After application of this law we obtain two behaviors that hold a specialization relation between them.

Finally we discussed the comprehensiveness of our laws. We define informally that a *normal form* must not include process inheritance relations and that our laws can be extensively applied to achieve this goal.

The next chapter summarizes all contributions of this work and its relations with other works. Future work are also emphasized.

CHAPTER 8

Conclusion

In this chapter we present the main contributions of our work. Furthermore, we discuss the related work involving refinements, algebraic laws and process inheritance. We conclude with the challenges that need to be tackled in future work.

8.1 Contributions

Concurrent and parallel systems are growing in number and complexity. Nevertheless the techniques to develop and maintain such systems are insufficient or unreliable in production environment. In order to support a rigorous development and maintenance process, formal specification languages, techniques and tools have been proposed. *OhCircus*, a combination of Z[43], CSP[23] and object-orientated paradigm, is a formal specification language whose semantics is based in the Unifying Theories of Programming [24]. *OhCircus* supports the construction of reactive processes in CSP style, with rich data representation in Z and object-oriented constructions. A process, in *OhCircus*, is a reactive component with its own control flow.

The main purpose of this work is the definition of sound algebraic laws to address *OhCircus* specifications with process inheritance. With this goal in mind we started defining what is process inheritance in *OhCircus*. Extending the model of process inheritance [47] for CSP, based on the failures model [23], we construct the semantics for process inheritance in *OhCircus*. To accomplish this we had to deal with Z and CSP refinement.

The original version of *OhCircus* makes process components invisible even for its subprocesses, which prevents code reuse. This forced us to change the syntax and semantics of *OhCircus* through the creation of a new access level to signalize the superprocess elements which will be visible (via **copy** semantics) to its subprocesses.

The new syntax for *OhCircus* demands the definition of new typing rules to validate a proper specification in its current version. Initially, we define a type environment (following the same principles for class inheritance in ROOL [12]) which registers the typing information of the declared processes, encompassing both its declared and inherited elements. Our typing rules, predicates over the typing environment under certain premisses, make the validation considering the process position in its inheritance tree, and whose elements are inherited and/or overridden. This contribution is a complement for the typing system developed for *Circus* in [52].

Another significant contribution of our work is a UTP semantics for process inheritance and its auxiliary constructs (**super**, **protected** clauses). To formalize inheritance, we also needed to introduce and define a UTP semantics for a new parallel operator, which allows the argument processes to share state elements, provided they agree on the final values of such elements. We have also defined a proper syntax for *OhCircus*, which

encourages code reuse. Then we move our efforts, in Chapter 6, to defining algebraic laws for process inheritance. They are a compatible extension¹ to those which have been defined for the original version of *OhCircus* [10, 35, 9, 41]. Finally our main contribution is proving the soundness of our proposed algebraic, such laws are classified as simple, when they are proved directly from the UTP semantics, and as composite if they are derived from other laws.

We conclude this work with a case study where we apply some of our simple and composite laws. We start defining a monolithic process encompassing two abstractions. After rewriting this process to merge with our laws, we extract two processes, related by inheritance, where a generic behavior is specialized by a specific one. Based on this we formulate an informal notion of completeness that can be formulated using our laws in combination with those already defined for *OhCircus* [10, 35, 9, 41].

In summary, in this work we:

- define the semantic of a new parallel operator as well as visibility controls mechanisms and **super**;
- define a semantics for process inheritance in *OhCircus*, based on [47];
- extend the grammar and semantics of *OhCircus* to allow code reuse;
- define a typing environment and typing rules for *OhCircus* based on [12, 52];
- propose a set of algebraic laws addressing process inheritance together with their soundness proofs;
- present the laws application in a real and complex situation;
- give an informal notion of completeness.

8.2 Related Work

Several work have addressed the notion of behavioral subtyping [37, 3, 28, 14, 8, 7, 4, 47]. In [3, 28] a subtype relation is defined in terms of invariants over a state plus pre/post conditions and constraint rules over methods. The other cited works define a subtype relation based on some process algebra model, like failures and failures-divergences proposed for CSP, relating process refinement with inheritance.

In [47] the failures model of CSP is used to define four types of inheritance concerning the adherence to the substitutability principle: the subtype exposes, for the environment, the same behavior exhibited by its supertype. Each subtype relation considers scenarios where the environment capabilities for detecting the differences between types increases from the weaker (weak) to the stronger (optimal) subtype relation.

In [28] a subtype is allowed to extend the behavior of its supertype adding new methods, provided there exists a function that maps these new methods as a combination

¹We extend *OhCircus* conservatively, since programs and refinement laws valid in the previous version are also valid here.

of the supertype methods; this is not allowed in [47]. Here we allow, in a subtype, new methods like in [28] and even more: new state components, method overriding, reducing the non-determinism, and methods that change both inherited and declared state components².

A consequence of the appropriate use of inheritance is code reuse [2]. As the initial version of *OhCircus* does not support inheritance in a manner that promotes code reuse, we extend its original grammar[11]. Still in [11] a process which schedules tasks is extended by a subprocess that deals with priority tasks. Process inheritance is reduced to a data refinement: the collection class (superprocess state component) that operates over a set of simple tasks is replaced by a collection that handles priority tasks (in the subprocess). There is no reuse of schemas nor state components. Through examples presented in this work it becomes evident that process inheritance is just a wrapper for interleaving.

Code reuse implies in the possibility for a subprocess to reuse elements from its superprocesses. We extend *OhCircus* creating a new access level, **protected**, similar to that found in Java[43]. Semantically there is no difference between a protected element declared or inherited, which is not the case of Java. We follow ROOL [12] concerning the protected access level.

Our typing rules formulated for *OhCircus* are a conservative extension of those proposed in [48], which defines and implements the type system of *Circus*, a combination of CSP and Z, but without considering process inheritance. A typing rule in *Circus* is a predicate, considering a possible empty set of premisses, over the typing environment, which registers the typing information of the declared processes. We first extend the typing environment of [48] to register, for a process, not only its declared but also its inherited elements. We have taken as basis the definition of the typing environment considering inheritance in the object oriented language ROOL[12]. With inheritance information properly registered we defined our typing rules, which is an important part of a formal definition of the semantics of the new *OhCircus* constructs.

Many works have proposed refinements and algebraic laws for *Circus* [39, 9, 10] and these are consequently applicable to *OhCircus*. In [39] the meaning of refinement of processes and their actions are defined based on forward simulation [22]. It also proposes laws in the process grain, as splitting and indexing processes, as a part of a general method of development in *Circus*, based on refinement. A complementary work is developed in [9] focusing on refinement laws for actions, which gives the meaning of processes. The work in [10] unifies the results in [39] and [9], and provides an accompanying iterative development strategy, involving the application of simulation, action and, most importantly, process refinement. It complements [39] providing a notion of backwards simulation [21] in *Circus*. None of these works propose refinements considering process inheritance. *OhCircus* is introduced in [11], which presents the grammar (extended in this work) and the notion of process inheritance explored through a simple data refinement, as already discussed. We are not aware of any work that proposes refinements considering process inheritance. This work is a contribution in this direction, by considering rich data in process state modeling

²As we are using *OhCircus*, a language that combines CSP, Z and object orientation, we cannot use, strictly, the terms subtype, as a process is not a type, nor methods (processes have an equivalent term called schema). Despite that we use interchangeably these terms.

and access control that allow state and schemas (consequently schema overriding) reuse.

Finally we emphasize that in the classic work about behavior subtyping [28] new methods in a subtype must be explained through the supertype methods. In our work we apply the same restriction only to the schema part that concerns superprocess state components. The extra subprocess state components are changed freely by the new schemas; it is completely necessary the independence of the superprocess state components from the subprocess ones.

8.3 Future Works

The mechanization of the formal semantics of *Circus* given in the UTP, is provided in [34], with some corrections concerning particular specifications developed by [53]. This mechanization is carried out in ProofPower-Z [29], a higher-order tactic-based theorem prover. It provides syntax and type checking, schema expansion, precondition calculation, domain checking, and general theorem proving. The extension of this work for *OhCircus*, in the form proposed here, is our next immediate goal.

In this context we plan:

- to encode the new constructions of *OhCircus* in ProofPower-Z and then perform a semi-mechanized proof of our algebraic laws and new ones concerning actions and Z schema refinements in the presence of inheritance;
- provide the complete unification of the theory that relates classes and processes initiated by [11]
- study the applicability of abstract class and interface concepts in the context of *OhCircus*, analysing whether abstract or interface processes would be useful, and defining their semantics and related laws
- construct an integrated environment for *OhCircus*, supporting both existing laws and those defined in this work.

One possibility, considering the latter goal, is to build such an environment as a plugin to Eclipse, based on the Model Driven Development paradigm. This would entail defining a metamodel for *OhCircus* and implementing the laws using tools like QVT [5] or ATL [27]. Another possibility is the extension of CRefine [36], a tool that can be used throughout the refinement of concurrent systems in a calculational approach, with our extended version of *OhCircus* and the laws developed in this work.

APPENDIX A

OhCircus original syntax

Program	::=	OhCircusParagraph*	
OhCircusParagraph	::=	Paragraph ChannelDefinition ChanSetDefinition OhProcessDefinition ClassDefinition	
OhProcessDefinition	::=	process N $\hat{=}$ [extends N] Process	
Process	::=	begin PParagraph* [state Schema-Exp Constraint] PParagraph* • Action end N Process; Process Process □ Process Process □ Process Process [[CSEExpression]] Process Process Process Process \ CSEExpression	
OhExpression	::=	Expression this null new N[(OhExpression ⁺)] <i>constructor</i> OhExpression.N <i>attribute selection</i> OhExpression.N(OhExpression*) <i>method call</i> super .N <i>attribute selection</i> super .N(OhExpression*) <i>super call</i> OhExpression instanceof N <i>type test</i> (N)OhExpression <i>type cast</i>	
ClassDefinition	::=	class N $\hat{=}$ [extends N] begin CParagraph* [state StateSchema Constraint] CParagraph* [initial Schema-Exp] CParagraph* end	

CParagraph	::=	Paragraph Qualifier N $\hat{=}$ ParametrisedCommand
Qualifier	::=	public protected private logical
ParametrisedCommand	::=	Schema-Exp Command ParameterDeclaration • Command
ParameterDeclaration	::=	ParameterQualifier Declaration ParameterQualifier Declaration; ParameterDeclaration
ParameterQualifier	::=	val res
Command	::=	N ⁺ : [Predicate, Predicate] N ⁺ := Expression ⁺ OhExpression.N(OhExpression*) super .N(OhExpression*) Command; Command μ N • Command if GuardedCommands fi var Declaration • Command
GuardedCommands	::=	Predicate $\rightarrow$ Command Predicate $\rightarrow$ Command $\square$ GuardedCommands

APPENDIX B

OhCircus extended syntax

```

Program          ::=    OhCircusParagraph*

OhCircusParagraph ::=    Paragraph
                        |    ChannelDefinition | ChanSetDefinition
                        |    OhProcessDefinition | ClassDefinition

OhProcessDefinition ::=  process N  $\hat{=}$  [extends N] Process

Process ::= begin
           PParagraph*
           [state N Schema-Exp | Constraint]
           PParagraph*
           • Action
           end
           | N | Process; Process
           | Process □ Process | Process □ Process
           | Process [[ CSEExpression ]] Process
           | Process ||| Process | Process \ CSEExpression

PParagraph ::= SchemaText | N  $\hat{=}$  Action
           | [PQualifier] N SchemaText

SchemaText ::= (⊙ N)+ [Declaration+] [super.N+] [Predicate]

Schema-Exp ::= ([PQualifier] Declaration)*

Declaration ::= N : Expression
             | N == Expression
             | Expression

PQualifier ::= protected

OhExpression ::= Expression
             | this | null
             | new N[(OhExpression+)] constructor
             | OhExpression.N attribute selection
             | OhExpression.N(OhExpression*) method call
             | super.N attribute selection
             | super.N(OhExpression*) super call
             | OhExpression instanceof N type test
             | (N)OhExpression type cast

```

ClassDefinition ::= **class** N $\hat{=}$ [extends N] **begin**
 CParagraph*
 [state StateSchema | Constraint] CParagraph*
 [initial Schema-Exp] CParagraph*
 end

CParagraph ::= Paragraph | Qualifier N $\hat{=}$ ParametrisedCommand

Qualifier ::= **public** | **protected** | **private** | **logical**

ParametrisedCommand ::= Schema-Exp | Command
 | ParameterDeclaration • Command

ParameterDeclaration ::= ParameterQualifier Declaration
 | ParameterQualifier Declaration; ParameterDeclaration

ParameterQualifier ::= **val** | **res**

Command ::= $N^+ : [\text{Predicate}, \text{Predicate}]$ | $N^+ := \text{Expression}^+$
 | OhExpression.N(OhExpression*) | **super**.N(OhExpression*)
 | Command; Command | μ N • Command
 | **if** GuardedCommands **fi** | **var** Declaration • Command

GuardedCommands ::= Predicate $\rightarrow$ Command
 | Predicate $\rightarrow$ Command $\square$ GuardedCommands

APPENDIX C

UTP framework

Semantics of processes, schemas and commands developed in [34].

C.1 OhCircus Processes

Derived from [34], since the action in the command is submitted to the ER function.

$$\mathbf{begin\ state}\ [decl \mid pred]PPars \bullet A \mathbf{end} \hat{=} \mathbf{var}\ decl \bullet ER(A, pred, PParas)$$

$$\begin{aligned} ER &: \text{Action} \rightarrow \mathbb{P} \text{Constraint} \rightarrow \mathbb{P} \text{PParagraph} \rightarrow \text{Action} \\ ER\ a\ \emptyset\ \emptyset &= a \\ ER\ a\ \emptyset\ p &= R(a, p) \\ ER\ a\ inv\ \emptyset &= E(a, inv) \\ ER\ a\ i : invs\ p : pps &= ER(E(R(a, E(p, i)), i), invs, pps) \end{aligned}$$

$$\begin{aligned} E &: \text{Action} \rightarrow \text{Constraint} \rightarrow \text{Action} \\ E\ a\ inv &= a; \mathbf{if}\ inv \rightarrow \text{Skip} \parallel \text{Stop} \\ E\ (a\ op\ b)\ inv &= E(a, inv)\ op\ E(b, inv) \end{aligned}$$

$$\begin{aligned} E &: \text{Paragraph} \rightarrow \text{Constraint} \rightarrow \text{Paragraph} \\ E\ [udecl; ddecl' \mid pred]\ inv &= [udecl; ddecl' \mid pred \wedge inv] \end{aligned}$$

$$\begin{aligned} R &: \text{Action} \rightarrow \text{Paragraph} \rightarrow \text{Action} \\ R\ a\ sc &\hat{=} [udecl; ddecl' \mid pred] = \mathbf{if}\ a = sc \mathbf{then}\ [udecl; ddecl' \mid pred] \mathbf{else}\ a \\ R\ (a\ op\ b)\ sc &= R(a, sc)\ op\ R(b, sc) \end{aligned}$$

C.2 Schema Expressions

$$[udecl; ddecl' \mid pred] \hat{=} ddecl : [\exists ddecl' \bullet pred, pred]$$

C.3 Command

$$\mathbf{var}\ x : T \bullet A = \exists\ x; x' : T \bullet A$$

APPENDIX D

Access Levels Equivalence

Some laws change the access level of schemas or state components, the proofs of these laws use the bellow lemmas based on the similar lemmas for ROOL[13]:

Lemma D.1 (access levels equivalence between schema in semantics level).

Consider Γ and Γ' proper typing environments, where $\Gamma.vis\ P\ sc = default$ and $\Gamma'.vis\ P = \Gamma.vis\ P \oplus \{sc \mapsto protected\}$. We have that:

$$\Gamma, pds \triangleright P : \mathbf{Process} = \Gamma', pds \triangleright P : \mathbf{Process}$$

provided

$$\begin{aligned} (\rightarrow) & (1) \forall Q \mid P < Q \bullet (\forall s \in Q \mid N(s) = N(sc) \rightarrow \neg PL(s)) \\ & (2) \forall R \mid R < P \bullet \neg occurs(sc, R.act) \wedge \forall s \in R.pps \bullet (\neg N(sc) = N(s) \wedge \neg occurs(sc, s)) \\ (\leftarrow) & (1) \forall Q \mid P < Q \bullet sc \notin P.pps \quad (2) \forall R \mid R < P \bullet sc \notin R.pps \end{aligned}$$

proof By induction, since the semantics is defined in terms of the extended typing system (see Subsection 4.1.2), which does not enforce the visibility constraints, the difference between Γ and Γ' is irrelevant.

Lemma D.2 (access levels equivalence between state components in semantics level).

Consider Γ and Γ' proper typing environments, where $\Gamma.vis\ P\ st_1 = default$ and $\Gamma'.vis\ P = \Gamma.vis\ P \oplus \{st_1 \mapsto protected\}$. We have that:

$$\Gamma, pds \triangleright P : \mathbf{Process} = \Gamma', pds \triangleright P : \mathbf{Process}$$

provided

$$\begin{aligned} (\rightarrow) & \forall R \mid R < P \bullet \neg occurs(R.st_1, R.act) \wedge \neg occurs(R.st_1, R.pps) \\ (\leftarrow) & \forall R \mid R < P \bullet st_1 \notin PS(R.st) \wedge \forall Q \mid P < Q \bullet st_1 \notin PS(Q.st) \end{aligned}$$

proof Similar to Lemma D.1.

APPENDIX E

Algebraic Laws

E.1 Access Modifications

Law E.1 (change schema access level).

provided

- $(\rightarrow) (1) \forall Q \mid P < Q \bullet (\forall s \in Q.pps \mid N(s) = N(sc) \rightarrow \neg PL(s))$
 $(2) \forall R \mid R < P \bullet \neg occurs(sc, R.act) \wedge \forall s \in R.pps \bullet (\neg N(sc) = N(s) \wedge \neg occurs(sc, s))$
 $(\leftarrow) (1) \forall Q \mid P < Q \bullet sc \notin P.pps \quad (2) \forall R \mid R < P \bullet sc \notin R.pps$

□

Law E.2 (change state component access level).

provided

- $(\rightarrow) \forall R \mid R < P \bullet \neg occurs(R.st_1, R.act) \wedge \neg occurs(R.st_1, R.pps)$
 $(\leftarrow) \forall R \mid R < P \bullet st_1 \notin PS(R.st) \wedge \forall Q \mid P < Q \bullet st_1 \notin PS(Q.st)$

□

E.2 Localized Eliminations

Law E.3 (process elimination).

$$pds \text{ } pd_1 = pds$$

where

$$pd_1 = \text{process } P \hat{=} \text{begin} \bullet \text{Skip end}$$

provided

$$(\leftrightarrow) \neg \text{occurs}(P, pds)$$

□

Law E.4 (default schema elimination).

<pre> process P $\hat{=}$ extends Q state st sc pps • act end </pre>	$=_{pds}$	<pre> process P $\hat{=}$ extends Q state st pps • act end </pre>
---	-----------	--

provided

$$\begin{aligned}
&(\rightarrow) \neg \text{occurs}(P.sc, P.act) \wedge \neg \text{occurs}(P.sc, P.pps) \\
&(\leftarrow) sc \notin P.pps \wedge (\forall Q \mid P < Q \bullet sc \notin Q.pps, \text{if } PL(sc))
\end{aligned}$$

□

Law E.5 (default state component elimination).

<pre> process P $\hat{=}$ extends Q state st $\wedge$ st₁ pps • act end </pre>	$=_{pds}$	<pre> process P $\hat{=}$ extends Q state st pps • act end </pre>
---	-----------	--

provided

$$\begin{aligned}
&(\rightarrow) \neg \text{occurs}(P.st_1, P.act) \wedge \neg \text{occurs}(P.st_1, P.pps) \\
&(\leftarrow) st_1 \notin P.st
\end{aligned}$$

□

Law E.6 (protected state component elimination).

provided

$$\begin{aligned}
 (\rightarrow) \quad & \forall R \mid R \leq P \bullet \neg \text{occurs}(R.st_1, R.act) \wedge \neg \text{occurs}(R.st_1, R.pps) \\
 (\leftarrow) \quad & \forall R \mid R \leq P \bullet st_1 \notin PS(R.st) \wedge \forall Q \mid P < Q \bullet st_1 \notin PS(Q.st)
 \end{aligned}$$

□

Law E.7 (protected schema elimination).

provided

$$\begin{aligned}
 (\rightarrow) \quad & \forall R \mid R < P \bullet \neg \text{occurs}(sc, R.act) \wedge \neg \text{occurs}(sc, R.pps) \vee \\
 & N(sc) \in N(R.pps) \Rightarrow P.sc \sqsubseteq R.sc \wedge \neg \text{occurs}(sc, R.sc) \\
 (\leftarrow) \quad & sc \notin P.pps \wedge (\forall R \mid R < P \bullet sc \notin R.pps \vee (sc \in R.pps \wedge P.sc \sqsubseteq R.sc)) \wedge \\
 & (\forall Q \mid P < Q \bullet sc \notin Q.pps \vee (sc \in Q.pps \wedge Q.sc \sqsubseteq P.sc))
 \end{aligned}$$

□

Law E.8 (super elimination).

<pre> process $P \hat{=}$ extends Q state st protected $sc \hat{=}$ $[\oslash st\ decls \mid pred]$ pps $\bullet act$ process $R \hat{=}$ extends P state st protected $sc \hat{=}$ $[\oslash st\ decls \textbf{super } sc \mid pred]$ pps $\bullet act$ </pre>	$=_{pds}$	<pre> process $P \hat{=}$ extends Q state st protected $sc \hat{=}$ $[\oslash st\ decls \mid pred]$ pps $\bullet act$ process $R \hat{=}$ extends P state st protected $sc \hat{=}$ $[\oslash st \oslash P.sc.st$ $decls\ P.sc.decls \mid$ $pred\ P.sc.pred]$ pps $\bullet act$ </pre>
--	-----------	---

□

Law E.9 (eliminating a trivial schema redefinition).

<pre> process $Q \hat{=}$ extends M state st protected sc pps $\bullet act$ end process $P \hat{=}$ extends Q state st protected $sc \hat{=}$ $[\textbf{super } sc]$ pps $\bullet act$ end </pre>	$=$	<pre> process $Q \hat{=}$ extends M state st protected sc pps $\bullet act$ end process $P \hat{=}$ extends Q state st pps $\bullet act$ end </pre>
---	-----	--

□

E.3 Element Interchanges

Law E.10 (moving state component to subprocess).

provided

- $(\rightarrow) \forall S \mid S \leq P \wedge \neg (S \leq R) \bullet \neg occurs(st_2, S.pps) \wedge \neg occurs(st_2, S.act)$
 $(\leftarrow) \forall S \mid S \leq P \wedge \neg (S \leq R) \bullet st_2 \notin PS(S.st)$

□

Law E.11 (move action to subprocess).

provided

- $(\leftrightarrow) \forall S \mid S \in pds \wedge S \neq R \bullet \neg occurs(P, S)$
 $(\rightarrow) PL(st_2)$

□

Law E.12 (splitting a schema among processes).

provided

$(\leftrightarrow) \forall S \mid S \leq P \wedge \neg (S \leq R) \bullet \neg occurs(st_2, S.pps) \wedge \neg occurs(st_2, S.act) \wedge$
 $\neg impact(st_1, st_2)$ ¹
 $(\rightarrow) PL(st_2) \wedge N(sc) \notin N(R.pps)$

□

¹ $impact(st_1, st_2)$ is true iff the value of a state component st_1 is affected by the value of st_2

Law E.13 (move a protected schema to subprocess).

provided

$$(\leftrightarrow) \forall S \mid S < P \wedge \neg (S < R) \bullet \neg occurs(sc, S.pps) \wedge \neg occurs(sc, S.act)$$

□

Law E.14 (move a default schema to subprocess).

provided

$$(\rightarrow) \neg occurs(sc, P.pps) \wedge \neg occurs(sc, P.act) \wedge N(sc) \notin N(R.pps)$$

$$(\leftarrow) \neg occurs(sc, R.pps) \wedge \neg occurs(sc, R.act) \wedge$$

$$\forall T \mid P < T \bullet N(sc) \notin N(PS(T.pps))$$

□

E.4 Subprocess Extraction

Law E.15 (subprocess extraction).

provided
 $(\leftrightarrow) R \notin N(pds)$

□

BIBLIOGRAPHY

- [1] J.-R. Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, New York, NY, USA, 1996.
- [2] W. Al-Ahmad and E. Steegmans. Modeling and reuse perspectives of inheritance can be reconciled. In *Proceedings of the 31st International Conference on Technology of Object-Oriented Language and Systems*, TOOLS '99, pages 31–, Washington, DC, USA, 1999. IEEE Computer Society.
- [3] Pierre America. Designing an object-oriented programming language with behavioural subtyping. In *Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages*, pages 60–90, London, UK, 1991. Springer-Verlag.
- [4] C. Balzarotti, Fiorella de Cindio, and Lucia Pomello. Observation equivalences for the semantics of inheritance. In *Proceedings of the IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pages 455–, Deventer, The Netherlands, The Netherlands, 1999. Kluwer, B.V.
- [5] Wim Bast, Mariano Belaunde, Xavier Blanc, Keith Duddy, Catherine Griffin, Simon Helsen, Michael Lawley, Michael Murphree, Sreedhar Reddy, Shane Sendall, Jim Steel, Laurence Tratt, R. Venkatesh, and Didier Vojtisek. MOF QVT final adopted specification, Nov 2005. OMG document `ptc/05-11-01`.
- [6] Paulo Borba, Augusto Sampaio, Ana Cavalcanti, and Márcio Cornélio. Algebraic reasoning for object-oriented programming. *Sci. Comput. Program.*, 52:53–100, August 2004.
- [7] H. Bowman, C. Briscoe-smith, J. Derrick, and B. Strulo. On behavioural subtyping in lotos, 1996.
- [8] Howard Bowman and John Derrick. A junction between state based and behavioural specification (invited talk). pages 213–239, Deventer, The Netherlands, The Netherlands, 1999. Kluwer, B.V.
- [9] A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. Refinement of actions in *circus*. In *Proceedings of REFINE'2002*, Eletronic Notes in Theoretical Computer Science, 2002. Invited paper.
- [10] A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. A Refinement Strategy for *Circus*. *Formal Aspects of Computing*, 15(2 - 3):146 — 181, 2003.

- [11] A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. Unifying Classes and Processes. *Software and System Modelling*, 4(3):277 – 296, 2005.
- [12] Ana Cavalcanti and David A. Naumann. A weakest precondition semantics for an object-oriented language of refinement. In *Proceedings of the World Congress on Formal Methods in the Development of Computing Systems-Volume II*, FM '99, pages 1439–1459, London, UK, 1999. Springer-Verlag.
- [13] M. L. Cornélio. *Refactorings as Formal Refinements*. PhD thesis, Centro de Informática - UFPE, Recife-Brazil, 2004. BC2004-481.
- [14] Elspeth Cusack. Refinement, conformance and inheritance. *Formal Aspects of Computing*, 3:129–141, 1991. 10.1007/BF01898400.
- [15] John Derrick and Eerke Boiten. *Refinement in Z and object-Z: foundations and advanced applications*. Springer-Verlag, London, UK, 2001.
- [16] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18:453–457, August 1975.
- [17] Robert Bruce Findler, Mario Latendresse, and Matthias Felleisen. Behavioral contracts and behavioral subtyping. *SIGSOFT Softw. Eng. Notes*, 26(5):229–236, 2001.
- [18] C. Fischer. Combining CSP and Z. Technical Report, Germany.
- [19] Clemens Fischer. Csp-oz: a combination of object-z and csp. In *Proceedings of the IFIP TC6 WG6.1 international workshop on Formal methods for open object-based distributed systems*, pages 423–438, London, UK, UK, 1997. Chapman & Hall, Ltd.
- [20] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1996.
- [21] J He, C A R Hoare, and J W Sanders. Data refinement refined. In *Proc. of the European symposium on programming on ESOP 86*, pages 187–196, New York, NY, USA, 1986. Springer-Verlag New York, Inc.
- [22] C. A. R. Hoare. Proof of correctness of data representation. In *Language Hierarchies and Interfaces, International Summer School*, pages 183–193, London, UK, 1976. Springer-Verlag.
- [23] C. A. R. Hoare. Communicating sequential processes. volume 21, pages 666–677, New York, NY, USA, August 1978. ACM.
- [24] C.A.R. Hoare and J. He. *Unifying theories of programming*, volume 14. Prentice Hall, 1998.
- [25] ISO/IEC. Information technology - z formal specification notation - syntax, type system and semantics. ISO 13568:2002 International Standard, 2002.

- [26] Cliff B. Jones. *Systematic software development using VDM (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.
- [27] F. Jouault and I. Kurtev. Transforming models with atl. In *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138, Berlin, 2006. Springer Verlag.
- [28] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994.
- [29] Lemma 1 Ltd. Proofpower, 2011.
- [30] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Trans. Softw. Eng.*, 30:126–139, February 2004.
- [31] Ralph Miarka, Eerke Boiten, and John Derrick. Guards, preconditions, and refinement in z. In *ZB 2000: Formal Specification and Development in Z and B*, volume 1878 of *Lecture Notes in Computer Science*, pages 286–303. Springer Berlin / Heidelberg, 2000.
- [32] R. Milner. *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.
- [33] Carroll Morgan. *Programming from specifications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.
- [34] M. V. M. Oliveira. *Formal Derivation of State-Rich Reactive Programs using Circus*. PhD thesis, Department of Computer Science - University of York, UK, 2006. YCST-2006-02.
- [35] M. V. M. Oliveira, A. L. C. Cavalcanti, and J. C. P. Woodcock. A UTP Semantics for *Circus*. *Formal Aspects of Computing*, 21(1):3 – 32, 2007. The original publication is available at www.springerlink.com.
- [36] M. V. M. Oliveira, A. C. Gurgel, and C. G. Castro. Crefine: Support for the circus refinement calculus. In *Proceedings of the 2008 Sixth IEEE International Conference on Software Engineering and Formal Methods*, pages 281–290, Washington, DC, USA, 2008. IEEE Computer Society.
- [37] Franz Puntigam. Types for active objects based on trace semantics. In *Proceedings FMOODS 96*, pages 4–19. Chapman and Hall, 1996.
- [38] A. W. Roscoe, C. A. R. Hoare, and Richard Bird. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.
- [39] A. C. A. Sampaio, J. C. P. Woodcock, and A. L. C. Cavalcanti. Refinement in *circus*. In L. Eriksson and P. A. Lindsay, editors, *FME 2002: Formal Methods – Getting IT Right*, volume 2391 of *Lecture Notes in Computer Science*, pages 451–470. Springer-Verlag, 2002.

- [40] Augusto Sampaio. *An Algebraic Approach to Compiler Design*. World Scientific, 1997.
- [41] Augusto Sampaio, Jim Woodcock, and Ana Cavalcanti. Refinement in *Circus*. *Lecture Notes in Computer Science*, 2391:451–470, 2002.
- [42] Steve Schneider and Helen Treharne. Communicating b machines. In *Proceedings of the 2nd International Conference of B and Z Users on Formal Specification and Development in Z and B*, ZB '02, pages 416–435, London, UK, UK, 2002. Springer-Verlag.
- [43] J. M. Spivey. *The Z notation: a reference manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.
- [44] J. M. Spivey. *The Z notation: a reference manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.
- [45] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.
- [46] Peter Wegner and Stanley B. Zdonik. Inheritance as an incremental modification mechanism or what like is and isn't like. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '88, pages 55–77, London, UK, 1988. Springer-Verlag.
- [47] Heike Wehrheim. Behavioral subtyping relations for active objects. *Form. Methods Syst. Des.*, 23(2):143–170, 2003.
- [48] J. C. P. Woodcock and A. L. C. Cavalcanti. A concurrent language for refinement. In A. Butterfield and C. Pahl, editors, *IWFM'01: 5th Irish Workshop in Formal Methods*, BCS Electronic Workshops in Computing, Dublin, Ireland, July 2001.
- [49] J. C. P. Woodcock and A. L. C. Cavalcanti. The semantics of *circus*. In D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, editors, *ZB 2002: Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science*, pages 184–203. Springer-Verlag, 2002.
- [50] Jim Woodcock and Ana Cavalcanti. The semantics of circus. In *Proceedings of the 2nd International Conference of B and Z Users on Formal Specification and Development in Z and B*, ZB '02, pages 184–203, London, UK, UK, 2002. Springer-Verlag.
- [51] Jim Woodcock and Jim Davies. *Using Z: specification, refinement, and proof*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.
- [52] M. A. Xavier, A. L. C. Cavalcanti, and A. C. A. Sampaio. Type Checking *Circus* Specifications. In A. M. Moreira and L. Ribeiro, editors, *SBMF 2006: Brazilian Symposium on Formal Methods*, pages 105 – 120, 2006.

- [53] F. Zeyda and A. Cavalcanti. Mechanical Reasoning about Families of UTP Theories. In P. Machado, A. Andrade, and A. Duran, editors, *SBMF 2008, Brazilian Symposium on Formal Methods*, pages 145–160, August 2008. Best paper award.