INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Université d'Ottawa · University of Ottawa

Non-Monotonic Concurrent Constraint Program Verification Using Phase Semantics

Yirgu Yilma

A MASTERS THESIS

submitted to the University of Ottawa
in partial fulfillment of the Masters
Degree of Computer Science

School of Information Technology and Engineering
University of Ottawa

16 December, 1999

Abstract

We present two kinds of Concurrent Constraint Programming languages. The first one is based on a classical logic framework, whereas the second one is based on linear logic. The latter will be at the center of our present discussion since it provides several features such as ability to consume constraints and a full translation of the language into the underlying logic. The provability semantics of this logic can then be used to test programs for safety properties. We will also investigate ways of automating this verification technique, notably by means of finite models of the logic.

Acknowledgments

I would like to extend my sincere thanks to my supervisor, Dr Philip J. Scott, whose guidance, patience and support was greatly appreciated.

I am also much indebted to Paul Ruet for the inspiring and fruitful discussions we have had while I was at the early stage of this work.

I would like to thank Dr Max Kanovich for his useful comments about some parts of this thesis.

Contents

Chapter 1

Introduction

Concurrent Constraint Programming (cc) has been the subject of growing interest as the focus of a new paradigm for concurrent computation. It derives from two computational approaches : *Constraint Logic Programming* which is a technology that has been proved very successful for complex system modeling and for solving "declaratively combinatorial search problems" [17], and *Concurrent Logic Programming.* And thus cc claims a close relationship to logic. Since its inception in 1987 by Saraswat [18], cc has emerged as an important model in the asynchronous computation framework. It provides a logical view of processes and their execution through the paradigm *program-as-formula* and *computation-as-proof-search.*

In parallel with this development, the rapidly growing area of Girard's *linear logic* (LL), has proved to be useful in theoretical computer science. More precisely, LL, which is a radical modification of traditional logic, happens to have strong connections with asynchronous computation and to be close to the concept of resource consumption, because of its resource-sensitive nature. Moreover, we will explain how LL also handles recursion very efficiently. Some works have been initiated to ground cc languages on a LL framework. A non-monotonic extension of cc , lcc is such a language that is based on intuitionistic linear logic (ILL), and thus on *linear constraint systems.* These works on lcc , by P. Lincoln and V. J. Saraswat, and by Fages, Ruet, and Soliman, are mostly from unpublished technical reports. We shall give a short, but self-contained, introduction. As we shall see below, lcc can be given a sound and complete logical semantics in ILL, and the phase semantics of LL can be used to provide simple and elegant "semantical" proofs

of safety properties for `lcc` programs [2].

Organization of the thesis

Chapter 2 is an introduction to intuitionistic linear logic (ILL): we will present its provability model, i.e. the phase space model and we will briefly sketch its capacity to describe asynchronous systems.

In Chapter 3, we introduce concurrent constraint programming languages (both monotonic and non-monotonic versions) and give a logical characterization for these languages.

In Chapter 4, we present a program verification strategy initiated by Fages, Ruet and Soliman [2, 3], followed by some examples.

Chapter 5 deals with sequentialy in the non-monotonic version of concurrent constraint programming language.

Finally in Chapter 6, we give a brief description of the verification tool we have implemented, together with examples for which we have verified some safety properties using the tool.

Contributions of this thesis

A new `lcc` program verification strategy using the phase semantics of ILL has been introduced by Fages, Ruet and Soliman [2, 3]. This approach is new and has, to our knowledge, yet to be thoroughly explored. One contribution of this thesis is to give an exposition of their work (see Chap. 4), taking into account unpublished technical reports on related material by Lincoln and Saraswat[13], and using machinery introduced by various people: Kanovich, Okada, Scedrov [9], Lafont [10], and Ruet [17]. This account also includes some of the recent work about linear logic (e.g. Okada's version of Girard's phase semantics [15] and Lafont's finite model property [10]), and is influenced by Girard's survey [6].

Many protocols require a sequentiality operator to express them in a natural way. In Chapter 5, we show that the basic language `lcc` is powerful enough to represent such a sequentiality operator. This analysis, which is a new contribution of this thesis, includes the study of *phantom constraints* (Section 5.4), as well as related Soundness and Completeness Theorems with respect to Operational Semantics (Theorems 5.4.10, 5.4.12) and the Simulation Theorem 5.4.17, which shows how to simulate sequentiality using phantom constraints.

In 1997, Lafont proved the multiplicative and additive fragment of linear logic (MALL) satisfies the finite model property [10], i.e., MALL provability is equivalent to satisfiability by all finite phase models. The original work of Fages, Ruet, and Soliman often used infinite phase models. In Chapter 6, we will investigate ways of automating the verification method through semantics (using finite phase models), and we will present the experimental tool we have implemented. Here we shall reexamine their work, using finite phase models, based on the finite cyclic monoids (groups) Z_n. This is a new contribution.

In Section 6.4, we reexamine the examples of Fages-Ruet (Dining Philosopher, Mutual Exclusion using Semaphores) using our tool. In Section 6.5, which is new, we discuss Real-time Verification, using techniques of Kanovitch, Okada, and Scedrov in analyzing the ABP (Alternating Bit Protocol) in `lcc` .

Chapter 2

Intuitionistic linear logic

2.1 Overview

Linear logic, introduced by Girard [5], is a refinement of classical logic. It is sometimes described as *resource sensitive* because it provides a natural accounting of resources. This is highlighted by the fact that in linear logic, two copies of the same formula A are distinguished from a single copy of A. Let us describe this in more detail.

Traditional Gentzen rules in proof theory for forming entailments $A_1, A_2, \cdots A_n \vdash B_1, B_2, \cdots, B_m$ are divided into two classes: structural and logical. The logical rules introduce each logical connective on the left and the right of the entailment symbol $\vdash$: the structural rules were historically considered merely basic manipulation rules: Exchange, Weakening, Contraction. In what follows we will illustrate these phenomena for sequents of the form $A_1, A_2, \cdots A_n \vdash B$ [1].

Exchange $\dfrac{\Gamma \vdash A}{\sigma(\Gamma) \vdash A}$ where $\sigma(\Gamma)$ is a permutation of the set Γ

Weakening $\dfrac{\Gamma \vdash B}{\Gamma, A \vdash B}$ **Contraction** $\dfrac{\Gamma, A, A \vdash B}{\Gamma, A \vdash B}$

The exchange rule basically says that the order of hypothesis is irrelevant

[1] Sequents of the form $\Gamma \vdash B$, where Γ is a multiset of formulas and B is a single formula are called *intuitionist* sequents. In this thesis we will restrict ourselves to this type of sequents. However Girard's linear logic uses more general sequents $\Gamma \vdash \Delta$, where both Γ and Δ are multisets of formulas.

to the deduction. The weakening gives us the green light to add to the set of premisses as many extra formulas as we want. The contraction states that the number of copies of a formula in the set of premisses doesn't matter.

Girard[5] pointed out the deep significance of these rules. Linear logic may be viewed as a modification of classical logic where we make the following transformations:

- Removal of the structural rules, *contraction* and *weakening*, which manipulate the use of assumptions and conclusions in logical deductions. The removal of the structural rules leads naturally to two forms of conjunctions : the *multiplicative* $\otimes$ and the *additive* &, and similarly to two forms of disjunction, the *par* $\parr$ (multiplicative) and the *plus* $\oplus$ (additive). This produces a logical system in which each assumption must be used or consumed exactly once. But the lack of contraction or weakening rules makes the logic too weak [5], [6]. So we do the following:

- We add to our system

  a storage or reuse operator !, called *of course*. Intuitively the assumption $!A$ provides unlimited use of the resource A and allows the recapturing of the above structural rules (contraction and weakening) in a more controlled way. We are then able to simulate both intuitionistic and classical logic [5].

This transformation yields the existence of two conjunctions $\otimes$ (time) and & (with) which correspond to two distinct uses of the word "and" [6]. Both conjunctions expresses the availability of two actions; but in the case of $\otimes$, both will be done, whereas in the case of &, only one of them will be performed.

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \& B} \qquad\qquad \frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B}$$

As an illustration consider the example given in [6] : let A, B, C mean:

A : to spend \$1,
B : to buy a pack of Camels,
C : to buy a pack of Marlboros.

Suppose a pack of cigarettes costs one dollar. An action of type A will be a way to spend \$1. Since we may own several \$1 bills, there may be many such actions. Similarly, there are several packs of Camels at the dealer's, hence there are many actions of type B. An action of type $A \multimap B$ is a way of replacing any specific dollar by a specific pack of Camels. In general in linear logic, implication $\multimap$ can be thought of as a process or function that reads its input premise exactly once: moreover, in the process of performing the inference $A, A \multimap B \vdash B$, the premise A is consumed. This process can be repeated depending on the availability of premise A.

Thus given an action of type $A \multimap B$ and an action of type $A \multimap C$, there is no way of forming an action of type $A \multimap B \otimes B$, because one dollar is not enough to buy two packs of cigarettes. By contrast the action $A \multimap B \& C$ is possible, because only one action between B and C will be performed. However we have first to decide which one takes place and then perform the chosen one.

For our work, we only need to present a special fragment of LL, namely *intuitionistic linear logic* (ILL).

Definition 2.1.1 *The intuitionistic formulas are built from*

- *atoms* $p, q, \cdots$
- *the multiplicative connectives :* $\otimes$ *(tensor) and implication* $\multimap$,
- *the additive connectives :* $\&$ *(with) and* $\oplus$ *(plus),*
- *the exponential connective* ! *(of course),*
- *the constants : multiplicative* $\mathbf{1}$ *and* $\bot$, *and additive* $\top$ *and* $\mathbf{0}$,
- *the quantifiers : universal* $\forall$ *and existential* $\exists$.

2.2 Sequent rules for ILL

Intuitionistic sequents are of the form $\Gamma \vdash A$ or $\Gamma \vdash$, where A is a formula and $\Gamma = \{A_1, A_2, ..., A_n\}$ is a finite multiset of formulas (often called *context*). The intended meaning of $\Gamma \vdash A$ is : A_1 or A_2 or ... or A_n implies A.

Axiom / cut

$A \vdash A$ (identity) $\qquad \dfrac{\Gamma \vdash A \quad \Delta, A \vdash B}{\Delta, \Gamma \vdash B}\ (Cut)$

Multiplicatives

$$\frac{\Gamma, A, B \vdash C}{\Gamma, A \otimes B \vdash C} \qquad \frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B}$$

$$\frac{\Gamma \vdash A \quad \Delta, B \vdash C}{\Gamma, \Delta, A \multimap B \vdash C} \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B}$$

$$\frac{\Gamma \vdash A}{\Gamma, \mathbf{1} \vdash A} \qquad \vdash \mathbf{1} \qquad \bot \vdash$$

Additives

$$\frac{\Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma, A \oplus B \vdash C} \qquad \frac{\Gamma \vdash A}{\Gamma \vdash A \oplus B} \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \oplus B}$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \& B} \qquad \frac{\Gamma, B \vdash C}{\Gamma, A \& B \vdash C} \qquad \frac{\Gamma, A \vdash C}{\Gamma, A \& B \vdash C}$$

$$\frac{\Gamma \vdash}{\Gamma \vdash \bot} \qquad \Gamma \vdash \top \qquad \Gamma, \mathbf{0} \vdash A$$

Bang

$$\frac{\Gamma, A \vdash B}{\Gamma, !A \vdash B} \qquad \frac{!\Gamma \vdash A}{!\Gamma \vdash !A}$$

$$\frac{\Gamma, !A, !A \vdash B}{\Gamma, !A \vdash B} \qquad \frac{\Gamma \vdash B}{\Gamma, !A \vdash B}$$

Quantifiers

$$\frac{\Gamma, \forall x A \vdash B}{\Gamma, A[t/x] \vdash B} \qquad \frac{\Gamma \vdash A}{\Gamma \vdash \forall x A}\ x \notin fv(\Gamma)$$

$$\frac{\Gamma, A \vdash B}{\Gamma, \exists x A \vdash B}\ x \notin fv(\Gamma, B) \qquad \frac{\Gamma \vdash A[t/x]}{\Gamma \vdash \exists x A}$$

Explanation

- ◇ *multiplicative and additive :* $\otimes$ involves concatenation of contexts whereas & uses the same context twice i.e. identifies two copies of the same context. These correspond to two different uses of conjunction in ordinary Gentzen rules (and are equivalent if we have the full structural rules):

$$\frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B} \qquad \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}$$

- ◇ *exponential :* $!A$ indicates the possibility of using A *ad libitum.*

Notation 2.2.1 *We follow usual logical convention and write* $\Gamma \vdash_{\mathcal{L}} A$ *to mean the sequent* $\Gamma \vdash A$ *is derivable (= is a theorem) by the rules of logic* $\mathcal{L}$. *It should always be clear when we are refering to a sequent as syntax versus as a theorem.*

2.3 Phase space semantics

Linear logic has many semantics. Two fundamental ones are: a proof semantics known as *coherent semantics*, which is a refinement of Dana Scott's denotational semantics for intuitionistic logic (and programming languages); and a provability semantics, the *phase semantics*. In the present section we will focus on the latter because it turns out to be useful in proving safety properties for programs.

We will present here Okada's definition of phase semantics for intuitionistic LL and extend it to constants $(1, \bot, \top)$ [15].

Definition 2.3.1 A phase space $\mathbf{P} = (P, \cdot, 1, \mathcal{F})$ is a commutative monoid $(P, \cdot, 1)$ together with a set $\mathcal{F} \subseteq \mathcal{P}(P)$, whose elements are called *facts*, such that :

(i) $\mathcal{F}$ is closed under arbitrary intersection ;

(ii) $\forall A \subset P,\ \forall F \in \mathcal{F},\ A \multimap F = \{x \in P \ :\ \forall a \in A,\ a \cdot x \in F\}$ is a fact.

Facts are interpretations of ILL formulas, and hence of `lcc` agents, as we shall see in the upcoming chapters. One can note that the greatest fact $\top$ is the whole monoid P, whereas the smallest fact $\mathbf{0}$ is the empty set $\emptyset$ and $\mathbf{1} = \cap\{F \in \mathcal{F} \ : \ 1 \in F\}$. Moreover facts are closed under linear implication $\multimap$.

Definition 2.3.2 *Let A, B be facts. Define the following facts :*

$$A \& B = A \cap B.$$
$$A \otimes B = \bigcap\{F \in \mathcal{F} \ : \ A \cdot B \subset F\}.$$
$$A \oplus B = \bigcap\{F \in \mathcal{F} \ : \ A \cup B \subset F\}.$$
$$\exists x A = \bigcap\{F \in \mathcal{F} \ : \ (\bigcup_x A) \subset F\}.$$

Clearly we have $A \& B$ as being the greatest fact contained in both A and B. Without much more difficulty we can see that :
- $A \otimes B$ is the smallest fact containing $A \cdot B$.
- $A \oplus B$ is the smallest fact containing $A \cup B$.

Definition 2.3.3 (Enriched phase space) An *enriched phase space* is a phase space $(P, \cdot, \mathcal{F})$ together with a subset $\mathcal{O}$ of $\mathcal{F}$, whose elements are called *open facts*, such that :

. $\mathcal{O}$ is closed under arbitrary $\oplus$ -in particular there is a greatest open fact:

. $\mathbf{1}$ is the greatest open fact:

. $\mathcal{O}$ is closed under finite $\otimes$

. $\otimes$ is idempotent on $\mathcal{O}$, i.e. $\forall A \in \mathcal{O} \ : \ A \otimes A = A$.

The unary connective *of course* can be defined now.

Definition 2.3.4 $!A$ is defined as the greatest open fact contained in A.

Definition 2.3.5 Given an enriched phase space $\mathbf{P} = (P, \cdot, 1, \mathcal{F})$, a valuation η is mapping from atomic formulas to facts such that $\eta(\top) = \top$, $\eta(\mathbf{1}) = \mathbf{1}$ and $\eta(\mathbf{0}) = \mathbf{0}$.

The interpretation $\eta(A)$ of a formula A is given inductively :

$$\begin{aligned} \eta(!A) &= !\eta(A). \\ \eta(A \otimes B) &= \eta(A) \otimes \eta(B). \\ \eta(A \multimap B) &= \eta(A) \multimap \eta(B). \\ \eta(A \& B) &= \eta(A) \& \eta(B). \end{aligned}$$

The interpretation of a context is given similarly :

$$\begin{aligned} \eta((\Gamma, \Delta)) &= \eta(\Gamma) \otimes \eta(\Delta), \\ \eta(\Gamma) &= \mathbf{1} \text{ if } \Gamma \text{ is empty.} \end{aligned}$$

We interpret sequents as follows :

$$\eta(\Gamma \vdash A) = \eta(\Gamma) \multimap \eta(A)$$

Definition 2.3.6 Here are some definitions related to *validity :*

- $\mathbf{P} \models_\eta (\Gamma \vdash A)$ *iff* $1 \in \eta(\Gamma \vdash A)$. *in other words* $\eta(\Gamma) \subset \eta(A)$.
- $\mathbf{P} \models (\Gamma \vdash A)$ *iff* $\forall \eta.\ \mathbf{P} \models_\eta (\Gamma \vdash A)$.
- $\models (\Gamma \vdash A)$ *iff for every phase space* $\mathbf{P}$ *:* $\mathbf{P} \models (\Gamma \vdash A)$

The phase space semantics of ILL satisfies the soundness and completness properties:

Theorem 2.3.7 (Soundness) *If there is a sequent calculus proof of* $\Gamma \vdash A$ *then* $\models (\Gamma \vdash A)$.

Theorem 2.3.8 (Completeness) *If* $\models (\Gamma \vdash A)$ *then there is a sequent calculus proof of* $\Gamma \vdash A$

As pointed out in [3], these results are useful when proving correctness of concurrent constraint programs. Thus *liveness properties* i.e. properties expressing that something good will eventually happen, are proved using ILL completeness results, whereas *safety properties* (i.e. properties expressing that nothing bad will happen) uses soundness results. For instance one can show that

$$A \odot T \vdash A \odot A \odot T$$

is not derivable by exhibiting a phase space and a valuation that does not validate it. Here we can take the phase space $(\mathcal{N}, \cdot, 1, \mathcal{P}(\mathcal{N}))$, where $\mathcal{N}$ is the set of natural numbers, and the valuation η defined by $\eta(A) = \{2\}$, $\eta(\top) = \mathcal{N}$ and observe that 2 is not a multiple of 4.

In this work will essentially focus on proving safety properties for programs.

2.4 Connection of LL to process algebras

With the extra declarations in the formulas of linear logic one could treat the use of resources and communicating processes in a straightforward way. In the multiplicative fragment each resource is used only once, in the additive fragment we may share resources, and in the exponential fragment we open up the possibility of using a resource an arbitrary number of times [8]. In particular, the use of "$\odot$" permits the representation of parallel (or concurrent) processes (essentially by linear conjunction of formulas with no common variables). The use of additives allows us to consider processes in which we split an environment Γ into two identical copies.

Chapter 3

Concurrent Constraint Programming

3.1 Logic as a framework for theoretical computer science

Logic has two important connections with computer science: the first is via the Curry-Howard-DeBruijn isomorphism and is related to: cut-elimination, normalization of proofs, operational semantics of functional languages, etc. This is familiar to all workers in denotational semantics and programming language theory. In this view, the types of a programming language correspond to formulas, and programs (of a given type) correspond to proofs of that type. The cut-elimination (or normalization) process in operational semantics seeks to find normal forms of programs of a given type; and typing rules (or judgments) say how to logically and consistently assign types to untyped programs. Computation by normalization is used in many theoretical and practical functional languages.

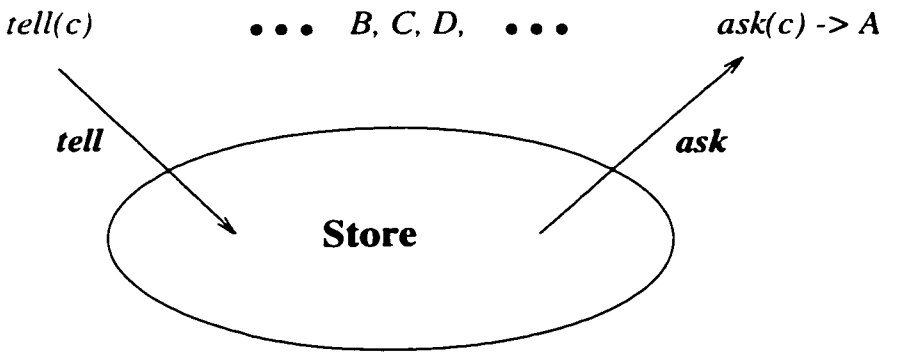
A second major connection of logic to computer science is via the proof-search paradigm, familiar from AI (especially logic-based languages like Prolog) as well as the Concurrent Constraint Programming languages (`cc`) family discussed here. In this paradigm, say especially for the `cc` family, programs themselves are viewed as formulas of a logic, and the *execution* of the program is viewed as proof search.

3.2 Concurrent Constraint Programming : an overview

Concurrent Constraint Programming (cc) languages introduced by Vijay Saraswat in 1987 are a combination of constraint logic programming and concurrent logic programming with a synchronization mechanism based on constraint entailment. In the logical constraint programming framework the operational behavior of a program is deeply entwined with proof theory through the paradigm *"program=formula"* and *"computation=proof search"*. cc programming is a model of concurrent(distributed) computation, where concurrent agents (or processes) communicate through a shared store, represented by a constraint, which expresses some partial information on the value of the variables involved in the computation.
The store is accessed by agents via two basic operations :

- agent *tell*(c), writes (or adds) the constraint c to the store.
- agent *ask*(c) $\rightarrow A$ (where A is an agent) suspends until the store has enough information to entail constraint c.

The *tell* operation is the mechanism for communication : it takes a constraint o and adds it to the common database. The *ask* construct is the mechanism for synchronization. Given a constraint o, *ask*(o) succeeds or fails depending upon whether the store entails o. In the former case the process continues, in the latter case the process suspends until (if ever) more data becomes available [14].



As in traditional process algebras, cc languages use parallel composition, non-deterministic choice, variable hiding and suspension operators. These process combinators may be thought as logical connectives and the processes can be related to formulas.

As for the store, it can be viewed as a theory $(\mathcal{L}, \models)$ (a logic $\mathcal{L}$ together with an entailment relation). This is the main feature of concurrent constraint *logic* programming : the store (or database) is considered as a logic and the inference mechanism of the logic *is* the dynamics of the language.

If we view cc stores as a classical logic formula, then it is clear that only the process $tell(c)$ is capable of altering the content of the store by adding constraints to it. However in the linear logic framework not only does the agent $tell(\ )$ transform the store but also $ask(\ )$ removes constraints from it. This retrieval capability derives from the interpretation of the ask operator by the linear implication which consumes premisses. Indeed in ILL, $\vdash A$, $\vdash A \multimap B$ implies $\vdash B$ and A is consumed in the inference.

In the upcoming subsections we will consider the following systems:

- cc : classical monotonic concurrent programming language as introduced by Saraswat.
- lcc : non-monotonic concurrent programming language as introduced by Lincoln and Saraswat and further studied by Fages and Ruet.

We also discuss three translations:

- From cc to IL (intuitionistic logic)
- From lcc to ILL (intuitionistic linear logic)
- From cc to lcc

3.3 Monotonic cc

The classical logic framework leads to a monotonic store, since no information is removed by logical inference. That is, if $\vdash A$, $\vdash A \rightarrow B$ then $\vdash B$ and A is still available in the store. The atomic agent $tell(-)$ adds data to the store, which can never be removed. Our treatment will basically follows [17], [2] and [3].

3.3.1 The syntax

The Store

The store is a constraint system $(\mathcal{C}, \models_{\mathcal{C}})$, where :

- $\mathcal{C}$ is a set of formulas obtained from a set V of variables, a set Σ of function and relation symbols and logical operators : 1 (true) conjunction $\wedge$ and the existential quantifier $\exists$:
- $\models$ is a subset of $\mathcal{C} \times \mathcal{C}$ which defines the non-logical axioms of the constraint system.

We will use $c \models_{\mathcal{C}} d$ to mean $(c, d) \in \models_{\mathcal{C}}$. $\models_{\mathcal{C}}$ will express the relation of derivation between constraints.

Denote by $\vdash_{\mathcal{C}}$ the smallest relation in $\mathcal{C}^* \times \mathcal{C}$ (where $*$ is the Kleene star) containing $\models_{\mathcal{C}}$ and closed under the rules of intuitionistic logic (IL) :

$$\Gamma, c \vdash c \qquad \frac{\Gamma, c \vdash d \quad \Gamma \vdash c}{\Gamma \vdash d} \qquad \vdash 1 \qquad \frac{\Gamma \vdash c}{\Gamma, 1 \vdash c}$$

$$\frac{\Gamma, d, d \vdash c}{\Gamma, d \vdash c} \qquad \frac{\Gamma \vdash c}{\Gamma, d \vdash c} \qquad \frac{\Gamma \vdash c[t/x]}{\Gamma \vdash \exists x c} \qquad \frac{\Gamma, c \vdash d}{\Gamma, \exists x c \vdash d}\ x \notin fv(\Gamma, d)$$

$$\frac{\Gamma \vdash c_1 \quad \Gamma \vdash c_2}{\Gamma \vdash c_1 \wedge c_2} \qquad \frac{\Gamma, c_1 \vdash c}{\Gamma, c_1 \wedge c_2 \vdash c} \qquad \frac{\Gamma, c_2 \vdash c}{\Gamma, c_1 \wedge c_2 \vdash c}$$

Agents

The syntax of the agents is given by the following grammar :

$$A ::= 1 \mid p(\vec{x}) \mid tell(c) \mid (A \| A) \mid A + A \mid \exists x A \mid c \rightarrow A$$

where 1 represents the *nil* process (or inaction), $\|$ is the parallel composition operator, $+$ is the non-deterministic choice operator, $\exists$ is the variable hiding operator and $\rightarrow$ is the suspension (or blocking) operator. The atomic agents $p(\vec{x})$.... are called *process names.*

Declarations

The syntax of declarations is given by the following grammar :

$$D ::= \epsilon \mid p(\vec{x}) = A \mid D.D$$

where A is an agent.

As we can observe, recursion is obtained through declarations. We make the very natural assumption that in the declaration $p(\vec{x}) = A$, the free variables of A have free occurrences in p.

Programs
A *program* P is defined as a declaration D together with an agent A.

Configurations
A *configuration* is a triple $(\vec{x}; c; \Gamma)$, where $\vec{x}$ is a set of variables (local variables involved in the computation), c is a constraint (the store), and Γ is a multiset of agents which are running in parallel. This represents a snapshot of a running program.

As we shall see, the operational semantics of cc languages is elegantly expressed through configurations.

3.3.2 Operational semantics

Structural congruence
The relation of structural congruence $\equiv$ is the smallest congruence relation such that :

Parallel composition	$(\vec{x}; c; A \| B, \Gamma) \equiv (\vec{x}; c; A, B, \Gamma)$
Inaction	$(\vec{x}; c; 1, \Gamma) \equiv (\vec{x}; c; \Gamma)$
α-Conversion	$\dfrac{y \notin fv(c, \Gamma)}{(\vec{x}; c; \exists y A, \Gamma) \equiv (\vec{x}, y; c; A, \Gamma)}$
Local variables	$\dfrac{y \notin fv(c, \Gamma)}{(\vec{x}; c; \exists y A, \Gamma) \equiv (\vec{x}, y; c; A, \Gamma)} \qquad \dfrac{y \notin fv(c, \Gamma)}{(\vec{x}, y; c; \Gamma) \equiv (\vec{x} y; c; \Gamma)}$

Transition
The transition relation $\longrightarrow_{cc}$ is the smallest transitive relation such that :

Tell	$(\vec{x}; c; tell(d), \Gamma) \equiv (\vec{x}; c \wedge d; \Gamma)$
Ask	$\dfrac{c \vdash_C d}{(\vec{x}; c; d \rightarrow A, \Gamma) \longrightarrow_{cc} (\vec{x}; c; A, \Gamma)}$

$$\text{Declarations} \qquad \frac{(p(\vec{y}) = A) \in P}{(\vec{x}; c; p(\vec{y}).\Gamma) \longrightarrow_{cc} (\vec{x}; c; A.\Gamma)}$$

$$\text{Congruence} \qquad \frac{(\vec{x}; c; \Gamma) \longrightarrow_{cc} (\vec{y}; d; \Delta)}{(\vec{x}; c; \Gamma) \equiv (\vec{y}; d; \Delta)}$$

$$\text{Non-determinism} \qquad (\vec{x}; c; A + B.\Gamma) \longrightarrow_{cc} (\vec{x}; c; A.\Gamma)$$

$$(\vec{x}; c; A + B.\Gamma) \longrightarrow_{cc} (\vec{x}; c; B.\Gamma)$$

P denotes a program.

Proposition 3.3.1 (Monotonicity of the store) *cc programs have the following two features :*

(i) If $(\vec{x}; c; \Gamma) \longrightarrow_{cc} (\vec{y}; d; \Delta)$ *then* $\exists yd \vdash \exists xc$.

(ii) If $(\vec{x}; c; \Gamma) \longrightarrow_{cc} (\vec{y}; d; \Delta)$ *then for all multiset of agents* Σ *and every constraint* e, $(\vec{x}; c \wedge e; \Gamma.\Sigma) \longrightarrow_{cc} (\vec{y}; d \wedge e; \Delta.\Sigma)$.

Because *tell* adds constraints to the store and *ask* doesn't alter the store. in *(i)* we have $d = c \wedge e$. for some constraint e. and $\exists yc \wedge e \vdash \exists xc$.
The first feature merely underlines the monotonic nature of the store: if the store is capable of entailing a constraint at a given step of the computation then it is able to entail the same constraint at any future step. Whereas the second feature allows the addition of dummy constraints to the store and dummy agents to the multiset of agents.

The following result justifies the fact that an agent that doesn't make use of the + operator is *deterministic* :

Proposition 3.3.2 (Confluence) *For all deterministic configuration* κ *(i.e. configurations that don't have non-deterministic rules), if* $\kappa \longrightarrow_{cc} \lambda_1$ *and* $\kappa \longrightarrow_{cc} \lambda_2$ *then there exists a configuration* λ' *such that* $\lambda_1 \longrightarrow_{cc} \lambda'$ *and* $\lambda_2 \longrightarrow_{cc} \lambda'$.

3.4 Non-monotonic Concurrent constraint Programming: Linear cc

Non-monotonic versions of cc programs have been introduced by Saraswat and Lincoln [18] and further studied by Ruet and Fages [2],[3]. They are based

on constraint consumption and thus are closer to some process algebras such as CCS and π-calculus. The constraint system is closed under the rules of a fragment of ILL, and the $Ask(-)$ agent consumes constraints. The idea is to use the inference mechanism of linear logic, which allows consumption of premisses. We shall call such languages `lcc` . Our presentation will closely follow the style of Ruet-Fages.

3.4.1 The syntax

The Store
The store is a constraint system $(\mathcal{C}, \models_{\mathcal{C}})$, where :

- $\mathcal{C}$ is a set of formulas obtained from a set V of variables, a set Σ of function and relation symbols and logical operators : 1 (true) multiplicative conjunction $\otimes$, the existential quantifier $\exists$ and the exponential connector ! ;
- $\models_{\mathcal{C}}$ is a subset of $\mathcal{C} \times \mathcal{C}$ which defines the non-logical axioms of the constraint system.

We will use $c \models_{\mathcal{C}} d$ to mean $(c, d) \subseteq \models_{\mathcal{C}}$. $\models_{\mathcal{C}}$ will express the relation of derivation between constraints.

Denote by $\vdash_{\mathcal{C}}$ the smallest relation in $\mathcal{C}^* \times \mathcal{C}$ (where $*$ is the Kleene star) containing $\models_{\mathcal{C}}$ and closed under the rules of intuitionistic linear logic (ILL) :

$$c \vdash c \qquad \frac{\Gamma, c \vdash d \quad \Delta \vdash c}{\Gamma, \Delta \vdash d} \qquad \vdash 1 \qquad \frac{\Gamma \vdash c}{\Gamma, 1 \vdash c}$$

$$\frac{\Gamma, d, e \vdash c}{\Gamma, d \otimes e \vdash c} \qquad \frac{\Gamma \vdash c \quad \Delta \vdash d}{\Gamma, \Delta \vdash c \otimes d} \qquad \frac{\Gamma \vdash c[t/x]}{\Gamma \vdash \exists x c} \qquad \frac{\Gamma, c \vdash d}{\Gamma, \exists x c \vdash d}\ x \notin fv(\Gamma, d)$$

$$\frac{\Gamma, c \vdash d}{\Gamma, !c \vdash d} \qquad \frac{!\Gamma \vdash c}{!\Gamma \vdash !c} \qquad \frac{\Gamma \vdash d}{\Gamma, !c \vdash d} \qquad \frac{\Gamma, !c, !c \vdash d}{\Gamma, !c \vdash d}$$

These are the rules for *intuitionistic linear logic* for 1, $\otimes$, $\exists$ and !.

Agents
The syntax of `lcc` agents is the same as that of cc agents :

$$A ::= 1 \mid p(\vec{x}) \mid tell(c) \mid (A \| A) \mid A + A \mid \exists x A \mid c \rightarrow A$$

Declarations
The syntax of declarations is given by the following grammar :

$$D ::= \epsilon \mid p(\vec{x}) = A \mid D.D$$

where A is an agent.

As we can observe, recursion is obtained through declarations. We make the very natural assumption that in the declaration $p(\vec{x}) = A$, the free variables of A have free occurrences in p.

Programs
A *program* P is defined as a declaration D together with an agent A.

Configurations
A *configuration* is a triple $(\vec{x}; c; \Gamma)$, where $\vec{x}$ is a set of variables (local variables involved in the computation), c is a constraint (the store), and Γ is a multiset of agents which are running in parallel. As in the monotonic case, a configuration represents a snapshot of a running program.

3.4.2 Operational semantics

Structural congruence
The relation of structural congruence $\equiv$ is the smallest congruence relation such that :

Parallel composition	$(\vec{x}; c; A \| B, \Gamma) \equiv (\vec{x}; c; A, B, \Gamma)$
Inaction	$(\vec{x}; c; 1, \Gamma) \equiv (\vec{x}; c; \Gamma)$
α-Conversion	$\dfrac{z \notin fv(c, \Gamma)}{(\vec{x}, y; c; \exists y A, \Gamma) \equiv (\vec{x}, z; c[z/y]; A, \Gamma[z/y])}$
Local variables	$\dfrac{y \notin fv(c, \Gamma)}{(\vec{x}; c; \exists y A, \Gamma) \equiv (\vec{x}, y; c; A, \Gamma)} \qquad \dfrac{y \notin fv(c, \Gamma)}{(\vec{x}, y; c; \Gamma) \equiv (\vec{x}; c; \Gamma)}$

We can note that these congruences are identical to those of the classical logic framework. This is also the case of the transitions system below except

for *tell* and *ask* :

Transition
The transition relation $\longrightarrow_{lcc}$ is the smallest transitive relation such that :

Tell	$(\vec{x}; c; tell(d).\Gamma) \equiv (\vec{x}; c \otimes d; \Gamma)$
Ask	$\dfrac{c \vdash_C d \otimes e}{(\vec{x}; c; e \multimap A.\Gamma) \longrightarrow_{lcc} (\vec{x}; d; A.\Gamma)}$
Declarations	$\dfrac{(p(\vec{y}) = A) \in P}{(\vec{x}; c; p(\vec{y}).\Gamma) \longrightarrow_{lcc} (\vec{x}; c; A.\Gamma)}$
Congruence	$\dfrac{(\vec{x}; c; \Gamma) \longrightarrow_{lcc} (\vec{y}; d; \Delta)}{(\vec{x}; c; \Gamma) \equiv (\vec{y}; d; \Delta)}$
Non-determinism	$(\vec{x}; c; A + B.\Gamma) \longrightarrow_{lcc} (\vec{x}; c; A.\Gamma)$
	$(\vec{x}; c; A + B.\Gamma) \longrightarrow_{lcc} (\vec{x}; c; B.\Gamma)$

P denotes a program.

3.4.3 Some Remarks on Recursion

Contrary to the case of classical logic, linear logic provides a natural framework for expressing recursion. Consider a recursive program

$$A = \cdots \| A$$

Any instance of this program, if it halts, will employ a finite number of self-calls. If we translate such a program into a (finite) proof in LL of the form $A \vdash \cdots \otimes A$, the corresponding derivation will consist of a tree with root A, with a finite number of identical subtrees (each with root A). The point is that there is a natural bijective correspondence between such derivations and unwinding of recursive calls. This view is used by many authors in naturally simulating machines in LL [7]. Although any finite computation will correspond to a derivation (even in classical logic), the converse fact–to capture finite recursive processes by propositional proofs–is only available in LL because of its resource sensitive aspect. On the other hand, classical logic has many indirect or superfluous proofs (even of $A \vdash B \wedge A$) which do not correspond to recursive unwindings.

3.5 Translation of cc into lcc : the translation $(-)^\circ$

The expressive power of ILL is able to emulate monotonic versions of concurrent constraint programming languages. This ability is essentially due to the capability of linear logic to interpret intuitionistic logic. As we have already seen, contraction and weakening rule are recaptured in linear logic by means of exponentials.

The intuitionistic implication $A \Rightarrow B$ appears as

$$A \Rightarrow B \;=\; (!A) \multimap B$$

in other words, A implies B when B is caused by some iterations of A. This formula is the essential ingredient of a faithful translation of intuitionistic logic into linear logic.

The monotonic nature of cc programs is restored in the case of lcc programs through the use of the exponential connective !, which allows *replication of hypothesis* and hence counters consumption of hypothesis.

We define the translation $(-)^\circ$: cc $\longrightarrow$ lcc of a constraint system $\mathcal{C} \models_c$ and cc agents into a linear constraint system $(\mathcal{C}, \models_c)^\circ$ as follows :

$c^\circ = !c$ if c is an atomic constraint	
$(c \wedge d)^\circ = c^\circ \odot d^\circ$	$(\exists x c)^\circ = \exists x c^\circ$
$tell(c)^\circ = tell(c^\circ)$	$p(\vec{x})^\circ = p(\vec{x})$
$(A \| B)^\circ = A^\circ \| B^\circ$	$(A + B)^\circ = A^\circ + B^\circ$
$(c \rightarrow A)^\circ = c^\circ \rightarrow A^\circ$	$(\exists x A)^\circ = \exists x A^\circ$

For constraints the above translation is a well-known translation of intuitionistic logic into linear logic.

The deduction relation $\models_c^\circ$ is defined as :

$$c^\circ \models_c^\circ d^\circ \text{ iff } c \models_c d$$

The relation $\vdash^\circ$ is derived from $\models^\circ$ using the rules of linear logic for 1, !, $\odot$ and $\exists$.

cc configurations are translated as follow :

$$(\vec{x}; c; \Gamma)^\circ = (\vec{x}; c^\circ; \Gamma^\circ)$$

As for the transition relation $\longrightarrow^{\circ}_{cc}$, it will be the one of `lcc` .

Proposition 3.5.1 *Let c and d be constraints, then : $c \vdash_C d$ iff $c^\circ \vdash^{\circ}_C d^\circ$.*

We recapture of course the monotonic behavior of configurations [2]:

Proposition 3.5.2 (Soundness of $(-)^\circ$) *Let $(\vec{x}; c; \Gamma)$ and $(\vec{y}; d; \Delta)$ be* cc *configurations :*

- $(\vec{x}; c; \Gamma) \equiv (\vec{y}; d; \Delta)$ *iff* $(\vec{x}; c^\circ; \Gamma^\circ) \equiv^\circ (\vec{y}; d^\circ; \Delta^\circ)$
- *if* $(\vec{x}; c; \Gamma) \longrightarrow_{cc} (\vec{y}; d; \Delta)$ *then* $(\vec{x}; c^\circ; \Gamma^\circ) \longrightarrow^{\circ}_{cc} (\vec{y}; d^\circ; \Delta^\circ)$
- *if* $(\vec{x}; c^\circ; \Gamma^\circ) \longrightarrow^{\circ}_{cc} (\vec{y}; d^\circ; \Delta^\circ)$ *then* $(\vec{x}; c; \Gamma) \longrightarrow_{cc} (\vec{y}; e; \Delta)$. *with* $e \vdash_C d$.

3.6 Logical semantics

As mentioned earlier, processes may be thought of as logical formulas and process combinators as logical connectives. This viewpoint was introduced in Mendler et al [14], and also Lincoln & Saraswat [13] , and emphasized in Ruet and Fages [17, 2]. It will be examined in the present section.

3.6.1 Characterizing the store in IL: the translation $(-)^\dagger$

The interpretation of monotonic cc programming in intuitionistic logic is perhaps less "interesting" because only the deterministic fragment is interpretable. Indeed one can be tempted to interpret the non-deterministic operator $+$ by the logical connective $\vee$, however the operational semantics for $+$ differs from the derivation rule for $\vee$ in the sense that the transitions $A + B \longrightarrow_{cc} A$ and $A + B \longrightarrow_{cc} B$ are valid but neither $\alpha \vee \beta \vdash \alpha$ nor $\alpha \vee \beta \vdash \beta$ are correct. Let $det($cc $)$ denotes the deterministic part of cc .

We now describe a translation $(-)^\dagger : det($cc $) \longrightarrow$ IL, as usual defined by structural induction on agents and configurations.

- **Deterministic agents**

$$tell(c)^\dagger = c \qquad (c \rightarrow A)^\dagger = c \Rightarrow A^\dagger$$
$$p(\vec{x})^\dagger = p(\vec{x}) \qquad (\exists x A)^\dagger = \exists x A^\dagger$$
$$(A||B)^\dagger = A^\dagger \wedge B^\dagger$$

- **Multiset of agents**
  Let $\Gamma = (A_1, ..., A_n)$ be a multiset of agents; then $\Gamma^\dagger = A_1^\dagger \wedge ... \wedge A_n^\dagger$, if $\Gamma = \emptyset$ then $\Gamma^\dagger = 1$.

- **Configurations**
  A configuration $(\vec{x}; c; \Gamma)$ is interpreted as $(\vec{x}; c; \Gamma)^\dagger = \exists \vec{x}(c \wedge \Gamma^\dagger)$

- **The deduction system**
  IL$(\mathcal{C}, \mathcal{D})$ denotes the deduction system obtained by adding to IL :
  - the non-logical axiom $c \vdash d$ for every (c, d) in $\models_{\mathcal{C}}$.
  - the non-logical axiom $p(\vec{x}) \vdash A^\dagger$ for every declaration $p(\vec{x}) = A$ in $\mathcal{D}$.

Theorem 3.6.1 (Soundness of $(-)^\dagger$) *Let $(\vec{x}; c; \Gamma)$ and $(\vec{y}; d; \Delta)$ be deterministic configurations.*

(i) If $(\vec{x}; c; \Gamma) \equiv (\vec{y}; d; \Delta)$ then $(\vec{x}; c; \Gamma)^\dagger \dashv\vdash_{IL(\mathcal{C},\mathcal{D})} (\vec{y}; d; \Delta)^\dagger$

(ii) If $(\vec{x}; c; \Gamma) \longrightarrow_{cc} (\vec{y}; d; \Delta)$ then $(\vec{x}; c; \Gamma)^\dagger \vdash_{IL(\mathcal{C},\mathcal{D})} (\vec{y}; d; \Delta)^\dagger$

where $\dashv\vdash$ denotes logical equivalence.

3.6.2 Characterizing the store in ILL: the translation $(-)^\ddagger$

The non-monotonic version of cc programming is fully interpreted by ILL, including the + operator which will be translated by the additive conjunction &. We now define the translation function $(-)^\ddagger : cc \longrightarrow ILL$ by structural induction.

- **Agents**

$tell(c)^\ddagger = c$	$(c \rightarrow A)^\ddagger = c \multimap A^\ddagger$
$p(\vec{x})^\ddagger = p(\vec{x})$	$(\exists x A)^\ddagger = \exists x A^\ddagger$
$(A \| B)^\ddagger = A^\ddagger \odot B^\ddagger$	$(A + B)^\ddagger = A^\ddagger \& B^\ddagger$

- **Multiset of agents**
  Let $\Gamma = (A_1, ..., A_n)$ be a multiset of agents; then $\Gamma^\ddagger = A_1^\ddagger \odot ... \odot A_n^\ddagger$, if $\Gamma = \emptyset$ then $\Gamma^\ddagger = 1$.

- **Configurations**
  A configuration $(\vec{x}: c: \Gamma)$ is interpreted as $(\vec{x}: c: \Gamma)^{\ddagger} = \exists \vec{x}(c \otimes \Gamma^{\ddagger})$

- **The deduction system**
  ILL$(\mathcal{C}, \mathcal{D})$ denotes the deduction system obtained by adding to ILL :

  - the non-logical axiom $c \vdash d$ for every (c, d) in $\models_{\mathcal{C}}$.
  - the non-logical axiom $p(\vec{x}) \vdash A^{\dagger}$ for every declaration $p(\vec{x}) = A$ in $\mathcal{D}$.

Theorem 3.6.2 (Soundness of the translation $(-)^{\ddagger}$) *Let* $(\vec{x}: c: \Gamma)$ *and* $(\vec{y}: d: \Delta)$ *be configurations.*

(i) If $(\vec{x}: c: \Gamma) \equiv (\vec{y}: d: \Delta)$ *then* $(\vec{x}: c: \Gamma)^{\ddagger} \dashv\vdash_{ILL(\mathcal{C},\mathcal{D})} (\vec{y}: d: \Delta)^{\ddagger}$

(ii) If $(\vec{x}: c: \Gamma) \longrightarrow_{\mathtt{lcc}} (\vec{y}: d: \Delta)$ *then* $(\vec{x}: c: \Gamma)^{\ddagger} \vdash_{ILL(\mathcal{C},\mathcal{D})} (\vec{y}: d: \Delta)^{\ddagger}$

Contrarily to `cc` , which has no a logical semantics, except for its deterministic fragment, we can see that `lcc` enjoys full logical interpretation.

3.6.3 Sequentiality in `lcc`

As we will see in the coming examples several real-world protocols make use of the sequentiality operator. However the modeling of sequentiality in ILL has many pitfalls. First there is no ILL connective that captures the exact operational semantics of the sequentiality operator which is

$$(\sigma: nil \triangleright B, \Gamma) \equiv (\sigma, B, \Gamma) \qquad \frac{(\sigma: A, \Gamma) \longrightarrow (\pi: A', \Delta)}{(\sigma: A \triangleright B, \Gamma) \longrightarrow (\pi: A' \triangleright B, \Delta).}$$

Second, if we let the tensor (which happens to have the closest operational meaning of $\triangleright$), interpret sequentiality then we lose the soundness result. And thereby the developed verification method will turn out to be false. Fortunately, sometimes, depending on the exact specification of the protocol, one might still get around this problem and we will show how on some examples.

Chapter 4

Proving Properties of `lcc` programs via phase semantics

4.1 The Fages-Ruet approach to Safety Properties

Safety properties informally mean "nothing bad happens". More precisely, in our operational semantics, transitions that don't fit the specification of our protocol or program must not occur. The technique developed by Fages-Ruet is very natural and can be summarized as follows.

1. We first write down the transition which is not expected to occur, and translate it into an ILL sequent.

2. Find an appropriate phase structure and a valuation that fulfill our purpose, i.e. that make our sequent not valid.

3. Show why the transition is impossible by exhibiting a concrete counter-example.

Let's now consider the correctness of their approach. As discussed in [2] this is achieved by first translating `lcc` programs into ILL. The soundness theorem says :

$$\Gamma \vdash_{ILL} A \text{ implies } \forall P.\ \forall \eta.\ P \models_{\eta} (\Gamma \vdash A)$$

We extend the soundness result to $ILL_{C,\mathcal{D}}$ by adding an extra condition for any valuation η so that it satisfies the inclusion coming from the non-logical axioms (i.e. if $c \vdash d$ then $\eta(c) \subset \eta(d)$).

Now by taking the contrapositive we get :

$$\exists P, \eta \text{ such that } P \not\models_\eta (\Gamma \vdash A) \text{ implies } \Gamma \not\vdash_{ILL_{C,\mathcal{D}}} A.$$

or equivalently :

$$\exists P, \eta \text{ such that } \eta(\Gamma) \not\subset \eta(A) \text{ implies } \Gamma \not\vdash_{ILL_{C,\mathcal{D}}} A.$$

By applying the Soundness Theorem from `lcc` to $ILL_{C,\mathcal{D}}$:

$$(\vec{x}; c; \Gamma)^\ddagger \not\vdash_{ILLC,\mathcal{D}} (\vec{y}; d; \Delta)^\ddagger \text{ implies } (\vec{x}; c; \Gamma) \not\longrightarrow_{\texttt{lcc}} (\vec{y}; d; \Delta)$$

Finally we get the following result :

Proposition 4.1.1 (Ruet) *To prove a safety property of the type :*

$$(\vec{x}; c; \Gamma) \not\longrightarrow_{\texttt{lcc}} (\vec{y}; d; \Delta)$$

it is enough to show that: there is a phase space $\mathbf{P}$*, a valuation* η *and an element* $a \in \eta((\vec{x}; c; \Gamma)^\ddagger)$ *such that* $a \notin \eta((\vec{y}; d; \Delta)^\ddagger)$.

This result, which is based on the Soundness Theorem, is a cornerstone for verification of `lcc` programs. We prove safety properties by showing some derivations are impossible in the ILL version of the program, i.e. by exhibiting a phase space, an interpretation and a counter-example.

The question regarding the automation of this technique will be discussed in Chapter 6.

Notation 4.1.2 For the upcoming discussions we will adopt the following conventions :

- For simplicity we will drop the variables field in a configuration since we won't use any. Consequently configurations will consist of a constraint and a multiset of agents.
- Because our discussion will be essentially centered on `lcc` , we will use $\longrightarrow$ for transitions instead of the cumbersome $\longrightarrow_{\texttt{lcc}}$.
- Moreover we will denote the agent $tell(\phi)$ by simply ϕ.

4.2 Examples

4.2.1 The critical section problem

Consider a system consisting of n processes. Each process has a segment of code, called a critical section, in which the process may be changing common variables, updating a table, writing a file and so on. The important feature is that, when a process is executing in its critical section, no other process is to be allowed to enter its critical section. The critical section problem is to design a protocol that the processes can use to cooperate. A solution to this problem must satisfy the following three requirements:

- *Mutual Exclusion* : Only one process is allowed to execute in its critical section at a time.
- *Progress* : If no process is executing in its critical section and there exists some processes that wish to enter their critical section, then only those processes that are not executing in their remainder section can participate in the decision of which will enter its critical section next, and this selection cannot be postponed indefinitely.
- *Bounded Waiting* : There must be a bound on the number of times that other processes are allowed to enter their critical section after a process has made a request to enter its critical section and before this request is granted.

We will restrict our attention to algorithms that are applicable to only two processes at a time. We will apply the Fages-Ruet verification style for the first requirement, namely the mutual exclusion.

4.2.2 First example

This protocol synchronizes the execution of two processes in their critical sections. We have two processes P_i and P_j, together with a variable *turn* that ranges over $\{i, j\}$. At the beginning *turn* is initialized to i or j (the choice is irrelevant). Here is what happens when P_i wants to enter its critical section:

```
while turn = j do no-op ;
```

```
critical section ;
    turn = j ;
remainder section ;
```

In order to codify this protocol in `lcc` , we replace *turn* with two atomic constraints t_i and t_j. Thus $turn = i$ (resp. j) is interpreted by the existence of t_i (resp. t_j) in the store. The subprocess A_i (resp. A_j) corresponds to the fragment of code that P_i (resp. P_j) executes in its critical section. Here is the code:

$$P_i = [t_j \rightarrow (t_j \| P_i)] + [t_i \rightarrow (t_i \| (A_i \triangleright (t_i \rightarrow t_j)))]$$
$$init = t_i \parallel P_i \parallel P_j.$$

$\triangleright$ denotes the sequentiality operator, which doesn't however belong to the fragment of `lcc` language we have described. One of the reasons why we discarded this operator from our construct is because there is no ILL connective that achieves a faithful translation of its operational semantics. In this example we allow ourselves to interpret $\triangleright$ by $\otimes$. Although this translation is not sound (since $\otimes$ is more general than $\triangleright$), we can still capture enough of the consequences of the sequentiality operator in the kinds of examples we look at below, in order to prove the safety properties we want ([3]).

Let's prove now that the mutual exclusion holds. The mutual exclusion is violated whenever P_i and P_j run at the same time in their critical region. i.e : whenever the following transition :

$$init \longrightarrow A_i \parallel A_j \parallel * \tag{4.1}$$

is possible (we represent by $*$ the other executing processes). But when A_i (resp. A_j) is running we have t_i (resp. t_j) in the store; thus we have the subsequent transition

$$init \longrightarrow t_i \parallel t_j \parallel * \tag{4.2}$$

If we translate the sequentiality operator $\triangleright$ by the ILL tensor, it is clear that we can derive

$$(init)^{\ddagger} \vdash (A_i)^{\ddagger} \otimes (A_j)^{\ddagger} \otimes \top \tag{4.3}$$

As we see above (and as mentioned in [2], [3]), the interpretation of the sequentiality operator by $\odot$ is not sound in general. However one can sometimes capture its operational behavior through its meaning in a model. Hence, in order to establish that (4.1) is not derivable, it suffices to show that (4.2) is not derivable either (note that (4.1) implies (4.2)).

1/ **The property** : $\forall c,\ \forall B.\ (1 : init) \not\longrightarrow (t_i . t_j . c ; B)$

2/ **Phase structure** : Consider the following phase space: $M = \{\mathcal{N}, \cdot, 1, \mathcal{P}(\mathcal{N})\}$, where $(\mathcal{N}, \cdot, 1)$ is the monoid of natural numbers under multiplication.

3/ **Valuation** : We define a valuation η on $A_i, A_j, t_i, t_j, P_i, P_j$ and $init$. We have to check the conditions coming from the declarations are satisfied.

Let η be given by :

$$\eta((A_i)^\ddagger) = \eta((A_j)^\ddagger) = \eta((P_i)^\ddagger)) = \eta((P_j)^\ddagger) = \{1\}$$
$$\eta(t_i) = \eta(t_j) = \eta((init)^\ddagger) = \{2\}$$

One can easily check (using the definition of $P_i^\ddagger$)

- $\eta(\text{body of } P_i^\ddagger) = [\{2\} \multimap (\{2\} \odot \{1\})]\ \&$
  $[\{2\} \multimap (\{2\} \odot \{1\} \odot (\{2\} \multimap \{2\}))]$
  $= [\{2\} \multimap \{2\}] \& [\{2\} \multimap \{2\}]$
  $= \{1\} \& \{1\}$
  $= \{1\}$

  Hence we obtain $\eta(P_i^\ddagger) \subseteq \eta(\text{body of } P_i^\ddagger)$
- $\eta(\text{body of } init^\ddagger) = \{1\} \odot \{1\} \odot \{2\} = \{2\}$
  Therefore $\eta(init^\ddagger) \subseteq \eta(\text{body of } init^\ddagger)$

- **Counter-example** : Because $\eta(t_i \odot t_j \odot \top) = \{2\}.\{2\}.\mathcal{N} = 4\mathcal{N}$ we get $\eta((init)^\ddagger) \not\subseteq \eta(t_i \odot t_j \odot \top)$, since 2 is not a multiple of 4.

4.2.3 A second example

The second protocol that we consider also synchronizes two processes in their critical region. Each process P_i has a flag $flag[i]$ initialized to false at the beginning. Here are the steps that process P_i takes in order to get into its critical section:

```
    flag[i] = true ;
while flag[j] do no-op ;
  critical section ;
  flag[i] = false ;
 remainder section ;
```

This protocol satisfies two of the three safety properties, which are the mutual exclusion and the bounded waiting. The proposed `lcc` code of this protocol uses the sequentiality operator twice:

$$P_i = flag[i] = F \rightarrow (flag[i] = T \parallel Wait(i))$$

$$Wait(i) = [flag[j] = T \rightarrow (flag[j] = T \parallel Wait(i))] \\ + \\ [flag[j] = F \rightarrow (flag[j] = F \triangleright A_i \triangleright (flag[i] = T \rightarrow flag[i] = F))]$$

$$init = P_i \parallel P_j \parallel [flag[i] = F \parallel [flag[j] = F$$

We will only prove that mutual exclusion holds. As in example 1 the interpretation in ILL of $\triangleright$ by $\odot$ has some problems, since it is clear that we can derive:

$$(init)^{\ddagger} \vdash (A_i)^{\ddagger} \odot (A_j)^{\ddagger} \odot \top$$

thus we can't really directly state the safety property as:

$$init \not\longrightarrow A_i \parallel A_j \parallel *$$

However note that $init \longrightarrow A_i \parallel A_j \parallel *$ implies

$$init \longrightarrow flag[i] = T \| flag[j] = T \| flag[i] = F \| flag[j] = F \| *$$

because whenever A_i (resp. A_j) is running we have $flag[i] = T \odot flag[j] = F$ (resp. $flag[j] = T \odot flag[i] = F$) in the store. Hence in order to check for mutual exclusion, it suffices to show that the derivation

$$(init)^{\ddagger} \vdash (flag[i] = T)^{\ddagger} \odot (flag[j] = T)^{\ddagger} \odot (flag[i] = F)^{\ddagger} \odot (flag[j] = F)^{\ddagger} \odot \top$$

is impossible. In order to achieve that, we will exhibit a phase space model together with a valuation that makes the above derivation impossible.

1/ **Formulation of the property :** $\forall c.\ \forall B.\ (1 : init) \not\longrightarrow ((flag[i] = T) \odot (flag[j] = T) \odot (flag[i] = F) \odot (flag[j] = F) \odot c : B)$

2/ **Phase structure :** Again, consider $M = \{N, .., 1, \mathcal{P}(N)\}$

3/ **Valuation :** We define a valuation on P_i, P_j, A_i, A_j, $flag[i] = T$, $flag[j] = T$, $flag[i] = F$, $flag[j] = F$ and $init$. Again we make sure that the conditions coming from the declarations are satisfied. Let η be such that :

$\eta(P_i^\ddagger) = \eta(P_j^\ddagger) = \eta(A_i^\ddagger) = \eta(A_j^\ddagger) = \eta(Wait(i)^\ddagger) = \eta(Wait(j)^\ddagger) = \{1\}$
$\eta((flag[i] = T)^\ddagger) = \eta((flag[j] = T)^\ddagger) = \{2\}$
$\eta((flag[i] = F)^\ddagger) = \eta((flag[j]) = F)^\ddagger) = \{2\}$
$\eta(init^\ddagger) = \{4\}$

It is easy to check that:

- $\eta(P_i^\ddagger) \subseteq \eta(body\ of\ P_i^\ddagger)$
- $\eta(Wait(i)^\ddagger) \subseteq \eta(body\ of\ Wait(i)^\ddagger)$
- $\eta(init^\ddagger) \subseteq \eta(body\ of\ init^\ddagger)$

4/ **Counter-example :** Now we have

$\eta((flag[i] = T)^\ddagger \odot (flag[j] = T)^\ddagger \odot (flag[i] = F)^\ddagger \odot (flag[j] = F)^\ddagger \odot \top) = 16 \cdot \mathcal{N}$: but $16.\mathcal{N} \not\supseteq \{4\} = \eta(init^\ddagger)$, because 4 is not a multiple of 16.

4.2.4 A third example

Here we present a slightly modified version of the Peterson algorithm. Although this algorithm combines the key ideas of the previous two algorithms, it is a correct solution for the critical-section problem, where all three requirements are met [4]. The processes share two variables:

```
var flag: array[0..1] of boolean;
    turn: 0..1;
```

Initially flag[i] = flag[j] = false and the value of turn is irrelevant.

```
turn:=j;
flag[i]:=true;
while flag[j] and turn=j do no-op
critical section;
flag[i]:=false;
Remainder section;
```

When expressed in `lcc` . the protocol becomes :

$$Wait(i) = flag[j] = F \multimap (flag[j] = F \rhd A_i \rhd (flag[i] = T \multimap flag[i] = F))$$
$$+$$
$$turn = i \multimap (turn = i \rhd A_i \rhd (flag[i] = T \multimap flag[i] = F))$$

$$P(i) = flag[i] = F \odot turn = i \multimap (flag[i] = T \odot turn = j \parallel Wait(i))$$
$$+$$
$$flag[i] = F \odot turn = j \multimap (flag[i] = T \odot turn = j \parallel Wait(i))$$

$$Init = P(i) \parallel P(j) \parallel flag[i] = F \odot flag[j] = F \odot turn = i$$

The mutual exclusion is expressed by the fact that the transition

$$Init \longrightarrow A_i \parallel A_j \parallel * \tag{4.4}$$

can't be performed. However here again we run into the usual problem: can show that

$$Init^{\ddagger} \vdash A_i^{\ddagger} \odot A_j^{\ddagger} \odot \top \tag{4.5}$$

is derivable in ILL. Luckily, as in the previous examples, we can get around this problem by using the operational semantics of sequentiality: if transition (4) is possible then it would result in the performance of one of the following transitions :

(α) $Init \longrightarrow (turn = i) \odot (turn = j) \odot (flag[i] = T) \odot (flag[j] = T) \parallel *$

(β) $Init \longrightarrow (turn = i) \odot (flag[i] = F) \odot (flag[i] = T) \odot (flag[j] = T) \parallel *$

(γ) $Init \longrightarrow (turn = j) \odot (flag[j] = F) \odot (flag[i] = T) \odot (flag[j] = T) \parallel *$

(δ) $Init \longrightarrow (flag[i] = F) \odot (flag[j] = F) \odot (flag[i] = T) \odot$
$(flag[j] = T) \parallel *$

So it suffices to show that the ILL translation of the above transitions are not derivable by finding a counter-phase space model.

1/ **Formulation of the property :**

(α) $\forall c.\ B.\ (1 : Init) \not\longrightarrow ((turn = i) \otimes (turn = j) \otimes (flag[i] = T) \otimes (flag[j] = T) \otimes c : B)$

(β) $\forall c.\ B.\ (1 : Init) \not\longrightarrow ((turn = i) \otimes (flag[i] = F) \otimes (flag[i] = T) \otimes (flag[j] = T) \otimes c : B)$

(γ) $\forall c.\ B.\ (1 : Init) \not\longrightarrow ((turn = j) \otimes (flag[j] = F) \otimes (flag[i] = T) \otimes (flag[j] = T) \otimes c : B)$

(δ) $\forall c.\ B.\ (1 : Init) \not\longrightarrow ((flag[i] = F) \otimes (flag[j] = F) \otimes (flag[i] = T) \otimes (flag[j] = T) \otimes c : B)$

2/ **Phase structure :** Take the usual structure $M = \{N, \cdot, 1, \mathcal{P}(N)\}$

3/ **Valuation :** Let η be such that :

$\eta(P(i)^{\ddagger}) = \eta(P(j)^{\ddagger}) = \{1\}$
$\eta(A_i^{\ddagger}) = \eta(A_j^{\ddagger}) = \{1\}$
$\eta(flag[i] = F) = \eta(flag[i] = F) = \{2\}$
$\eta(Init^{\ddagger}) = \{2^2.3\}$
$\eta(Wait(i)^{\ddagger}) = \{1\}$
$\eta(flag[i] = T) = \eta(flag[j] = T) = \{2\}$
$\eta(turn = i) = \eta(turn = j) = \{3\}$

The valuation η is correct since

- $\eta(P(i)^{\ddagger}) \subseteq \eta(body\ of\ P_i)$
- $\eta(Wait(i)^{\ddagger}) \subseteq \eta(body\ of\ Wait(i))$
- $\eta(Init^{\ddagger}) \subseteq \eta(body\ of\ Init)$

4/ **Counter-example :** Now we get

(α) $\eta((turn = i) \otimes (turn = j) \otimes (flag[i] = T) \otimes (flag[j] = T) \otimes \top) = 2^2.3^2.N \not\geq 2^2.3$

(β) $\eta((turn = i) \otimes (flag[i] = F) \otimes (flag[i] = T) \otimes (flag[j] = T) \otimes \top) = 2^3.3.N \not\geq 2^2.3$

(γ) $\eta((turn = j) \otimes (flag[j] = F) \otimes (flag[i] = T) \otimes (flag[j] = T) \otimes \top) = 2^4.N \not\geq 2^2.3$

(δ) $\eta((flag[i] = F) \odot (flag[j] = F) \odot (flag[i] = T) \odot$
$(flag[j] = T) \odot \top) = 2^3.3.\mathcal{N} \not\geq 2^2.3$

Finally we remark that we constantly used the sequentiality operator in our examples without discussing much about it. This operator is not essential in `lcc` . in the sense that it is not primitive. We merely used it to "abbreviate" or make programs more readable. It can be shown that the sequentiality operator can be expressed in terms of parallel composition and ask operators, and this will be the topic of the next chapter.

Chapter 5

Simulation of the sequentiality operator

5.1 A brief scheme of the approach

The `lcc` language version of Fages-Ruet has no explicit sequentiality operator. However since it derives from constraint programming, it ought to have some way of expressing sequentiality notably through its blocking mechanism.
The operational semantics of the sequentiality operator is

$$(\sigma: nil \rhd B, \Gamma) \equiv (\sigma, B, \Gamma) \qquad \frac{(\sigma: A, \Gamma) \longrightarrow (\pi: A', \Delta)}{(\sigma: A \rhd B, \Gamma) \longrightarrow (\pi: A' \rhd B, \Delta).}$$

$A \rhd B$ says : process B remains idle while A is running, and starts executing if and when A reduces to nil. So if we could signal the ending of A by means of a flag, a special constraint for instance, then all we have to do is make use of the already defined blocking operator on process B. We can suspend B until the flag is available in the store so that B synchronizes with the termination of process A.

In the present chapter we will investigate some way of *simulating* a process of the form $A \rhd B$. The concept of simulating a process is something that must be formally defined. This notion will involve some set-theoretic results that we will establish first.

5.2 Multisets of possible actions

Definition 5.2.1 A multiset E is called *multiset of actions* if E is a multiset possibly empty of first order ILL formulas constructed from a set V of variables, from a set Σ of function and relation symbols and from logical operators : 1 and $\otimes$.

Definition 5.2.2 A multiset F is called a *multiset of possible actions* and is denoted by PA, if $F = \{E_1, E_2, ... E_n\}$, where each E_i is a multiset of actions. We define the empty PA as $\emptyset_{PA} =_{def} \{\emptyset\}$.

Let $\mathcal{F}$ be the set of PA's over a set of formulas $\mathcal{C}$.

Definition 5.2.3 Define the operation $@ : \mathcal{F} \times \mathcal{F} \rightarrow \mathcal{F}$, as follows :

$$\{D_1, ... D_m\} @ \{E_1, ... E_n\} =_{def} \{D_1, ... D_m, E_1, ... E_n\}$$

$@$ is basically a concatenation operation of PA's.

Definition 5.2.4 Define the operation $* : \mathcal{F} \times \mathcal{F} \rightarrow \mathcal{F}$ as follow :

- $\{F\} * \{G\} =_{def} \{F \cup_{mult} G\}$ if F and G are both multisets of actions: where $\cup_{mult}$ stands for the $\cup$ of multisets
- $(F_1 @ F_2) * G =_{def} (F_1 * G) @ (F_2 * G)$;
- $F * (G_1 @ G_2) =_{def} (F * G_1) @ (F * G_2)$;

Proposition 5.2.5 *Here are some properties of* $*$ *and* $@$*:*

- *They are associative and commutative;*
- $*$ *has* $\emptyset_{PA}$ *as neutral element.*

Since $@$ is distributive with respect to $*$, it is straightforward to establish that $(\mathcal{F}, *, \emptyset_{PA})$ is a commutative monoid.

Definition 5.2.6 Let F and G, two PA's. We say G *derives* from F iff one of the following conditions is true :

(i) $F = G$;

(ii) $F = G @ H$, for some PA H;

(iii) $F = G * H$, for some PA H;

This definition has no an immediate purpose, but it will be useful in the upcoming discussions (see section 5.4).

5.3 Frontier of an agent

The fragment of the lcc language we will consider will be given by the following grammar :

$$A ::= nil \mid tell(c) \mid (A \parallel A) \mid A + A \mid c \rightarrow A$$

We assume that all agents we work with are finite in length, i.e. contain a finite number of sub-agents. Moreover we won't allow agents of the form $c \rightarrow nil$ and $A + nil$ although they are derivable from the grammar. Practically speaking the agent $c \rightarrow nil$ removes the constraint c from the store and then halts. It is therefore semantically equivalent to $c \rightarrow tell(1)$. Similarly $A + nil$ is semantically equivalent to $A + tell(1)$.

Definition 5.3.1 The *frontier* $\mathcal{F}(A)$ of an agent A is a multiset of possible actions (PA) and is given inductively :

(i) $\mathcal{F}(nil) := \emptyset_{PA}$

(ii) $\mathcal{F}(tell(c)) := \{\{c\}\}$

(iii) $\mathcal{F}(c \rightarrow A) := \mathcal{F}(A)$

(iv) $\mathcal{F}(A + B) := \mathcal{F}(A) \textcircled{@} \mathcal{F}(B)$

(v) $\mathcal{F}(A \parallel B) := \mathcal{F}(A) * \mathcal{F}(B)$

Example 5.3.2 *Let A be a process given by*

- *If* $A = (tell(c_1) \parallel tell(c_2)) + c_3 \rightarrow tell(c_4)$.
  then $\mathcal{F}(A) = (\{\{c_1\}\} * \{\{c_2\}\}) \textcircled{@} \{c_4\} = \{\{c_1, c_2\}\} \textcircled{@} \{\{c_4\}\} = \{\{c_1, c_2\}, \{c_4\}\}$
- *If* $A = c_1 \rightarrow tell(c_2) \parallel (tell(c_3) + c_4 \rightarrow (c_5 \rightarrow tell(c_6)))$.
  then $\mathcal{F}(A) = \{\{c_2\}\} * (\{\{c_3\}\} \textcircled{@} \{\{c_6\}\})$
  $= (\{\{c_2\}\} * \{\{c_3\}\}) \textcircled{@} (\{\{c_2\}\} * \{\{c_6\}\}) = \{\{c_2, c_3\}, \{c_2, c_6\}\}$

Lemma 5.3.3 *For any agent A, we can write* $\mathcal{F}(A)$ *as* $\textcircled{@}_{i=1}^{m} E_i$, *for some* $m \geq 1$, *and where each* E_i *is a multiset of actions.*

Proof. By induction on the structure of agents :

- $A = tell(c)$: clear;

- $A = c \multimap B$:
  Note that $\mathcal{F}(A) = \mathcal{F}(B)$. and apply the induction hypothesis:

- $A = B \parallel C$:
  suppose $\mathcal{F}(B) = \bar{\mathbb{Q}}_i^m E_i$ and $\mathcal{F}(C) = \bar{\mathbb{Q}}_j^n F_j$: then $\mathcal{F}(A) = (\bar{\mathbb{Q}}_i^m E_i) * (\bar{\mathbb{Q}}_j^n F_j)$. By applying the very definition of $*$. we obtain

$$\mathcal{F}(A) = \bar{\mathbb{Q}}_i^m (E_i * (\bar{\mathbb{Q}}_j^n F_j))$$

- $A = B + C$:
  Again we know by induction hypothesis that $\mathcal{F}(B) = \bar{\mathbb{Q}}_i^m E_i$ and $\mathcal{F}(C) = \bar{\mathbb{Q}}_i^n F_i$. By renaming the multisets of actions as follow : $G_1 = E_1, \ldots, G_m = E_m, G_{m+1} = F_1, \ldots, G_{m+n} = F_1$, we obtain $\mathcal{F}(A) = \bar{\mathbb{Q}}_i^{m+n} G_i$.

□

5.4 Phantom constraints and simulation

We borrow a partial ordering relation on linear constraints from Ruet [3] that will be useful in our present discussion.

Definition 5.4.1 The *subsumption preorder* $>$ is defined by

$$c > c' \text{ iff } c \vdash_{\mathcal{C}} c' \otimes \top$$

Because $\top$ is the "true" connective of linear logic, we have

$c > c'$ iff there exists a formula A such that $c \vdash_{\mathcal{C}} c' \otimes A$

Definition 5.4.2 We define a *context* Γ of an agent A to be a multiset of agents that execute concurrently with A.

Note that if $\Gamma = \{A_1, \ldots, A_n\}$ then we can write Γ as $A_1 \parallel \ldots \parallel A_n$. Recall we have introduced the notion of context in our earlier discussion on ILL: we called *context* the multiset of formulas which corresponds to the left-handside of a ILL sequent. According to our concurrent constraint programming paradigm *program-as-formula*, these two concepts of context coincide.

Definition 5.4.3 (phantom constraints) A constraint $e \neq 1$ is said to be a *phantom* for a multi-set of agents Γ and a store σ iff for all constraint $d \neq 1$ such that $\sigma, \Gamma^{\ddagger} > d$. and for all S s.t $\sigma, \Gamma^{\ddagger} > S$ we have :

$$\text{If } S, e > d \text{ then } S > d, e \not> d \text{ and } e \neq d$$

We assume naturally that if a constraint e is phantom with respect to a multiset Γ, then it is a phantom with respect to each element of Γ.

Given an agent A, we would like to enumerate all its actions. If the PA of A ($\mathcal{F}(A)$) can be written as $\tilde{\oplus}_i^m E_i$, (bear in mind that every $E_i \in \mathcal{F}(A)$ is finite) then we can label the elements of each E_i as $\{c_{i1}, c_{i2}, \ldots, c_{in}\}$, where n is the cardinality of E_i.

Now given an agent A, we are able to construct from its PA $\mathcal{F}(A) = \tilde{\oplus}_{i=1}^m E_i$ another multiset $\mathcal{P} = \tilde{\oplus}_{i=1}^m P_i$ where the elements of each P_i are phantoms with respect to A and such that for each $E_i \in \mathcal{F}(A)$, $P_i \in \mathcal{P}$ is of the same cardinality as E_i.

Definition 5.4.4 *A* phantom multiset *(PM) of an agent A is a multiset of phantom constraints with respect to A constructed in the above manner.*

Lemma 5.4.5 *If e is a phantom constraint with respect to A and Γ and if $(\sigma: A, \Gamma) \longrightarrow (\pi: B, \Gamma)$ then e is also a phantom constraint with respect to B.*

Proof. Suppose e is a phantom constraint with respect to A and let $d \neq 1$ be a constraint such that $\pi, \Gamma^{\ddagger}, B^{\ddagger} > d$. Let S be such that $\pi, \Gamma^{\ddagger}, B^{\ddagger} > S$ and $S, e > d$. We know $\sigma, \Gamma^{\ddagger}, A^{\ddagger} \vdash \pi, \Gamma^{\ddagger}, B^{\ddagger}$, hence $\sigma, \Gamma^{\ddagger}, A^{\ddagger} > S$. Therefore e being a phantom with respect of A, we get

$$S > d \text{ and } e \not> d.$$

In other words e is a phantom with respect to B. □

Proposition 5.4.6 *We can easily extend this result to assert that if e is a phantom constraint with respect to an agent A, then e is a phantom constraint for every sub-agent A' of A.*

Proof. The proof is easy and comes from the transitivity of the transition relation. □

Proposition 5.4.7 *Let A be a process in a context Γ and let $\mathcal{E}$ be a PM of A and multiset of phantoms with respect to Γ.*

(i) If $A = c \rightarrow B$ then $\mathcal{E}$ is a PM of B:

(ii) If $A = A_1 + A_2$ then we can write $\mathcal{E}$ as $\mathcal{E}_1 \hat{\oplus} \mathcal{E}_2$ where $\mathcal{E}_i$ is a PM of A_i. $i = 1, 2$.

(iii) If $A = A_1 \parallel A_2$ then we can write $\mathcal{E}$ as $\mathcal{E}_1 * \mathcal{E}_2$, where $\mathcal{E}_i$ is a PM A_i. $i = 1, 2$.

Proof.

(i) $A = c \rightarrow B$
Since $(\sigma; A, \Gamma) \longrightarrow (\pi; B, \Gamma)$, by lemma 5.4.5 we know that $\mathcal{E}$ is a multiset of phantoms with respect to B. From the fact $\mathcal{F}(A) = \mathcal{F}(B)$ we conclude that $\mathcal{E}$ is a PM of B.

(ii) $A = A_1 + A_2$
Because $(\sigma; A, \Gamma) \longrightarrow (\sigma; A_i, \Gamma)$, $i = 1, 2$, again using the lemma 5.4.5 we know $\mathcal{E}$ is a multiset of phantoms with respect to A_i. Moreover since $\mathcal{F}(A) = \mathcal{F}(A_1) \hat{\oplus} \mathcal{F}(A_2)$ we have $\mathcal{E} = \mathcal{E}_1 \hat{\oplus} \mathcal{E}_2$. Now the fact that each $\mathcal{E}_i$ is a PM of A_i derives from the observation that $\mathcal{E}$ is a multiset of phantoms with respect to A_i.

(iii) By Proposition 5.4.6, $\mathcal{E}$ is a multiset of phantoms for both A_1 and A_2. Moreover, because $\mathcal{F}(A) = \mathcal{F}(A_1) * \mathcal{F}(A_2)$, we know we can decompose $\mathcal{E}$ as $\mathcal{E}_1 * \mathcal{E}_2$. The fact that each $\mathcal{E}_i$ is a PM of A_i, $i = 1, 2$, derives from Proposition 5.4.6. □

Let A be an agent in a context Γ and $\mathcal{E}$ a PM of A which is a multiset of phantoms with respect of Γ. Now we can introduce another agent $A_{\mathcal{E}}$ which is identical to A except we replace each frontier action c_{ij} of A by $c_{ij} \odot e_{ij}$.

$A_{\mathcal{E}}$ is an agent that has "almost" the same operational behavior as A, in the sense that it can can perform every transition that A is capable of. The only difference lies on the ability of $A_{\mathcal{E}}$ to add some extra information into the store. Luckily this information happens to be phantom constraints, in other words it doesn't interfere with the actual execution of the program. We call $A_{\mathcal{E}}$ a *simulation* or a *simulator* of A.

For instance if $A = (tell(c_1) \parallel tell(c_2)) + c_3 \rightarrow tell(c_4)$ in a context Γ and $\mathcal{E} = \{\{e_1, e_2\}, \{e_4\}\}$ is a PM of A and multiset of phantoms with respect to

Γ then $A_{\mathcal{E}} = (tell(c_1 \odot e_1) \parallel tell(c_2 \odot e_2)) + c_3 \rightarrow tell(c_4 \odot e_4)$

More formally :

Definition 5.4.8 Let A be an agent and Γ its context. We define agent $A_{\mathcal{E}}$, where $\mathcal{E}$ is a multiset of phantoms with respect to Γ and a PM of A as follows:

(0) if $A = nil$ then $\mathcal{E} = \emptyset_{PA}$ and $A_{\emptyset_{PA}} = nil$

(1) if $A = tell(c)$ then $\mathcal{E} = \{\{e\}\}$ and $A_{\{\{e\}\}} = tell(c \odot e)$;

(2) if $A = c \rightarrow B$ then $(c \rightarrow B)_{\mathcal{E}} = c \rightarrow B_{\mathcal{E}}$;

(3) if $A = B + C$ then $(B + C)_{\mathcal{E}} = B_{\mathcal{E}_1} + C_{\mathcal{E}_2}$,where $\mathcal{E}_1$ and $\mathcal{E}_2$ are PM's of B and C respectively and such that $\mathcal{E} = \mathcal{E}_1 @ \mathcal{E}_2$;

(4) if $A = B \| C$ then $(B \| C)_{\mathcal{E}} = B_{\mathcal{E}_1} \parallel C_{\mathcal{E}_2}$, where $\mathcal{E}_1$ and $\mathcal{E}_2$ are PM's of B and C respectively and such that $\mathcal{E} = \mathcal{E}_1 * \mathcal{E}_2$;

¿From the definition of $A_{\mathcal{E}}$, we can see the mapping that associates an agent to a simulator is surjective, but not injective however. This is related to the fact that we may construct more than one PM for any agent A.

Example 5.4.9

Let $Cl(i) = tell(tick) \parallel Cl(i-1)$. and $Cl(0) = nil$.
Then $\mathcal{F}(Cl(i)) = \{\{tick, tick, \ldots . tick\}\}$ where $tick$ is repeated i times. Let $\mathcal{E}$ be a PM of $Cl(i)$. so $\mathcal{E} = \{\{e_1, e_2, \cdots, e_i\}\}$.
$Cl(i)_{\mathcal{E}} = tell(tick \odot e_i) \| Cl(i-1)_{\{\{e_1, \cdots, e_{i-1}\}\}}$. and $Cl(0)_{\{\emptyset\}} = nil$.

Of course we can observe the choice of $\mathcal{E}$ is not limited, we can consider any multiset of constraints provided it is a PM for $Cl(i)$.

□

Theorem 5.4.10 (Soundness) *Let A be an agent in a context Γ. Suppose $\mathcal{E}$ is a PM of A that is a multiset of phantoms with respect to Γ. Then*

(i) $A = nil$ *iff* $A_{\mathcal{E}} = nil$

(ii) $(\sigma: A, \Gamma) \longrightarrow (\pi; A', \Delta)$ *implies* $(\sigma: A_{\mathcal{E}}, \Gamma) \longrightarrow (\pi \odot e: A'_{\mathcal{E}'}, \Delta)$. *where* $\mathcal{E}'$ *derives from* $\mathcal{E}$.

Proof. Part *(i)* is straightforward and part *(ii)* will be proven by induction.

(i) $A_{\mathcal{E}} = nil$ iff $\mathcal{E} = \{\emptyset\}$ iff $\mathcal{F}(A) = \{\emptyset\}$, which means $A = nil$.

(ii) By induction on the structure of agents :

(case 1) $A = tell(c)$:
We know that $(\sigma: tell(c), \Gamma) \longrightarrow (\sigma \odot c: nil, \Gamma)$. On the other hand $(\sigma: tell(c \odot e), \Gamma) \longrightarrow (\sigma \odot c \odot e: nil, \Gamma)$, and of course $nil_{\emptyset_{PA}} = nil$.

(case 2) $A = c \rightarrow B$:
We have $(\sigma: c \rightarrow B, \Gamma) \longrightarrow (\pi: B, \Gamma)$, where $\sigma = \pi \odot c$. Similarly $(\sigma: (c \rightarrow B)_{\mathcal{E}}, \Gamma) = (\sigma; c \rightarrow B_{\mathcal{E}}, \Gamma) \longrightarrow (\pi: B_{\mathcal{E}}, \Gamma)$ (with again $\sigma = \pi \odot c$), which yields the desired result.

(case 3) $A = B + C$:
We will see only one of the possible two 1-step transitions.
We know $(\sigma: B + C, \Gamma) \longrightarrow (\sigma: B, \Gamma)$ and $(\sigma: c \rightarrow (B + C)_{\mathcal{E}}, \Gamma) = (\sigma: B_{\mathcal{E}_1} + C_{\mathcal{E}_2}, \Gamma)$ with $\mathcal{E} = \mathcal{E}_1 \hat{\oplus} \mathcal{E}_2$ and such that $\mathcal{E}_1$ is a PM of B and $\mathcal{E}_2$ a PM of C. The transition $(\sigma: c \rightarrow (B+C)_{\mathcal{E}}, \Gamma) \longrightarrow (\sigma; c \rightarrow B_{\mathcal{E}_1}, \Gamma)$ implies the desired result.

(case 4) $A = B \| C$:
Here, it suffices to note the fact that $(\sigma: B \parallel C, \Gamma) \equiv (\sigma: B, C, \Gamma)$ and $(\sigma: (B \| C)_{\mathcal{E}}, \Gamma) = (\sigma: c \rightarrow B_{\mathcal{E}_1} \| C_{\mathcal{E}_2}, \Gamma) \equiv (\sigma: B_{\mathcal{E}_1}, C_{\mathcal{E}_2}, \Gamma)$ and then apply the induction hypothesis.

□

Before presenting the completeness theorem, we have one more concept to define, namely the closure of a multiset of phantoms.

Definition 5.4.11 Define $\mathcal{E}^* = @_{i=1}^{m} E_i^*$ where each E_i^* is the closure of E_i under 1 and $\odot$.

Theorem 5.4.12 (Completeness) *Let an agent A in a context Γ, and let $\mathcal{E}$ be a PM of A. If $(\sigma: A_{\mathcal{E}}, \Gamma) \longrightarrow (\pi: B, \Gamma)$ then there is a constraint π', a phantom constraint $e \in E_k^*$, for some $E_k^* \in \mathcal{E}^*$, such that $\pi = \pi'$ or $\pi = \pi' \odot e$ and there exists an agent B' such that $B = B'_{\mathcal{E}'}$ with $(\sigma, A) \longrightarrow (\pi', B')$ and with $\mathcal{E}'$ deriving from $\mathcal{E}$.*

Proof. By induction on agents:

- $A = tell(c)$
  So $A_{\{\{e\}\}} = tell(c \odot e)$. We know $(\sigma: tell(c \odot e), \Gamma) \longrightarrow (\sigma \odot c \odot e: nil, \Gamma)$ hence $\pi' = \sigma \odot c$, $\pi = \pi' \odot e$, $B' = nil$, $\mathcal{E}' = \emptyset_{PA}$ and $(\sigma: A, \Gamma) \longrightarrow (\pi': B', \Gamma)$

- $A = (c \rightarrow C)$
  Then $A_{\mathcal{E}} = (c \rightarrow C_{\mathcal{E}})$. From $(\sigma, (c \rightarrow C_{\mathcal{E}}), \Gamma) \longrightarrow (\pi, C_{\mathcal{E}}, \Gamma)$ we obtain $\pi = \pi'$.$B' = C$, $\mathcal{E}' = \mathcal{E}$ and $(\sigma: A, \Gamma) \longrightarrow (\pi': B', \Gamma)$

- $A = (C + D)$
  So $A_{\mathcal{E}} = (C_{\mathcal{E}_1} + D_{\mathcal{E}_2})$, with $\mathcal{E} = \mathcal{E}_1 @ \mathcal{E}_2$ and such $\mathcal{E}_1$ is a PM of C and $\mathcal{E}_2$, a PM of D. We will look at one of the two possible transitions only. From $(\sigma, C_{\mathcal{E}_1} + D_{\mathcal{E}_2}), \Gamma) \longrightarrow (\sigma, C_{\mathcal{E}}, \Gamma)$ we conclude $\pi = \pi'$, $B' = C$, $\mathcal{E}' = \mathcal{E}_1$ and $(\sigma: A, \Gamma) \longrightarrow (\pi', B', \Gamma)$

- $A = (C \parallel D)$
  Hence $A_{\mathcal{E}} = (C_{\mathcal{E}_1} \| D_{\mathcal{E}_2})$, with $\mathcal{E} = \mathcal{E}_1 @ \mathcal{E}_2$ and such $\mathcal{E}_1$ is a PM of C and $\mathcal{E}_2$, a PM of D. We use a combination of the following congruence : $(\sigma, C_{\mathcal{E}_1} \parallel D_{\mathcal{E}_2}), \Gamma) \equiv (\sigma, C_{\mathcal{E}_1}, D_{\mathcal{E}_2}, \Gamma)$ together with the induction hypothesis.

□

Theorem 5.4.13 *Let $A \neq nil$ be an agent in a context Γ and $\mathcal{E} = @_{i=1}^{m} E_i$ a PM of A. $(\sigma, A_{\mathcal{E}}, \Gamma) \longrightarrow^{*} (\pi, nil, \Delta)$ implies there exists a $k \in \{1, \ldots, m\}$ such that*

$$\pi > (\otimes_{j=1}^{|E_k|} e_{kj})$$

where each $e_{kj} \in E_k$.

Proof. By induction on the structure of agents.

- $A = tell(c)$:
  Suppose $\{\{e\}\}$ is a PM of A. Then $A_{\{\{e\}\}} = tell(c \odot e)$ and we obtain $(\sigma: tell(c \odot e), \Gamma) \longrightarrow (\pi \odot c \odot e: nil, \Delta)$. And of course one can easily see that $\pi \odot c \odot e > e$.

- $A = c \rightarrow B$:
  Here it is crucial to recall our earlier assumption, in which we have thrown out agents of the form $c \rightarrow nil$. Therefore B being different from nil. $c \rightarrow B$ can't halt after a single transition. Noting that $A_{\mathcal{E}} = c \rightarrow B_{\mathcal{E}}$, we apply the induction hypothesis on B.

- $A = B + C$:
  Recall we don't accept in our grammar agents of the form $B + nil$ and $nil + C$. In other words $B \neq nil$ and $C \neq nil$, which means that A can't halt after a single transition.

  We have $A_{\mathcal{E}} = B_{\mathcal{E}_1} + C_{\mathcal{E}_2}$ such that $\mathcal{E} = \mathcal{E}_1 @ \mathcal{E}_2$, with $\mathcal{E}_1$ being a PM of B and $\mathcal{E}_2$ a PM of C. One step transition of $A_{\mathcal{E}}$ will lead to $B_{\mathcal{E}_1}$ (or $C_{\mathcal{E}_2}$). Consider the first case : $(\sigma: A_{\mathcal{E}}, \Gamma) \longrightarrow (\pi: B_{\mathcal{E}_1}, \Delta)$, and then apply the induction hypothesis on $B_{\mathcal{E}_1}$.

- $A = B \parallel C$:
  Write $A_{\mathcal{E}}$ as $B_{\mathcal{E}_1} \parallel C_{\mathcal{E}_2}$ where $\mathcal{E} = \mathcal{E}_1 * \mathcal{E}_2$, with $\mathcal{E}_1$ being a PM of B and $\mathcal{E}_2$ a PM of C. Then use the congruence $(\sigma, C_{\mathcal{E}_1} \parallel D_{\mathcal{E}_2}), \Gamma) \equiv (\sigma, C'_{\mathcal{E}}, D_{\mathcal{E}_2}, \Gamma)$ together with the induction hypothesis.

  If $\mathcal{E}_1 = @_{i=1}^{p} F_i$ and $\mathcal{E}_2 = @_{j=1}^{q} G_j$, from the induction hypothesis we know that when $(\sigma, B_{\mathcal{E}_1}, C_{\mathcal{E}_2}, \Gamma) \longrightarrow^{*} (\pi, nil, nil, \Delta)$, then $\pi > \otimes_{i=1}^{|F_r|} f_i$, and $\pi > \otimes_{j=1}^{|G_s|} g_j$. Hence from the fact $\pi \odot \pi > \pi$, we deduce $\pi > (\otimes_{i=1}^{|F_r|} f_i) \odot (\otimes_{j=1}^{|G_s|} g_j)$ Moreover note that

$$\begin{aligned}\mathcal{E} = \mathcal{E}_1 * \mathcal{E}_2 &= (@_{i=1}^{p} F_i) * (@_{i=j}^{q} G_j) \\ &= @_{i=1}^{p}(F_i * (@_{j=1}^{q} G_j)) \\ &= @_{i=1}^{p}(@_{j=1}^{q}(F_i * G_j)) \\ &= @_{i=1}^{p+q}(F_i * G_j))\end{aligned}$$

Since $@_{i=1}^{m} E_i = @_{i=1}^{p+q}(F_i * G_j))$ we obtain $E_k = F_r * G_s$ for some k.

□

Lemma 5.4.14 *Let $A \neq nil$ be an agent in a context Γ and $\mathcal{E} = @_{i=1}^{m} E_i$ a PM of A. If we have the transition $(\sigma, A_{\mathcal{E}}, \Gamma) \longrightarrow^* (\pi \odot \epsilon, A'_{\mathcal{E}'}, \Delta)$ with ϵ being the tensor product of phantoms added by $A_{\mathcal{E}}$ into the store, then each atomic phantom that is part of ϵ comes from a single E_k, for a certain $k \in \{1, \cdots, m\}$.*

Proof. Suppose the phantom constraints come from two different multisets, say E_k and E_l. Recall $\mathcal{F}(A)$ is of the form $@_{i=1}^{m} F_i$. By construction we know E_k and E_l are associated respectively to F_k and F_l. Which means some frontier actions that belong to F_k and some others that are elements of F_l have been executed. However from the construction of F_k and F_l we know that their actions lie in two different branchs of the execution tree, and each branch corresponds to a non-deterministic choice for A. From the operational semantics of the non-deterministic choice we deduce it is impossible to perform actions on two distinct paths of the execution: as soon as one path in the execution tree is chosen, all the other possibilities are removed. Thus we obtain a contradiction. □

Theorem 5.4.15 *Let $A \neq nil$ be an agent in a context Γ and $\mathcal{E} = @_{i=1}^{m} E_i$ a PM of A. If $(\sigma, A_{\mathcal{E}}, \Gamma) \longrightarrow^* (\pi \odot \epsilon, A'_{\mathcal{E}'}, \Delta)$ with $A'_{\mathcal{E}'} \neq nil$ and ϵ being the tensor product of phantoms added by $A_{\mathcal{E}}$ into the store, then $\epsilon = \bigotimes_{i=1}^{k} e_{li}$, for some $l \in \{1, \cdots, m\}$ and some $k < |E_l|$.*

Proof. Because elements of E_l have been added to the store, by Lemma 5.4.14, elements of this same multiset will keep on being added exclusively to the store until $A'_{\mathcal{E}'}$ eventually terminates. Since $A'_{\mathcal{E}'} \neq nil$ not all elements of E_l have been put into the store. Hence $k < |E_l|$. □

Corollary 5.4.16 *Let A be an agent in a context Γ and $\mathcal{E} = \mathbin{@}_{i=1}^{m} E_i$ a PM of A. Then*

$$(\sigma, A_{\mathcal{E}}, \Gamma) \longrightarrow^{*} (\pi, nil, \Delta) \text{ iff } \pi \vdash \bigoplus_{i=1}^{m}((\bigotimes_{j=1}^{|E_i|} e_{ij}) \odot \top)$$

Proof. ($\Longrightarrow$)The proof uses Theorem 5.4.13 and the ILL rule for $\oplus$. For simplicity write each $\bigotimes_{j=1}^{|E_i|} e_{ij}$ as o_i, $i = 1, \cdots, m$.

¿From Theorem 5.4.13 we know that $(\sigma, A_{\mathcal{E}}, \Gamma) \longrightarrow^{*} (\pi, nil, \Delta)$ implies $\pi > o_k$, for some $k \in \{1, \cdots, m\}$ and hence $\pi \vdash \bigoplus_{i=1}^{m}(o_i \odot \top)$, from the rule for $\oplus$.

($\Longleftarrow$)if $\pi \not\vdash \bigoplus_{i=1}^{m}((\bigotimes_{j=1}^{|E_i|} e_{ij}) \odot \top)$ then for every $i \in \{1, \cdots, m\}$ we have $\pi \not\vdash (\bigotimes_{j=1}^{|E_i|} e_{ij}) \odot \top$. Therefore by Theorem 5.4.15 $A_{\mathcal{E}} \longrightarrow^{*} B$, for a certain $B \neq nil$. □

Recall that the operational semantic of the sequentiality operator is

$$(\sigma: nil \rhd B, \Gamma) \equiv (\sigma, B, \Gamma) \qquad \frac{A \longrightarrow A'}{(\sigma; A \rhd B, \Gamma) \longrightarrow (\pi; A' \rhd B, \Delta)}$$

In words $A \rhd B$ means that process B is blocked until A terminates. Now that we have set up a way to signal the termination of A, we can run A and B concurrently by suspending however the execution of B until this signal becomes available in the store. The next theorem tells us how to simulate sequentiality in `lcc`

Theorem 5.4.17 (Simulation Theorem) *Let A be an agent and $\mathcal{E} = \mathbin{@}_{i=1}^{m} E_i$ its PM. Put $\omega_i = \bigotimes_{j=1}^{|E_i|} e_{ij}$, for $i = 1 \cdots, m$. If B is some agent then in the configuration $(\sigma; A_{\mathcal{E}} \| (\bigoplus_{i=1}^{m} \omega_i \rightarrow B), \Gamma)$, the process $\bigoplus_{i=1}^{m} \omega_i \rightarrow B$ suspends until $A_{\mathcal{E}}$ halts.*

Proof. Suppose $(\sigma; A_{\mathcal{E}} \| (\bigoplus_{i=1}^{m} \omega_i \rightarrow B), \Gamma) \longrightarrow (\pi; A'_{\mathcal{E}'} \| (\bigoplus_{i=1}^{m} \omega_i \rightarrow B, \Gamma))$

(Case 1.) $A'_{\mathcal{E}'} = nil$
By Corollary 5.4.16, $\pi \vdash \bigoplus_{i=1}^{m}(\omega_i \odot \top)$, hence the query $Ask(\bigoplus_{i=1}^{m} \omega_i)$ can be satisfied and agent B may proceed.

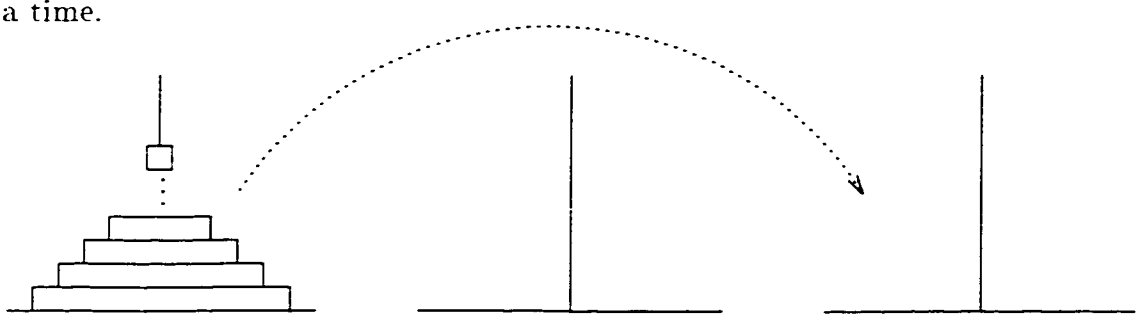
(Case 2.) $A'_{\mathcal{E}'} \neq nil$
From Corollary 5.4.16 we know the store is unable to entail the constraint $\bigoplus_{i=1}^{m} \omega_i$ therefore the process $\bigoplus_{i=1}^{m} \omega_i \rightarrow B$ is blocked.

□

5.5 Tower of Hanoi in lcc and the sequentiality problem

5.5.1 Motivation

The Tower of Hanoi problem is sequential by nature : only one disk is moved at a time.



Here is a recursive algorithm that solves this problem:

```
Hanoi(X, Y, Z, n) = if n = 1 then Move (X, Y)
                    else begin
                           Hanoi(X, Z, Y, n-1);
                           Move(X, Z);
                           Hanoi(Y, X, Z, n-1);
                         end
```

`Move(M, N) : moves the top disk of M to N.`

It is evident that the above algorithm is sequential: only one disk is moved at a time. Note that :

(i) each call to Hanoi(X, Y, n) moves $2^n - 1$ disks ;

(ii) If $n \neq 1$, Hanoi(X, Y, Z, n) may be decomposed into three parts:

- one call to Hanoi(X, Y, Z, n-1);

- one call to Move(X, Z) which moves the n^{th} disk, provided the disks are labeled in increasing order of their diameter;
- one call to Hanoi(Y, X, Z, n-1);

As we will see further, these remarks will be useful.

5.5.2 Tower of Hanoi in lcc

We will use the following variables : to each call of Hanoi(X, Y, Z, n), we associate a integer variable x_n which counts the number of disks moved by the latter procedure.
Move(X, Y) will be modified to Move(X, Y, i) with i being the label of the displaced disk. For each Move (X, Y, i) we associate a boolean variable M_i assigned to true only by Move(X, Y, i).

The constraint $M_i = true$ is used as a phantom constraint and signals the termination of the agent $Move(X, Y, i)$. Similarly $x_n = 2^n - 1$ is a phantom constraint put into the store by $Hanoi(X, Y, Z, n)$ upon its termination.

And finally a counter *Count* will label each disk movement so that we are able to identify the order of displacements. Roughly speaking, $Count(i, j)$ means " At step j, move the i^{th} disk".

$$\begin{aligned} Hanoi(X,Y,Z,n) = (n=1) &\rightarrow (Move(X,Z,1) \parallel M_1 = true \rightarrow x_1 = 1) \\ &+ \\ (n>1) &\rightarrow [Hanoi(X,Y,Z,n-1) \parallel \\ &\quad (x_{n-1} = 2^{n-1}-1) \rightarrow Move(X,Z,n) \parallel \\ &\quad (M_n = true) \rightarrow Hanoi(Y,X,Z,n-1) \parallel \\ &\quad (x_{n-1} = 2^{n-1}-1) \rightarrow x_n = 2^n - 1] \end{aligned}$$

$$\begin{aligned} Move(X,Y,n) = (Count = (j,s)) &\rightarrow ((Count = (n,s+1)) \oplus c) \parallel \\ &\quad c \rightarrow (M_n = true) \end{aligned}$$

$$Init = Hanoi(A,B,C,k) \parallel (Count = (1,0))$$

5.5.3 Safety

One of the safety properties we would like to verify is the that fact only one disk is moved at a time. This can be done by exhibiting a counter phase space model.

We will represent by $p_n!$ the product of the first n prime numbers.

(i) The phase space : as usual we will consider the structure $M = (\mathcal{N}, \cdot, 1, \mathcal{P}(\mathcal{N}))$

(ii) The valuation : Let η be a valuation such that

$$\eta(count = (i.s)) = \begin{cases} \{(p_n!)^s\} & \text{if } s = 1 \\ \{(p_i!)^s\} & \text{otherwise} \end{cases}$$

$$\eta(Move(X, Y, i)^\ddagger) = \begin{cases} \{p_n!\} & \text{if } i = 1 \\ \{p_i!\} & \text{otherwise} \end{cases}$$

$\eta(Hanoi(X, Y, Z, i)^\ddagger) = \{p_i!\}$
$\eta(M_i = true) = \{1\}$
$\eta(n > 1) = \{p_n!\}$
$\eta(n = 1) = \{p_k!/p_n!\}$
$\eta(x_i = 2^i - 1) = \{p_i!\}$
$\eta(Init^\ddagger) = \{p_k!\}$

The correctness of η can be established by showing that :

- $\eta(Hanoi(X, Y, Z, i)^\ddagger) \subseteq \eta(body\ of\ Hanoi(X, Y, Z, i)^\ddagger)$
- $\eta(Move(X, Y, i)^\ddagger) \subseteq \eta(body\ of\ Move(X, Y, i)^\ddagger)$
- $\eta(Init^\ddagger) \subseteq \eta(body\ of\ Init^\ddagger)$

(iii) The counter-example : If two disks, say the i^{th} and the j^{th}, are moved simultaneously then we will have in store $x_i = 2^i - 1$ and $x_j = 2^j - 1$. Hence our goal is to show

$$\forall i, j \in \{1..k\}, \quad Init \longrightarrow (x_i = 2^i - 1) \parallel (x_j = 2^j - 1) \parallel *$$

Or translating it into LL :

(5.1) $\forall i, j \in \{1..k\}, \quad Init^\ddagger \not\vdash (x_i = 2^i - 1) \odot (x_j = 2^j - 1) \odot \top$

Indeed we can see that $\eta((x_i = 2^i - 1) \odot (x_j = 2^j - 1) \odot \top) =$ $\{p_i!\}.\{p_j!\}.\mathcal{N} = p_i!.p_j!\mathcal{N} \not\supseteq \{p_k!\}$ because $p_k!$ is not divisible by $p_i!.p_j!$

Chapter 6

Toward an automated tool

6.1 Algebraic background

6.1.1 Preliminaries

Definition 6.1.1 *For arbitrary integers a, b and a positive integer n, we say that a is congruent to b modulo n, and we write $a \equiv b \bmod n$, if the difference $a - b$ is a multiple of n, that is if $a = b + kn$, for some integer k.*

It is obvious that "congruence modulo n" is an equivalence relation, i.e. reflexive, symmetric and transitive. Consider now the equivalence classes into which the relation of congruence modulo n partitions the set Z. They will be the sets :

$[0] = \{\cdots, -2n, -n, 0, n, 2n, \cdots\}$
$[1] = \{\cdots, -2n+1, -n_1, 1, n+1, 2n+1, \cdots\}$
$\vdots$
$[n-1] = \{\cdots, -n-1, -1, n-1, 2n-1, 3n-1, \cdots\}$

We may define on the set $\{[0], [1], \cdots, [n-1]\}$ of equivalence classes a binary operation by

$$[a] \cdot [b] = [a \cdot b]$$

where a and b are any elements of the respective sets $[a]$ and $[b]$ and the product ab on the right hand side is the ordinary product of a and b. It is easily shown that this new operation is well-defined. Associativity of this operation follows from associativity of ordinary product. Moreover we can note that [1] is the identity element. Consequently :

Proposition 6.1.2 *The elements of the set* $\{[0], [1], \cdots, [n-1]\}$ *form a monoid for any integer* $n > 0$. *We will denote this set by* Z_n.

Example 6.1.3

Here is the Cayley table for Z_6 :

$\cdot$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	1	2	3	4	5
2	0	2	4	0	2	4
3	0	3	0	3	0	3
4	0	4	2	0	4	2
5	0	5	4	3	2	0

□

We denote by (a, b) the greatest common divisor of a and b. Given a positive integer n, we define the greatest common divisor of two congruence classes $[a], [b] \in Z_n$ by the class of $[(a, b)]$.

Let n be a positive number. An element $[m] \in Z_n$ is said to be a divisor of $[0]$ if there is an non-zero element $[l] \in Z_n$ such that $[m] \cdot [l] = [0]$

Theorem 6.1.4 *Let* $a.b$ *and* m *be integers with* $m > 0$ *and* $(a, m) = d$. *If* $d \nmid b$, *then* $a \cdot x \equiv b \bmod m$ *has no solutions. If* $d \mid b$, *then* $a \cdot b \bmod m$ *has exactly* d *incongruent solutions modulo* m.

The proof can be found in [16].

Remark 6.1.5 *It follows that in any monoid* Z_n, *the equation* $[a] \cdot [x] = [b]$, *has no solution if* $(a, n) \nmid b$ *and has exactly* (a, n) *distinct solutions otherwise.*

Proposition 6.1.6 *Let* n *be a positive integer.* $[1] \in [m] \cdot Z_n$ *iff* $(m, n) = 1$

Proof. ($\Longrightarrow$) If $[1] \in [m] \cdot Z_n$ then $[m] \cdot [k] = [1]$, for some $[k] \in Z_n$, i.e. $m \cdot k = 1 + a \cdot n$, for some integer a. In other words $(m, n) = 1$

($\Longleftarrow$)The equation $[m] \cdot [x] = [1]$ has exactly (m, n) solutions because (m, n) divides 1. □

Proposition 6.1.7 *Let n be a non-prime positive number. Then an element $[m] \in Z_n$ that is different from [0] is invertible iff $(m, n) = 1$.*

Proof. $[m]$ is invertible in Z_n iff $\exists![l]$ such that $[m] \cdot [l] = [1]$ iff $m.l = n.a + 1$ for some integer a, i.e. $m.l + n.a = 1$, which is equivalent to saying $(m, n) = 1$. □

Proposition 6.1.8 *Let n be a positive integer :*

(i) $(Z_n^*, \cdot, 1)$ *is a group iff n is prime.*

(ii) Suppose n is not prime. Then any for positive integer m such that $(m, n) \neq 1$. $[m]$ is a divisor of [0].

Proof.

(i) This is a well-known fact.

(ii) Note that $(m, n)|n$, which means there exists an integer $k > 0$ such that $k \cdot (m, n) = n$. Since $(m, n) \neq 1$, we have $0 < k < n$. Hence $k \not\equiv 0 \ mod \ n$. Suppose l is such that $l \cdot (m, n) = m$, we have :

$$\begin{aligned} m \cdot k &\equiv l \cdot (m, n) \cdot k \ mod \ n \\ &\equiv l \cdot 0 \ mod \ n \qquad \text{because } k \cdot (m, n) \text{ is a multiple of } n. \\ &\equiv 0 \ mod \ n \end{aligned}$$

□

Lemma 6.1.9 *Let n be a positive integer. An element $[m] \in Z_n$ is invertible in Z_n, i.e. there exists an integer k such that $[m] \cdot [k] = 1$ iff m and n are relatively prime, in other words $(m, n) = 1$.*

Proof. $(m, n) = 1$ iff $a \cdot m + b \cdot n = 1$ iff $a \cdot m \equiv 1 \ mod \ n$ iff $[a \cdot m] = [1]$, in Z_n iff $[a] \cdot [m] = [1]$ iff $[a]$ is the inverse of $[m]$ in Z_n. □

6.1.2 Finite phase spaces: $\multimap$ and $\otimes$ sets

We are interested in finite phase structures of the type $(Z_n, \cdot, 1, \mathcal{P}(Z_n))$. We focus our attention on the cases where we assign to atomic formulas singleton facts, or facts with two elements. It is helpful to analyze what kinds of general properties are satisfied by facts of the form $A \otimes B$ and $A \multimap B$, where A and B are facts associated to atomic formulas. In particular it will be shown that these facts depend essentially on the structure of A and B as well as the properties of n.

As usual $\top$ corresponds to the whole monoid Z_n, the unit of $\otimes$, $\mathbf{1}$ is the singleton $\{[1]\}$ and $\mathbf{0}$ is represented by $\emptyset$.

Proposition 6.1.10 *Let n be a positive integer, and $m \in Z$. Then*

$$\{[m]\} \otimes \top \vdash [1] \text{ iff } (m, n) = 1.$$

Proof. $\{[m]\} \otimes \top \vdash [1]$ iff $\{[1]\} \subseteq \{[m] \cdot Z_n\}$, i.e. $[m] \cdot [k] = [1]$, for some class $[k]$ in Z_n, and this holds iff $(m, n) \mid 1$, from Theorem 6.1.4. And of course $(m, n) \mid 1$ is equivalent to $(m, n) = 1$ □

Proposition 6.1.11 *Let n be a prime number. Then for all integers m, l, if $m \not\equiv 0 \bmod n$, then $| \{[m]\} \multimap \{[l]\} | = 1$*

Proof. By definition $\{[m]\} \multimap \{[l]\} = \{[x] \in Z_n \mid [x] \cdot [m] = [l]\}$.
Since Z_n is a group and $[m] \neq [0]$, there exists a unique class $[k] \in Z_n$ such that $[k] \cdot [m] = [1]$ (all elements of Z_n are invertible).
$\Longrightarrow ([k] \cdot [m]) \cdot [l] = [l]$. By uniqueness of $[k]$, $|\{[m]\} \multimap \{[l]\}| = 1$ □

Proposition 6.1.12 *Let n be a non-prime positive integer.*

(i) Let m be a positive integer. Then $\{[m]\} \otimes \top = \top$ iff $(m, n) = 1$.

(ii) If $m > 0$ is such that $(m, n) \neq 1$, then $\{[m]\} \otimes \top$ is a proper subset of $\top$.

(iii) $\{[0]\} \otimes \top = \{[0]\}$.

Proof.

(i) Since $(m, n) = 1$, from Remark 6.1.5 it turns out that for each class $[k] \in Z_n$, the equation $[m] \cdot [x] = [k]$ has exactly one solution in Z_n. Moreover for distinct classes $[k_1], [k_2] \in Z_n$ the equations $[m] \cdot [x] = [k_1]$ and $[m] \cdot [x] = [k_2]$ have distinct solutions. This can be shown by assuming for instance $k_1 > k_2$; then if $[l]$ is a common solution to the two previous equations we deduce that $[m] \cdot (k_1 - k_2) = 0$, violating the assumption that m is not a divisor of zero.

(ii) Since $(m, n) \neq 1$, $[m]$ is a zero-divisor, i.e. $\exists [k] \in Z_n$ with $[k] \neq [0]$ and such that $[m] \cdot [k] = [0]$. Keeping in mind that $[m] \cdot [0] = [0]$ we deduce that $[m] \cdot Z_n$ has at most $n - 1$ distinct elements. And from the fact that $[m] \cdot Z_n \subseteq Z_n$ we conclude that $m \cdot Z_n$ is strictly contained in Z_n.

(iii) Clear.

□

Proposition 6.1.13 *Let n be a non-prime positive and m, an integer.*

(a) if $[m]$ is not invertible in Z_n then for all invertible $[l] \in Z_n$, $\{[m]\} \multimap \{[l]\} = \emptyset$:

(b) if $[m]$ is invertible in Z_n then $\forall\ [l] \in Z_n$, $\{[m]\} \multimap \{[l]\}$ has exactly one element:

(c) $\{[m]\} \multimap \{[0]\} \neq \emptyset$ iff $[m]$ is not invertible in Z_n.

Proof.

(a) Suppose $\{[m]\} \multimap \{[l]\} \neq \emptyset$. Then there exists at least an element, say $[k] \in Z_n$ such that $[k] \cdot [m] = [l]$. Using the invertibility of $[l]$ and lemma 6.1.9 we get $(k \cdot m, n) = 1$, which implies $(m, n) = 1$, contradicting the fact that $[m]$ is non-invertible in Z_n.

(b) If $[m]$ is invertible then there exists a unique class $[k] \in Z_n$ such that $[m] \cdot [k] = [1]$. We can derive then $[m] \cdot ([k] \cdot [l]) = [l]$, meaning $[k] \cdot [l] \in \{[m]\} \multimap \{[l]\}$. The fact that $\{[m]\} \multimap \{[l]\}$ is a singleton derives from the unicity of k.

(c) ($\Longrightarrow$) The contrapositive is pretty clear : $[m]$ is invertible $\Longrightarrow \{[m]\} \multimap \{[0]\} = \emptyset$
($\Longleftarrow$) Suppose $[m]$ is not invertible: then $[m]$ is a zero-divisor, i.e. there is an $[l] \in Z_n$ such that $[m] \cdot [l] = [0]$: hence $[l] \in \{[m]\} \multimap \{[0]\}$.

□

6.2 The tool

6.2.1 The underlying theory : Finite model properties

In a logic $\mathcal{L}$, in order to show a formula A is not provable, it suffices to exhibit a model of $\mathcal{L}$ that does not satisfy A. We say $\mathcal{L}$ satisfies the *finite model property* if for every unprovable formula A of $\mathcal{L}$, there is a finite model to falsify it. Unfortunately, there may be an arbitrary number of finite models that are candidates for this task. If we are able to answer the question "*Is A not provable?*" by only checking a bounded number of such models (say bounded by the length or by the number of subformulas of A) then we say $\mathcal{L}$ is *decidable* in the sense that there is a "semi-algorithm" for proof search or provability and a "semi-algorithm" for model search, i.e. non-provability. Most logics in the literature that have the finite model property (e.g. certain modal logics, propositional linear logics, etc.) satisfy this stronger "boundedness" condition, and hence are decidable. The multiplicative and additive fragment of propositional linear logic (MALL) is known to satisfy the finite model property. The follwing theorem is due to Lafond [10]:

Theorem 6.2.1 *For any formula A in* MALL, *the following statements are equivalent:*

(i) A is provable in MALL*:*

(ii) A is satisfied by all finite phase models.

Moreover it has been shown it is decidable and in fact PSPACE-complete [12]. The use of finite structures (such as phase spaces that derive from monoids of the form Z_n) to verify protocols is well known. Our results below do not directly depend on these observations. Nevertheless, our work can be connected to these practices if these protocols can be expressed in the MALL fragment of linear logic. Even though we make use of predicates to

represent agents, we could have used literals labelled by the parameters of the predicates. That is: since there are only at most denumerably many function symbols and constants, we can do the following:

- We observe that all protocols below are translated into a Horn fragment of ILL.
- Predicates $P(i,j)$ can be represented by literals p_{ij}. Distinct atomic formulas $P(t_1,\cdots,t_n)$ (containing terms t_i) should become distinct literals.

As already hinted in Chapter 4, the procedure of verifying a program through phase space semantics is "mechanical enough" to produce an automated tool. Indeed we implemented "a semi-automated " tool which works on a restricted class of monoids, namely the monoids of the type Z_n. In what follows we will describe the main features of this program.

6.2.2 Representing an agent

We adopt the postfix notation for agents, for instance if

$$A = B \parallel (C + (\phi \multimap D))$$

then its postfix format will correspond to

$$A = \parallel B \; + \; C \; \multimap \; \phi \; D$$

The $tell(\psi)$ agent, which we usually abbreviate to ψ and a blocking constraint in $ask(\phi)$ are represented by a record with three fields :

Agent name	Range	Fact

The *agent name* is none other than the name of the constraint: ψ for $tell(\psi)$ and ϕ for $ask(\phi)$. The *range* is an interval of Z_n, over which the search of a fact associated to the agent is done. Finally *fact* represents the actual value of the constraint in the model. It is itself implemented by a dynamic linked list whose values are elements of Z_n.

The operations $\parallel$, $\multimap$ and $+$ will be implemented by the same structure, with the fields *range* and *fact* remaining empty. Finally a non-atomic agent is implemented by a linked list where each node is a record of the above format.

6.2.3 Implementation of some basic operations

As noted earlier, a fact is represented by a dynamic linked list. It is necessary to implement the basic operations on facts, namely the *tensor* $\otimes$, the *with* $\&$ and the *linear implication* $\multimap$.

The *with* corresponds to the intersection operation on sets.

```
ResultList ⟵ &(RightList,LeftList)
 Begin
   For each element R of RightList
   Compare R to each element of LeftList
   If R = L, for some element L of LeftList
   Then Insert R to ResultList
 End;
```

Recall the semantics of the tensor is given by

$$\otimes(\textbf{RightList},\textbf{LeftList}) = \{r \cdot l; r \in \textbf{RightList},\ l \in \textbf{LeftList}\ \}$$

we can write a function for the tensor in a very simple way :

```
ResultList ⟵ ⊗(RightList,LeftList)
 Begin
   For each element R in RightList
     For each element L of LeftList
     Insert (R * L mod N) into ResultList
 End;
```

Perhaps the least trivial implementation would be for the linear implication. Let's recall how $\multimap$ is defined :

$$\multimap(\textbf{PremissesList},\ \textbf{ConclusionList}) = \{x \in Z_n; x \cdot \textbf{PremissesList} \subseteq \textbf{ConclusionList}\}$$

The implementation is as follows :

```
ResultList ⟵ ⊸(PremissesList , ConclusionList)
 Begin
```

```
    For each element C of ConclusionList do
     For V := 1 to N do
       For each element P  of PremissesList do
       If P * V mod N = C
       Then Insert V to ResultList
End;
```

The Insert function must avoid redundancy of elements in the **ResultList**. since **ResultList** is a set in the usual sense.

As we can see, these operations are implemented in a postfix notation. That is why we have adopted a postfix notation for processes. However this way of representing processes might be a bit tedious from the point of view of a user. In order to let the user write agents in an infix format, we also implemented in the program a subroutine that translates infix agents into postfix agents.

6.2.4 I/O's

Now let's give a brief summary on how the I/O's of the program are managed. We first let the user write an agent, and then we allow him to set the range of values that may be put into the fact of the agent and also into each of its sub-agents. Next we ask the user whether the fact associated to the agent is to be equal, say, to one of its sub-agents. This is precisely how recursivity is supported by the program.
To check the correctness of grammar for agents we introduce a basic compiler.

To summarize, inputs are an agent's body together with a list of constraints on the ranges of potential facts that might be associated with the sub-agents and the size of the monoid. The output will be a list of feasible facts of all sub-agents that constitute the main agent.

6.2.5 The search strategy

The program tries to find a fact for every agent such that the requirements are met, and writes the result to a file. There may be more than one answer to the query of the user, so it is up to the user to pick the result that best fits her needs.

Hence as far as the program is concerned, the problem of finding a counter-example is reduced to a search problem. The search strategy adopted for this tool is perhaps not very efficient in terms of complexity because we use a brute force search over the space of potential solutions. It is of the order of $O(n^l)$, where n is the cardinality of the monoid and l the length of the program in terms of agents. Hence we run into a state-explosion problem, and for relatively big agents the program doesn't terminate normally (as expected) because of the huge amount of dynamic allocation that is made.

6.3 Examples revisited

We have run our tool on the simple examples given in Chapter 4, and we have found several finite phase spaces that prove mutual exclusion. Here we will present one for each example. For simplicity, we represent elements of a monoid Z_n by their least positive member (For instance $[23] \in Z_4$ will be represented by 3).

6.3.1 Example 1

Recall the protocol :

$$P_i = [t_j \multimap (t_j \| P_i)] + [t_i \multimap (t_i \| (A_i \triangleright (t_i \multimap t_j)))]$$
$$init = t_i \parallel P_i \parallel P_j.$$

- **Property** : Mutual exclusion is expressed by the ILL expression $init^\ddagger \not\vdash t_i \odot t_j \odot \top$.
- **Phase space** : We have run our program on the phase space $(Z_4, \cdot, 1, \mathcal{P}(Z_4))$
- **Valuation** : One of the valuations found by our tool is :

$$\begin{array}{ll} \eta(t_i) = \{2\} & \eta(P_i^\ddagger) = \{1,3\} \\ \eta(t_j) = \{2\} & \eta(P_j^\ddagger) = \{1,3\} \\ \eta(A_i^\ddagger) = \{1\} & \eta(init^\ddagger) = \{2\} \\ \eta(A_j^\ddagger) = \{1\} & \end{array}$$

- **Counter-example** :
  $\eta(t_i \odot t_j \odot \top) = \{2\} \cdot \{2\} \cdot Z_4 = \{0\}$, but $\eta(init^\ddagger) = \{2\} \not\subseteq \{0\}$.

6.3.2 Example 2

The protocol of example 2 is:

$$P_i = flag[i] = F \rightarrow (flag[i] = T \parallel Wait(i))$$

$$Wait(i) = [flag[j] = T \rightarrow (flag[j] = T \parallel Wait(i))] \overset{+}{} [flag[j] = F \rightarrow (flag[j] = F \triangleright A_i \triangleright (flag[i] = T \rightarrow flag[i] = F))]$$

$$init = P_i \parallel P_j \parallel [flag[i] = F \parallel [flag[j] = F$$

- **Property :** Recall we have expressed the mutual exclusion by the ILL expression
  $\eta(init^{\ddagger}) \not\vdash \eta(flag[i] = T) \odot \eta(flag[i] = F) \odot \eta(flag[j] = T) \odot \eta(flag[i] = F) \odot \eta(flag[j] = F) \odot \top$.

- **Phase space :** Here we have chosen to run or tool on $P = (Z_{16}, \cdot, 1, \mathcal{P}(Z_{16}))$

- **Valuation :** One of the valuations found by the program is :

  $\eta(flag[i] = T) = \{2\}$ $\quad$ $\eta(Wait(i)^{\ddagger}) = \{1.9\}$
  $\eta(flag[j] = T) = \{2\}$ $\quad$ $\eta(Wait(j)^{\ddagger}) = \{1.9\}$
  $\eta(flag[i] = F) = \{2\}$ $\quad$ $\eta(A_i^{\ddagger}) = \{1\}$
  $\eta(flag[j] = F) = \{2\}$ $\quad$ $\eta(A_j^{\ddagger}) = \{1\}$
  $\eta(P_i^{\ddagger}) = \{1.9\}$ $\quad$ $\eta(P_j^{\ddagger}) = \{1.9\}$
  $\eta(init^{\ddagger}) = \{4\}$

- **Counter-example :** We get

  $\eta(flag[i] = T) \odot \eta(flag[i] = T) \odot \eta(flag[j] = T) \odot \eta(flag[i] = F) \odot \eta(flag[j] = F) = \{2\}^4 \cdot Z_{16} = \{0\}$.

  Thus $\eta(init^{\ddagger}) \not\subseteq \{0\}$.

6.4 Examples from Fages-Ruet

In [2], [3], safety properties of some classical protocols have been tested using infinite models. Here we show that finite structures are enough to get the desired results.

6.4.1 Dining philosopher

In [2] the `lcc` dining philosopher program has been shown to be safe using infinite phase structures. However we have found some simpler phase spaces to verify safety properties.
Recalling the program as proposed in [2] :

$$
\begin{array}{l}
philosopher(i,n) \;=\\
\qquad chopstick(i) \odot chopstick(i+1 \;mod\; n) \rightarrow (eat(i,n)) \;\|\\
\qquad eat(i,n) \rightarrow (chopstick(i) \odot chopstick(i{+}1 \;mod\; n) \;\|\; philosopher(i,n))
\end{array}
$$

$$
\begin{array}{l}
recphilo(m,p) \;=\\
\qquad m \neq p \rightarrow chopstick(m) \rightarrow (philosopher(m,p) \;\|\; recphilo(m+1,p))\\
\qquad\qquad\qquad\qquad\qquad\qquad \| \\
\qquad m = p \rightarrow (philosopher(m,p) \| chopstick(m))
\end{array}
$$

$$
init(n) \;=\; recphilo(1,n)
$$

We express the fact that we don't want two neighbors to eat at the same time as follow :

$$
\forall n.\forall i.\forall c.\forall A.\;\; (\emptyset; 1; init(n)) \not\longrightarrow (\emptyset; eat(i,n) \odot eat(i+1 \;mod\; n, n) \odot c; A)
$$

In other words we need to find a phase space P and a valuation η such that :

$$
\begin{array}{c}
\forall n.\forall i.\; \exists x \in \eta(init(n)) \text{ such that}\\
x \notin \eta(eat(i,n)) \odot \eta(eat(i+1 \;mod\; n, n) \odot \top)
\end{array}
$$

- **Phase space.** We have run the tool on the finite the structure defined by $(Z_6, \cdot, 1, \mathcal{P}(Z_6))$.
- **Valuation.** Our program found this valuation in addition to several others :

$\eta(philosopher) = \{1.4\}$ $\quad$ $\eta(chopstick(i)) = \{2\}$
$\eta(eat(i.n)) = \{2\}$ $\quad$ $\eta(i = n) = \{2\}$
$\eta(i \neq n) = \{2\}$ $\quad$ $\eta(recphilo) = \{1.4\}$
$\eta(init) = \{1.4\}$

Let's check the valuation given above is indeed correct:
Checking for philosopher :
RHS of $philosopher(i, n)$

$$= (\{2\} \odot \{2\} \multimap \{2\}) \odot (\{2\} \multimap (\{2\} \odot \{2\} \odot \{1.4\}))$$
$$= (\{4\} \multimap \{2\}) \odot (\{2\} \multimap \{4\})$$
$$= \{2.5\} \odot \{2.5\} = \{1.4\} = philosopher(i, n)$$

Checking for recphilo :
RHS of $recphilo = (\{2\} \multimap (\{1.4\} \odot \{2\} \odot \{1.4\})) \odot (\{2\} \multimap \{1.4\} \odot \{2\})$

$$= (\{2\} \multimap \{2\}) \odot (\{2\} \multimap \{2\})$$
$$= \{1.4\} \odot \{1.4\} = \{1.4\}$$

- **Counter-example.**
  We get $\eta(eat(i.n) \odot eat(i + 1 \ mod \ n) \odot \top) = \{2\}.\{2\}.Z_6 = \{0.2.4\}$.
  And as we can see $1 \notin \{0.2.4\}$.

6.4.2 Mutual exclusion using semaphore

The verification of mutual exclusion properties on a very simple but elegant protocol (which is presented below) that uses a semaphore involved also the use of infinite phase structure in [2] and [3]. But as we shall see a phase space built on a 4-element monoid is enough to check for safety.

$$P_i = sem \rightarrow cs||(A; (cs \rightarrow sem||P_i))$$
$$init = sem||P_1||...||P_n$$

P_i is a process with A as its fragment of codes which are executed upon entering the critical section.

We want to show that two processes don't enter in their critical section at the same time:

$$\forall B, init \not\longrightarrow cs||cs||B||*. \text{ e.i. } init^{\dagger} \not\Vdash cs \odot cs \odot \top$$

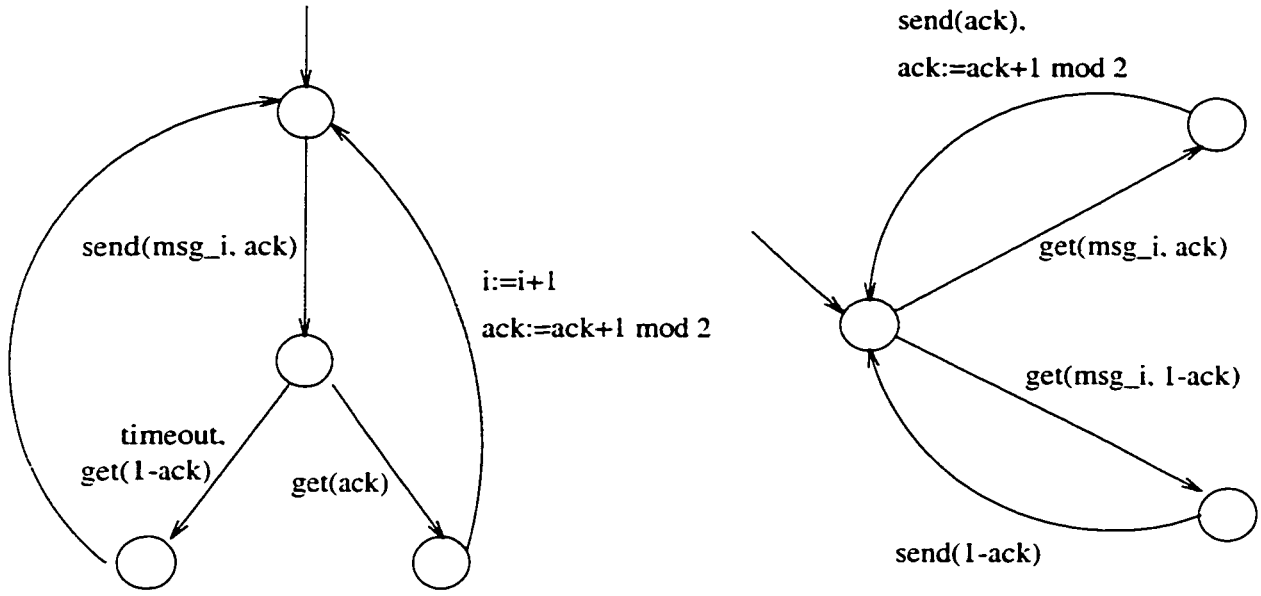
- **Phase space.** We take $(Z_4, \cdot, 1, \mathcal{P}(Z_4))$ as phase structure.
- **Valuation.**
  Here is one valuation found :
  $\eta(P_i) = \{1, 3\}$, $\eta(cs) = \eta(sem) = \eta(init) = \{2\}$ and $\eta(A) = \{1\}$.
- **Counter-example.**
  We have $\eta(cs \odot cs \odot \top) = \{0\} \not\supseteq \eta(init)$.

6.5 Real-time verification

We would like to explore some avenues on real-time verification of safety properties via phase space on an example. We use the real-time specification formalism based on linear logic as proposed in [9]. Time is represented by a unary predicate $time(t)$, where t is assumed to range over a countable dense linear ordering $\mathcal{R}$ together with a "top" element $+\infty$ (e.g. the ordinal $\eta + 1$, where η is the order type of the rationals). For timeouts, we use an alarm $timer(r)$ where $r \in \mathcal{R}$ is finite. Moreover we adopt the convention $timer(+\infty)$ to mean the timer is off.

6.5.1 The alternating bit protocol (ABP)

ABP is used to deliver packets between two nodes in a network in the correct order over an unreliable transmission line.It considers packets from the source one at a time and cannot proceed to the next packet until the current packet is received by the destination. Both the sender and the receiver maintain single bits, initially 0, that record the sequence number of the next packet the sender will send and the receiver expects to receive, respectively.



Sender

Reciever

6.5.2 ABP in lcc

The sender is represented by a 2-ary predicate $Sender(Msg_i, ack)$, where Msg_i is the i^{th} packet and ack the sequence number associated. The 1-ary predicate $Receiver(ack)$ represents the receiver, where ack is the sequence number of the next packet to be received. The sender (resp. the receiver) uses channel Chs for delivery (resp. for reception) and channel Chr for reception (resp. for delivery).

The sender delivers both the packet Msg_i and sequence number ack over channel Chs and sets the alarm $timer$ to some finite value $timeout \in \mathcal{R}$. Then it expects to receive the sequence number back over Chr. In case of reception of the right sequence it carries on with the next packet. However if it receives the wrong message or if a timeout occurs it resends the present message.

$Sender(Msg_i, ack) =$
$time(t) \rightarrow [tell(Chs(Msg_i) \otimes Chs(ack) \otimes ok_1) \parallel$

$$tell(timer(t + timeout) \odot ok_2)] \parallel$$
$$(ok_1 \odot ok_2) \multimap [Chr(ack) \multimap Sender(Msg_{i+1}, ack + 1 \ mod \ 2)$$
$$+ \ Chr(1 - ack) \multimap Sender(Msg_i, ack)$$
$$+ \ time(r) \odot timer(s) \odot (r \geq s) \multimap Sender(Msg_i, ack)]$$

It is worthwhile to note that we used two atomic constraints ok_1 and ok_2 in order to "sequentialize" our program. Indeed we want the sender to wait for the acknowledgment once it has sent the packet and set the timer. Here we use explicitly two phantoms ok_1 and ok_2 to simulate sequentiality.

The receiver waits for the packet and acknowledgment over channel Chs, and upon reception of the expected message, sends back the acknowledgment ack over channel Chr. However in case of reception of the wrong message, it still sends back the sequence number attached to it.

$$Receiver(ack) =$$
$$[(Chs(ack) \odot Chs(m)) \multimap tell(Chr(ack)) \parallel Receiver(ack + 1 \ mod \ 2)]$$
$$+$$
$$[(Chs(1 - ack) \odot Chs(m)) \multimap tell(Chr(1 - ack) \parallel Reciever(ack)]$$

Both sender and receiver work in parallel:

$$System = Sender(Msg_0, 0) \parallel Receiver(0)$$

6.5.3 Verification

One of the features we would like to verify is no packet is lost while it is being transmitted. A particular instance of this problem can be expressed by the following transition :

$$Chs(Msg_i) \odot Chs(ack) \parallel * \not\multimap Chs(Msg_{i+2}) \odot Chs(1 - ack) \parallel *$$

which corresponds to the following interpretation in ILL:

$$Chs(Msg_i) \odot Chs(ack) \odot \top \not\vdash Chs(Msg_{i+2}) \odot Chs(1 - ack) \odot \top$$

1/ **Phase space** : We have run the tool on $(Z_6, \cdot, 1, \mathcal{P}(M))$.

2/ **Valuation** : One of the valuation η found is

$\eta(time(t)^{\ddagger}) = \{1\}$ $\quad$ $\eta(x \geq y) = \{1\}$

$\eta(ok_i) = \{1\}$ $\quad$ $\eta(Sender(Msg_i.ack)\ddagger) = \{0\}$

$\eta(receiver(ack)^{\ddagger}) = \{4\}$ $\quad$ $\eta(timer(t)^{\ddagger}) = \{1\}$

$$\eta(Chs(ack)^{\ddagger}) = \begin{cases} \{2\} & \text{if } ack = 0 \\ \{3\} & \text{if } ack = 1 \end{cases}$$

$$\eta(Chr(ack)^{\ddagger}) = \begin{cases} \{2\} & \text{if } ack = 0 \\ \{3\} & \text{if } ack = 1 \end{cases}$$

$$\eta(Chs(msg_i)^{\ddagger}) = \begin{cases} \{1\} & \text{if } i \text{ is even} \\ \{3\} & \text{otherwise} \end{cases}$$

$$\eta(Chr(msg_i)^{\ddagger}) = \begin{cases} \{1\} & \text{if } i \text{ is even} \\ \{3\} & \text{otherwise} \end{cases}$$

3/ **Counter-Example :**

Case 1. i even: We get

$$\eta(Chs(Msg_i) \odot Chs(ack) \odot \top) = \{2\}.M = \{0, 2, 4\}$$

however

$$\eta(Chs(Msg_{i+2}) \odot Chs(1 - ack) \odot \top) = \{3\}.M = \{0, 3\}$$

And we can see that $\{0.2.4\} \not\subseteq \{0.3\}$.

Case 2. i odd:

$$\eta(Chs(Msg_i) \odot Chs(ack) \odot \top) = \{3\}.M = \{0, 3\}$$

and

$$\eta(Chs(Msg_{i+2}) \odot Chs(1 - ack) \odot \top) = \{0\}.M = \{0\}$$

We get our contradiction :$\{0, 3\} \not\subseteq \{0\}$.

Chapter 7

Conclusion and future works

Because its underlying logic is resource-sensitive and well-suited to describe asynchronous systems, `lcc` can express protocols in a very simple and natural way. It has been shown in [2, 3] that `cc` is "simulated" in `lcc` in the sense that there exists a sound translation from `cc` to `lcc` . In this work, we explore yet another feature of the expressive power of `lcc` , namely its capability of handling sequentiality in a very natural way.

The *program-as-formula* and *execution-as-proof search* paradigms that concurrent constraint programming languages possess has enabled us to achieve an interesting level of abstraction for reasoning about programs. In particular the ability to confer a sound and complete logical semantics for `lcc` has opened up the possiblity to test programs at the logical level by means of a provability semantics of the logic itself. Indeed phase semantics seems to reveal an interesting avenue for protocol verification. In our work, we showed a way of automating this "semantical" proof strategy, by considering a particular class of finite monoids. Our implementation although not very efficient in terms of complexity has proved to be successful on some simple and "short" protocols. Although we have essentially focused on verifying safety properties, liveness properties can be proved by using the completeness property of phase semantics as suggested in [3].

It remains to establish what has been gained vis-a-vis traditional verification methods such as model checking. Also, further investigation should focus on how to improve the efficiency by considering other types of monoids and/or by optimizing the search strategy in the implementation with some kind of heuristic or other methods from AI.

Bibliography

[1] R. Cleaveland. Analyzing Concurrent Systems Using the Concurrency Workbench. in: *Functional Programming, Concurrency, Simulation and Automated Reasoning*. volume 693 of Lecture Notes in Computer Science. P.E. Lauer. editor. Springer-Verlag, 1993. pp. 129–144

[2] F. Fages. P. Ruet. S. Soliman. Linear concurrent constraint programming: operational and phase semantics. 1998.(This is a LIENS research report R98-5. Available from *http://www.ens.fr/ fages/Papers*

[3] F. Fages. P. Ruet. S. Soliman. Phase Semantics and Verification of Concurrent Constraint Programs. *Proc. Logic in Computer Science* (LICS). IEEE Press. 1998. pp. 141-152.

[4] P. B. Galvin. A. Silberschatz. Operating systems concepts. Fourth Edition. Addison- Wesley Publishing Company. 1994.

[5] J.Y. Girard. Linear logic. Theoretical Computer Science. 50(1), 1987. pp. 1–102.

[6] J.Y. Girard. Linear logic: Its syntax and semantics. Advances in linear logic. et al. Eds.. London Math. Soc. Series 222, Cambridge University Press. pp. 1-42.

[7] M.I. Kanovich. The direct simulation of Minsky machines in linear logic. Advances in Linear Logic. Cambridge University Press. pp 123-145. 1995.

[8] H. R. Jervell. Review of two key works on linear logic. Journal of Symbolic Logic, vol. 61, no. 1, 1996, pp 336-338.

[9] M.I. Kanovich, M. Okada, A. Scedrov. Specifying Real-Time Finite-State Systems in Linear Logic. Electronic Notes in Theoretical Computer Science. Vol. 16 Issue 1, 1998, 15 pp.

[10] Y. Lafont. The finite model property for various fragments of linear logic. The Journal of Symbolic Logic. Vol 62. No 4, 1997, pp. 1202-1208.

[11] R. Lidl, H. Niederrater. Introduction to finite fields and their applications. Revised Edition. Cambrigde University Press, 1994.

[12] P. Lincoln, J. Mitchell, A. Scedrov, N. Shankar. Decision problems for propositional linear logic. Annals of Pure and Applied Logic, vol 56, 1992, pp. 239-311.

[13] P. Lincoln, V.A. Saraswat. Higher-order linear concurrent constraint programming. Technical report, Xerox Parc, 1992.

[14] N. Mendler, P. Panangaden, P.J. Scott, R.A.G. Seely. A logical view of concurrent constraint programming. *Nordic Journal of Computing*, 2, 1995, pp. 181-220.

[15] M. Okada. Girard's phase semantics and a higher- order cut-elimination proof. Technical report, Institut de Mathématiques de Luminy, 1994.

[16] K. H. Rosen. Elementary number theory and its applications. Third Edition. Addison-Wesley Publishing Company, 1993.

[17] P. Ruet. Logique non-commutative et programmation concurrente par contraintes. *PhD thesis*, Universite Denis Diderot, Paris VII, 1997.

[18] V. J. Saraswat. Concurrent constraint programming. ACM Doctoral Dissertation Awards. MIT Press, 1993.

[19] A. Scedrov. A Brief Guide to Linear Logic. *Current Trends in Theoretical Computer Science*. World Scientific Publishing Co., 1993.

[20] A. S. Troelstra. Lectures on Linear Logic. CSLI Lecture Notes No. 29. Center for the Study of Languages and Information, Stanford University, 1992.