Models and Languages for Global Computing Transactions

Hernán Melgratti

January 2005

Addr: Largo B. Pontecorvo 3, I-56127 Pisa – Italy
Tel: +39-050-2213146 — Fax: +39-050-2212726 — E-mail: melgratt@di.unipi.it
Web page: http://www.di.unipi.it/~melgratt
Contents

1 Introduction ................................................................. 13
   1.1 Motivations ............................................................ 13
   1.2 Transactional Frameworks ........................................... 16
      1.2.1 Database transactions ...................................... 17
      1.2.2 Transactional workflows .................................. 19
      1.2.3 Committed Choice Languages ............................... 19
      1.2.4 Transition synchronisation ................................ 20
      1.2.5 Process calculi with transactions ......................... 21
   1.3 Main Contributions ................................................... 21
   1.4 Outline of this thesis .............................................. 23
   1.5 Origins of the Chapters ............................................ 26

2 Background ........................................................................ 29
   2.1 Notation and Symbols ................................................ 29
   2.2 Petri Nets ................................................................... 30
      2.2.1 Operational Semantics ...................................... 31
      2.2.2 Deterministic processes .................................... 32
   2.3 The Join calculus ...................................................... 34
      2.3.1 Syntax .................................................................. 34
      2.3.2 The Chemical Abstract Machine ............................ 35
      2.3.3 Operational semantics ........................................ 36
      2.3.4 Abstract semantics ............................................. 36
      2.3.5 High-level Nets as Typed Fragments of Join ............. 37
   2.4 The d2PC protocol ...................................................... 39
      2.4.1 Informal presentation ......................................... 39
      2.4.2 Formal definition of the d2PC ............................... 43

I Adding mobility to ZS nets ................................................. 47

3 Preliminaries on Zero-Safe Nets ........................................ 51
   3.1 From Petri nets to Zero-safe nets .................................. 51
   3.2 Operational Semantics ............................................... 51
3.3 Abstract Semantics .................................. 55

4 Adding colours to ZS nets 59
4.1 Background: Coloured P/T nets .......................... 59

4.1.1 Operational semantics .............................. 62
4.2 Coloured Deterministic Processes ...................... 62
4.3 Correspondence between processes and firings ............ 67
4.4 Coloured ZS nets ................................ 77
4.5 Abstract Semantics ................................ 79
4.6 ZS nets as C-ZS nets ................................ 88
4.7 Related works .................................. 89

5 Reconfigurable ZS nets 91
5.1 Background: Reconfigurable nets ...................... 91

5.1.1 Operational semantics .............................. 92
5.2 Reconfigurable Deterministic Processes .......... 93
5.3 Correspondence between processes and firings ...... 98
5.4 Reconfigurable ZS nets ................................ 105
5.5 Abstract Semantics ................................ 106
5.6 C-ZS nets as R-ZS nets .............................. 110

6 Dynamic ZS nets 111
6.1 Background: Dynamic nets ............................ 111
6.2 Flat Dynamic ZS nets .............................. 116
6.3 No Abstract View for Flat Dynamic ZS nets .......... 118

II Committed Join 123

7 Committed Join 127
7.1 Motivations .................................. 127
7.2 Committed Join .................................. 128

7.2.1 Syntax .................................. 128
7.2.2 Operational Semantics ......................... 129
7.2.3 Abstract Semantics .......................... 133
7.3 Examples ...................................... 133
7.4 Serializability and Big-Step Semantics .......... 139
7.5 Encoding Zero-Safe nets .......................... 143
7.6 Related works .................................. 146

8 Flat cJoin in Join 153
8.1 A type system for flat cJoin ....................... 154

8.1.1 Properties of the type system .................. 155
8.2 A canonical form for flat processes ................ 157
8.3 Writing flat cJoin process in canonical form ............................ 159
8.3.1 The canonical form of $P$ is bisimilar to $P$ ...................... 162
8.4 Compiling flat cJoin into Join .............................................. 180
8.4.1 d2PC with Compensations (d2PCC) ............................... 180
8.4.2 Correctness of the d2PCC ............................................. 184
8.4.3 Translating flat cJoin in Join .......................................... 186
8.5 Extension of JoCaml ......................................................... 199
8.6 Related works ............................................................... 201

III Compensations and Flow Composition 203

9 Compensations and Flow Composition 207
9.1 Motivations ................................................................. 207
9.2 Sequential Sagas .......................................................... 210
9.2.1 Big Step Semantics .................................................... 211
9.2.2 Proof of Adequacy Theorem ....................................... 214
9.3 Parallel Sagas ............................................................. 216
9.3.1 Naïve definition for the semantics of $'|'$ ........................... 217
9.3.2 Parallel Sagas Revised .............................................. 220
9.3.3 Proof of Adequacy Theorems ...................................... 223
9.4 Adding Nesting to Sagas ............................................... 229
9.5 Additional features ....................................................... 233
9.5.1 Programmable compensations .................................... 233
9.5.2 Exception handling .................................................. 234
9.5.3 Alternatives to aborted subtransactions ......................... 235
9.5.4 Choices ................................................................. 236
9.5.5 Link dependencies .................................................... 237
9.6 Implementation issues ................................................. 238

10 Conclusions ................................................................. 239

Bibliography ................................................................. 243
## List of Figures

2.1 Operational semantics of P/T nets. ............................................. 31
2.2 A simple P/T net. ................................................................. 32
2.3 A simple process. ............................................................... 33
2.4 Join Calculus: Syntax. ......................................................... 34
2.5 Defined, received, and free names. ........................................ 34
2.6 Join Calculus: CHAM Semantics. .......................................... 36
2.7 Nets as Typed Fragments in Join ........................................... 38
2.8 D2PC algorithm. ................................................................. 40
2.9 Example of commit. ............................................................. 41
2.10 Example of abort. ............................................................... 42
2.11 The encoding of coordinators. ............................................ 43
2.12 States of coordinators ....................................................... 43
2.13 The hierarchy of transactional nets. ...................................... 50

3.1 Operational semantics of ZS nets. ........................................... 52
3.2 Free choice net for the assignment problem. ............................ 53
3.3 Free choice net for the assignment problem. ............................ 53
3.4 Computations of the free choice net for the assignment problem. . 54
3.5 An instance of the mobile lessees. ......................................... 55
3.6 ZS net of the mobile lessees example. .................................... 55
3.7 A proof for the execution of a transaction in the mobile lessees ZS net. 56
3.8 Abstract net of the mobile lessees example. ............................... 57

4.1 A simple C-P/T net. ............................................................. 62
4.2 A computation of a coloured net. ............................................ 63
4.3 Simple coloured net morphism. .............................................. 65
4.4 C-ZS net of the mobile lessees example. ................................ 78
4.5 Two coloured processes for the mobile lessees example. ............. 80
4.6 Partial abstract net of the C-ZS net of the mobile lessees example. 87
4.7 First level in the hierarchy of transactional nets. ...................... 88

5.1 A simple R-P/T net. ............................................................. 93
5.2 A computation on a reconfigurable net. ................................... 93
5.3 Reconfigurable net morphism. .............................................. 95
5.4 A non-general causal net morphism ........................................ 96
5.5 Two isomorphic reconfigurable processes ............................ 98
5.6 A $r$-$p/t$ morphism using pattern matching ......................... 99
5.7 A $r$-$z$ net for the mailing list example .............................. 107
5.8 A reconfigurable connected transaction $\xi_1$ for the mailing list example .................................................. 108
5.9 A partial view of the abstract net for the mailing list example. . 109
5.10 Second level in the hierarchy of transactional nets .................... 110

6.1 A simple dynamic net ..................................................... 113
6.2 Operational semantics of dynamic nets ............................... 114
6.3 Operational semantics of flat dynamic Sz nets ....................... 117
6.4 Private Mailing Lists .................................................... 118
6.5 Private Mailing Lists after firing $t$ with colours $p_1 = (a_{s_1}, d_{m_1})$ 119
6.6 Private Mailing Lists after firing $t$ with colours $p_2 = (a_{s_2}, d_{m_2})$ 120
6.7 A simple $d$-$z$ net ...................................................... 122

7.1 Syntax of cJoin .......................................................... 129
7.2 Defined, received, and free names for cJoin processes ............... 129
7.3 Operational semantics of cJoin ........................................ 130
7.4 A possible computation of the Mailing list example ................. 135
7.5 Trip booking example .................................................. 136
7.6 Encoding of AKL committed rules ................................... 139
7.7 Big-step semantics of cJoin ........................................... 142
7.8 Proof sketch for Lemma 7.3 .......................................... 143
7.9 Encoding of $z$ nets in cJoin ......................................... 144

8.1 Flat cJoin Typing ....................................................... 155
8.2 Flat Solution Typing .................................................. 156
8.3 Definition of $\text{count}(P)$ ........................................... 157
8.4 Flat Processes in Canonical Form ................................... 158
8.5 Canonical form of flat processes .................................... 159
8.6 Annotated translation ................................................ 163
8.7 States of coordinators ............................................... 181
8.8 Join code of coordinators ......................................... 183
8.9 Encoding of canonical flat processes ................................ 187
8.10 Encoding of canonical flat definitions ............................. 190

9.1 A sequential saga ...................................................... 208
9.2 Semantics of sequential sagas ....................................... 212
9.3 A sequential saga for handling orders ................................ 213
9.4 Naive semantics of parallel composition ............................ 218
9.5 A parallel saga for handling orders .................................. 219
9.6 Semantics of parallel sagas .......................................... 221
9.7 Semantics of nested Sagas .......................................... 230
9.8  Possible executions of $S \equiv \{ \llbracket P \rrbracket \mid A \div B \}$ when $\Gamma = A \leftrightarrow \Xi, \Gamma'$. 231
9.9  A nested saga for handling orders. ................................. 233
9.10 Semantics of exception handling ................................. 235
9.11 Semantics for the choice of a successful branch: $P \boxplus Q$ 236
9.12 A flow with links. ..................................................... 238
Abstract

Transactions are a common coordination problem of global computing applications. Typically, in order to take a final decision, several autonomous components, which are distributed over the web, require to be guaranteed about the successful execution of the others. Consequently, primitives for handling transactional interactions among independent components become essential.

This thesis is aimed at contributing to the development of formal models and languages with primitives for global computing transactions. In particular we concentrate on the two most representative kind of transactions: ACID and compensable.

For the first case, we progressively extend the basic zero safe nets model (i.e., a transactional version of Petri nets) with: (1) the value passing mechanism of coloured nets; (2) the dynamic interconnection mechanism of reconfigurable nets; and (3) the high-order features of dynamic nets. In this way we obtain a model of atomic transactions that are mobile and multiway.

As far as compensable long running transactions is concerned, we present two different approaches, each of them accounts for a particular style for composing applications. For the conversational approach (i.e., when any component declares the ways in which it can be engaged in a larger process) we introduce an extension of the Join calculus with compensable transactions, called committed Join (cJoin). Transactions in cJoin are multiway and can be nested. We show that an important fragment of cJoin (that of non-nested transactions) can be implemented in Join by exploiting a distributed commit protocol.

Finally, for the case in which composition is specified as a flow, i.e., by describing the flow of control between the components, we give a hierarchy of flow composition languages with compensations. This family is defined in a modular way, by starting with a simple core language in which processes are just sequences of compensated activities, and by progressively adding parallel composition, nesting, programmable compensations and exception handling.
Chapter 1

Introduction

1.1 Motivations

Wide area network computing, web programming, and, more generally global computing (GC) have emerged as some of the most important contexts for developing distributed applications. GC applications integrate several components designed, implemented and distributed autonomously, that interact over a computational infrastructure with variable guarantees for communication, security, mobility, resource usage, which, moreover, are not static but change dynamically.

In this context, the development of applications concerns mainly the aggregation of components or services that are autonomous, decentralised, mobile, dynamically reconfigurable and open-ended. Clearly, an ad hoc strategy for integrating services does not scale up well when applications get complex and several non functional aspects are considered. For this reason, primitives for handling interaction patterns among independent components become essential. A common requirement when integrating applications is that of coordinating all involved components so that a dependable outcome can be generated. Consider the typical scenario where a user organises her holidays by accessing several independent services on the web, which allow her to book, e.g., a flight, a hotel and a car. All these tasks can be executed in any order, even concurrently, but all of them must be successfully completed in order to arrange a consistent trip plan. Moreover, it is required that all activities succeed or fail as a whole. More generally, the problem is that of committing the results of long distributed decision processes as soon as the participants reach partial agreements. Note that the problem is not just to coordinate the updates of a distributed repository (e.g., a database), since components are independent, and any of them is responsible for maintaining the consistency on local data. Instead, we also require any component in an interaction to be assured that (i) either all activities have been completed successfully, or (ii) some activity has failed and all the remaining activities in the interaction are aware of this.

A GC transaction (also called negotiation, agreement or contract) is a widely
Atomicity, A transaction’s changes to the state are atomic: either all happen or none happen.

Consistency, A transaction is a correct transformation of the state. The actions taken as a group do not violate any of the integrity constraints associated with the state.

Isolation, Even though transactions execute concurrently, it appears to each transaction, T, that others executed either before T or after T, but not both.

Durability, Once a transaction completes successfully (commits), its changes to the state survive failures.

Table 1.1: ACID properties as defined in [61].

distributed process in which every party either completes successfully or it is made aware of the execution failure.

Although some solutions have been proposed in the literature, still we lack foundational models that accurately explain the crucial aspects of the problem. For this reason, academy and industry are showing a growing interest in the study of programming languages and calculi with well-disciplined transactional mechanisms for managing electronic GC negotiations. Usually, the features of GC programming languages are studied with the help of suitable process description languages (PDL’s) (e.g. π-calculus [89], join calculus [52], spi-calculus [1], ambient calculus [37]), where new primitives can be easily introduced and experimented with. Although the transactional aspect of executions can be seen somehow orthogonal to usual PDL’s operators, and possibly it could be encoded into standard operators, the extension of PDLs with this kind of abstraction is desirable (at least) for the following reasons: (i) to lay the foundations for the formal definition of orchestration languages; (ii) to facilitate the comparison of different semantics; (iii) to reason about the relation among operators; (iv) as the starting point for comparing the expressive power of newly proposed primitives (i.e., whether they can be conveniently defined in term of usual operators or not) and for studying the properties preserved by different encodings; and (v) to give insights about implementation details.

This thesis is aimed at contributing to the development of formal models and languages with primitives for GC transactions.

Clearly, transactional mechanisms are not new, and have been largely studied by several communities in computer science. Originally, the term transaction was coined to designate sequences of operations/actions which preserve database consistency [48]. Traditional database transactions are called ACID (an acronym for atomicity, consistency, isolation and durability, as described in Table 1.1). Nevertheless, many different ACID transaction models have appeared in literature, e.g.,
1.1. MOTIVATIONS

the flat model [48], flat transactions with save-points and chained transactions [61], the nested model [90] (which allows transactions to be part of another transaction), and multi-level transactions [82, 97, 107]. Since ACID transactions are often based on locking mechanisms [10, 77, 49, 61], they are suitable for handling activities with short duration. Later approaches have focused on long-running transactions and have introduced the concept of ad hoc compensations, i.e., user programmed actions that handle anomalous situations (whose most representative proposal is Sagas [55]). Moreover, compensations are the standard mechanism used for managing transactions in workflow systems [100] (i.e., systems that coordinate the execution of multiple tasks according to a precise schedule [96]), and in many recent proposals for web service composition languages (like XML driven technologies BPML [17], XLANG [104], WSFL [81], BPEL4WS [16]).

Following these two mainstreams (i.e., ACID and compensable transactions), and considering that applications may require different flavours of transactions, in this thesis we tackle both kind of transactional models. The first part of this thesis is devoted to the proposal of a suitable scheme of mobile ACID transactions. The model we present is based on the so called dynamic zero-safe nets. They are the mobile version of zero-safe nets (ZS nets), which are themselves an extension of Petri nets with a simple synchronisation mechanism for transitions. The attractive features of ZS nets are that: (i) transactions are multiway, i.e., have multiple entry and exit points, which allows transactions to change dynamically the number of participants; (ii) they provide a two-level description of systems, one concrete, where the coordination mechanism is explicit, and the other abstract, where transactions are represented as atomic activities; and (iii) they can be implemented in an asynchronous name passing calculus (e.g. Join calculus [52]) in a fully distributed way. On the other hand, (name) mobility provides a natural way to model the dynamic interconnection of components.

As far as long running negotiations are concerned, we provide two different approaches depending on the style used for describing composed services. Usually, the approaches presented in literature for specifying composition are either [81]: (i) interaction based composition (also known as conversational patterns or global models) and (ii) flow composition (also known as hierarchical patterns). Conversational based languages are aimed at describing the interaction protocols or patterns that services should follow in order to achieve a specific goal. In this case, any service declares the ways in which it can be engaged in a larger process. Instead, flow composition is reminiscent of workflow systems [57], where a composed service is described by a process that states precisely the flow of both control and data between the component parts.

As a model of conversational transactions, in the second part of this thesis, we propose committed Join (cJoin for short), which is an extension of the Join calculus with high-level primitives for programming compensable negotiations. Transactions in cJoin are processes that execute in a controlled environment until completion, when they commit and make their results observable to the rest of the system.
Alternatively, they can be explicitly aborted, in which case suitable compensation programs are activated to restore a locally consistent state. Transactions in c\textit{Join} are also multiway, in the sense that several negotiations can be merged into a larger transaction if they communicate during their execution. Moreover, negotiations in c\textit{Join} can be nested, i.e. transactions may contain processes that are themselves transactions. Roughly, negotiations in c\textit{Join} may be viewed as the compensable counterpart of the ACID transactions in dynamic ZS nets.

For transactions in flow composition languages, the third part of this thesis introduces a hierarchy of languages for describing compensable flows. We maintain the usual view given by flow composition languages, which describe the interaction of services by giving a precise flow of control between the component parts and by hiding the details about low-level computations performed by components. In a modular way, we formally define the semantics for a family of flow composition languages with compensations. We start by a simple core language, where processes are sequences of compensated activities, and progressively add parallel composition, nesting, programmable compensations and exception handling.

\section{Transactional Frameworks}

In this section we summarise several notions of commitment proposed in different contexts and with diverse motivations that provided us with useful insights about the commit problem. We also explain the origin of several concepts and terms that will be reprise through the different sections of this thesis. First of all, we survey \textbf{database transaction models}, that have originated several interesting notions such as the flat model, nesting, serializability, ACID properties, commit protocols (e.g. the well-known Two Phase Commit). Moreover, several notions were introduced to handle activities with long duration, e.g. compensation, cooperation between transactions, dynamic organisation of transactions.

Then we move to \textbf{Transactional Workflows}, which are intended to orchestrate the execution of several activities. In particular, transactional workflows leave out the assumption of coordinating the update of databases. In this case, the term transactional means that a particular execution satisfies a set of user-defined constraints.

Another slightly different notion of commitment is the notion of \textbf{committed choice} introduced in concurrent logic/constraint programming, meaning that a non-deterministic selection is solved by choosing a particular alternative and discarding all the other possibilities. A remarkable fact is that commit is associated with the notion of \textit{termination}, i.e., the selection of a rule is committed when its guard is proved. It is at commit time that locally computed constraints update global constraints.

On the other hand, the \textbf{zero-safe net model} is based on a distinction between observable and hidden states, and it defines transactions as those computations that
transform observable states into observable states, possibly traversing several hidden states. Moreover, transactions in zero-safe nets are naturally multiway transactions.

Finally, we summarise some approaches for extending process description languages with transactions.

1.2.1 Database transactions

Database transactions are sequences of operations that preserve the consistency of data [48], i.e., the execution of a transaction on a consistent state is guaranteed to produce a consistent state, independently of the activities that are being executed concurrently. This property is also known as serializability, i.e., any concurrent execution $T_1 \| \ldots \| T_n$ that interleaves the steps of any ACID transactions $T_i$ is serializable if there exists a sequence $T_{i_1}; T_{i_2}; \ldots; T_{i_n}$ that executes all transactions one at a time (without interleaving steps from different transactions) and that produces the same result [10].

Many different models were proposed in order to meet ACID properties and to assure serializable executions. In the basic flat model [48], all actions in a transaction are committed at the same time, and all executed actions are rolled back when the transaction aborts. Due to its simplicity, the flat model has many limitations that have been largely discussed in the literature [60]. The main drawback of flat transactions is that they are implemented by using locking mechanisms, which prevent concurrent transactions to access objects simultaneously [10, 77, 49, 61]. Consequently, their execution may suffer a considerable delay while waiting for others transactions to commit.

Several extensions of the flat model, like flat transactions with save-points and chained transactions, have been proposed to handle partial failures [61, 43], i.e., to allow transactions to be rolled back to a previous internal state and restart the execution from that point instead of being aborted. A more general approach is introduced by the nested model [90], where transactions can be decomposed into a hierarchy of sub-transactions. The root of the hierarchy is usually referred to as the top-level transaction. In this scheme, any sub-transaction executes atomically and concurrently with respect to its parent and siblings, deciding freely to commit or abort. All changes done by a sub-transaction become visible to its parent transaction upon the sub-transaction’s commit, nevertheless if the parent aborts all its sub-transactions are aborted and consequently rolled back.

A different proposal based on a hierarchical organisation of activities is the multi-level model [82, 97, 107], where any level collects operations at a particular level of abstraction. Operations at level $n$ are implemented by using operations of the lower layer. Moreover, this scheme assumes that any operation $o$ has an inverse operation $o^{-1}$ that can reverse semantically all the effects of $o$, called the perfect compensation.

Since ACID transactions are regarded as not suitable for dealing with long running activities, alternative models leave out the ACID properties. For instance, in Sagas [55], a long lived transaction is a sequence of ACID sub-transactions (called
steps) that can be interleaved in any way with other transactions. Additionally, any step has a compensating activity. Any partial execution of a saga is undesirable, and if it occurs, it must be compensated for. A saga of the form \( T_1, \ldots, T_n \) where each \( T_i \) has compensation \( C_i \) is guaranteed to execute either \( T_1, \ldots, T_n \) or \( T_1, \ldots, T_j, C_j, \ldots, C_1 \) for some \( j \leq n \). Sagas is not aimed at satisfying ACID properties, hence compensations are not required to semantically reverse the original transactions, as in the multi-level model. In this case, they are appropriate actions to handle anomalous situations. Moreover, Sagas are just a particular case of Open Nested transaction, which are a relaxed version of multi-level transactions where perfect compensations are substituted by programmed compensations.

Several other models are aimed at dealing with open-ended activities [74], i.e., tasks that are long running, and whose actions are unforeseeable at the beginning of the task. These models usually provide particular operations over transactions, such as split or join [70], or flexible ways of defining dependencies [47]. Moreover, many further models were proposed by combining or extending notions such as nesting, hierarchical composition, compensation, dynamic composition of transactions. Some of them are intended to fulfil ACID properties while increasing the concurrency of activities on the database. Others deal with relaxed forms of ACID properties that make transactions suitable for handling requirements such as long-running activities, sharing of partial results between cooperating activities, and dynamic dependencies between transactions. Most of them have remained just as academic proposals suitable for handling specific situations.

An interesting problem introduced by distributed databases is that of distributed transactions. That is, a transaction that updates a distributed data repository is divided into several parts, any of which modifies a specific portion of the global repository. In this case, the whole transaction can commit only when all local updates have been done. The distributed model relies on the well-known Two Phase Commit Protocol (2PC) to achieve the agreement, where a Distributed Transaction Coordinator (DTC) running on a server machine is responsible for coordinating all components. Roughly, the 2PC works as follows [61]:

1. the DTC sends a vote request to all participants;
2. upon vote request, each participant either votes no and aborts, or votes yes;
3. the DTC collects all votes. If all votes are yes, then the DTC sends commit to all participants. Otherwise, the DTC sends abort to all participants that voted yes;
4. each participant that voted yes waits for DTC response and behaves accordingly.

Several variants of the 2PC protocol have appeared in literature. Notably, the decentralised 2PC, the the three phase commit (3PC) [101], and the distributed two phase commit (D2PC) citeBLM:OTJC.
1.2.2 Transactional workflows

A workflow is an activity that requires the coordinated execution of multiple tasks performed by different processing entities according to a precise (possibly concurrent) schedule [96]. A software system responsible for coordinating the execution of a workflow is called workflow management system (WMS). Generally, workflows involve several independent applications and database systems together with special requirements related to the correctness of computations and data integrity. Primitive workflow systems were the Job Control Languages (JCL) of batch operating systems, which allowed the organisation of the execution of several programs. Basically a JCL script states the sequence in which a set of programs must be executed, also specifying how the outputs of an application become the inputs of others. Modern WMS support more sophisticated descriptions of workflows. Typically, WMS provide tools to describe workflows as the composition of several activities, which are generally implemented by some application.

Basically, given a workflow description, a WMS schedules the execution of activities by invoking applications, providing the appropriate data, and gathering the outputs produced at each step (see [57, 96] for detailed description of WMS).

As workflow complexity increases, several correctness aspects must be addressed, for instance, what does it happen when a task fails? Should the workflow continue or abort? Are the effects of the activities executed previously still valid? Should they be undone? Transactional workflows provide a robust process execution and a clear semantics for dealing with these problems [75]. In this context “transactional” means coordinated execution of multiple related tasks that require access to several heterogeneous, autonomous and distributed systems. In transactional workflows (like Contracts [94], METEOR [78] or METEOR2 [76]) a script (i.e., a process) describes the execution flow of atomic activities. Such flows may specify conditional behaviours depending on whether a step ends successfully or not. Additionally, WMS allows for the definition of compensation mechanisms to recover from partial executions.

We remark that all the flow composition languages for web services (like BPML [17], XLANG [104], WSFL [81], BPEL4WS [16]) are modern versions of workflow systems.

1.2.3 Committed Choice Languages

Logic programs can be viewed as non-deterministic transition systems in which program rules define the transitions of the system [99]. Concrete implementations, such as Prolog, accomplish non-determinism by searching for all solutions, attempting, in this way, all possible computations. Sequential Prolog explores the search tree in a depth-first left-to-right fashion, applying just one program rule in each step. Differently, parallel implementations were developed (see for instance PARLOG [41] and Concurrent Prolog [98]) to optimise the exhaustive search by executing steps in parallel. Then, successful computations are selected (known as Don't-know non-determinism).
Instead, in concurrent logic / constraint programming the alternatives to explore are selected in advance (known as Don’t-care non-determinism or Indeterminism). Consequently, concurrent languages provide mechanisms to control the choices, in particular guards. A guard is a condition associated to a program rule, which is tested before selecting such alternative. Program rules are usually given as guarded Horn clauses of the form

\[ \text{Head} \leftarrow \text{Guard} \mid \text{Body} \]

where Head and Body are defined as in ordinary Horn clauses; and ‘|’ is the commit operator.

A guarded clause can be chosen to reduce a goal if and only if the conditions specified by the head (e.g., successful unification) and the guard are satisfied. Once a guarded clause is applied, all the other alternatives are pruned and the computation is committed.

In concurrent logic programming, the commitment is defined in terms of successful termination. In fact, the selection of a rule can be committed only when its guard is proved, i.e., when the computation of the guard has finished successfully. And it is at commit time that all restrictions associated with the head of the selected clause and those computed while proving the corresponding guard are applied to the rest of the goal.

1.2.4 Transition synchronisation

Zero-Safe nets (ZS nets) [28] extend place/transition Petri nets (P\(\tau\) nets) by providing a built-in mechanism for synchronising transitions. Transactions are described as the coordinated executions of several transitions, and in particular the commit of a transaction synchronises the end of all participant transitions. Anticipating from Section 2.2 some basics on P\(\tau\) nets, places are repositories of resources (tokens), and transitions fetch and produce tokens. Net configurations, called markings, are multisets of places, recording the number of tokens available in each place.

The places of zero-safe nets are partitioned into ordinary and transactional ones (called stable and zero, respectively). Correspondingly, markings \(U\) can be seen as pairs \((S, Z)\) with \(U = S + Z\), where \(S\) and \(Z\) are the multisets of stable and zero resources, respectively. A transaction goes from a multiset of stable places (stable marking) to another stable marking. The key point is that stable tokens produced during a transaction are made available only at commit time, when no zero tokens are left.

An interesting aspect of ZS nets is that for any ZS net it is possible to build an ordinary net (called abstract net), whose places are the stable places of the ZS net and whose transitions are the atomic moves of the original ZS net.

Moreover, based on the ZS mechanism, a general notion of distributed transaction was introduced in [21], where transactions retain several entry and exit points,
and admit a number of participants which is statically unknown. Such transactions can be implemented in a fully distributed way by using a *distributed two phase commit* (d2PC) presented in [21]. The model was proposed for extending the transactional features of Biztalk [95, 80], which is a commercial WMS.

### 1.2.5 Process calculi with transactions

In the last few years some proposals for extending process calculi with transactions have appeared in the literature. We give here a short overview of the most representative approaches, while a more detailed discussion is postponed to Section 7.6.

The data-driven coordination language Linda [56] has been extended in [2, 33, 34, 71] with transaction primitives that resemble the usual database transactions. In particular, transactions are defined by putting a process inside a transactional context (usually by using new prefixes like *start* and *commit*). The intended meaning is that the process should execute until completion in isolation. Only when it reaches the commit, the produced data are made available to the rest of the system.

A different approach is taken by TraLinda [29], which is based on the ZS approach and relies on a typing mechanism for partitioning messages into two classes, *low-level* and *high-level*. Low-level messages are used during the execution of a transaction, while high-level messages are observable only when a transaction commits.

Several proposals extend the π-calculus [88] with compensable processes (for instance [14, 13]). Roughly, any long running transaction is described as process $P$ that has an associated compensation $Q$. In general $P$ can execute (possibly by interacting with the environment) until either it is completed (in which case the compensation is discarded) or aborts by activating $Q$. A version extended with timed transactions, called webπ, has been proposed in [79].

A different approach is taken by StAC [35], a CSP-like [66] language, where processes install compensations to be executed in the reverse order w.r.t. that of completion of original activities.

### 1.3 Main Contributions

The work in this thesis is aimed at providing formal models and languages for describing and reasoning about GC applications that require transactional capabilities. In this dissertation we provide three different kinds of models:

- A model for mobile multiway ACID transactions (*FIRST PART*).
- A calculus of interaction-driven compensable transactional processes (*SECOND PART*).
- A hierarchy of compensable flow compositions calculi (*THIRD PART*).
This thesis has been organised accordingly in three different parts. In the **FIRST PART** we enrich the ZS model to be able to express mobile multiway ACID transactions. The model is obtained by combining two orthogonal proposals: (i) the ZS approach, and (ii) dynamic high-level nets (a mobile version of Petri nets). The former accounts for flat multiway atomic transactions, the latter allows mobility. Since high-level nets can be obtained from Petri nets by successively adding colours, reconfigurable capabilities, and element creation, we analogously enriched the basic ZS model giving birth to:

- a model for multiway atomic transactions with value passing (Chapter 4);
- a model for reconfigurable multiway atomic transactions (Chapter 5);
- a model for multiway atomic transactions with mobility (Chapter 6).

For the first two extensions, we still have the two level descriptions of the system like in the basic ZS model, in which each concrete net has a correspondent abstract net that associates a single transition to each transaction in the concrete counterpart. We remark that in both cases the abstract net represents with a unique parametric transition all those transactions that differ only on the values of parameters. Differently, we show that the construction of the abstract net cannot be built for dynamics ZS nets, since dynamic ZS transactions allow for the description of nets that can produce new transitions fetching tokens from already existent places (a feature not available in ordinary dynamic nets).

Moreover, we prove that any model in the hierarchy is an extension of the previous one, in the sense that, e.g. the basic ZS model is a particular case of the coloured model, and so on. Furthermore, the construction of the abstract view obtained at one level coincides with the construction produced by the upper levels, i.e., the abstract net of a basic ZS net is isomorphic to the abstract net built as a coloured net.

A side contribution of the **FIRST PART** is the definition of suitable notions for deterministic processes of coloured and reconfigurable nets. The most important feature of our definitions is that a single deterministic process is a pattern that describes several concrete computations, which differ only on the value associated to the involved tokens. We propose a novel definition of process, given in term of variables for representing the values of the computation and of a constraint over those variables (actually a unification).

The **SECOND PART** proposes a formal conversational language, called cJoin, for describing long running negotiations while retaining all the characteristics of a name passing calculus. In general, since cJoin follows a conversational style for specifying composition, there is no global description of the complete behaviour of a transaction. Instead, each participant to a negotiation describes the role it plays in it. Then, the interaction (i.e., communication) mechanism, assures that all cooperating transactions reach the same decision. Additionally, negotiations in cJoin can be nested,
1.4. OUTLINE OF THIS THESIS

which allows transactions to be described recursively in terms of sub-transactions. Moreover, we show that cJoin is expressive enough to encode the models proposed in the FIRST PART.

For a fragment of cJoin, called shallow, we show that it is possible to reason about processes at different levels of abstraction. This property is analogous to the usual requirement of serializability, which allows to consider the execution of one transaction at a time. We show that the operational semantics of cJoin guarantees the serializability of shallow processes.

An important contribution of the SECOND PART is that a significant sub-calculus of cJoin, i.e. that of flat transactions, can be implemented in Join. We characterise flat cJoin as a typed fragment of cJoin, and we prove that the whole fragment can be written by using only a few definition patterns (called the canonical form). Then, we show that negotiations in canonical flat cJoin can be implemented as distributed agreements in Join. Finally, we propose a prototype extension of the programming language JoCaml with the primitives of flat cJoin.

The THIRD PART studies primitives for long running transactions in flow composition languages, and in particular in structured control flows, i.e., flows defined in terms of a fixed set of primitives, like sequencing and branching. We provide a formal semantics for a hierarchy of transactional languages starting from a very small language in which activities can only be composed sequentially. Then, we progressively introduce parallel composition, nesting, programmable compensations and exception handling. One of the main features of our approach is that it provides a nice abstract level description of a compensable flow, without taking into account the low-level computations performed by individual services, i.e. at the composition level it is enough to know whether a service executes successfully or fails. Moreover, the incremental definition of the language makes clear the relation among the different operators, which have a neat semantics (in the sense that they preserve the most common uses).

1.4 Outline of this thesis

In Chapter 2 we introduce some technical background that is essential to understand the notation and the formal content of this thesis. Nevertheless, the presentation of more specific background topics is postponed to the chapter where they are needed (e.g., the presentation of zero-safe nets is in Chapter 3). More precisely:

- Section 2.1 introduces some basic notation and symbols that will be used through this thesis.

- In Section 2.2 we recall the basics of Petri nets. In particular, their operational semantics and the characterisation of their concurrent semantics in terms of deterministic processes.
• In Section 2.3 we describe the syntax and operational semantics (in the CHAM style) of the Join calculus. We also give the definition of its abstract semantics in terms of barbed bisimilarity: the equivalence we will use in the second part of this thesis to compare processes. Finally, we sketch the typing framework proposed in [31], where a hierarchy of nets (of increasing expressiveness) is defined in terms of typeable fragments of the Join calculus.

• Section 2.4 gives an informal presentation of the Distributed two phase commit (D2PC), and its formalisation in the Join calculus, as presented in [21]. We will use the D2PC for implementing transaction primitives in Section 8.4.

The original contributions of this thesis are organised in three different parts. At the beginning of each part, a short roadmap describes the contents and the main results. Although this introduces some redundancy in the presentation, it should facilitate a separate reading of the parts.

In the first part (Adding Mobility to ZS Nets) we extend the basic ZS model to account for name passing, reconfigurability and dynamic creation of components. In this part we are interested on characterising both the operational and abstract semantics of the proposed models. The content of this part makes use of concepts introduced in Section 2.2 and 2.3.5.

The first part is organised in four chapters.

• In Chapter 3 we review the fundamentals of the ZS model proposed in [27]. In particular, Section 3.1 introduces ZS nets, while their operational semantics is reported in Section 3.2. Finally, Section 3.3 summarises the abstract semantics for the individual token philosophy.

• Chapter 4 presents the extension of the basic ZS model with parameter passing capabilities and a basic pattern matching mechanism. The model is obtained by applying the ZS approach to coloured nets (C-P/T nets). Section 4.1 gives the definition of C-P/T nets, which is a variant of those proposed in [4].

In Section 4.2 we give a novel definition for deterministic processes of C-P/T nets, called coloured deterministic processes, which exploits the notion of substitution that takes place while computing when formal parameters are replaced by actual parameter. The rather technical Section 4.3 shows the relation between coloured processes and the operational semantics of C-P/T nets.

Coloured ZS nets are presented in Section 4.4, while their abstract semantics is in Section 4.5. We show in Section 4.6 that C-P/T nets are a generalisation of the basic model by proving that both ZS nets and their abstract nets can be recovered in the coloured setting.

Finally, we conclude in Section 4.7 by comparing the contents of the chapter with related works.
1.4. OUTLINE OF THIS THESIS

- Chapter 5 is organised along the lines of Chapter 4. It starts by presenting reconfigurable nets (Section 5.1) and proposing a suitable notion for reconfigurable deterministic processes (Section 5.2). The correspondence between the operational semantics of reconfigurable nets and of processes is formally stated in Section 5.3.

Reconfigurable ZS nets are introduced in Section 5.4, while their abstract semantics is given in Section 5.5.

The chapter finishes by showing that both the concrete and the abstract view of reconfigurable nets are generalisations of the coloured model (Section 5.6).

- In Chapter 6 we deal with dynamic nets, which are described in Section 6.1. The extension of dynamic nets with ZS flat transactions is presented in Section 6.2. The chapter finishes by illustrating the difficulties of giving the characterisation of the abstract net corresponding to a dynamic ZS net (Section 6.3).

Different examples shows the main features of any transactional model in the hierarchy.

In the second part (Committed Join) we propose a linguistic extension of the Join calculus, called Committed Join (cJoin). It provides primitives for handling long running negotiations. We discuss its expressiveness on the basis of a few examples and encodings. Finally, we show that a significant fragment of cJoin can be implemented in Join.

The second part is divided in two chapters.

- Chapter 7 introduces the cJoin calculus as an extension of Join. The background needed for reading this chapter is presented in Section 2.3.

We discuss in Section 7.1 the motivations and principles that guided the choice of negotiation primitives included in cJoin. Then, cJoin syntax and operational semantics (given in the CHAM style) is presented in Section 7.2, while Section 7.3 presents three different examples that illustrate the main features of cJoin.

In Section 7.4 we study the serializability property for a sub-calculus of cJoin, called shallow. In particular, we give a big-step semantics for shallow processes and we show that shallow processes are serializable by proving a correspondence between their CHAM and big-step semantics.

Finally, Section 7.5 shows that the ZS model (and some of the extended versions presented in FIRST PART) can be encoded in cJoin.

- In Chapter 8 we show that a significant fragment of cJoin consisting of processes whose negotiations cannot be nested (called flat cJoin), can be encoded in the ordinary Join calculus. We first define a type system that singles out flat
processes and we prove subject reduction for it (Section 8.1). Then, Section 8.2 shows that flat cJoin processes can be written in an equivalent canonical form, where a few elementary definition patterns are used. The result in this section is quite technical and makes intensive use of the abstract semantics of Join, summarised in Section 2.3.

Section 8.4 gives an implementation of canonical flat processes in Join, where negotiation primitives are encoded as fully distributed agreements. Hence, Section 8.4.1 describes the commit protocol used by the encoding, which is a variant of the D2PC protocol described in Section 2.4. The actual encoding and the proofs of its correctness and completeness are presented in Section 8.4.3.

Finally, we illustrate (in Section 8.5) how the proposed encoding can be used to extend running implementations of Join. In particular, we describe our prototype extension of JoCaml.

The third part (Compensations and Flow Composition) presents a hierarchy of calculi for specifying compensable flow of composed services. We start from a very small language in which activities can only be composed sequentially. Then, we progressively introduce parallel composition, nesting, programmable compensations and exception handling. A running example illustrates the main features of each calculus in the hierarchy.

The third part consists of Chapter 9. Section 9.1 gives the motivations for formalising compensable flow composition languages and discusses previous approaches in literature (for convinience we presenting our approach, we anticipate the presentation of related approaches to the first section of the chapter). In Section 9.2 we present the core language that allows only the sequential composition of compensated activities. In Section 9.3 we extend the core language with parallel composition, and in Section 9.4 we add nesting. In Section 9.5 we discuss how the language with nesting sagas can be further extended with additional features like programmable compensations, exception handling, choices, and dependencies. In Section 9.6 we mention some implementation issues about flow composition languages.

In the conclusions we summarise the main results and techniques used in this thesis, and sketch possible future lines of research.

1.5 Origins of the Chapters

Many results of this thesis have been already presented in some conferences and some of the have been already published in preliminary form. Although some improvements have been done w.r.t. published material, and many concepts have been extended or refined, the basic ideas behind the results remain the same. Hence, we give here a list of pointers to their initial versions.
• **THE EXTENSIONS OF ZS NETS** with colours, reconfigurable transitions, and dynamic creation of components have been published in the LNCS Springer series on Advances in Petri Nets [22].

• **CJOIN** has been introduced in the Proceedings of IFIP International conference on Theoretical Computer Science (IFIP-TCS 2004) [24], together with the characterisation of serializability for shallow processes. The encoding of ZS nets has also appeared in [24].

• **IMPLEMENTATION OF FLAT CJOIN IN JOIN** has been reported in the ENTCS Proceedings of the Final Workshop of the Italian MIUR Project CoMeta [23]. An initial coding of the commit protocol in JoCaml has been appeared in the ENTCS Proceedings of the International Workshop FOCLASA 2004 [6].

• **FOUNDATIONS FOR COMPENSATIONS IN FLOW COMPOSITION LANGUAGES** have been published in the ACM Proceedings of the International Symposium POPL 2005 [25].
Chapter 2

Background

2.1 Notation and Symbols

This section summarises the the notations that will be used through the different
parts of this thesis. More specific notation will be introduced as they are needed.

Sets. We use the usual notation for sets and operations over sets. In order to
avoid clumsy parenthesised expressions we will write singletons \{a\} simply as a. In
addition, we write \(|S|\) for denoting the cardinality of \(S\), and \(S_1 \cup S_2\) for the union
of the disjoint sets \(S_1\) and \(S_2\). Given a set \(S\), \(\wp(S)\) denotes the powerset of \(S\), while
\(\wp_f(S) \subseteq \wp(S)\) stands for the set of all \textit{finite} subsets of \(S\).

Given two naturals \(n_1 \leq n_2\), we let \(n_1..n_2 = \{n \in \mathbb{N} \land n_1 \leq n \leq n_2\}\).

Functions. We will note a total function \(f\) from \(A\) to \(B\) as \(f : A \to B\), while
\(f : A \rightharpoonup B\) stands for a partial function. Moreover, we define

\[
dom(f) = \{a \in A | f(a) \text{ is defined} \}
\]

Given a partial function \(f\), \(f(a) = n\) means that \(f(a)\) is defined and has value \(n\).

Usually, we write a finite domain function \(f\) as \(\{a_1 \mapsto f(a_1), \ldots, a_n \mapsto f(a_n)\}\). The union of two functions \(f_1\) and \(f_2\) with disjoint domains is indicated by \(f_1 \cup f_2\). The usual composition of functions is denoted by \(f_1 \circ f_2\), where \((f_1 \circ f_2)(x) = f_2(f_1(x))\).

Relations. Given two relations \(R_1 \subseteq A \times B\) and \(R_2 \subseteq B \times C\), then the composition
of \(R_1\) and \(R_2\) is \(R_1;R_2 = \{(a,c) | (a,b) \in R_1 \land (b,c) \in R_2\}\). Given a binary relation
\(R \subseteq A \times A\), then \(R^0 = \{(a,a) | a \in A\}, R^{n+1} = R; R^n\). Moreover \(R^*\) is the reflexive
and transitive closure of \(R\) (i.e. \(R^* = \cup_{i \in \mathbb{N}} R^i\)). As usual \(R^{-1}\) stands for the inverse
of \(R\).
Substitutions. Given a (possibly infinite) set of names \( \mathcal{N} \), a substitution on \( \mathcal{N} \) is a function \( \sigma : \mathcal{N} \to \mathcal{N} \) such that \( \{ a \in \mathcal{N} | \sigma(a) \neq a \} \) is finite, whereas for infinitely many names \( \sigma \) behaves as the identity. This definition allows us a compact notation for substitutions, where \( \{ b_1/a_1, \ldots, b_n/a_n \} \) denotes the substitution that replaces \( a_i \) with \( b_i \) for \( i \in 1..n \), and it is the identity otherwise.

Multisets. Given a set \( S \), a multiset over \( S \) is a function \( m : S \to \mathbb{N} \). Let \( \text{dom}(m) = \{ s \in S | m(s) > 0 \} \). The set of all finite multisets (i.e., with finite domain) over \( S \) is written \( \mathcal{M}_S \). The empty multiset (i.e., with \( \text{dom}(m) = \emptyset \)) is written \( \emptyset \). The multiset union \( \oplus \) is defined as \( (m_1 \oplus m_2)(s) = m_1(s) + m_2(s) \) for any \( s \in S \). Given a multiset \( m \), \( |m| = \sum_{s \in S} m(s) \) denotes the size of \( m \).

Note that \( \oplus \) is associative and commutative, and \( \emptyset \) is the identity for \( \oplus \). Hence, \( \mathcal{M}_S \) is the free commutative monoid \( S^\oplus \) over \( S \). We write \( s \) for a singleton multiset \( m \) such that \( \text{dom}(s) = \{ s \} \) and \( m(s) = 1 \). Moreover, we write \( \{ s_1, \ldots, s_n \} \) for \( s_1 \oplus \ldots \oplus s_n \).

By abusing notation we will apply functions (i.e., \( f_S \)) over (multi)sets, meaning the multiset obtained by applying the function element-wise: \( f_S(\{ a_0, \ldots, a_n \}) = f_S(a_0) \oplus \ldots \oplus f_S(a_n) \).

### 2.2 Petri Nets

Several versions where proposed in literature since Petri nets were introduced [91]. In this section we report the basic model, while some extensions are discussed informally in Section 2.3.5 and with full details presented in Chapter 3 and in the background sections of Chapters 4, 5 and 6.

In Petri nets, places are repositories of tokens (i.e., resources, messages) and transitions fetch and produce tokens (i.e., instances of resources). We consider an infinite set \( \mathcal{P} \) of resource names.

**Definition 2.1 (Net).** A net \( N \) is a 4-tuple \( N = (S_N, T_N, \delta_0N, \delta_1N) \) where \( S_N \subseteq \mathcal{P} \) is the (nonempty) set of places, \( a, a', \ldots, T_N \) is the set of transitions, \( t, t', \ldots \) (with \( S_N \cap T_N = \emptyset \)), and the functions \( \delta_0N, \delta_1N : T_N \to \wp(S_N) \) assign respectively, source and target to each transition.

We denote \( S_N \cup T_N \) by \( N \), and omit subscript \( N \) whenever no confusion arises. We abbreviate a transition \( t \in T \) such that \( \delta_0(t) = s_1 \) and \( \delta_1(t) = s_2 \) as \( s_1 \rightarrow s_2 \), where \( s_1 \) is the preset of \( t \) (written \( \bullet t \)) and \( s_2 \) is the postset of \( t \) (written \( t^* \)). Similarly for any place \( a \) in \( S \), the preset of \( a \) (written \( \bullet a \)) denotes the set of all transitions of which \( a \) is target (i.e., \( \bullet a = \{ t | a \in t^* \} \)), and the postset of \( a \) (written \( a^* \)) denotes the set of all transitions of which \( a \) is source (i.e., \( a^* = \{ t | a \in \bullet t \} \)). Moreover, let \( \mathsf{N} = \{ x \in N | \bullet x = \emptyset \} \) and \( \mathsf{N} = \{ x \in N | x^* = \emptyset \} \) denote the sets of initial and final elements of \( N \) respectively. A place \( a \) is said to be isolated if \( \bullet a \cup a^* = \emptyset \).
Remark. We consider only nets whose transitions have a non-empty preset, i.e. nets $N$ where $\delta N \subseteq S$.

Note that the target and the source of a transition are sets of places, and thus transitions can consume and produce at most one token in each state. More generally, a transition in $p/t$ nets can fetch and produce several tokens in a particular place, i.e., the pre- and postsets of transitions are multisets instead of sets.

Definition 2.2 ($p/t$ net). A marked place / transition Petri net ($p/t$ net) is a tuple $N = (S_N, T_N, \delta _{0N}, \delta _{1N}, m_{0N})$ where $S_N \subseteq P$ is a set of places, $T_N$ is a set of transitions, the functions $\delta _{0N}, \delta _{1N} : T_N \rightarrow M_{S_N}$ assign respectively, source and target to each transition, and $m_{0N} \in M_{S_N}$ is the initial marking.

The notions of pre- and postset, initial and final elements, and isolated places are straightforwardly extended to consider multisets instead of sets.

### 2.2.1 Operational Semantics

The operational semantics of $p/t$ nets is given by (the least relation inductively generated by) the inference rules in Figure 2.1. Given a net $N$, the proof for $m \rightarrow_T m'$ means that a marking $m$ evolves to $m'$ under a step, i.e., the concurrent firing of several transitions. We omit the subscript $T$ whenever the set of transitions is clear from the context. Rule (Firing) describes the evolution of the state of a net (represented by the marking $m \oplus m''$) by applying a transition $m[\rightarrow] m'$, which consumes the tokens $m$ corresponding to its preset and produces the tokens $m'$ corresponding to its postset. The multiset $m''$ represents idle resources, i.e. the tokens that persist during the evolution. Rule (Step) stands for the parallel composition of computations, meaning that several transitions can be applied in parallel as far as there are enough tokens to fire all of them. The sequential composition of computations is the reflexive and transitive closure of $\rightarrow$, which is written $\rightarrow^*$, i.e. $m \rightarrow^* m'$ denotes the evolution of $m$ to $m'$ under a (possibly empty) sequence of steps.

Example 2.1 (A simple $p/t$ net). Let $N$ be a $p/t$ net s.t. $S = \{a,b,c,d\}$, $T = \{t_1, t_2\}$, $\delta _0(t_1) = \delta _0(t_2) = \{a, b\}$, $\delta _1(t_1) = \{c\}$, $\delta _1(t_2) = \{d\}$, $m_0 = a \oplus a \oplus b \oplus b$. Figure 2.2(a) shows the graphical representation of $N$. As usual, places are represented with circles, transitions with boxes, tokens with black bullets, and the pre- and postset functions are represented with directed arcs. Figure 2.2(b) shows
a possible computation in $N$ for the marking $a \oplus a \oplus b \oplus b$, which corresponds to the concurrent firing of $t_1$ and $t_2$.

### 2.2.2 Deterministic processes

There are two different schools of thought for defining the concurrent semantics of Petri nets well summarised in [58]: (i) the *collective token philosophy* (CTph), and (ii) the *individual token philosophy* (ITph). The net semantics under the CTph does not distinguish among different instances of the idealised resources (i.e., tokens). This is a valid interpretation of the behaviour of a system only when any such instance is operationally equivalent to all the others. Nevertheless, tokens may have different origins and histories, thus the *causality* information carried on by different tokens is disregarded when identifying equivalent computations w.r.t. CTph, which turns to be the main drawback of this approach. Alternatively, the ITph takes into account the causal dependencies arising in concurrent executions.

In particular, the distinction between tokens with different origins and history relies on the notion of deterministic processes. A deterministic process represents a set of causal equivalent computations in a net, where the causal dependency between firings is made explicit [59]. The definition of processes relies on two other concepts: net morphisms and causal nets.

**Definition 2.3** (Net morphisms). Let $N, N'$ be $\pi/T$ nets. A pair $f = (f_S : S_N \rightarrow S_{N'}, f_T : T_N \rightarrow T_{N'})$ is a *net morphism* from $N$ to $N'$ (written $f : N \rightarrow N'$) if $f_S(\delta_i(t)) = \delta_i(f_T(t))$ for any $t$ and $i = 0, 1$. Moreover, $N$ and $N'$ are said *isomorphic*, and thus equivalent, if $f$ is bijective. Given two morphisms $f_1 : N \rightarrow N'$ and $f_2 : N' \rightarrow N''$, then composition of $f_1$ and $f_2$ is the morphism $f_1; f_2 = (f_S : S_N \rightarrow S_{N''}, f_T : T_N \rightarrow T_{N''})$, where $f_S = f_{1S}; f_{2S}$ and $f_T = f_{1T}; f_{2T}$.

In what follows we usually omit subscripts when they are clear from the context.

**Definition 2.4** (Causal Net). A net $K = (S_K, T_K, \delta_0K, \delta_1K)$ is a *causal net* (also called *deterministic occurrence net*) if it is acyclic and $\forall t_0 \neq t_1 \in T_K : \delta_iK(t_0) \cap \delta_iK(t_1) = \emptyset$, for $i = 0, 1$. 

![Figure 2.2: A simple $\pi/T$ net.](image-url)
**Definition 2.5** (Deterministic Causal Process). A *deterministic causal process* for a P/T net $N$ is a net morphism $P$ from a causal net $K$ to $N$.

**Definition 2.6** (Process isomorphism). Two processes $P_1 : K_1 \to N$ and $P_2 : K_2 \to N$ of the same net $N$ are *isomorphic* (written $P_1 \approx P_2$) and thus equivalent if there exists a net isomorphism $\psi : K_1 \to K_2$ such that $\psi; P_2 = P_1$. We denote the equivalence class of $P$ as $[P]_\approx$.

**Notation 2.1.** Given a process $P : K \to N$, the set of origins and destinations of $P$ are defined as $O(P) = \circ K$ and $D(P) = K^c \cap S_K$, respectively. We write $pre(P)$ and $post(P)$ for the multisets denoting the initial and final markings of the process, i.e. $pre(P) = P(O(P))$ and $post(P) = P(D(P))$. Moreover, as isomorphisms respect initial and final markings, $pre(P) = pre(P')$ and $post(P) = post(P')$ for any $P \approx P'$, thus we say that $O([P]_\approx) = pre(P)$ and $D([P]_\approx) = post(P)$.

We refer to the set of places that are neither initial nor final as the *evolution places* of a process $P$.

**Definition 2.7** (Evolution places). Let $P$ be a process, the set of *evolution places* of $P$ is defined as $E_P = \{P(a)|a \in K,|\bullet a| = |a^*| = 1\}$.

The following definition identifies those processes that describe computations that cannot be split as two unrelated computations.

**Definition 2.8** (Connected Process). A deterministic process $P : K \to N$ is a *connected process* if $T_K$ is non-empty, and for all $t_0, t_1 \in T_K$ there exists an undirected path connecting $t_0$ and $t_1$ (i.e., an undirected path in the graph induced by the flow functions $\delta_{0N_K}$ and $\delta_{1N_K}$).
\[(\text{PROC})\] \[P, Q ::= 0 \mid x \langle \bar{y} \rangle \mid \text{def } D \text{ in } P \mid P|Q\]

\[(\text{DEF})\] \[D, E ::= J \triangleright P \mid D \land E\]

\[(\text{PAT})\] \[J, K ::= x \langle \bar{y} \rangle \mid J|K\]

Figure 2.4: Join Calculus: Syntax.

dn(x⟨\bar{y}⟩) = \{x\}  
dn(J) = dn(J) \cup dn(K)

dn(J|K) = \{x\} \cup \{\bar{y}\}

\[\begin{align*}
    \text{fn}(D \land E) &= \text{fn}(D) \cup \text{fn}(E) \\
    \text{fn}(J \triangleright P) &= \text{fn}(P) \uplus (\text{fn}(P) \setminus \text{rn}(J)) \\
    \text{fn}(\{0\}) &= \emptyset \\
    \text{fn}(\langle \bar{y} \rangle) &= \{x\} \cup \{\bar{y}\}
\end{align*}\]

\[\text{fn}(P|Q) = \text{fn}(P) \uplus \text{fn}(Q) \quad \text{fn}(\text{def } D \text{ in } P) = (\text{fn}(P) \uplus \text{fn}(D)) \setminus \text{dn}(D)\]

Figure 2.5: Defined, received, and free names.

**Example 2.2.** Figure 2.3 shows a deterministic process of the p/T net N. Note that K is a causal net, since there are neither cycles nor conflicts among the pre- and postset of its transitions. The morphism P from K to N is graphically shown by dotted lines. Moreover, O(P) = a₁, D(P) = a₃, pre(P) = post(P) = E_P = a.

## 2.3 The Join calculus

The Join calculus [52] is a well-known PDL with asynchronous name-passing communication. It has the same expressive power as the asynchronous π-calculus and it has distributed running implementations, e.g. JoCaml [42] and Polyphonic C⁵ [8].

### 2.3.1 Syntax

Join relies on an infinite set \(\mathcal{N}\) of names \(x, y, u, v, \ldots\). Name tuples are written \(\bar{u}\). Join processes, definitions and patterns are in Figure 2.4. A process is either the inert process \(0\), the asynchronous emission \(x \langle \bar{y} \rangle\) of message \(\bar{y}\) on port \(x\), the process \text{def } D \text{ in } P \text{ equipped with local ports defined by } D, \text{ or a parallel composition of processes } P|Q. \text{ A definition is a conjunction of basic reactions } J \triangleright P \text{ that associate join-patterns } J \text{ with guarded processes } P. \text{ Names defined by } D \text{ in } \text{def } D \text{ in } P \text{ are bound in } P \text{ and in all the guarded processes contained in } D. \text{ The sets of defined names } dn, \text{ received names } rn \text{ and free names } fn \text{ are defined in Figure 2.5.}
2.3.2 The Chemical Abstract Machine

The semantics of the Join calculus relies on the reflexive CHAM. In a CHAM \[11\] computation states \( S \) (called solutions) are finite multisets of terms \( m \) (called molecules), and computations are multiset rewrites. Multisets are denoted by \( m_1, \ldots, m_n \) and abbreviated with \( \oplus_i m_i \). Solutions can be structured in a hierarchical way by using the operator \textit{membrane} \( \{ . \} \) to group a solution \( S \) into a molecule \( \{ [S] \} \). In \[11\] molecules can be built also with the constructor \textit{airlock}, but it is not needed in our presentation.

Transformations are described by a set of \textit{chemical rules}, which specify how solutions react. In a CHAM there are two different kinds of chemical rules: \textit{heating / cooling} (or \textit{structural}) rules \( \hookrightarrow \) representing syntactical rearrangements of molecules in a solution, and \textit{reaction} rules \( \rightarrow \). Structural rules are reversible: a solution obtained by applying a cooling rule can be heated back to the original state, and vice versa. Instead, reaction rules cannot be undone.

The laws governing CHAM computations are the following:

- \textit{Reaction law}: Given a rule, an instance of its left-hand-side can be replaced by the corresponding instance of the right-hand-side.

\[
\begin{align*}
\text{(REACTION LAW)} \\
& m_1, \ldots, m_k \rightarrow m'_1, \ldots, m'_i \in \text{set CHAM rules} \\
& m_1 \sigma, \ldots, m_k \sigma \rightarrow m'_1 \sigma, \ldots, m'_i \sigma
\end{align*}
\]

- \textit{Chemical law}: Reactions can be applied in every larger solution

\[
\begin{align*}
\text{(CHEMICAL LAW)} \\
& S \rightarrow S' \\
& S, S'' \rightarrow S', S''
\end{align*}
\]

Note that CHAM rules have no premises and are purely local. They specify only the part of the solution that actually changes.

- \textit{Membrane Law}: Reactions may occur at any level in the hierarchy of solutions

\[
\begin{align*}
\text{(MEMBRANE LAW)} \\
& S \rightarrow S' \\
& \{ [S] \} \rightarrow \{ [S'] \}
\end{align*}
\]

Note that, since solutions are multisets, rules can be applied concurrently.
2.3.3 Operational semantics

The CHAM for the Join calculus is defined as follow. Molecules correspond to terms of the Join calculus denoting processes or definitions. The chemical rules are shown in Figure 2.6. Rule STR-NULL states that 0 can be added or removed from any solution. Rules STR-JOIN and STR-AND stand for the associativity and commutativity of $|$ and $\land$, because $\ldots$ is such. STR-DEF denotes the activation of a local definition, which implements a static scoping discipline by properly renaming defined ports by \textit{globally fresh} names. A name $x$ is fresh w.r.t. a process $P$ (resp. a definition $D$) if $x \notin \text{fn}(P)$ (resp. $x \notin \text{fn}(D)$). Moreover, $x$ is fresh w.r.t. a solution $s$ if it is fresh w.r.t. every term in $s$. A set of names $X$ is fresh if every name in $X$ is such. We write the substitution of names $x_1 \ldots x_n$ by names $y_1 \ldots y_n$ as $\sigma = \{y_1/x_1, \ldots, y_n/x_n\}$, with $\text{dom}(\sigma) = \{x_1, \ldots, x_n\}$ and $\text{range}(\sigma) = \{y_1, \ldots, y_n\}$. We indicate with $\sigma_N$ an injective substitution $\sigma$ such that $\text{dom}(\sigma) = N$. We require names to be globally fresh, i.e. fresh w.r.t the implicit context in which the rule is applied.

Consider, for instance, $s = \{z\langle x,z \rangle, \text{def } x\langle y \rangle \triangleright z\langle y, x \rangle \text{ in } x\langle a \rangle\}$, whose second molecule contains a definition of a local port $x$ different from the homonym free port in the first molecule. When STR-DEF is applied, the local definition of $x$ is renamed by using a fresh name, obtaining, for instance, the solution $s' = \{z\langle x, z \rangle, x_1\langle y \rangle \triangleright z\langle y, x_1 \rangle, x_1\langle a \rangle\}$.

Finally, RED describes the use of an active reaction rule ($J \triangleright P$) to consume messages forming an instance of $J$ (for a suitable substitution $\sigma$, with $\text{dom}(\sigma) = \text{rn}(J)$), and produce a new instance $P\sigma$ of its guarded process $P$. By applying RED to $s'$ for $\sigma = \{a/y\}$, we get $s' \rightarrow \{z\langle x, z \rangle, x_1\langle \bar{y} \rangle \triangleright z\langle y, x_1 \rangle, z\langle a, x_1 \rangle\}$. Note that the local port $x_1$ has been extruded on the free channel $z$.

2.3.4 Abstract semantics

Several notions of Join process equivalences have been proposed in literature [51, 53]. In this thesis (particularly in PART II) we will use a barbed bisimilarity to compare processes.

\textbf{Definition 2.9} (Barb). The observation predicate $\downarrow_x$, also known as the \textit{strong barb}, detects whether a process emits on some free name $x$:

$$P \downarrow_x \text{ iff } \exists P', \bar{u} : P \equiv \text{def } D \text{ in } P'|x\langle \bar{u} \rangle \text{ and } x \notin \text{dn}(D)$$
2.3. THE JOIN CALCULUS

The \textit{(weak) barb} $\downarrow_x$ detects whether a process may satisfy the basic observation predicate $\downarrow_x$, possibly after performing a sequence of reductions.

\[ P \downarrow_x \text{ iff } \exists P' : P \rightarrow^* P' \land P' \downarrow_x \]

\textbf{Definition 2.10} (Strong barbed bisimulation). A binary relation $\mathcal{R}$ over processes is a \textit{strong barbed simulation} if for all $P \mathcal{R} Q$ then:

1. $\forall x : P \downarrow_x \Rightarrow Q \downarrow_x$.
2. $\forall P'$ s.t. $P \rightarrow P'$ then $\exists Q' : Q \rightarrow Q'$ and $P' \mathcal{R} Q'$

$\mathcal{R}$ is a \textit{strong barbed bisimulation} when both $\mathcal{R}$ and $\mathcal{R}^{-1}$ are strong barbed simulations. \textit{Strong Barbed bisimilarity} $\approx$ is the largest strong barbed bisimulation.

\textbf{Definition 2.11} (Barbed simulation, bisimulation). A binary relation $\mathcal{R}$ over processes is a \textit{(weak) barbed simulation} if for all $P \mathcal{R} Q$ then:

1. $\forall x : P \downarrow_x \Rightarrow Q \downarrow_x$.
2. $\forall P'$ s.t. $P \rightarrow P'$ then $\exists Q' : Q \rightarrow^* Q'$ and $P' \mathcal{R} Q'$

$\mathcal{R}$ is a \textit{barbed bisimulation} when both $\mathcal{R}$ and $\mathcal{R}^{-1}$ are barbed simulations. \textit{Barbed bisimilarity} $\approx$ is the largest barbed bisimulation.

\textbf{Notation 2.2.} We often write $P \equiv P'$ instead of $P \equiv^* P'$. Moreover, $\equiv_e$ denotes the least equivalence relation satisfying

1. if $P \equiv Q$ then $P \equiv_e Q$, and
2. $P \equiv_e P \mid \text{def } D \in 0$.

It is easy to notice that $\equiv_e$ is a strong barbed bisimulation.

2.3.5 High-level Nets as Typed Fragments of Join

The work in [31] provides a full correspondence between Join calculus and a hierarchy of Petri nets. Although the proposed hierarchy will be precisely formalised through the chapters in the first part of this thesis, we summarise here the line followed by the approach. At the bottom of the hierarchy we locate ordinary place/transition Petri nets ($\Pi_0$). The next levels enrich the basic model by adding one by one the following features: (i) \textit{value-passing}, in the form of \textit{coloured nets} ($\Pi_1$), (ii) \textit{network reconfigurability} ($\Pi_2$), and (iii) \textit{dynamically growing, open networks} ($\Pi_3$).

Four different type systems $\Delta_i$ for $i = 0, \ldots, 3$ (shown in Figure 2.7) were designed in order to single out those terms in Join that correspond to nets in $\Pi_i$. Then, a translation $\llbracket \_ \rrbracket$ maps terms into nets. The defined translation preserves the semantics in the sense that a reduction in Join corresponds to the firing of a transition on the net. Finally, it has been proved that for any Join term $P$, $P$ is typable in $\Delta_i$ if and only if $\llbracket P \rrbracket$ belongs to $\Pi_i$. Since any Join term is typable in $\Delta_3$, it can be concluded that dynamic nets coincide with join terms.
**Type judgements:**

\[ \Gamma; \nabla \vdash P : i^\infty \] (\(P\) well-typed, in \(\Pi_i\), and containing no \texttt{def} in \(\cdot\))

\[ \Gamma; \nabla \vdash P : i^\circ \] (\(P\) well-typed and in \(\Pi_i\))

\[ \Gamma; \nabla \vdash D : i^\circ \] (\(D\) well-typed and containing terms in \(\Pi_i\))

where \(i^\infty \in \{0^\circ, 1^\circ, 2^\circ, 3^\circ\}\), \(i^\circ \in \{0^\circ, 1^\circ, 2^\circ, 3^\circ\}\)

**Typing rules**

\[(\text{P-Mess}_0)\]

\[ \Gamma; \nabla \vdash x(\bullet) : 0^\circ \quad (x \notin \nabla) \]

\[(\text{P-Mess}_1)\]

\[ \Gamma; \nabla \vdash x(y) : 1^\infty \quad (x \notin \nabla, y \notin \Gamma) \]

\[(\text{P-Mess}_2)\]

\[ \Gamma; \nabla \vdash x(y) : 2^\circ \]

\[(\text{P-Def})\]

\[ \Gamma, d\eta(D) ; \nabla \vdash D : i^\circ \quad \Gamma, d\eta(D) ; \nabla \vdash P : i^\circ \]

\[ \Gamma; \nabla \vdash \text{def} D \text{ in } P : i^\circ \]

\[(\text{P-Par})\]

\[ \Gamma; \nabla \vdash P : \tau \quad \Gamma; \nabla \vdash Q : \tau \]

\[ \Gamma; \nabla \vdash P \mid Q : \tau \]

\[(\text{P-Sub})\]

\[ \Gamma; \nabla \vdash P : \tau \quad \tau < \tau' \]

\[ \Gamma; \nabla \vdash P : \tau' \]

\[(\text{D-Pat}_0)\]

\[ \Gamma; \nabla \vdash P : 0^\circ \]

\[ \Gamma; \nabla \vdash J \triangleright P : 0^\circ \quad (\eta(J) = \{i\}) \]

\[(\text{D-Pat}_1)\]

\[ \Gamma; \nabla, \epsilon_n(J) \vdash P : i^\infty \]

\[ \Gamma; \nabla \vdash P : i^\circ \]

\[(\text{D-Pat}_2)\]

\[ \Gamma; \nabla, \epsilon_n(J) \vdash P : i^\circ \]

\[ \Gamma; \nabla \vdash J \triangleright P : 3^\circ \]

\[(\text{D-And})\]

\[ \Gamma; \nabla \vdash D : i^\circ \quad \Gamma; \nabla \vdash E : i^\circ \]

\[ \Gamma; \nabla \vdash D \land E : i^\circ \]

\[(\text{D-Sub})\]

\[ \Gamma; \nabla \vdash D : i^\circ \quad i < j \]

\[ \Gamma; \nabla \vdash D : j^\circ \]

where \(\tau\downarrow\) stands for \(i\) in \(\tau = i^\circ\), and \(\tau\downarrow\) for \(i\), \(\tau < \tau'\) denotes \(\tau\downarrow < \tau'\downarrow\) and \(\tau\uparrow \leq \tau'\uparrow\), with \(\circ \circ < \circ\).

**Figure 2.7:** Nets as Typed Fragments in Join
2.4 The D2PC protocol

This section describes the Distributed two Phase Commit Protocol (D2PC) proposed in [21] to implement zero-safe nets. We start by giving an intuitive presentation of the protocol and then its formalisation in the Join calculus.

2.4.1 Informal presentation

Roughly, the D2PC is a variant of the decentralised 2PC protocol [10] that implements a distributed agreement protocol among a set of participants that have a partial knowledge about the whole set of parties. The algorithm assumes a reliable asynchronous communication between participants. Moreover, participants can abort, but do not crash. The D2PC has been proved to be correct in such setting assuring that all participants will asynchronously take the same decision.

All participants in the D2PC act as transaction managers, all of them having the same behaviour and communicating in an asynchronous way. Any participant \( P_j \) maintains three lists:

- \( \ell_j \) records the set of participants to be notified with the commit vote;
- \( \ell'_j \) contains the set of all known parties in the transaction, called the synchronisation set,
- \( \ell''_j \) records the parties from the synchronisation set that have voted commit.

When a participant \( P_j \) initiates the execution of the protocol voting commit, the sets \( \ell_j \) and \( \ell'_j \) are initialised with the identities of those parties that have interacted directly with \( P_j \). Instead, the list of committing parties \( \ell''_j \) contains only \( P_j \). The three lists are updated during the execution of the protocol, until either an abort decision is reached or all participants to the transaction are discovered. More precisely, any participant performs the algorithm described in Figure 2.8. (Perhaps the meaning of the notation Lock for those messages including synchronisation sets is not obvious to the reader: it means that the parties in the synchronisation sets are “locked” until an agreement / abort is established.)

In Figure 2.9 we illustrate a run of the D2PC with three coordinators, namely \( A, B \) and \( C \), any of them willing to commit. The initial configuration (Figure 2.9(a)) shows the partial view that any participant has about the other parties in the agreement (see the local synchronisation sets \( \ell'_j \)): \( A \) and \( B \) know that, apart from themselves, only \( C \) is part of the agreement, while \( C \) knows both \( A \) and \( B \). Moreover, every participant initialises the set \( \ell \) with the known participants that have to be notified, and \( \ell'' \) with its own proper identity (i.e. at the beginning there is no information about remaining participants).

When the protocol starts (Figure 2.9(b)) every participant sends its ready to commit vote together with its synchronisation set \( \ell'' \) to any known participant. After
Initial State of the \( j \)-th participant \( P_j \).

- \( \ell_j \): parties to notify (those with whom \( P_j \) cooperated directly).
- \( \ell'_j \): all known parties (those with whom \( P_j \) cooperated directly).
- \( \ell''_j = \{ P_j \} \)
- \( state_j \in \{ committing, finished \} \)

Algorithm.

- **Committing.** While in state \textit{committing} perform the following steps
  1. If \( \ell \neq \emptyset \) then send the own synchronisation set \( \ell'_j \) to every participant in \( \ell_j \)
     (message \textsc{lock}), and make \( \ell = \emptyset \).
  2. Otherwise, if \( \ell'_j = \ell''_j \) then finish with \textit{"commit"}.
  3. Wait for messages from other parties
     - For any received message \textsc{lock}(\( \ell''_i \)) from the participant \( P_i \) update the state
       in the following way:
       * \( \ell_j = \ell'_j \cup \ell''_i \)
       * \( \ell''_j = \ell''_j \cup \{ P_i \} \)
     - If a message \textsc{abort} is received, send all \textsc{lock} messages and then pass to the
       state \textit{finished}.
  4. goto 1.

- **Finished.** When the state \textit{finished} is reached, finish with \textit{"abort"}.
  
  While in state \textit{finished} answer with \textsc{abort} to any received message of type \textsc{lock}.

Figure 2.8: D2PC algorithm.

This round (Figure 2.9(c)), all participants update their states with the information contained in the received messages. Note that \( C \) has received votes from both \( A \) and \( B \) without information about other participants. In this case the sets \( \ell' \) and \( \ell'' \) of \( C \) coincide and thus \( C \) knows that all parties in the negotiation are willing to commit. At this time \( C \) can commit, because no party has decided to abort. Differently, \( A \) and \( B \) have received the commit vote from \( C \) containing participants not known previously, thus they update their state and must continue the execution of the protocol. In the next step, \( A \) and \( B \) send their decisions to the recently known participants (Figure 2.9(d)). After that, they update their state and commit (Figure 2.9(e)).

Consider a different scenario in which \( A \) and \( C \) are willing to commit but \( B \) decides to abort. The initial situation is shown in Figure 2.10(a). We do not show the synchronisation set of aborted components because it is useless. When
the protocol starts, every participant in committing state (i.e., $A$ and $C$) sends its vote to the known parties. Similarly to the previous case, committing participants update their status (Figure 2.10(c)). Note that $C$ cannot commit because it has not received yet the confirmation from $B$. Moreover, $A$ cannot commit because it has received the identity $B$, discovering a new participant to contact. In the next round (Figure 2.10(d)), $A$ sends its vote to $B$. Instead, $B$ answers the message received in the previous round from $C$ with $\texttt{abt}$, signalling the abort of the negotiation. After the second round (Figure 2.10(e)) $C$ aborts because of the message $\texttt{abt}$ received from $B$, while $A$ is still waiting the corresponding vote from $B$. Finally, in the third round (Figure 2.10(f)), $B$ answers to the commit vote from $A$ with $\texttt{abt}$. After this round (Figure 2.10(g)) all participants have aborted.

Figure 2.9: Example of commit.
Figure 2.10: Example of abort.
2.4. THE D2PC PROTOCOL

\[
D = 
\begin{align*}
\text{state}(\alpha) \mid \text{put} (\ell, \kappa) & \triangleright \text{commit}(\ell \setminus \{\text{lock}\}, \ell, \text{lock}, \alpha, \kappa) \\
\land 
\text{state}(\alpha) \triangleright \text{finished()} \mid \text{release}() \\
\land 
\text{commit}\{\{l\} \cup \ell, \ell', \ell'', \alpha, \kappa\} \triangleright \text{commit}(\ell, \ell', \ell'', \alpha, \kappa) \mid l(\ell, \text{lock}, \text{abort}) \\
\land 
\text{commit}(\ell, \ell', \ell'', \alpha, \kappa) \mid \text{lock}(\ell'', l, a) \triangleright \text{commit}(\ell \cup (\ell'' \setminus \ell), \ell' \cup \ell'', \ell'' \cup \{l\}, \alpha, \kappa) \\
\land 
\text{commit}(\emptyset, \ell, \ell', \alpha, \kappa) \triangleright \text{release}(\kappa) \\
\land 
\text{commit}(\emptyset, \ell', \ell'', \alpha, \kappa) \mid \text{abort}() \triangleright \text{finished()} \mid \text{release}(\alpha) \\
\land 
\text{finished()} \mid \text{put} (\ell, \kappa) \triangleright \text{finished()} \\
\land 
\text{finished()} \mid \text{lock}(\ell, l, a) \triangleright \text{finished()} \mid a() \\
\land 
\text{finished()} \mid \text{abort}() \triangleright \text{finished()} 
\end{align*}
\]

Figure 2.11: The encoding of coordinators.

\begin{center}
\begin{tikzpicture}
\node (initial) at (0,0) {initial};
\node (commit) at (1,-1) {commit};
\node (finished) at (2,-1) {finished};
\node (abort) at (1,-2) {\text{abort}};
\path (initial) edge [->] node [above] {put} (commit);
\path (commit) edge [->] node [above] {\text{abort}} (finished);
\path (commit) edge [->] node [above] {\text{Abort}} (abort);
\end{tikzpicture}
\end{center}

Figure 2.12: States of coordinators

2.4.2 Formal definition of the D2PC

The D2PC has been formally defined as the Join definition presented in Figure 2.11. We analyse the whole protocol by considering the behaviour of a coordinator on its different states (a state transition diagram for coordinators is shown in Figure 2.12) and considering that coordinators communicate through three ports put, lock and abort:

1. **Initial**: A participant in the initial state has not decided whether to vote commit or abort. This state is modelled by the message state, which records the messages (if any) that must be released if the transaction aborts.

2. **Finished**: When a participant aborts, it transits to the state finished (modelled with the message finished()).

3. **Commit**: A participant is in the state commit if it is willing to commit. This state is modelled by the message commit, which carries the values \(\langle \ell, \ell', \ell'', \alpha, \kappa\rangle\), where

   - \(\ell, \ell', \ell''\) are the lists that records the set of participant to notify, the synchronisation set, and the committing parties as explained in Section 2.4.1;
   - \(\alpha\) stores the messages to be released in case of abort;
• $\kappa$ contains the messages to be sent if the transaction finally commits.

A participant in the initial state may decide non-deterministically to abort by firing the rule

$$\text{state}(\alpha) \triangleright \text{finished}(\emptyset) | \text{release}(\alpha)$$

In this case, the coordinator transits to the state $\text{finished}(\emptyset)$ and releases the compensation $\alpha$.

Instead, when a coordinator in the initial state is required to start the execution of the protocol by voting commit (i.e. when it receives the message $\text{put}(\ell, \kappa)$), then the following rule

$$\text{state}(\alpha) | \text{put}(\ell, \kappa) \triangleright \text{commit}(\ell \setminus \{\text{lock}\}, \ell, \{\text{lock}\}, \alpha, \kappa)$$

can be fired to produce the message $\text{commit}$.

While in state $\text{commit}$, coordinators performs the following steps:

1. **First phase.** The participant sends a request message to every party in its own synchronisation set. This task is performed by the rule

   $$\text{commit}(\{l\} \cup \ell, \ell', \alpha, \kappa) \triangleright \text{commit}(\ell, \ell', \alpha, \kappa | l(\ell, \text{lock}, \text{abort}).$$

   The request message carries the synchronisation set of the participant, together with the local names $\text{lock}$ and $\text{abort}$ on which the coordinator waits for commit and abort votes.

2. **Second phase.** The participant collects the messages sent by other coordinators and updates its own synchronisation set. The port used to receive this information is $\text{lock}$, and the rule collecting the synchronisation sets is:

   $$\text{commit}(\ell, \ell', \ell'', \alpha, \kappa) | \text{lock}(\ell''', l, a) \triangleright \text{commit}(\ell \cup (\ell''' \setminus \ell'), \ell' \cup \ell''', \ell'' \cup \{l\}, \alpha, \kappa).$$

   A message $\text{lock}$ is sent to the items added to the synchronisation set. This means that the first and the second phases proceed in parallel.

3. When the synchronisation set is transitively closed, i.e. when $\ell'$ is equal to the received confirmations $\ell''$, then the commit protocol of the participant terminates locally, and the continuation $\kappa$ is released.

4. In case of abort, the coordinators reaches the state $\text{finished}$ and releases the compensation $\alpha$. This is achieve by activating the rule

   $$\text{commit}(\emptyset, \ell, \ell'', \alpha, \kappa) | \text{abort}(\emptyset) \triangleright \text{finished}(\emptyset) | \text{release}(\alpha)$$
While in state $\text{finished}()$, a coordinator will reply to any received message $\text{lock}$ with $\text{abort}$. This is coded by the following rule

$$
\text{finished}() \parallel \text{lock}(\ell, l, a) \triangleright \text{finished}() \parallel a()
$$

The encoding has been proven to be correct, in the sense that if all coordinators are ready to commit, then all continuations are released (assuming fairness) and when several coordinators abort, then all participants to the transactions abort. The proof of correctness of the D2PC is split in two steps: (part 1) shows that if all coordinators are ready to commit, then all continuations will be released (assuming fairness); (part 2) deals with aborts.

In the following theorems from [21], let $\sigma_i$ for $i \in \mathbb{N}$ be the renaming that indexes with $i$ all the defined names in $D$. Also, we write $D_i$ for $D\sigma_i$. We write $A()$, when $A$ is an empty set or a singleton $\{a\}$: in the former case it means $0$, in the latter case it represents $a()$. Moreover, we let a symmetric lock covering be a finite family $\{\ell_i\}_{i \in I}$ such that $\ell_i \subseteq \{\text{lock}_j | j \in I\}$, with $\text{lock}_j \in \ell_i$ if and only if $\text{lock}_i \in \ell_j$ for all $i, j \in I$.

The D2PC will be extended in Section 8.4.1 in order to implement the cJoin calculus. The correctness proof will be extended as well.

**Theorem 2.1** (Correctness of the D2PC, part 1). Let $P = \Pi_{i \in I} \text{commit}_i(\ell_i \setminus \{\text{lock}_i\}, \ell_i, \{\text{lock}_i\}, \alpha_i, \kappa_i)$, where $\{\ell_i\}_{i \in I}$ is a symmetric lock covering. The process

$$
def \bigwedge_{i \in I} D_i \text{ in } P
$$

is strongly confluent, in the sense that it always converges after a finite number of steps bound by $O(n^2)$ to the configuration

$$
def \bigwedge_{i \in I} D_i \text{ in } \Pi_{i \in I} \kappa_i()
$$

The second theorem states that, when several coordinators abort, then all the participants to those transactions abort. To determine the coordinators which participate to a transaction, the transitive closure of the synchronisations sets.

**Theorem 2.2** (Correctness of the D2PC, part 2). Let $P = \Pi_{i \in I} P_i$, such that $P_i$ may have one of the following shapes

- $\text{commit}_i(\ell_i \setminus \{\text{lock}_i\}, \ell_i, \{\text{lock}_i\}, \alpha_i, \kappa_i)$; or
- $\text{state}_i(\alpha_i)$; or
- $\text{finished}_i()|\alpha_i()$;

where $\{\ell_i\}_{i \in I}$ is a symmetric lock covering. Let $L \subseteq I$ be the least set such that
1. if \( P_i = \text{state}_i(\alpha_i) \) then \( i \in L \); 
2. if \( P_i = \text{finished}_i()|\alpha_i \) then \( i \in L \); and 
3. \( L \) is transitively closed, namely if \( i \in L \) and \( \text{lock}_j \in \ell_i \), then also \( j \in L \).

Then, the process \( \text{def} \bigwedge_{i \in I} D_i \text{ in } \Pi_{i \in I} P_i \) is strongly confluent, in the sense that it always converges after a finite number of steps bound by \( O(n^2) \) to 

\[
\text{def} \bigwedge_{i \in I} D_i \text{ in } (\Pi_{i \in I \setminus L} \kappa_i()) | (\Pi_{i \in L} \alpha_i()|\text{finished}_i()).
\]

The above result assures that the coordinators of the transactions that commit (i.e., coordinators in \( I \setminus L \)) release their continuations, while all coordinators involved in transactions that abort (i.e., coordinators in \( L \)) release the compensations and pass to the state \textit{finished}. 
Part I

Adding mobility to ZS nets
Roadmap to PART I

Zero-safe nets (zs nets) have been introduced to model transactions in concurrent systems [27]. The basic model extends p/T nets with a mechanism for expressing serializable concurrent (multiway) transactions. The transactional mechanism of zs nets relies on a distinction between observable and transient markings. Hence, the places (and, consequently, the tokens) of a zs net are classified either as stable (i.e., observable) and zero-safe (i.e., hidden). Roughly, a transaction on a ZS net is a concurrent computation that departs from and arrives to a multiset of stable tokens. In addition, zs nets offer a two-level view of the modelled system: (1) the concrete operational view, where transient places and the coordination mechanism between activities participating to a transaction are fully exposed; and (2) the abstract view, where transactions are seen as atomic activities involving only stable places, while transient places are hidden. In fact, the abstract view is given by an ordinary p/T net, whose places are the stable places of the zs net and whose transitions are the transactions of the zs net.

In this part of this thesis, zs nets are progressively enriched following the hierarchy proposed in [31] (see Section 2.3.5), i.e. by adding: (1) the value passing mechanism of coloured nets; (2) the dynamic interconnection mechanism of reconfigurable nets; (3) the high-order features of dynamic nets.

For the first two cases we show that the two-level view of the zero-safe approach is fully preserved, in the sense that, e.g. the abstract net of a coloured zs net is a coloured p/T net, and so on. Moreover, the constructions are consistent with the obvious embedding derived from the hierarchy, in the sense that, e.g. if we regard a coloured zs net as a reconfigurable zs net and take the corresponding abstract reconfigurable p/T net, then we get a net that is isomorphic to the coloured abstract net regarded as a reconfigurable net. In other words, the diagram in Figure 2.13 commutes up-to isomorphism (vertical arrows are the obvious embedding, while horizontal arrows stand for the construction of abstract nets). For dynamic nets we show that the construction of the abstract net it is not possible, i.e. in general a dynamic zs net allows the description of behaviour that cannot be modelled with a dynamic net.

For each layer of the tower we give several examples for illustrating the main features of the corresponding model. The two main case studies we present are the mobile lessees problem and the mailing list. Regarding the mobile lessees problem, first it is shown that an instance of the problem can be always represented as a zs net, then it is shown that colours allow for modelling all the instances with a unique coloured zs net (actually, different instances of the problem correspond to different initial markings). Regarding the mailing list example, first it is shown that reconfigurable arcs are needed for modelling dynamic message delivery, and then it is shown that the example can be extended with dynamic creation of new mailing lists by exploiting reflection in dynamic zs nets.
Figure 2.13: The hierarchy of transactional nets.

Content of Part 1  Chapter 3 is a background chapter, where we recall the basics of ZS nets. In particular, we define the operational semantics of such model, the notion of atomic transactions, and the notion of an abstract net, which are later extended to account for colours, reconfiguration and high-order. The modelling of (instances of) the mobile lessees problem is instead original.

Chapters 4 and 5 contain the original proposals for extending the zero-safe approach to coloured and reconfigurable nets. In both cases, the operational and abstract semantics are defined and related by strong correspondence theorems.

Chapter 6 extends the zero-safe approach to dynamic nets. The operational semantics of dynamic ZS nets is presented and discussed on the basis of the mailing list example, whereas the abstract semantics is just informally discussed to put in evidence the difficulties in completing the tower in Figure 2.13.
Chapter 3

Preliminaries on Zero-Safe Nets

In this chapter we summarise the basics of the zero-safe approach by following the presentation given in [27].

3.1 From Petri nets to Zero-safe nets

Zero-safe nets are an extension of Petri nets suitable to express transactions. Differently from P/T nets, the places of zero-safe nets are partitioned into ordinary and transactional ones (called stable and zero, respectively). Accordingly to the ordinary terminology, the places of a '0-safe' net cannot contain any token in all reachable markings. Zero-safe net — note the word 'zero' instead of the digit '0' — is used to denote that the net has zero places that cannot contain tokens in any observable marking. The role of zero places is to coordinate the atomic execution of complex collections of transitions.

Definition 3.1 (zs net). A Zero-Safe net ( zs net) is a 6-tuple $B = (S_B, T_B, \delta_0B, \delta_1B, m_0B, Z_B)$ where $N_B = (S_B, T_B, \delta_0B, \delta_1B, m_0B)$ is the underlying P/T net and the set $Z_B \subseteq S_B$ is the set of zero places. The places in $S_B \setminus Z_B$ (denoted by $L_B$) are called stable places. A stable marking $m$ is a multiset of stable places (i.e., $m \in M_{L_B}$), and the initial marking $m_0B$ must be stable.

Note that markings $m \in M_{S_B}$ can be seen as pairs $(s, z)$ with $m = s \oplus z$, where $s \in M_{L_B}$ is a stable marking and $z \in M_{Z_B}$ is a multiset of zero resources, because $M_{S_B} \cong M_{L_B} \times M_{Z_B}$. Transitions are written $m \rightarrow m'$, with $m$ and $m'$ multisets of stable and zero places as assigned by $\delta_0B$ and $\delta_1B$. As usual, we omit subscripts when referring to components of a zs net if they are clear from the context.

3.2 Operational Semantics

The operational semantics of zs nets is given by the two relations $\Rightarrow_T$ and $\rightarrow_T$, which are defined inductively by the inference rules in Figure 3.1. Rules (FIRING)
and *(step)* are the ordinary ones for the execution of one/many transition(s) in a Petri nets. However, sequences of steps differ from the ordinary transitive closure of \( \rightarrow_T \). Instead, the rule *(concatenation)* composes zero tokens in series but stable tokens in parallel, hence stable tokens produced by the first step cannot be consumed by the second one. The transactions of the nets are defined by rule *(close)*. Note that a transaction goes from a stable marking to another stable marking. The combination of rules *(concatenation)* and *(close)* fixes the main policy of zs nets: *stable tokens produced during a transaction are made available only at commit time, when no zero tokens are left.*

The moves \((s, \emptyset) \rightarrow_T (s', \emptyset)\) define all the atomic activities of the net, and hence they can be performed in parallel and sequentially like the transitions of any ordinary net. It is worth noting that a step \((s, \emptyset) \rightarrow_T (s', \emptyset)\) can be itself the parallel composition of several concurrent transactions (by rule *(step)*).

**Example 3.1** *(The free choice problem).* This example has been borrowed (with slightly adjustments) from [26]. Consider the net shown in Example 2.1 (see Figure 2.2(a)) to code the assignment of two resources \( a \) and \( b \) either to the activity \( c \) or to \( d \). By firing \( t_1 \), the resources are assigned to \( c \), and by \( t_2 \) to \( d \). The nondeterministic choice encoded by the net corresponds to a centralised coordination mechanism that guarantees that both resources are assigned atomically to the same activity. Nevertheless, if one wants to model the system using a free choice net, i.e. where all decisions are made locally to each place, the situation is different. Consider the free choice net shown in Figure 3.2. It models the system with two independent decisions: one for the assignment of \( a \), the other for the assignment of \( b \). The free choice net admits computations not allowed in the abstract system in Figure 2.2(a). In fact, the free choice net has deadlocks: consider the firing of \( \text{assign}_{a,c} \) and \( \text{assign}_{b,d} \). In this case, the net cannot evolve to either \( c \) or \( d \), which is a computation not possible in the original net.

ZS nets can be used to overcome this problem, by defining intermediate places as zero places. The assignment problem can be modelled as the zs net in Figure 3.3, where smaller circles stand for zero places. This zs net avoids deadlocks because
computations ending in markings containing zero tokens are recoverable and not observable.

The only atomic movements of the net in Figure 3.3, are \((a \oplus b, \emptyset) \rightarrow_{r} (c, \emptyset)\) and \((a \oplus b, \emptyset) \rightarrow_{r} (d, \emptyset)\) (the corresponding proof are in Figure 3.4, where \(F, S, C\) and \(CL\) stand resp. for \(FIRING\), \(STEP\), \(CONCATENATION\) and \(CLOSE\)). Note that computations like \((a \oplus b, \emptyset) \rightarrow_{r} (\emptyset, a_{to.b} \oplus b_{to.c})\) are not observable, since they are obtained with the relation \(\rightarrow_{r}\) but not with \(\Rightarrow_{r}\), which identifies the valid computations of the system.

Example 3.2 (Mobile lessees). The general problem we want to model consists of: (i) a set of apartments that can be rented immediately, (ii) a group of people looking for an apartment, and (iii) people willing to move to another apartment if somebody else can rent their actual apartments. Consider an instance of the problem with three apartments \(A, B, C\) and four people \(P, Q, R, S\). The initial state
\[
\begin{align*}
\frac{a \mid a \text{to}\_c \in T}{(a, \emptyset) \rightarrow_T (\emptyset, a \text{to}\_c)} & \quad \frac{b \mid b \text{to}\_c \in T}{(b, \emptyset) \rightarrow_T (\emptyset, b \text{to}\_c)} \\
\frac{(a \oplus b, \emptyset) \rightarrow_T (\emptyset, a \text{to}\_c \oplus b \text{to}\_c)}{a \text{to}\_c \oplus b \text{to}\_c} & \quad \frac{(\emptyset, a \text{to}\_c \oplus b \text{to}\_c) \rightarrow_T (c, \emptyset)}{c \in T} \\
\frac{(a \oplus b, \emptyset) \rightarrow_T (c, \emptyset)}{c \in T} & \quad (\text{cl})
\end{align*}
\]

\[
\begin{align*}
\frac{a \mid a \text{to}\_d \in T}{(a, \emptyset) \rightarrow_T (\emptyset, a \text{to}\_d)} & \quad \frac{b \mid b \text{to}\_d \in T}{(b, \emptyset) \rightarrow_T (\emptyset, b \text{to}\_d)} \\
\frac{(a \oplus b, \emptyset) \rightarrow_T (\emptyset, a \text{to}\_d \oplus b \text{to}\_d)}{a \text{to}\_d \oplus b \text{to}\_d} & \quad \frac{(\emptyset, a \text{to}\_d \oplus b \text{to}\_d) \rightarrow_T (d, \emptyset)}{d \in T} \\
\frac{(a \oplus b, \emptyset) \rightarrow_T (d, \emptyset)}{d \in T} & \quad (\text{cl})
\end{align*}
\]

Figure 3.4: Computations of the free choice net for the assignment problem.

can be represented as in Figure 3.5(a), where the apartment A is available for rent, P and S are searching for an apartment, Q wants to leave B, and R wants to leave C. Figure 3.5(b) shows the preferences of people for renting a new apartment (although for simplicity we associate one apartment to one person, in general people may have a set of preferences).

The zs net in Figure 3.6 formulates the instance in Figure 3.5 of the mobile lessees problem. Note that there is a place for any apartment available for immediate rent in the initial state (A_{free}), a place for any person looking for an apartment (S_{wants} and P_{wants}), and a place for any person willing to change apartment (Q_{changes B} and R_{changes C}). There is also a place for any possible rent (i.e., accordingly to the preference matrix in 3.5(b)). For instance, the transition S_{takes A} states that person S can rent the apartment A whenever A is free and S is searching for an apartment, and a token in S_{moves A} means that the person S has rented the apartment A. The more interesting transitions are Q_{leaves B} and R_{leaves C}, where any of them starts a transaction. In fact, they describe the activity of changing an apartment as the orchestration of two different activities, one in which a person finds a new apartment, and other in which the apartment is rented. For instance, the firing of Q_{leaves B} produces two tokens: one in Q_{search} and the other in B_{avail}. Note this transaction can finish only when both produced tokens are consumed, meaning that both Q has rented a new apartment and B has been rented. The initial marking denotes the initial state of the problem.

In Figure 3.7 we show a proof for a transaction in which Q leaves B and takes A, R leaves C and takes B; and P takes C, while S remains without apartment. For space reason, we abbreviate the name of places (i.e. A_f for A_{free}, QB_c for Q_{changes B}, and similarly for the rest) and the name of applied rules (F stands for FIRING and S
Figure 3.5: An instance of the mobile lessees.

Figure 3.6: ZS net of the mobile lessees example.

for step). Moreover, we write stable places with capital letters, while zero places are written with lower case. The computation corresponds to the parallel beginning of two transactions (where Q.changes_B and R.changes_C are decomposed into two sub-activities) followed by the parallel execution of Q.takes_A, P.takes_C and R.takes_B.

### 3.3 Abstract Semantics

As stated by the operational semantics of ZS nets (Figure 3.1), the observable states of a system are those represented by stable markings, while the meaningful computations (i.e., the atomic activities) are the stable steps of the net, i.e. the steps consuming and producing stable markings (relation $\Rightarrow$). Since stable steps can be composed in sequence and in parallel, a stable step can be thought of as the execution of several basic transactions, i.e., stable steps that cannot be decomposed
Aux Proof:

\[
\begin{align*}
A_x & \vdash q_s \mid QA_m \in T \quad b_a \vdash r_s \mid RB_m \in T \\
& \quad \Rightarrow (A_x, q_s) \rightarrow_T (QA_m, \emptyset) \quad (\emptyset, b_a \vdash r_s) \rightarrow_T (RB_m, \emptyset) \\
& \quad \Rightarrow (A_x, q_s \land b_a \land r_s) \rightarrow_T (QA_m \land RB_m, \emptyset) \\
& \quad \Rightarrow (S_y \land P_y \land A_x \land QB_c \land RC_c, \emptyset) \rightarrow_T (S_y \land QA_m \land RB_m \land PC_m, \emptyset)
\end{align*}
\]

Figure 3.7: A proof for the execution of a transaction in the mobile leasees ZS net.

into other stable steps. Consequently, all observable behaviors of the system can be derived from the set of basic transactions of the net. The abstract semantics is intended to capture the behavior of a ZS net by identifying its basic transactions.

Since transactions are concurrent computations, the abstract semantics depends on the token philosophy adopted when defining equivalent executions. The abstract semantics of zero-safe nets has been studied under both philosophies [27]. We summarise here the basics of the abstract semantics under the ITph, which is the most significant (because it is more precise in the sense explained in Section 2.2.2) and the one that we will use in the following chapters.

**Definition 3.2** (Connected transaction). Let \( B \) be a ZS and \( P \) a process of the underlying \( P/T \) net \( N_B \). The equivalence class \([P]_\xi\) of all processes isomorphic to \( P \) (see Definition 2.6) is a connected transaction of \( B \) if:

- \( pre(P) \) and \( post(P) \) are stable markings, i.e., the process starts by consuming stable tokens and produces only stable tokens;

- \( E_P \subseteq Z_B \), i.e. the evolution places of the process (see Notation 2.1) are zero places, this means that stable tokens produced during the transaction cannot be consumed during in the same transaction;

- \( P \) is connected (see Definition 2.8); and

- \( P \) is full, i.e., it does not contain idle (i.e., isolated) places (i.e., \( \forall a \in S_K: |\beta a| + |a^*| \geq 1 \)).

We denote by \( \Xi_B \) (ranged by \( \xi \)) the set of all connected transactions of \( B \).
A connected transaction $\xi$ can be executed when the state of the net contains enough stable tokens to enable all the transitions in $\xi$. At the end of its execution no token may be left on zero places (nor may be found on them at the beginning of the step). This means that all the zero tokens produced during a transaction are also consumed by the same transaction. Moreover, in a connected transaction no intermediate marking is stable.

**Definition 3.3** (Causal Abstract Net). Let $B = (S_B, T_B, \delta_0B, \delta_1B, m_0B, Z_B)$ a zs net. The net $I_B = (S_B \setminus Z_B, \Xi_B, \delta_0B, \delta_1B, m_0B)$, with $\delta_0B(\xi) = \text{pre}(\xi)$ and $\delta_1B(\xi) = \text{post}(\xi)$, is the causal abstract net of $B$ (we recall that $\text{pre}(\xi)$ and $\text{post}(\xi)$ denote the multisets $P_\xi(O(\xi))$ and $P_\xi(D(\xi))$, respectively, and that $\Xi_B$ is the set of all the connected transactions of $B$).

The correspondence between the concrete and the abstract view is stated by the following theorem [28].

**Theorem 3.1.** Let $B = (S_B, T_B, \delta_0B, \delta_1B, m_0B, Z_B)$ be a zs net and $I_B$ its abstract net. Then $m_0B \overset{t_n}{\rightarrow} m'$ iff $m_0B \overset{t_n}{\Rightarrow} m'$.

**Example 3.3** (Abstract Net for the Mobile lessees problem). Figure 3.8 shows the abstract net corresponding to the zs net in Figure 3.6. In the abstract net there are only two transitions, each of them representing an abstract transaction of the zs net: $S\_\text{takes}_\text{A}$, corresponding to the homonymous transition in the zs net; $Q\_\text{takes}_\text{A} \& P\_\text{takes}_\text{C} \& R\_\text{takes}_\text{B}$, for the atomic negotiation in which Q leaves B and takes A, R leaves C and takes B; and P takes B. These two transitions are enough to model the abstract behaviour of the system. In fact, any other combination is not possible because it would imply that some exchanged apartment (i.e., B or C) remains available for rent or a person willing to change apartment (Q or R) remains without apartment, which is an inconsistent state (with pending negotiations).

Note that one of the main advantages of the approach is that it allows to fully specify the behaviour of a system without analysing all possible global combinations. Consider an instance of the lessees problem with a larger number of apartments...
and people, and a more complicated set of preferences. It could be tedious to figure out which are all the possible combinations that correspond to consistent transformations in the system. This is one of the main advantages of the zero safe approach, which prevents the combinatorial explosion at the specification level. In fact, atomic activities can be defined in terms of several sub-activities, which keeps the description of the system small, tractable and modular. Moreover, zs nets with a finite number of transitions can yield abstract P/T nets with infinitely many transitions, like the multicasting system presented in [27] or the generalised version of the mobile lessees problem analysed in Section 4.7. From a system designer point of view, this means that the combinatorial features of zs nets can be exploited to keep small the size of the model.
Chapter 4

Adding colours to zs nets

In this section we add transactional capabilities to a simple version of coloured P/T nets [73] (also known as high-level nets) or equivalently we extend zs nets with value passing. Tokens in a coloured P/T net are not “anonymous black” tokens, but they can carry information, which is given by their colours. Actually, colours are values/data associated with a particular instance of a resource. Hence, the state and transitions of a net exploit also this information. Generally, arcs between places and transitions are labelled by arc expressions, which evaluate to multisets of coloured tokens after binding free variables to colours. Expressions on input and output arcs describe respectively the resources needed for the firing and those generated by it.

The version of coloured nets presented here simplifies considerably those in literature [73]. In particular, colours are only tuples of names, and arc expressions are only simply pattern matching between the colours of fetched tokens (a more detailed discussion is given in Section 4.7).

4.1 Background: Coloured P/T nets

We consider an infinite set of constant colour values $\mathcal{V}$, ranged over by $v, w, \ldots$ and an infinite set of colour variables $\mathcal{X}$, ranged over by $x, y, \ldots$. We denote the set of constants and variables with $\mathcal{C} = \mathcal{V} \cup \mathcal{X}$, where $c, c_1, \ldots$ range over $\mathcal{C}$. Moreover, we require constant and variable colours to be disjoint ($\mathcal{V} \cap \mathcal{X} = \emptyset$) and different from place names, i.e. $\mathcal{C} \cap \mathcal{P} = \emptyset$. We remark that this last constraint will be relaxed by reconfigurable and dynamic nets, which will allow colours to refer also to places in the net. Given a set of colours $C \subseteq \mathcal{C}$, $C^*$ stands for the set of all finite (possible empty) sequences of colours in $C$, i.e. $C^* = \{(c_1, \ldots, c_n) | \forall i : 0 \leq i \leq n \land c_i \in C\}$. The empty sequence is denoted by $\epsilon$, and $c, c_1, \ldots$ range over $C^*$. The underlying set of a sequence $c = (c_1, \ldots, c_n)$ is defined as:

$$\overline{c} = (c_1, \ldots, c_n) = \bigcup_i \{c_i\}$$
Definition 4.1 (Coloured net). A coloured net $N$ is a 5-tuple $N = (S_N, C_N, T_N, δ_{0N}, δ_{1N})$, where $S_N \subseteq P$ is the (nonempty) set of places, $C_N \subseteq C$ is the set of colours (we remark that $C \cap P = \emptyset$), $T_N$ is the set of transitions (with $S_N \cap T_N = \emptyset$), and the functions $δ_{0N}, δ_{1N} : T_N \rightarrow ω_1(S_N \times C_N^*)$ assign respectively, source and target to each transition. To assure that a transition fetches and produces at most one token in a place we require that $∀t \in T_N : \text{if} (a, c_1), (a, c_2) \in δ_{iN}(t) \text{ then } c_1 = c_2$, for $i = 0, 1$.

The pre- and postset of a transition are defined similarly to Section 2.2, but now they are coloured sets instead of sets. Analogously, for any place $a$ in $S_N$, the preset of $a$ (written $\ast a$) denotes the set of all transitions with target in $a$ (i.e., $\ast a = \{t|(a, c) \in t^*\}$, and the postset of $a$ (written $a^*$) denotes the set of all transitions with source in $a$ (i.e., $a^* = \{t|(a, c) \in *t\}$). The definitions for the sets of initial and final elements, and isolated places are identical to those given in Section 2.2.

Note that in coloured nets a transition $s_1 \rightarrow s_2$ denotes a pattern that should be matched/instantiated with appropriate colours in order to be applied. In particular, constant colours appearing in $s_1$ act as values that should be matched in order to fire the transition, while variables in $s_1$ should be instantiated with appropriate colours. Variables are binders of colours occurring in $s_2$ and their scope is local to each single transition. For instance, the transition $t = \{(a_1, x), (a_2, x), (a_3, v_1)\} \{((a_1, v_2), (a_2, x))\}$ can be fired by fetching two tokens with the same colour from $a_1$ and $a_2$ (but they can be of any constant colour because they are bound by the same variable $x$) and a token with constant colour $v_3$ from $a_3$. When $t$ is fired, the tokens matching the preset are consumed, and a new token with constant colour $v_2$ is produced in $a_1$, while a token with colour $x$ (corresponding to the consumed tokens in $a_1$ and $a_2$) is generated on $a_4$. Consequently, the firing of $t$ over $s = \{(a_1, v_3), (a_2, v_3), (a_3, v_1)\}$ will produce $s' = \{(a_1, v_2), (a_4, v_3)\}$. From a functional point of view, colour variables occurring in the preset of a transition act as the formal parameters, which are called received colours.

Definition 4.2 (Received colours of a transition). The colour of a set $s \subseteq S \times C^*$ is defined as $\text{col}(s) = \bigcup_{(a, c) \in s} c$, the set of constants is $\text{col}_c(s) = \text{col}(s) \cap C$, and the set of variables $\text{col}_X(s) = \text{col}(s) \cap X$. Note that $\text{col}(s) = \text{col}_c(s) \cup \text{col}_X(s)$. Given a transition $t = s \rightarrow s'$, the set of received colours (also received names) of $t$ is given by $\text{rn}(t) = \text{col}_X(s)$.

Remark. As variables are used to describe parameters in a transition, we will consider only coloured nets where any transition $t = s \rightarrow s'$ satisfies $\text{col}_X(s') \subseteq \text{rn}(t)$. This restriction states that all variables occurring in the postset of a transition are bound to some variable in the preset.

Clearly, all previous definitions (coloured nets, col, etc.) can be straightforwardly extended to consider coloured multisets instead of sets.

Definition 4.3 (Coloured Multiset). Given two sets $S$ and $C$, a coloured multiset over $S$ and $C^*$ is a function $m : S \rightarrow C^* \rightarrow N$. The set of all finite multisets
over $S$ and $C^*$ is written $\mathcal{M}_{S,C}$. With abuse of notation, we write $dom(m)$ for both \\
\{$(s, c) \in S \times C^* \mid m(s)(c) > 0$\} and \{s $\in$ S $\mid m(s)(c) > 0$\}. Additionally, $(s, c) \in m$ is a shorthand for $(s, c) \in dom(m)$, while $s \in m$ means $(s, c) \in m$ for some $c$. We write $s(c)$ for a multiset $m$ such that $dom(m) = \{(s, c)\}$ and $m(s)(c) = 1$. The multiset union is defined as $(m_1 \oplus m_2)(s)(c) = m_1(s)(c) + m_2(s)(c)$, and the difference as $(m_1 \ominus m_2)(s)(c) = \max\{0, m_1(s)(c) - m_2(s)(c)\}$. Given a multiset $m \in \mathcal{M}_{S,C}$ and a partial function $f : S \rightarrow S'$ s.t $dom(m) \subseteq dom(f)$, $f$ applied to $m$ is the multiset $f(m) \in \mathcal{M}_{S',C}$ s.t. $f(m)(s')(c) = \sum_{s \in f^{-1}(s')} m(s)(c)$.

**Definition 4.4 (C-P/T net).** A coloured marked place / transition net (C-P/T net) is a 6-tuple $N = (S_N, C_N, T_N, \delta_{0N}, \delta_{1N}, m_{0N})$, where $S_N \subseteq \mathcal{P}$ is the set of places, $C_N \subseteq C$ is the set of colours, $T_N$ is a set of transitions, the functions $\delta_{0N}, \delta_{1N} : T \rightarrow \mathcal{M}_{S_N, C_N}$ assign respectively, source and target to each transition, and $m_{0N} \in \mathcal{M}_{S_N, C_N \cup \mathcal{V}}$ is the initial marking. Moreover, $\forall t \in T_N : col_X(t^*) \subseteq r_n(t)$ (i.e., variables in the postset are bound to received names).

Note that the initial marking $m_{0N}$ is a multiset were colours are constants (in fact $m_{0N} \in \mathcal{M}_{S_N, C_N \cup \mathcal{V}}$). We remark that markings in a coloured net do not contain variables, and that variables are used only as formal parameters of transitions.

**Definition 4.5 (Coloured marking).** A multiset $m$ is a marking of a coloured net $N$ iff $m \in \mathcal{M}_{S_N, C_N}$ and $col(m) \subseteq \mathcal{V}$.

**Notation 4.1.** By abusing notation, we often write $t^*$ (similarly $t^*$) also to denote just the set of places in the postset of $t$ (resp. in the preset of $t$), i.e. \{a | (a, c) \in t^*\} (resp. \{a | (a, c) \in t^*\}).

As aforementioned, the firing of a transition $t$ in a coloured net requires to instantiate $t$ with appropriate colours, i.e., those corresponding to tokens present in places. Consequently, the instantiation of a transition corresponds to a substitution on colour variables.

**Definition 4.6 (Substitution on colours).** Let $\sigma : \mathcal{X} \rightarrow \mathcal{V} \cup \mathcal{X}$ be a partial function. The substitution $x \sigma$ on a colour variable $x$ is $c$ if $\sigma(x) = c$, otherwise it is $x$, i.e., the identity when $\sigma$ is not defined on $x$. Instead, the substitution $v \sigma$ on a constant colour $v$ produces $v$, i.e., it has no effect. The substitution on a colour sequence is the simultaneous substitution of the names appearing in the sequence, i.e., $(c_1, \ldots, c_n) \sigma = (c_1 \sigma, \ldots, c_n \sigma)$. The colour substitution on a multiset $m \in \mathcal{M}_{S,C}$ is given by $(m \ast \sigma)(s)(c) = \sum_{c \in S} m(c)(c)$. The composed substitution $(\sigma_1; \sigma_2)$ applied over a multiset (and similarly for variables and constants) is defined as $m \ast (\sigma_1; \sigma_2) = (m \ast \sigma_1) \ast \sigma_2$.

Note that the names of colour variables used in the preset of a transition are meaningless. Actually, they act as binders whose scope is just that transition, and consequently they can be renamed without modifying the meaning of a transition. We define the following relation over transitions, called $\alpha$-conversion on received colours.
**Definition 4.7** (α-equivalence of transitions). Two transitions \( t_1 = m_1 \parallel m'_1 \) and \( t_2 = m_2 \parallel m'_2 \) are \( \alpha \)-convertible if there exists an injective substitution \( \sigma : X \rightarrow X \), where \( \text{rn}(t_1) \subseteq \text{dom}(\sigma) \), such that \( m_1 \circ \sigma = m_2 \) and \( m'_1 \circ \sigma = m'_2 \). The \( \alpha \)-conversion is an equivalence relation, which is denoted by \( \equiv_\alpha \). (Note that since \( \sigma \) is injective, then \( \sigma^{-1} \) exists and \( m_2 \circ \sigma^{-1} = m_1 \) and \( m'_2 \circ \sigma^{-1} = m'_1 \) hold). We usually talk about transitions up-to \( \alpha \)-equivalence.

### 4.1.1 Operational semantics

The operational semantics of C-P/T nets is given by replacing the rule (Firing) in Figure 2.1 by the following version:

\[
\frac{t = m \mid m' \in T \quad m'' \in \mathcal{M}_{S,C\mathcal{N}V} \quad \text{rn}(t) \subseteq \text{dom}(\sigma) \quad \text{range}(\sigma) \subseteq Y}{m \circ \sigma \oplus m'' \rightarrow_T m' \circ \sigma \oplus m''}
\]

This rule states that a transition \( t \) can be fired whenever the marking contains an instance of its preset obtained by substituting all colour variables by constants. In this case, the tokens corresponding to the instance of the preset \( m \circ \sigma \) are consumed and a suitable instance of the postset is produced accordingly.

**Example 4.1.** Figure 4.1 shows a simple C-P/T net, with the initial marking \( a_1(v) \oplus a_2(v_1) \oplus a_2(v_2) \oplus a_3(v_2) \), while Figure 4.2 shows a possible computation in which \( t_1 \) and \( t_2 \) are fired concurrently.

### 4.2 Coloured Deterministic Processes

In this section we revise the notions of morphism, causal net and processes in order to take into account colours.
\[ a_1(x) \oplus a_2(v_1) \mid b_1(v_2) \in T \quad \sigma = \{ v/x \} \] (CF) \\
\[ a_2(y) \oplus a_3(y) \mid b_2(y) \in T \quad \sigma = \{ v_2/y \} \] (CF) \\
\[ a_1(w) \oplus a_2(v_2) \rightarrow_T b_1(v_2) \] (Step) \\
\[ a_2(v_2) \oplus a_3(v_2) \rightarrow_T b_2(v_2) \] (Step) \\
\[ a_1(w) \oplus a_2(v_1) \oplus a_2(v_2) \oplus a_3(v_2) \rightarrow_T b_1(v_2) \oplus b_2(v_2) \] (Step)

Figure 4.2: A computation of a coloured net.

**Definition 4.8** (Coloured net morphism). Let \( N, N' \) be c-p/T nets. A tuple \( f = (f_S: S_N \rightarrow S_{N'}, f_T: T_N \rightarrow T_{N'}, \sigma = \{ \sigma_t \}_{t \in T_N}) \), where \( \sigma \) is a family of substitutions (one for each \( t \in T_N \)) s.t. \( \sigma_t: rn(f_T(t)) \rightarrow col(\ast t) \), is said a coloured net morphism from \( N \) to \( N' \) (written \( f: N \rightarrow_N N' \)) if \( f_S(\ast t) \mid f_S(\ast t) = \ast f_T(t) \ast \sigma_t \ast f_T(t) \ast \sigma_t \).

Note that a morphism explains the correspondence not only between places and transitions on both nets, but also between the colours used by transitions. Note that the direction of the \( \sigma_t \)'s is contravariant w.r.t. that of \( f \). Each \( \sigma_t \) relates the colour variables appearing in \( f_T(t) \) with the colours used by \( t \). Moreover, transitions in \( N \) are required to be a particular case of those in \( N' \).

**Definition 4.9** (Composition of coloured morphisms). Given two coloured morphisms \( f_1: N \rightarrow_{\sigma_1} N' \) and \( f_2: N' \rightarrow_{\sigma_2} N'' \), the composition of \( f_1 \) and \( f_2 \) (written \( f_1; f_2 \)) is the morphism \( f: N \rightarrow_{\sigma} N'' \), where \( f_S = f_{1S}; f_{2S}, f_T = f_{1T}; f_{2T} \) (i.e., the mappings for places and transitions are the usual composition of the original mappings) and \( \sigma = \{ \sigma_{2f_1(t)}; \sigma_{1t} \}_{t \in T_N} \), i.e., the names used by a transition \( t \) in \( N \) are related with those on \( f_T(t) \in N'' \) by substituting the variables of \( f_T(t) \) first as said by \( \sigma_2 \) for the transition \( f_{1T}(t) \) (i.e., for the transition in \( N' \) associated with \( t \)) and then by using \( \sigma_{1t} \).

It is easy to check that \( f = f_1; f_2 \) is a morphism from \( N \) to \( N'' \). First of all, by \( f_1 \) we have that \( \forall t \in T_N : \)

\[ f_{1S}(\ast t) \mid f_{1S}(\ast t) = \ast f_{1T}(t) \ast \sigma_{1t} \ast f_{1T}(t) \ast \sigma_{1t} \] (4.1)

Since \( f_{1T}(t) \in T_{N'} \), by \( f_2 \) we have that

\[ f_{2S}(\ast f_{1T}(t)) \mid f_{2S}(\ast f_{1T}(t)) = \ast f_{2T}(f_{1T}(t)) \ast \sigma_{2f_1(t)} \ast f_{1T}(f_{1T}(t)) \ast \sigma_{2f_1(t)} \ast \sigma_{1t} \]

Then instantiating both transitions with the substitution \( \sigma_{1t} \), we have

\[ f_{2S}(\ast f_{1T}(t)) \ast \sigma_{1t} \mid f_{2S}(f_{1T}(t)) \ast \sigma_{1t} = \ast f_{2T}(f_{1T}(t)) \ast \sigma_{2f_1(t)} \ast \sigma_{1t} \ast f_{1T}(f_{1T}(t)) \ast \sigma_{2f_1(t)} \ast \sigma_{1t} \ast f_{1T}(f_{1T}(t)) \ast \sigma_{1t} \]

As \( f_{2S} \) and \( \sigma_{1t} \), have disjoint domains, they commute, and hence the previous equation can be written as

\[ f_{2S}(\ast f_{1T}(t)) \ast \sigma_{1t} \mid f_{2S}(f_{1T}(t)) \ast \sigma_{1t} = \ast f_{2T}(f_{1T}(t)) \ast \sigma_{2f_1(t)} \ast \sigma_{1t} \ast f_{1T}(f_{1T}(t)) \ast \sigma_{2f_1(t)} \ast \sigma_{1t} \ast f_{1T}(f_{1T}(t)) \ast \sigma_{1t} \]
By equation (4.1), \( f_{1T}(t) \star \sigma_{1t} = f_{1S}(t) \) and \( f_{1T}(t) \star \sigma_{1t} = f_{1S}(t') \). By using these equalities in the previous equation,

\[
\begin{align*}
&f_{2S}(f_{1S}(t)) \circ f_{2S}(f_{1S}(t')) = \\
&\quad \cdot f_{2T}(f_{1T}(t)) \star \sigma_{2t} \star \sigma_{1t} \circ f_{1T}(t) \star \sigma_{2t} \star \sigma_{1t}
\end{align*}
\]

And finally, by definition of composition of functions and substitutions, the following holds

\[
\begin{align*}
&f_{1S} ; f_{2S}(t) \circ f_{1S} ; f_{2S}(t') = \\
&\quad f_{1T} ; f_{2T}(t) \star \sigma_{2T} \star \sigma_{1T} \circ f_{1T} ; f_{2T}(t) \star \sigma_{2T} \star \sigma_{1T}
\end{align*}
\]

Note that coloured nets and coloured net morphisms form a category. The identity of the composition is \( id = (id_{ST}, id_{T}, \{\sigma_{t}\}_{t \in T}) \), where \( \sigma_{t} \) is the identity on \( \mathbb{R}(t) \).

**Definition 4.10** (Isomorphic Coloured Nets). Two nets \( N_{1} \) and \( N_{2} \) are isomorphic, written \( N_{1} \cong N_{2} \), if there exists a morphism \( (f_{S}, f_{T}, \{\sigma_{t}\}_{t \in T}) \) s.t. both \( f_{S} \) and \( f_{T} \) are injective, and all \( \sigma_{t} \) are injectiveRenameings (i.e. substitute variables by variables).

**Definition 4.11** (Deterministic Coloured Causal Net). A coloured net \( K = (S_{K}, C_{K}, T_{K}, \delta_{0K}, \delta_{1K}) \) is a **deterministic coloured causal net** if it is acyclic and transitions do not share:

- places in their pre- and postsets, i.e. if \( a \in \delta_{iN}(t_{0}) \) and \( a \in \delta_{iN}(t_{1}) \) then \( t_{0} = t_{1} \) for \( i = 0, 1 \); nor
- colour variables, i.e. \( \forall t_{1}, t_{2} \in T_{K} : t_{1} \neq t_{2} \) implies \( \mathbb{R}(t_{1}) \cap \mathbb{R}(t_{2}) = \emptyset \).

**Remark 4.1.** The condition about disjoint colour variables is not a restriction, since \( \alpha \)-conversion on received names allows us to rewrite any net into an isomorphic net s.t. transitions do not to share variables.

**Example 4.2.** Figure 4.3 shows three C-P / T nets \( K_{1}, N \) and \( K'_{1} \), and two coloured morphisms \( P_{1} : K_{1} \rightarrow_{\sigma} N \) and \( P_{1}' : K'_{1} \rightarrow_{\sigma'} N \). Nets \( K_{1} \) and \( K'_{1} \) in Figure 4.3 are deterministic coloured causal nets. Differently, \( N \) is not a deterministic coloured causal net because it has a cycle. The morphisms are depicted by showing with dotted lines the mapping between elements. The substitutions used to instantiate transitions are shown over the arrows connecting transitions. Note that all transitions in \( K_{1} \) corresponds to instances of transitions in \( N \) where all variables have been substituted by constants. Differently, the variables of the transitions in \( K'_{1} \) are just a renaming of those in \( N \).
When viewing deterministic coloured causal nets like $K_1$ and $K'_1$ as descriptions of (concurrent) executions, it is easy to notice that their execution flow can be blocked because the constraints on coloured tokens (like variable bindings) of adjacent transitions are inconsistent. Consider the net $K_1$ in Figure 4.3. This net does not describe a correct execution from the initial place $a_1$ to the final place $a_3$ because $t_1$ produces a token coloured with the constant $v$ in $a_2$ while $t_2$ needs a token coloured with the constant $w$. The following definition characterises all those nets that describe consistent computations from the initial places to the final ones.

**Definition 4.12** (Compatible execution of $K$). Let $K$ be a coloured causal. A substitution $\rho$ is said a compatible execution of $K$ if $\forall t_1, t_2 \in T_K : (a, c_1) \in t_1^*$ and $(a, c_2) \in t_2^*$ then $c_1\rho = c_2\rho$. If such $\rho$ exists, then we call $K$ compatible.

**Example 4.3.** Net $K_1$ in Figure 4.3 is not compatible. In fact there is no substitution $\rho$ that unifies $v$ (the colour of tokens produced in $a_2$) with $w$ (the colour of tokens consumes from $a_2$). Differently, $K'_1$ is compatible. In fact, transitions $t'_1$ and $t'_2$ do not share variables, and the substitution $\rho = \{y/z\}$ unifies the colours of tokens produced in and consumed from $b_2$, which is the only place of $K'_1$ shared by transitions.

A compatible execution captures the notion of unification that takes place when computing in a coloured net. Nevertheless, there can be many possible substitutions. For instance, also $\rho' = \{v/y, v/z\}$ is a compatible execution of $K'_1$. As we are interested on capturing the most general definition for equivalent executions, we associate to a compatible causal net the less restrictive constraints on colours that allow a complete execution, which are called the most general compatible execution.

**Definition 4.13** (mgce). A compatible execution $\rho_K$ is said the most general compatible execution (shorten as mgce) of $K$ if for every other compatible execution $\rho$
there exist a substitution \( \gamma \) s.t. \( \rho = \rho_K; \gamma \).

A different way in which nets can be blocked is when tokens produced in and consumed from a place are sequences with different length. In these cases, there is not unification for colours. In what follows we defined a particular kind of causal nets (called \textit{plain}) that are always compatible. We will use a particular case of plain nets in Section 4.6, when considering P/T and ZS nets respectively as C-P/T and coloured ZS nets.

**Definition 4.14** (Plain nets). A causal net \( K \) is a \textit{plain net} if \( \forall a \in S_K : \exists k \in \mathbb{N} \) s.t. \( \forall (t, c) \in \cdot a \cup \cdot a^* \), \( \tau \subseteq X \land |c| = k \), where \( |c| \) denotes the length of \( c \).

**Proposition 4.1.** If \( K \) is a plain net, then \( K \) is compatible.

Proof. The proof follows by considering any substitution \( \rho \) s.t. \( \rho(x) = \rho(y) \) for all \( x, y \in C_K \). It is straightforward to prove that \( \rho \) is a compatible execution of \( K \) because \( \forall a \in S_K \), if \( (a, c_1) \in t_1^* \) and \( (a, c_2) \in t_2^* \) then \( |c_1| = |c_2| \). Moreover, \( \tau \subseteq X \) and \( \tau \subseteq X \). Hence, \( c_1 \) and \( c_2 \) are sequence of variables with the same length, and consequently \( c_1 \rho = c_2 \rho \). \( \Box \)

Plain nets can be thought as nets where places are typed (in this simple model types refer to the length of colour sequences), the flow relation preserves types, and transitions do not use pattern matching with constants on consumed tokens.

The notion of processes is given in terms of compatible causal nets.

**Definition 4.15** (Process of a coloured net). A \textit{deterministic causal process} for a \( \text{C-P/T} \) net \( N \) is coloured net morphism \( P : K \rightarrow_\sigma N \) from a compatible coloured causal net \( K \) to \( N \), s.t. \( \forall \sigma_t \in \sigma, \sigma_t \) is a renaming (i.e. \( \sigma_t : X \rightarrow X \) and injective).

A process \( P \) associates a compatible coloured causal net \( K \) to a \( \text{C-P/T} \) net \( N \). Since \( K \) is itself a coloured net, its transitions can be fired for any compatible execution. Therefore, a process describes a set of runs with the same shape that start from markings that only differ in their colours. For instance, the process \( P'_1 : K'_1 \rightarrow N \) shown in Figure 4.3 describes a computation that fires twice the transition \( t \), independently from the colours of tokens. (Our approach is similar to that presented in [5], a more detailed discussion is in Section 4.7).

As for processes of \( \text{P/T} \) nets, the set of \textit{origins} and \textit{destinations} of a process \( P : K \rightarrow N \) are \( O(P) = ^oK \) and \( D(P) = K^o \cap S_K \), respectively. The set of \textit{evolution places} of \( P \) is the set \( E_P = \{ P(a) | a \in K, |\cdot a| = |\cdot a^*| = 1 \} \).

**Definition 4.16** (Process Isomorphism). Two coloured processes \( P_1 : K_1 \rightarrow_{\sigma_1} N \) and \( P_2 : K_2 \rightarrow_{\sigma_2} N \) are \textit{isomorphic} (written \( P_1 \approx P_2 \)) and thus equivalent if there exists an injective coloured net morphism \( f : K_1 \rightarrow_{\sigma_1} K_2 \) (i.e., \( f_S \) and \( f_T \) injective and all \( \sigma_t \) injective variable renamings) such that \( f; P_2 = P_1 \).

Note that the isomorphism between processes defines an equivalence relation over processes. We denote with \( [P]_\approx \) the equivalence class of \( P \).
4.3 Correspondence between processes and firings

This section is devoted to show the relation between coloured processes and firings in C-P/T nets. In particular, we prove that for each computation on C-P/T (i.e., a sequence of steps) there exists (at least) an associated process and vice versa. We start by fixing the notation needed for showing the correspondence. We recall that the set of origins and destinations of $P : K \rightarrow N$ are respectively $O(P) = \circ K$ and $D(P) = K^c \cap S_K$, while the isolated places are $I(P) = O(P) \cup D(P)$. Moreover, $pre(P)$ and $post(P)$ stand for the multisets of initial and final markings of $P$, i.e. $pre(P) = P(O(P))$ and $post(P) = P(D(P))$.

**Notation 4.2.** Let $P : K \rightarrow N$ be a coloured process, the sets $O_c(P)$ and $D_c(P)$ of connected initial and final places are $O_c(P) = O(P) \setminus I(P)$ and $D_c(P) = (D(P) \setminus I(P))$ respectively. The sets of coloured connected initial and final places are defined as follow:

$pre_c(P) = \{(a, c) | a \in O_c(P) \land t \in T_K \land (a, c) \in \bullet t\}$

$post_c(P) = \{(a, c) | a \in D_c(P) \land t \in T_K \land (a, c) \in t^*\}$

Given a total function $a : I(P) \rightarrow \nu^*$, i.e. a colour assignment for isolated places, the set of coloured isolated places is defined as

$I(P, a) = \{(a, a(a)) | a \in I(P)\}$

Given a set of places $S \subseteq S_K$, we write $pre_c(P) |_S$, as a shorthand for $\{(a, c) \in pre_c(P) | a \in S\}$. Similarly for $post_c(P) |_S$ and $I(P, a) |_S$.

In order to build the processes associated to a sequence of steps we will use the following auxiliary relations defining suitable mappings between places, markings and colours.

**Definition 4.17 (Suitable place mapping (F)).** Given a set of places $S_K$ and a coloured multiset $m \in \mathcal{M}_{S_K,c}$, a function $f_S : S_K \rightarrow S_N$ is a suitable place mapping from $S_K$ to $S_N$ if $F(S_K, m, f_S)$, where $F$ is the relation $F : S_K \times \mathcal{M}_{S_N,c} \times (S_K \rightarrow S_N)$ defined recursively as

\[
\begin{align*}
& \{ F(\emptyset, \emptyset, \emptyset) \\
& F(a \uplus S, b(c) \oplus m, a \mapsto b \uplus f) \text{ if } F(S, m, f) \}
\end{align*}
\]

**Definition 4.18 (Pre-marking (G)).** Given a set of places $S_K$, a coloured multiset $\mathcal{M}_{S_N,c}$, and a suitable place mapping $f_S$ from $S_K$ to $S_N$, a pre-marking of $m$ is a multiset $m' \in \mathcal{M}_{S_K,c}$ s.t. $G(S_K, m, f_S, m')$, where $G$ is the relation $G : S_K \times \mathcal{M}_{S_N,c} \times (S_K \rightarrow S_N) \times \mathcal{M}_{S_K,c}$ defined recursively by

\[
\begin{align*}
& \{ G(\emptyset, \emptyset, \emptyset, \emptyset) \\
& G(a \uplus S, b(c) \oplus m, a \mapsto b \uplus f, a(c) \oplus m') \text{ if } G(S, m, f, m') \}
\end{align*}
\]
Remark 4.2 (Properties of $F$).

1. For every $m'$ s.t. $G(S, m, f, m')$ then there is no $a \in S$ s.t. $m' = a(c_1) \oplus a(c_2) \oplus m''$. Proof by contradiction. Note if this is the case then $f$ is not a function.

2. For every $m'$ s.t. $G(S, m, f, m')$, then $f(m') = m$ and $f(m' \ast \sigma) = m \ast \sigma$. Proof by induction on $|m|$. The equality $f(m' \ast \sigma) = m \ast \sigma$ holds because the domain of $f$ is disjoint to the domain and range of $\sigma$, and therefore $f(m' \ast \sigma) = f(m') \ast \sigma$.

Definition 4.19 (Suitable colour assignment $(A)$). Given a set of places $S_K$, a coloured multiset $m \in M_{S_N, C}$, and a suitable place mapping $f_S$ from $S_K$ to $S_N$, a suitable colour assignment is a function $a : S_K \rightarrow C$ s.t. $A(S_K, m, f_S, a)$, where $A$ is the relation defining all possible colour assignments, i.e., $A : S_K \times M_{S_N, C} \times (S_K \rightarrow C)$ defined by the following rules.

\[
\begin{align*}
& A(\emptyset, \emptyset, \emptyset) \\
& A(a \uplus S, b(c) \oplus m, a \mapsto b \uplus f, a \mapsto c \uplus a) \text{ if } A(S, m, f, a)
\end{align*}
\]

Remark 4.3 (Property of $A$). $f\{(a, a(a)) | a \in S \land A(S, m, f, a)\} = m$. Proof by induction on $|m|$.

Example 4.4. Consider the set of places $S_K = \{a_1, a_2, b_1\}$ and $S_N = \{a, b\}$ and a marking $m = a(c_1) \oplus a(c_2) \oplus b(c_3)$. Then, the suitable place mappings $f_S$ s.t. $F(S_K, m, f_S)$ are the following

\[
\begin{align*}
& f_{S_1} = \{a_1 \mapsto a, a_2 \mapsto a, b_1 \mapsto b\} \\
& f_{S_2} = \{a_1 \mapsto a, a_2 \mapsto b, b_1 \mapsto a\} \\
& f_{S_3} = \{a_1 \mapsto b, a_2 \mapsto a, b_1 \mapsto a\}
\end{align*}
\]

Note any $f_{S_i}$ fixes a mapping between the sets $S_K$ and $S_N$ that is consistent with the marking $m$. Then, by selecting a mapping $f_{S_i}$, a pre-marking $m'$ of $m$ is a marking s.t. $f_{S_i}(m') = m$. In general, given a marking and a mapping there can be several pre-markings. For instance, the possible pre-markings $m'$ of $m$ when considering $f_{S_1}$ are

\[
\begin{align*}
m'_1 & = a_1(c_1) \oplus a_2(c_2) \oplus b_1(c_3) \\
m'_2 & = a_2(c_1) \oplus a_1(c_2) \oplus b_1(c_3)
\end{align*}
\]

Similarly, a suitable colour assignment fixes a colour for any state in $S_K$ that satisfies the marking $m$. For instance all the assignments $a$ s.t. $A(S_K, m, f_{S_1}, a)$ are below

\[
\begin{align*}
a_1 & = \{a_1 \mapsto c_1, a_2 \mapsto c_2, b_1 \mapsto c_3\} \\
a_2 & = \{a_1 \mapsto c_2, a_2 \mapsto c_1, b_1 \mapsto c_3\}
\end{align*}
\]
Lemma 4.2. Let $N = (S_N, C_N, T_N, \delta_{0N}, \delta_{1N})$ be a C-P/T net. If $m_1 \rightarrow_{T_N} m_2$, then there exist a process $P : K \rightarrow N$, a compatible execution $\rho : X \rightarrow V$ of $K$, and a colour assignment $a$ s.t.

- $m_1 = P(\text{pre}_c(P)) \ast \rho \oplus P(I(P, a))$
- $m_2 = P(\text{post}_c(P)) \ast \rho \oplus P(I(P, a))$

Proof. By rule induction.

- **Rule(coloured-firing)**: $\exists t = m \mid m' \in T_N$ s.t. $m_1 = m \ast \sigma_1 \oplus m''$ and $m_2 = m' \ast \sigma_1 \oplus m''$, where $\text{dom}(\sigma_1) \subseteq \text{rn}(t)$ and $\text{range}(\sigma_1) \subseteq V$. Define a coloured causal net $K = (S_K, C_K, T_K, \delta_{0K}, \delta_{1K})$ as follow:
  
  - $S_K = S_{K_m} \uplus S_{K_m'} \uplus S_{K_m''}$, where $|S_{K_m}| = |m|$, $|S_{K_m'}| = |m'|$, $|S_{K_m''}| = |m''|$. (We recall that $\uplus$ denotes the union of disjoint sets).
  
  - $C_K = \text{col}_V(m \uplus m') \subseteq X$, where $X \subseteq X$ and $|X| = |\text{col}_X(t)|$
  
  - $T_K = \{t\}$

  Then define $f_s = f_{S_{K_m}} \uplus f_{S_{K_m'}} \uplus f_{S_{K_m''}}$ by choosing any $f_{S_{K_m}}$, $f_{S_{K_m'}}$, and $f_{S_{K_m''}}$ such that $F(S_{K_m}, m, f_{S_{K_m}}), F(S_{K_m'}, m', f_{S_{K_m'}})$ and $F(S_{K_m''}, m'', f_{S_{K_m''}})$. Note such mappings exist since, e.g., $S_{K_m}$ and $m$ has the same size. Finally, take any bijective function $\sigma_t : \text{col}_X(t) \rightarrow X$ (i.e., a variable renaming) and define

  $\delta_{0K}(t') = n_1 \ast \sigma_{t'}$ for any $n_1$ s.t. $\mathcal{g}(S_{K_m}, m, f_{S_{K_m}}, n_1)$

  $\delta_{1K}(t') = n_2 \ast \sigma_{t'}$ for any $n_2$ s.t. $\mathcal{g}(S_{K_m'}, m', f_{S_{K_m'}}, n_2)$

By construction, $K$ is a coloured net. In fact, by Remark 4.2(1), all places on the pre- and postset of the unique transition are different. Moreover $K$ is a causal net. Clearly there is no sharing of places among transitions and $K$ is acyclic, since the pre- and postset of $t'$ are disjoint (we recall that $K_m$ and $K_m'$ are disjoint sets).

Note that $K$ has only one transition, and hence there is no sharing of colour variables. Moreover, since places are part of either the preset or the postset of the unique transition, any substitution is a compatible execution of $K$.

It remains to prove that there exists a process $P : K \rightarrow N$. Consider the triple $P = (f_s, f_T, \{\sigma_{t'}\})$, where $f_s$ and $\sigma_{t'}$ are the functions chosen previously, and $f_T$ maps $t'$ to $t$, i.e. $f_T(t') = t$. This triple is a morphism from $K$ to $N$, in fact

\[
\begin{align*}
  f_s(t') = f_s(n_1 \ast \sigma_{t'}) & = f_s(n_1 \ast \sigma_{t'}) & \text{By def. of } K \\
  & = m \ast \sigma_{t'} \ast m' \ast \sigma_{t'} & \text{By Remark 4.2(2)} \\
  & = \ast t \ast \sigma_{t'} \ast t' \ast \sigma_{t'} & \text{By def. of } N \\
  & = \ast f_T(t') \ast \sigma_{t'} \ast f_T(t') \ast \sigma_{t'} & \text{By def. of } f_T
\end{align*}
\]

Since $\sigma_{t'}$ is bijective, also $\sigma_{t'}^{-1}$ is a well-defined renaming. Then, take the substitution $\rho = \sigma_{t'}^{-1}; \sigma_1$ as a compatible execution of $K$. (We recall that $\cdot$ stands for the composition of substitutions, i.e. $m \ast (\sigma_i; \sigma_j) = m \ast \sigma_i \ast \sigma_j$). Note that
\[ P(\text{pre}_c(P)) \ast \rho = P(\text{pre}_c(P)) \ast (\sigma^{-1}_t; \sigma_1) \] By def. of \( \rho \)
\[ = P(\ast t') \ast \sigma^{-1}_t \ast \sigma_1 \] As \( t' \) is the unique transition
\[ = P(m_1 \ast \sigma_t) \ast \sigma^{-1}_t \ast \sigma_1 \] By definition of \( \delta_{0K}(t) \)
\[ = f_S(m_1 \ast \sigma_t) \ast \sigma^{-1}_t \ast \sigma_1 \] By definition of \( P \)
\[ = (m \ast \sigma_t) \ast \sigma^{-1}_t \ast \sigma_1 \] By Remark 4.2(2)
\[ = m \ast (\sigma_t; \sigma^{-1}_t) \ast \sigma_1 \] By associativity
\[ = m \ast \sigma_1 \] By composition of inverses

Similarly, it can be show that \( P(\text{post}_c(P)) \ast \rho = m' \ast \sigma_1. \)

As far as isolated places are concerned, by definition of \( K \), the isolated places of \( P \) are \( S_{K_{m''}} \). Moreover, \( |m''| = |S_{K_{m''}}| = |I(P)| \). Then, select a colour assignment \( a \) s.t. \( K(S_{K_{m''}}, m'', f_S_{K_{m''}}, a) \). With this assignment we have

\[ P(I(P, a)) = P(\{(a, a(a)) \mid a \in I(P)\}) \] By def. of \( I(P, a) \)
\[ = P(\{(a, a(a)) \mid a \in S_{K_{m''}}\}) \] By def. of \( K \) and \( \delta_{iK}(t) \)
\[ = f_S(\{(a, a(a)) \mid a \in S_{K_{m''}}\}) \] By def. of \( P \)
\[ = f_{S_{K_{m''}}}(\{(a, a(a)) \mid a \in S_{K_{m''}}\}) \] By def. of \( f_S \)
\[ = m'' \] By Remark 4.3

From the above equalities we have that there exist \( P, \rho \) and \( a \) s.t.: 
\[ P(\text{pre}_c(P)) \ast \rho \oplus P(I(P, a)) = m \ast \sigma_1 \oplus m'' = m_1 \]
\[ P(\text{post}_c(P)) \ast \rho \oplus P(I(P, a)) = m' \ast \sigma_1 \oplus m'' = m_2 \]

- Rule (step): Then \( m_1 = m'_1 \oplus m'_2, m_2 = m''_1 \oplus m''_2, m'_1 \rightarrow_T \sigma_1 m''_1 \) and \( m'_2 \rightarrow_T \sigma_1 m''_2 \).

By inductive hypothesis there are two processes \( P_1 \) and \( P_2 \) defined as \( P_i : K_i \rightarrow \sigma_i N \) with compatible executions \( \rho_i \), and colour assignments \( a_i \) s.t. the following conditions hold
\[ m'_i = P_i(\text{pre}_c(P_i)) \ast \rho_i \oplus P_i(I(P_i, a_i)) \]
\[ m''_i = P_i(\text{post}_c(P_i)) \ast \rho_i \oplus P_i(I(P_i, a_i)) \]

As the names of places \( S_{K_i} \), transitions \( T_{K_i} \), and colour variables \( C_{K_i} \cap \chi \) can be chosen arbitrarily, w.l.o.g. we can assume that \( S_{K_1} \cap S_{K_2} = \emptyset, T_{K_1} \cap T_{K_2} = \emptyset \) and \( C_{K_1} \cap C_{K_2} \cap \chi = \emptyset \). Then define \( K = (S_{K_1} \cup S_{K_2}, C_{K_1} \cup C_{K_2}, T_{K_1} \cup T_{K_2}, \delta_{0K_1} \cup \delta_{0K_2}, \delta_{1K_1} \cup \delta_{1K_2}) \). Functions \( \delta_{iK} = \delta_{iK_1} \cup \delta_{iK_2} \) are well-defined because \( \text{dom}(\delta_{iK_1}) \cap \text{dom}(\delta_{iK_2}) = \emptyset \). Moreover, \( K \) is a compatible coloured causal net because it is the disjoint union of two compatible causal nets.

A coloured net morphism \( f \) from \( K \) to \( N \) can be defined in terms of the morphisms \( f_1 \) and \( f_2 \) (corresponding to the processes \( P_1 \)) as follow
\[ f = (f_S = f_{1S} \uplus f_{2S}, f_T = f_{1T} \uplus f_{2T}, \sigma = \sigma_1 \uplus \sigma_2) \]

The functions \( f_S \) and \( f_T \) and the family of renamings \( \sigma \) are well-defined because \( P_1 \) and \( P_2 \) are processes with disjoint causal nets. Moreover, \( f \) is a coloured net
4.3. CORRESPONDENCE BETWEEN PROCESSES AND FIRINGS

morphism (it is trivial to show that  \( f_S(t^*) | f_S(t^*) = f_T(t) * \sigma_i | f_T(t) * \sigma_i \) by conveniently using \( f_i \) for \( t \in K_i \)). Since all substitutions in \( \sigma \) are renamings (i.e., they are either the renamings used by \( P_1 \) or \( P_2 \)), the morphism \( f \) is a process, and consequently we note \( f \) as \( P \).

As \( K \) does not add transitions, the sets of isolated, initial and final places correspond to the union of those in \( K_1 \) and \( K_2 \). Then, consider \( \rho = \rho_1 \uplus \rho_2 \) and \( a = a_1 \uplus a_2 \). Note they are well-defined functions because \( \rho_1 \) and \( \rho_2 \) (similarly \( a_1 \) and \( a_2 \)) have disjoint domains (in fact it is enough to consider \( \rho_i \) and \( a_i \) restricted to the colour variables of \( P_i \)). Then

\[
P(pre_c(P)) \ast \rho \uplus P(I(P, a)) = \quad \text{By def. of } P
\]
\[
P(pre_c(P_1) \cup pre_c(P_2)) \ast \rho \uplus P(I(P_1, a) \cup I(P_2, a)) = \quad \text{By def. of } P
\]
\[
(P(pre_c(P_1)) \uplus P(pre_c(P_2))) \ast \rho \uplus P(I(P_1, a)) \uplus P(I(P_2, a)) = \quad \text{by distributivity of } \ast
\]
\[
P(pre_c(P_1)) \ast \rho \uplus P(I(P_1, a)) \uplus P(I(P_2, a)) = \quad \text{since variables are different in } P_1 \text{ and } P_2 \text{ (and also for } \rho_i \text{ and } a_i)
\]
\[
P_1(pre_c(P_1)) \ast \rho_1 \uplus P_2(pre_c(P_2)) \ast \rho_2 \uplus P_1(I(P_1, a_1)) \uplus P_2(I(P_2, a_2)) = \quad \text{By associativity and commutativity of } \uplus
\]
\[
(P_1(pre_c(P_1)) \ast \rho_1 \uplus P_1(I(P_1, a_1))) \uplus (P_2(pre_c(P_2)) \ast \rho_2 \uplus P_1(P_2, a_2)) = \quad \text{As } m'_i = P_i(pre_c(P_i)) \ast \rho_i \uplus P_i(I(P_i, a_i))
\]
\[
m_1 \ast m_2 = m_1
\]

Similarly, it can be shown that \( P(post_c(P)) \ast \rho \uplus P(I(P, a)) = m''_1 \ast m''_2 = m_2 \).

\[\square\]

Lemma 4.3. Let \( N = (S_N, C_N, T_N, \delta_0N, \delta_1N) \) be a c-p/T net. If \( m_1 \rightarrow^*_T m_2 \), then there exists a process \( P : K \rightarrow_{\sigma} N \), a compatible execution \( \rho : X \rightarrow V \) of \( K \), and a colour assignment \( a \) s.t.

- \( m_1 = P(pre_c(P)) \ast \rho \uplus P(I(P, a)) \)
- \( m_2 = P(post_c(P)) \ast \rho \uplus P(I(P, a)) \)

Proof. By induction on the length \( n \) of the derivation.

- **Base case** \( n = 0 \): In this case \( m_1 = m_2 \). Then, define a coloured causal net \( K = (S_K, C_K, T_K, \delta_0K, \delta_1K) \) as follow:
  
  - Choose \( S_K \) s.t. \( |S_K| = |m_1| \).
  - \( C_K = colV(m_1) \).
  - \( T_K = \delta_0K = \delta_1K = \emptyset \).

Then select \( f_S \) s.t. \( F(S_K, m_1, f_S) \) and a colour assignment \( a \) s.t. \( A(S_K, m_1, f_S, a) \), as in the proof of Lemma 4.2. Clearly \( P(pre_c(P)) = P(post_c(P)) = \emptyset \) as there are no transitions in \( K \). For isolated places we have
\[ P(I(P, a)) = P(\{(a, a(a))|a \in I(P)\}) \quad \text{By def. of } I \]
\[ = P(\{(a, a(a))|a \in S_K\}) \quad \text{By def. of } K, I(P) = S_K \]
\[ = f_S(\{(a, a(a))|a \in S_K\}) \quad \text{By def. of } P \]
\[ = m_1 \quad \text{By Remark 4.3} \]
\[ = m_2 \quad \text{Because } n = 0 \]

- **Inductive Step**: \( m_1 \rightarrow_{T_N} m_3 \rightarrow_{T_N}^{*} m_2 \). Since \( m_1 \rightarrow_{T_N} m_3 \), by Lemma 4.2, there exists a process \( P_1 \) with a compatible execution \( \rho_1 \) and a colour assignment \( a_1 \) s.t.

\[ - m_1 = P_1(\text{pre}_c(P_1)) \ast \rho_1 \oplus P_1(I(P_1, a_1)) \]
\[ - m_3 = P_1(\text{post}_c(P_1)) \ast \rho_1 \oplus P_1(I(P_1, a_1)) \]

By inductive hypothesis applied on \( m_3 \rightarrow_{T_N}^{*} m_2 \), there exists a process \( P_2 \) with a compatible execution \( \rho_2 \) and a colour assignment \( a_2 \) s.t.

\[ - m_3 = P_2(\text{pre}_c(P_2)) \ast \rho_2 \oplus P_2(I(P_2, a_2)) \]
\[ - m_2 = P_2(\text{post}_c(P_2)) \ast \rho_2 \oplus P_2(I(P_2, a_2)) \]

Since the names used for defining places, transitions and colour variables in \( K_i \) can be chosen freely, select them in the following way

- \( T_{K_1} \cap T_{K_2} = \emptyset \), i.e. transitions are named differently;
- \( C_{K_1} \cap C_{K_2} \cap \mathcal{X} = \emptyset \), i.e. colour variables are different;
- \( (S_{K_1} \cap S_{K_2}) = O(P_2) = D(P_1) \), i.e. the initial places of \( P_2 \) and the final places of \( P_1 \) are named equally. All remaining places are named differently.
- \( \forall a \in D(P_1) = O(P_2), (a, c_1) \in \text{post}_c(P_1) \cup I(P_1, a_1) \) and \( (a, c_2) \in \text{pre}_c(P_2) \cup I(P_2, a_2) \) then \( P_1(a, c_1) \ast \rho_1 = P_2(a, c_2) \ast \rho_2 \). This condition imposes that shared places are mapped to the same place of \( N \) by both \( P_1 \) and \( P_2 \), and that the colours that both processes associate to shared places are the same. This can be done always because \( P_1(\text{post}_c(P_1)) \ast \rho_1 \oplus P_1(I(P_1, a_1)) = m_3 = P_2(\text{pre}_c(P_2)) \ast \rho_2 \oplus P_2(I(P_2, a_2)) \), by construction of \( P_1 \) and \( P_2 \).

Note that the above condition allows different ways to select the correspondence between final places of \( P_1 \) and initial places of \( P_2 \), any of them corresponds to a different concatenable process [86].

Define \( K = (S_{K_1} \cup S_{K_2}, C_{K_1} \cup C_{K_2}, T_{K_1} \cup T_{K_2}, \delta_{0_{K_1}} \cup \delta_{0_{K_2}}, \delta_{1_{K_1}} \cup \delta_{1_{K_2}}) \). Note that \( \delta_{i_{K}} = \delta_{i_{K_1}} \cup \delta_{i_{K_2}} \) are well-defined functions because \( \text{dom}(\delta_{i_{K_1}}) \cap \text{dom}(\delta_{i_{K_2}}) = \emptyset \). \( K \) is a coloured causal net because it introduces neither cycles nor conflicts between \( \delta_{i_{K}} \). In fact, shared places are either isolated in at least one \( K_1 \) and \( K_2 \), or they appear in the postset of a transition in \( T_{K_1} \) and in the preset of a transition in \( T_{K_2} \). Consider the substitution \( \rho = \rho_1 \oplus \rho_2 \), which is well-defined because it is enough to consider any \( \rho_i \) restricted to the variables of \( K_i \). Note that \( \rho \) unifies the colours on shared places, i.e. the places appearing in both \( D_c(P_1) \) and \( O_c(P_2) \), hence \( \rho \) is
a compatible execution of $K$. Similarly, the colour assignment $\alpha$ can be defined as
$\alpha = \alpha_1 \uplus \alpha_2$.

A coloured net morphism $f$ from $K$ to $N$ can be defined in terms of the morphisms $f_1$ and $f_2$ (corresponding to the processes $P_1$) as follow

$$ f = (f_{1S} \cup f_{2S}, f_{1T} \uplus f_{2T}, \sigma_1 \uplus \sigma_2) $$

Since $T_{K_1}$ and $T_{K_2}$ are disjoint, then both $f_{1T} \uplus f_{2T}$ and $\sigma_1 \uplus \sigma_2$ are defined. By construction of $P_1$ and $P_2$, the following holds $\forall a \in S_{K_1} \cap S_{K_2} : P_1(a) = P_2(a)$, and hence $f_{1S} \cup f_{2S}$ is a well-defined function. Moreover, $f$ is a net morphism (it is trivial to show that $f_{1S}(\cdot \cdot) \circ f_{2S}(\cdot \cdot) = \cdot f_{1T}(\cdot \cdot) \circ \sigma_1 \circ \cdot f_{2T}(\cdot \cdot) \circ \sigma_2$ by conveniently using $f_i$ for $t \in K_i$). Since all substitutions in $\sigma$ are renamings (i.e., they are either the renamings used by $P_1$ or $P_2$), the morphism $f$ is a process, and consequently we note $f$ as $P$.

It remains to prove that initial and final places of $P$ correspond to the markings $m_1$ and $m_2$. First of all we write the set of initial connected and isolated places of $P$ in terms of the initial and final places of $P_1$ and $P_2$. By construction, the isolated places of $P$ are the isolated places of $P_1$ and $P_2$ that are not connected in the other net, i.e.,

$$ I(P) = I(P_1) \setminus O_c(P_2) \cup I(P_2) \setminus D_c(P_1) $$

(4.2)

Additionally, by construction of $P_1$ and $P_2$, we have $D(P_1) = O(P_2)$, and then by definition of $D(P_1)$ and $O(P_2)$

$$ D_c(P_1) \cup I(P_1) = D(P_1) = O(P_2) = O_c(P_2) \cup I(P_2) $$

Then, by subtracting $D_c(P_1) \cup O_c(P_2)$ on both sides, we have

$$ (D_c(P_1) \cup I(P_1)) \setminus (D_c(P_1) \cup O_c(P_2)) = (O_c(P_2) \cup I(P_2)) \setminus (D_c(P_1) \cup O_c(P_2)) $$

By noting that the following conditions hold for every process, $D_c(P_1) \cap I(P_1) = \emptyset$, $O_c(P_2) \cap I(P_2) = \emptyset$, $I(P_1) \cap D_c(P_1) = \emptyset$, and $I(P_2) \cap O_c(P_2) = \emptyset$, the previous equation is rewritten as

$$ I(P_1) \setminus O_c(P_2) = I(P_2) \setminus D_c(P_1) $$

Therefore, the equation (4.2) can be simplified as

$$ I(P) = I(P_1) \setminus O_c(P_2) = I(P_2) \setminus D_c(P_1) $$

(4.3)

As far as connected initial places of $P$ are concerned, they are the connected initial places of $P_1$ plus the connected initial places of $P_2$ that are not final places of $P_1$, i.e.,
\[ O_c(P) = O_c(P_1) \cup O_c(P_2) \setminus D_c(P_1) \] (4.4)

By using equations (4.3) and (4.4), we have

\[ O_c(P) \cup I(P) = O_c(P_1) \cup (O_c(P_2) \setminus D_c(P_1)) \cup (I(P_2) \setminus D_c(P_1)) = O_c(P_1) \cup ((O_c(P_2) \cup I(P_2)) \setminus D_c(P_1)) \]

Then, the coloured version of above equality is the following

\[ P(pre_c(P)) * \rho \oplus P(I(P, a)) = P(pre_c(P_1)) * \rho \oplus ((P(pre_c(P_2)) * \rho \oplus P(I(P_2, a))) \oplus P(post_c(P_1)) * \rho) \]

Note we correctly use \( \oplus \) in the previous equality because \( D_c(P_1) \subseteq (O_c(P_2) \cup I(P_2)) \).

Finally, by definition of \( P \) and \( a \), we have

\[ P(pre_c(P_1)) * \rho \oplus P(I(P_1, a_1)) = P_1(pre_c(P_1)) * \rho_1 \oplus ((P_2(pre_c(P_2)) * \rho_2 \oplus P_2(I(P_2, a_2))) \oplus P_1(post_c(P_1)) * \rho_1) \] (4.5)

By construction of \( P_1 \) and \( P_2 \), the following equality holds

\[ P_1(post_c(P_1)) * \rho_1 \oplus P_1(I(P_1, a_1)) = m_3 = P_2(pre_c(P_2)) * \rho_2 \oplus P_2(I(P_2, a_2)) \]

Then, by subtracting \( P_1(post_c(P_1)) * \rho_1 \) on both sides we have

\[ P_1(I(P_1, a_1)) = (P_2(pre_c(P_2)) * \rho_2 \oplus P_2(I(P_2, a_2))) \oplus P_1(post_c(P_1)) * \rho_1 \]

Then, by substituting \( P_1(I(P_1, a_1)) \) in the expression corresponding to \( m_1 \), we have

\[ m_1 = P_1(pre_c(P_1)) * \rho_1 \oplus ((P_2(pre_c(P_2)) * \rho_2 \oplus P_2(I(P_2, a_2))) \oplus P_1(post_c(P_1)) * \rho_1) \] (4.6)

Finally, by equations (4.5) and (4.6),

\[ P(pre_c(P)) * \rho \oplus P(I(P, a)) = m_1 \]

Similarly, it can be shown that \( P(post_c(P)) * \rho \oplus P(I(P, a)) = m_2 \)

\( \square \)

**Lemma 4.4.** Let \( N = (S_N, C_N, T_N, \delta_{0N}, \delta_{1N}) \) be a \( C-P/T \) net and \( P : K \to \sigma N \) a compatible process of \( N \). For every compatible execution \( \rho : \mathcal{X} \to \mathcal{V} \) substituting all colour variables in \( pre_c(P) \), i.e. \( \text{col}_X(pre_c(P)) \subseteq \text{dom}(\rho) \), and for any colour assignment \( a \), then there exist \( m_1 \) and \( m_2 \) s.t.
\[ m_1 \rightarrow^{*}_{T_N} m_2 \]
\[ m_1 = P(\text{pre}_c(P)) * \rho \oplus P(I(P, a)) \]
\[ m_2 = P(\text{post}_c(P)) * \rho \oplus P(I(P, a)) \]

**Proof.** By induction on the number of transitions \( n = |T_K| \).

- **Base case:** \( n = 0 \): Since there are no transitions, given any colour assignment \( a \), then \( m_1 = P(I(P, a)) = m_2 \). Therefore \( m_1 \rightarrow^{*}_{T_N} m_2 \).

- **Inductive step:** \( |T_K| = n + 1 \). The proof follows by showing that it is possible to decompose \( P \) into two processes, one corresponding to \( m_1 \rightarrow^{*}_{T_N} m_3 \) and the other to \( m_2 \rightarrow^{*}_{T_N} m_2 \). In order to do that, select a transition \( t \) s.t. \( t \subseteq O_c(P) \). This is always possible because \( K \) is a causal net (and hence acyclic). Then consider the following two processes \( P_1 : K_1 \rightarrow_{\sigma_1} N \) and \( P_2 : K_2 \rightarrow_{\sigma_2} N \), where

- \( S_{K_1} = O(P) \cup t^* \), \( T_{K_1} = \{t\} \), \( \delta_{K_1}(t) = \delta_K(t) \) for \( i = 0, 1 \). First note that \( K_1 \) has only one transition, and hence any substitution is a compatible execution. Then define the process \( P_1 \) as the restriction of \( P \) to the elements of \( K_1 \).

Since \( K_1 \) has only one transition \( t \), \( \text{pre}_c(P_1) = \bullet t \) and \( \text{post}_c(P_1) = t^* \). Moreover, since \( P \) is a process, there exists \( t' = f_t(t) \in N \), s.t. \( t' \mid t^* = \bullet t' * \sigma_t \). Hence, \( \text{pre}_c(P_1) = \bullet t' * \sigma_t \) and \( \text{post}_c(P_1) = t'^* * \sigma_t \). For all \( \rho : X \rightarrow \mathcal{V} \) s.t. \( \text{col}_X(\text{pre}_c(P)) \subseteq \text{dom}(\rho) \), it is easy to notice that \( \text{rn}(t') \subseteq \text{dom}(\sigma_t; \rho) \) and \( \text{range}(\sigma_t; \rho) \subseteq \mathcal{V} \). Therefore, it is possible to construct the following derivation in \( N \) by using rule \((\text{COLOURED-FIRING})\)

\[
\begin{align*}
  t' &= \bullet t' \mid t'^* \in T_N \quad m_1^t, m_1^t \in M_{S,C} \\
  \quad \text{rn}(t') &\subseteq \text{dom}(\sigma_t; \rho), \\
  \quad \text{range}(\sigma_t; \rho) &\subseteq \mathcal{V} \\
  \bullet t'^* (\sigma_t; \rho) \oplus m_1^t &\rightarrow_{T_N} t'^* (\sigma_t; \rho) \oplus m_1^t
\end{align*}
\]

Hence, for any compatible execution \( \rho \) of \( P \)

\[ \text{pre}_c(P_1) * \rho \oplus m_1^t \rightarrow_{T_N} \text{post}_c(P_1) * \rho \oplus m_1^t \]

In what follows we show that for any colour assignment \( a \) there exists a particular colour assignment \( a_1 \) extending \( a \) to the isolated places of \( P_1 \) and connected in \( P \), s.t. \( \text{pre}_c(P_1) * \rho \oplus P(I(P, a_1)) = m_1 \). Consider a colour assignment \( a_1 \) s.t. (i) \( a \subseteq a_1 \) (i.e., isolated places of \( P \) are assigned with the same colour as in \( a \)) and (ii) \( \forall a \in O_c(P) \setminus t : (a, a_1(a)) \in \text{pre}_c(P) * \rho \) (i.e., initial connected places in \( P \) are assigned with the colour given by the compatible execution \( \rho \)). Consequently
\[ P_1(\text{pre}_c(P_1)) \star \rho \oplus P_1(I(P_1, a_1)) = \text{By def. of } K_1 \]
\[ P_1(\text{pre}_c(P_1)) \star \rho \oplus P_1(I(P_1, a_1)\mid I(P) \cup I(P_1, a_1)\mid I(P)) = \text{By definition of } I(P_1) \]
\[ P_1(\text{pre}_c(P_1)) \star \rho \oplus P_1(I(P_1, a_1)\mid I(P) \cup I(P_1, a_1)\mid O_c(P)\mid \{t\}) = \text{By distributing the application of } P_1 \]
\[ P_1(\text{pre}_c(P_1)) \star \rho \oplus P_1(I(P_1, a_1)\mid I(P) \cup P_1(I(P_1, a_1)\mid O_c(P)\mid \{t\}) \star \rho = \]
\[ \text{As } O_c(P_1) = \{t\}, \text{ and by associativity and commutativity of } \oplus \]
\[ P_1(\text{pre}_c(P)) \star \rho \oplus P_1(I(P, a)) = \text{Since } P_1 \subseteq P \text{ and } a_1(a) = a(a) \text{ for } a \in I(P) \]
\[ m_1 \]

For this reason \( m_1 \rightarrow_{T_N} m_3 \), where \( m_3 = \text{post}_c(P_1) \star \rho \oplus P_1(I(P_1, a_1)) \). Note that in general there are several possible \( a_1 \), all of them differ on

- Consider \( K_2 \) defined as follows: \( S_{K_2} = S_K \setminus \{t\} \), and \( T_{K_2} = T_K \setminus \{t\} \), and \( \forall t \in T_{K_2} : \delta_{K_2}(t) = \delta_K(t) \) for \( i = 0, 1 \).

By inductive hypothesis there exists \( n_1 \) and \( n_2 \) satisfying the equations corresponding to initial and final places s.t. \( n_1 \rightarrow_{T_N}^* n_2 \). Note that \( \rho \) is a compatible execution of \( P_2 \) (since \( P_2 \subseteq P \)).

We show that for any \( \rho \) and \( a \), it is also possible to build and assignments \( a_2 \) such that \( n_1 = m_3 \). We require

\[ (a, a_2(a)) \in \text{post}_c(P_1) \star \rho \text{ if } a \in I(P_2) \cap D_c(P_1) \]
\[ a_2(a) = a(a) \text{ for } a \in I(P) \]

The first condition makes \( a_2 \) to assign to the isolated places of \( P_2 \) that are final connected places of \( P_1 \) the same colour assigned by \( \rho \). The second one assures that isolated places of the original process \( P \) (and hence isolated both in \( P_1 \) and \( P_2 \)) are coloured with the assignment \( a \).

Since \( D(P_1) = O_c(P_1) \cup I(P_1) = \{t\} \cup (O_c(P) \setminus \{t\} \cup I(P)) = O(P_2) \), it is straightforward to show that \( n_1 = m_3 \) for any \( \rho, a_1 \) and \( a_2 \).

Finally, we show that \( n_2 = m_2 \). First note that \( D(P_2) = D_c(P) \cup I(P) \cup (\{t\} \setminus O_c(P_2)) \). Note that when \( a \in (\{t\} \setminus O_c(P_2)) \) then \( a \in D_c(P) \). Therefore \( D(P_2) = D_c(P) \cup I(P) \), and therefore \( n_2 = m_2 \) for any \( \rho \) and \( a \).

Finally, \( m_1 \rightarrow_{T_N} m_3 = n_1 \rightarrow_{T_N}^* n_2 = m_2 \)

\[ \square \]

Remark 4.4. The proof presented before gives a strictly sequential derivation, nevertheless a proof with concurrent steps can be obtained by selecting all transitions whose presets are in \( O(P) \) and by constructing a net for any of them. Finally, the combination of all nets corresponds to a single step obtained with rule (STEP).
4.4. COLOURED ZS NETS

fact, a stronger relation could be stated between firing sequences and processes, i.e., it is possible to show that a process represents all possible linearizations of a derivation.

**Theorem 4.5.** Let \( N = (S_N, C_N, T_N, \delta_{0N}, \delta_{1N}) \) be a coloured P/T net. Then, \( m_1 \rightarrow^*_{N} m_2 \) iff there exists a process \( P : K \rightarrow^\sigma N \), a compatible execution \( \rho : X \rightarrow \mathcal{Y} \) substituting all colour variables in \( \text{pre}_c(P) \), i.e. \( \text{col}_X(\text{pre}_c(P)) \subseteq \text{dom}(\rho) \), and a colour assignment \( \alpha, s.t.

- \( m_1 = P(\text{pre}_c(P)) \ast \rho \oplus P(I(P, \alpha)) \)
- \( m_2 = P(\text{post}_c(P)) \ast \rho \oplus P(I(P, \alpha)) \)

*Proof.* Immediate from lemmata 4.3 and 4.4. \( \square \)

4.4 Coloured ZS nets

The ZS version of a coloured net is obtained by distinguishing stable places from zero ones. We remark that colours do not interfere with the ZS mechanism, and that the definitions are straightforward extensions of those in Chapter 3.

**Definition 4.20 (c-ZS net).** A coloured ZS net (c-ZS net for short) is a 7-tuple \( B = (S_B, C_B, T_B, \delta_{0B}, \delta_{1B}, m_{0B}, Z_B) \) where \( N_B = (S_B, C_B, T_B, \delta_{0B}, \delta_{1B}, m_{0B}) \) is the underlying C-P/T net and the set \( Z_B \subseteq S_B \) is the set of zero places. The places in \( S_B \setminus Z_B \) (denoted by \( L_B \)) are called stable places. A stable marking \( m \) is a coloured multiset of stable places coloured with constants (i.e., \( m \in M_{L_B,C_B\cap\mathcal{Y}} \)), and the initial marking \( m_{0B} \) must be stable.

The operational semantics of c-ZS nets is a straightforward extension of rules given in Figure 3.1, where the firing rule is replaced by the following one:

\[
\begin{aligned}
&\text{(Coloured-firing)} \\
&t = s \oplus z \mid s' \oplus z' \in T \quad s'' \in M_{L,C\cap\mathcal{Y}} \quad z'' \in M_{Z,C\cap\mathcal{Y}} \\
&(s \ast \sigma \oplus s', z \ast \sigma \oplus z'') \rightarrow_T (s' \ast \sigma \oplus s'', z' \ast \sigma \oplus z'')
\end{aligned}
\]

\( \text{rn}(t) \subseteq \text{dom}(\sigma), \quad \text{range}(\sigma) \subseteq \mathcal{Y} \)

We still write a marking \( m = s \oplus z \) as a pair \( (s, z) \) to denote that \( s \in M_{L,C} \) and \( z \in M_{Z,C} \).

**Example 4.5 (c-ZS net for the mobile lessees problem).** A more general representation for the mobile lessees problem introduced in the Example 3.2 can be given in terms of c-ZS nets. Consider the net in Figure 4.4, where labels on arcs correspond to the colours of the pre- and postset of a transition. Tokens in the place free represent apartments that are available for being rented immediately. The actual identity of an apartment is given by the colour of the corresponding token. Similarly people looking for an apartment are represented as coloured tokens in wants,
Figure 4.4: C-ZS net of the mobile lessees example.

and those willing to change apartment as tokens \((x', y')\) in changes, meaning that the person \(x'\) changes the apartment \(y'\). A token \((x'', y'')\) in place pref denotes the fact that a person named \(x''\) likes the apartment \(y''\) for renting. The transition freeing initiates a transaction by making available for rent an offered apartment. Analogously, searching initiate a transaction in which a person is looking for an apartment. Transition changing starts a transaction by rendering available the offered apartment and putting a token in the place of people looking for an apartment. Hence, the transition taking states that a person \(x''\) searching for an apartment can take the available apartment \(y''\) if she likes it (i.e., a token with colour \((x'', y'')\) is in the set of preferences). A token with colour \((x'', y'')\) produced in the place moves means that the person \(x''\) has moved to the apartment \(y''\). It is worth noting that, according to the zs mechanism, tokens are actually produced on place moves when no token is left in the zero places (i.e., avail or search).

While in zs nets different instances of this problem (i.e., different set of apartments, people and preferences) correspond to different nets (i.e., places, transitions and flow functions), in the coloured version the structure of the net is the same for every instance of the problem, the only thing that changes is the initial marking. In fact, all the information about a particular instance of the problem is represented by colours. The initial marking for the C-ZS in Figure 4.4 that corresponds to the instance of the problem presented in Example 3.2 can be obtained directly from the relations in Figure 3.5. In particular, any pair \((p, a)\) in the initial situation (Figure 3.5.a) where the person \(p\) is leaving the apartment \(a\) corresponds to a token changes\((p, a)\) in the initial marking. Similarly, the initial marking contains a token free\((a)\) for any apartment \(a\) which is free in Figure 3.5, and a token wants\((p)\) for any person \(p\) looking for an apartment. Finally, the initial marking contains a token pref\((p, a)\) for any person - apartment in the table of preferences (Figure 3.5.b). Consequently the initial marking of the coloured zs net
is \( \text{changes}(Q,B) \oplus \text{changes}(R,C) \oplus \text{wants}(S) \oplus \text{wants}(P) \oplus \text{free}(A) \oplus \text{pref}(P,C) \oplus \text{pref}(Q,A) \oplus \text{pref}(R,B) \oplus \text{pref}(S,A) \).

The previous example shows how colours allow for a description at a higher level of abstraction, i.e., several problems that differ only on their data can be described by a unique coloured net.

Remark 4.5 (Contextual nets). The self-loop introduced to model the preference set in the Example 4.5 can be better modelled as read arcs. Nets with read arcs allow for modelling “read without consuming”, where many readers can access concurrently to the same resource. Consider transition taking in Figure 4.4. Actually, there is no need to consume the token in \text{pref} when firing \text{taking}, it will be enough to check the presence of a token with suitable colours on \text{pref}. The extension of the ZS model to contextual nets have been studied in [29]. Nevertheless, we do not analyse here the extension of coloured nets with contextual transitions, because this capability is somehow orthogonal to the hierarchy we are considering (that proposed in [31]). Although it could be interesting to extend coloured, reconfigurable and dynamic nets with contextual transitions, this goal is out of the scope of this work, which is aimed mainly at providing a mobile version of ZS nets.

4.5 Abstract Semantics

The definition for connected transactions is analogous to Definition 3.2 but considers coloured markings.

Definition 4.21 (Coloured Connected Transaction). Given a c-ZS net \( B \), and a coloured deterministic process \( P \) of the underlying c-P/T net \( N_B \). The equivalence class \( \xi = \llbracket P \rrbracket \) of all processes isomorphic to \( P \) (see Definition 2.6) is a coloured connected transaction of \( B \) if:

1. \( P(\text{prec}_c(P)) \) and \( P(\text{post}_c(P)) \) are stable markings, i.e., the process starts by consuming stable tokens and produces only stable tokens;

2. \( E_P \subseteq Z_B \); i.e. the evolution places of the process (see Notation 2.1) are zero places, this means that stable tokens produced during the transaction cannot be consumed during in the same transaction;

3. \( P \) is connected (see Definition 2.8); and,

4. \( P \) is full, i.e., it does not contain idle (i.e., isolated) places (i.e., \( \forall a \in S_K : |a^*| \geq 1 \)).

Given a transaction \( \xi \), we write \( \text{pre}(\xi) \) and \( \text{post}(\xi) \) for \( P(\text{prec}_c(P)) \) and \( P(\text{post}_c(P)) \) respectively.
**Example 4.6.** Two simple coloured connected transactions for the C-ZS net in Figure 4.4 are presented in Figure 4.5. We depicted mainly the causal nets of the transactions, while the morphisms are thought to map places and transitions of the causal net to the homonymous elements of the C-ZS net in Figure 4.4. The substitutions $\sigma_t$ of the morphisms are shown next to the corresponding transition. The first net represents the transaction in which a person looking for an apartment takes a free apartment, while the second shows the transaction in which two people exchange their apartments.

The substitution $\rho = \{y/y', x/x'\}$ is a compatible execution for the first process. Note that $x/x'$ in $\rho$ captures the idea that the token consumed from the state *wants* refers to the same person of the token from the preferences (i.e., place *pref*). Similarly, $y/y'$ relates the different tokens referring to the same apartment. Observe also that the substitution $\rho' = \{v/x, v/x', v/y, v/y'\}$ is a compatible execution for
the same process, which requires all names to be equals to the constant \( v \). Clearly, \( \rho' \) is more restrictive than \( \rho \), which is in fact the mgce.

Similarly, for the second process the mgce is \( \rho'' = \{ x_0/x_3, x_1/x_2, y_0/y_2, y_1/y_3 \} \).

The following results state the correspondence between derivations in the ZS nets and connected transactions. We start by showing that for a ZS net \( B \), a weaker notion of transactions (where the initial marking may contain zero places) describes a computation \( \rightarrow_B \). We remark that these results cannot be obtained directly from Section 4.3, because the derivations concerns to ZS semantics (and not on the underlying coloured net \( N_B \)), which does not allow concatenation on stable places.

**Lemma 4.6.** Let \( B \) be a C-ZS net and \( P : K \rightarrow N_B \) be a process of the underlying net \( N_B \). If \( P \) is non-empty (i.e., \( T_K \neq \emptyset \)), full (i.e., \( I(P) = \emptyset \)) and \( E_P \subseteq Z_B \), then for any compatible execution \( \rho : X \rightarrow V \) substituting all colour variables in \( P(\text{pre}_c(P)) \) (i.e., \( \text{col}_X(P(\text{pre}_c(P))) \subseteq \text{dom}(\rho) \)), \( P(\text{pre}_c(P)) \times \rho \rightarrow_T B \ P(\text{post}_c(P)) \times \rho \).

**Proof.** The proof follows by induction on \( n = |T_K| \)

- **Base case:** \( n = 1 \), \( T_K = \{ t \} \); \( \text{pre}_c(P) = \bullet t \) and \( \text{post}_c(P) = t \bullet \). As \( P \) is a process, then there exists \( t' \in T_B \) s.t. \( P(\text{pre}_c(P)) = P(\bullet t) = \bullet t' \circ \sigma_t \) and \( P(\text{post}_c(P)) = P(t') = t' \circ \sigma_t \). Since \( \text{col}_X(P(\text{pre}_c(P))) \subseteq \text{dom}(\rho) \), it is easy to notice that \( \text{rn}(t') \subseteq \text{dom}(\sigma_t; \rho) \) and \( \text{range}(\sigma_t; \rho) \subseteq V \).

Then by rule (coloured-firing) we have

\[
\frac{t' = m [m] m' \in T_B \quad m \circ (\sigma_t; \rho) \oplus \emptyset \rightarrow_T B m' \circ (\sigma_t; \rho) \oplus \emptyset}{\text{rn}(t) \subseteq \text{dom}(\sigma_t; \rho), \text{ and} \quad \text{range}(\sigma_t; \rho) \subseteq V}
\]

As \( m \circ \sigma_t = P(\text{pre}_c(P)) \) and \( m' \circ \sigma_t = P(\text{post}_c(P)) \), we have \( P(\text{pre}_c(P)) \times \rho \rightarrow_T B \ P(\text{post}_c(P)) \times \rho \)

- **Inductive step.** It follows by showing that a proof for the whole computation can be built as the concatenation of a step firing all transitions concurrently enabled by minimal places and another computation. Hence, consider the set \( T = \{ t_i | t_i \in T_K \land t_i \subseteq O(P) \} \) of all transitions enabled by the initial places of the process. Since \( K \) is a causal net, \( T \) is not empty.

For any \( t_i \in T \) define a net \( K_i = (\bullet t_i \cup t_i \bullet, \{ t_i \}, \delta_{0K_i}, \delta_{1K_i}) \) and the processes \( P_i : K_i \rightarrow N \) as the restriction of \( P \) to the elements of \( K_i \). For any \( \rho : X \rightarrow V \) s.t. \( \text{col}_X(P(\text{pre}_c(P))) \subseteq \text{dom}(\rho) \) we can obtain a proof for \( P(\text{pre}_c(P_i)) \times \rho \rightarrow_T B \ P(\text{post}_c(P_i)) \times \rho \) by using rule (coloured-firing) like in the base case.

If \( |T| > 1 \) (i.e., there are several transitions concurrently enabled by the minimal places), then all proofs corresponding to the transitions in \( T \) are combined by using repeatedly rule (step), which gives a proof for

\[
\bigoplus_i P(\text{pre}_c(P_i)) \times \rho \rightarrow_T B \bigoplus_i P(\text{post}_c(P_i)) \times \rho
\] (4.7)
If there are no remaining transitions after firing concurrently $T$ (i.e., $T_K \setminus T = \emptyset$), the proof is completed by noting that $P(\text{pre}_c(P)) = \bigoplus_i P(\text{pre}_c(P_i))$ and $P(\text{post}_c(P)) = \bigoplus_i P(\text{post}_c(P_i))$, since $P$ has no isolated places.

Otherwise, i.e. $T_K \setminus T \neq \emptyset$, the proof proceeds as follows. Firstly, we introduce some notation. We abbreviate $\forall_i P_i$ with $P^\prime$. Then note that $P(\text{pre}_c(P^\prime)) = P(\text{pre}_c(P)|_{\tau^\star})$, and $P(\text{post}_c(P^\prime))$ can be written as two markings: one stable and the other zero-safe, i.e. $P(\text{post}_c(P^\prime)) = P(\text{post}_c(P)|_{\tau^\star}) \oplus P(\text{post}_c(P^\prime)|_{\tau^\star \setminus D(P)})$. Then, the equation (4.7) can be written as

$$P(\text{pre}_c(\bullet_T)) \ast \rho \rightarrow_{T_B} P(\text{post}_c(P)|_{\tau^\star}) \ast \rho \oplus P(\text{post}_c(P^\prime)|_{\tau^\star \setminus D(P)} \ast \rho \quad (4.8)$$

For the remaining transitions in $P$, i.e. transitions in $T_K \setminus T$, we build the following net $K_r$:

- $S_{K_r} = S_K \setminus (\bigcup_i \bullet t_i \cup DS(T))$, where $DS(T) = \bigcup_i \bullet t_i \cap D(P)$ is the set of places in the postset of $t_i$ that are final in $P$.
- $T_{K_r} = T_K \setminus T$
- $\delta_{jK_r}(t) = \delta_{jK}(t)$ for $j = 0, 1$

Note that the flow relation is well-defined, because the removed places correspond either to final places or to the presets of removed transitions. Note also that $K_r$ is still a causal net because it is obtained from $K$ just by removing minimal elements. Then define the process $P_r$ from $K_r$ to $N_B$, by restricting $P$ to the elements of $K_r$. Clearly, $P_r$ is non-empty and full, where $E_{P_r} \subseteq Z_B$. Note that $O(P_r) = O(P)|_{\bullet T \cup \tau^\star \setminus D(P)}$. Then by inductive hypothesis, there exists a proof for

$$P(\text{pre}_c(P_r)) \ast \rho = P(\text{pre}_c(P)|_{O(P) \setminus \bullet_T}) \ast \rho \oplus P(\text{pre}_c(P)|_{\tau^\star \setminus D(P)} \ast \rho \rightarrow_{T_B} P(\text{post}_c(P_r)) \ast \rho = P(\text{post}_c(P)|_{D(P) \setminus \tau^\star}) \ast \rho$$

Since $\rho$ is a compatible execution, the colours of tokens produced in and consumed from evolution places are the same. Then, $P(\text{pre}_c(P_r)|_{\tau^\star \setminus D(P)} \ast \rho = P(\text{post}_c(P^\prime)|_{\tau^\star \setminus D(P)} \ast \rho$. Consequently,

$$P(\text{pre}_c(P)|_{O(P) \setminus \bullet_T}) \ast \rho \oplus P(\text{post}_c(P^\prime)|_{\tau^\star \setminus D(P)} \ast \rho \rightarrow_{T_B} P(\text{post}_c(P)|_{D(P) \setminus \tau^\star}) \ast \rho \quad (4.9)$$

Note that the computations given by equations (4.8) and (4.9) share only zero-safe markings, and consequently they can be concatenated by using rule (concatenation), giving a proof for

$$P(\text{pre}_c(P)|_{\bullet_T}) \ast \rho \oplus P(\text{pre}_c(P)|_{O(P) \setminus \bullet_T}) \ast \rho \rightarrow_{T_B} P(\text{post}_c(P)|_{\tau^\star \setminus D(P)} \ast \rho \oplus P(\text{post}_c(P)|_{D(P) \setminus \tau^\star}) \ast \rho \quad (4.10)$$

Finally, by noting that $P(\text{pre}_c(P)|_{\bullet_T}) \ast \rho \oplus P(\text{pre}_c(P)|_{O(P) \setminus \bullet_T}) \ast \rho = P(\text{pre}_c(P)) \ast \rho$ and $P(\text{post}_c(P)|_{\tau^\star \setminus D(P)} \ast \rho \oplus P(\text{post}_c(P)|_{D(P) \setminus \tau^\star}) \ast \rho = P(\text{post}_c(P)) \ast \rho$, we have
\[ P(\text{pre}_c(P)) \times \rho \rightarrow_B P(\text{post}_c(P)) \times \rho \]  

The following theorem states that a connected transaction of a zs net \( B \) corresponds to an atomic movement of \( B \).

**Theorem 4.7.** Let \( B \) be a c-zs net and \( \xi \) a connected transaction of \( B \), for any compatible execution \( \rho : X \rightarrow Y \) substituting all colour variables in \( \text{pre}(\xi) \) (i.e. \( \text{col}_X(\text{pre}(\xi)) \subseteq \text{dom}(\rho) \)), we have \( \text{pre}(\xi) \times \rho \rightarrow_B \text{post}(\xi) \times \rho \).

**Proof.** From Lemma 4.6, there is a proof for \( \text{pre}(\xi) \times \rho \rightarrow_B \text{post}(\xi) \times \rho \). Since \( \text{pre}(\xi) \) and \( \text{post}(\xi) \) are stable markings, then the whole proof is completed by applying rule (close). \( \square \)

The correspondence between atomic movements of zs nets and connected transactions are assured by the following two results. We start by showing that every computation in a zs net is represented by a family of connected, full processes. Note that a family is consider instead of a single process, because the rule (step) allows to combine unrelated computations, which produces non connected processes. Then, we show that atomic movements are described by a family of connected transactions.

**Lemma 4.8.** Let \( B \) be a c-zs net. If \( m \rightarrow_B m' \), then there exists a family \( \{P_i\}_{i \in I} \) of compatible processes and a substitution \( \rho \), that is a compatible execution for every \( P_i \) s.t.

- \( m = \bigoplus_{i \in I} P_i(\text{pre}_c(P_i)) \times \rho \oplus m'' \) and \( m' = \bigoplus_{i \in I} P_i(\text{post}_c(P_i)) \times \rho \oplus m'' \);
- \( E_{P_i} \subseteq Z_B \), for all \( i \in I \);
- \( P_i \) is connected, for all \( i \in I \);
- \( P_i \) is full, for all \( i \in I \).

**Proof.** By induction on the structure of the proof of \( m \rightarrow_B m' \).

- **Rule (coloured-firing).** The construction of the unique \( P_1 \) is identical to the proof of Lemma 4.2. The net has only one transition, and hence all conditions are satisfied.

- **Rule (step).** Then

\[
\begin{align*}
m_1 \rightarrow m'_1 & \quad m_2 \rightarrow m'_2 \\
m = m_1 \oplus m_2 & \rightarrow m'_1 \oplus m'_2 = m'
\end{align*}
\]

By inductive hypothesis there exists a family \( \{P^1_i\}_{i \in I_1} \) s.t.

- \( m_1 = \bigoplus_{i \in I_1} P^1_i(\text{pre}_c(P^1_i)) \times \rho_1 \oplus m'' \) and
- \( m'_1 = \bigoplus_{i \in I_1} P^1_i(\text{post}_c(P^1_i)) \times \rho_1 \oplus m'' \)}
where all $P^1_i$ are full, connected and $E_{P^1_i} \subseteq Z_B$. And similarly, there exists \{\(P^2_j\)\}_{j \in I_2}$ s.t.

\[ m_2 = \bigoplus_{j \in I_2} P^2_j (\text{pre}_c(P^2_j)) \ast \rho_2 \oplus m''_2 \quad \text{and} \quad m'_2 = \bigoplus_{j \in I_2} P^2_j (\text{post}_c(P^2_j)) \ast \rho_2 \oplus m''_2 \]

where all $P^2_j$ are full, connected and $E_{P^2_j} \subseteq Z_B$. As done for proving Lemma 4.2, the proof follows by noting that is possible to define both families s.t. $P^1_i$ and $P^2_j$ do not share names for places, transitions nor colour variables, for all $i \in I_1, j \in I_2$. The proof is completed by taking family \(\{P^1_i\}_{i \in I_1} \cup \{P^2_j\}_{j \in I_2}\), in which any process is full, connected and with evolution places mapped to zero places. Finally, consider the execution $\rho = \rho_1 \cup \rho_2$ for the whole proof ($\rho$ can be always defined because it is enough to consider $\rho_k$ restricted to the variables of $\{P^k_{\ell}\}_{\ell \in I_k}$, which do not share variable names).

- **Rule** (concatenation). Then

\[
\begin{align*}
  m_1 \rightarrow s'_1 \oplus z'_1 & \quad s_2 \oplus z'_1 \rightarrow m'_2 \\
  m = m_1 \oplus s_2 \rightarrow s'_1 \oplus m'_2 & = m''
\end{align*}
\]

By inductive hypothesis there exists a set $\{P^1_i\}$ s.t.

\[
m_1 = \bigoplus_i P^1_i (\text{pre}_c(P^1_i)) \ast \rho_1 \oplus m''_1 \quad \text{and} \quad s'_1 \oplus z'_1 = \bigoplus_i P^1_i (\text{post}_c(P^1_i)) \ast \rho_1 \oplus m''_1
\]

(4.10)

where any $P_i$ is full, connected and $E_{P^1_i} \subseteq Z_B$. And similarly $\{P^2_j\}$ s.t.

\[
\begin{align*}
  s_2 \oplus z'_1 & = \bigoplus_j P^2_j (\text{pre}_c(P^2_j)) \ast \rho_2 \oplus m''_2 \quad \text{and} \\
  m'_2 & = \bigoplus_j P^2_j (\text{post}_c(P^2_j)) \ast \rho_2 \oplus m''_2
\end{align*}
\]

(4.11)

where any $P^2_j$ is full, connected and $E_{P^2_j} \subseteq Z_B$. As before, select processes s.t. they do not share names for places, transitions nor colour variables. As done in the inductive step of the proof of Lemma 4.3, select a bidirectional mapping of final places in $\{P^1_i\}$ and initial places $\{P^2_j\}$ that are mapped to the same zero place in the net $B$. Then, use the same name for pair of places identified before. Finally, combine the processes that share places into a larger process. Note that the obtained net for the joint process is a causal net since the construction adds neither transitions nor cycles (in fact places that join processes were final in $P^1_i$ and initial in $P^2_j$, which were originally disjoint). Moreover, any new larger process $P_k$ is connected (because it is the union of connected processes that share at least one place), full (since it is the union of full processes), and $E_{P_k} \subseteq Z_B$ (since the new evolution places are those identified when joining processes, which are mapped to zero-places).

From equality $s'_1 \oplus z'_1$ in equation (4.10), we have

\[
s'_1 = \left( \bigoplus_i P^1_i (\text{post}_c(P^1_i)) \ast \rho_1 \oplus m''_1 \right) \ominus z'_1
\]

(4.12)
From equality $s_2 \oplus z'_1$ in equation (4.11), we have

$$m''_2 = s_2 \oplus z'_1 \oplus (\bigoplus_j P^2_j(\text{pre}_c(P^2_j)) \ast \rho_2)$$

By substituting $m''_2$ in expression $m'_2$ of equation (4.11), the following holds

$$m'_2 = \bigoplus_j P^2_j(\text{post}_c(P^2_j)) \ast \rho_2 \oplus s_2 \oplus z'_1 \oplus (\bigoplus_j P^2_j(\text{pre}_c(P^2_j)) \ast \rho_2) \quad (4.13)$$

Finally, by calculating $s'_1 \oplus m'_2$ from equations (4.12) and (4.13) we have

$$s'_1 \oplus m'_2 = \bigoplus_j P^2_j(\text{post}_c(P^2_j)) \ast \rho_2 \oplus (\bigoplus_i P^1_i(\text{post}_c(P^1_i)) \ast \rho_1) \oplus m''_2 \oplus s_2 \oplus (\bigoplus_j P^2_j(\text{pre}_c(P^2_j)) \ast \rho)$$

Moreover, as processes can be defined in order not to share variables, we can assume without loss of generality that $\rho = \rho_1 \oplus \rho_2$.

$$s'_1 \oplus m'_2 = \bigoplus_j P^2_j(\text{post}_c(P^2_j)) \ast \rho \oplus (\bigoplus_i P^1_i(\text{post}_c(P^1_i)) \ast \rho) \oplus m''_2 \oplus s_2 \oplus (\bigoplus_j P^2_j(\text{pre}_c(P^2_j)) \ast \rho)$$

Note that $(\bigoplus_i P^1_i(\text{post}_c(P^1_i)) \ast \rho \oplus m''_2 \oplus s_2 \oplus (\bigoplus_j P^2_j(\text{pre}_c(P^2_j)) \ast \rho)$ stands for the markings corresponding to the final places of the first set of processes that are not initial places of some process in the second set, i.e., $(\bigoplus_i P^1_i(\text{post}_c(P^1_i)) \ast \rho \oplus m''_2 \oplus (\bigoplus_j P^2_j(\text{pre}_c(P^2_j)) \ast \rho)$, and the marking of the idle resources, i.e. $m''_2 \oplus s_2 \oplus (\bigoplus_j P^2_j(\text{pre}_c(P^2_j)) \ast \rho$. Then the whole expression stands for the multiset of coloured final places of the composition.

Reasoning analogously, it can be shown that $m_1 \oplus s_2$ corresponds to the colouring of initial places of the composition.

\[\square\]

**Theorem 4.9.** Let $B$ be a C-ZS net. If $m \Rightarrow_T m'$, then there exists a set of connected transactions \(\{\xi_i\}\), and a substitution $\rho$ that is a compatible execution of all $\xi_i$ s.t.

- $m = \bigoplus_i \text{pre}(\xi_i) \ast \rho \oplus m''$ and $m' = \bigoplus_i \text{post}(\xi_i) \ast \rho \oplus m''$.

**Proof.** The proof is immediate by Lemma 4.8.  \[\square\]

The definition of the abstract net for a C-ZS is straightforward. The difference with the analogous definition for ZS nets (Definition 3.3) is that abstract transitions depend on the mgc of the connected transactions. We recall that the mgc of a coloured net gives the binding of the variables appearing on transitions. In particular, it represents the constraint on the colours of the initial and final places of the net.
Definition 4.22 (Coloured Causal Abstract Net). Let $B = (S_B, C_B, T_B, \delta_0B, \delta_1B, m_{0B}, Z_B)$ be a c-zs net. The coloured causal abstract net of $B$ is defined as $I_B = (S_B \backslash Z_B, C_B, \Xi_B, \delta_{0I}, \delta_{1I}, m_{0I})$, with $\delta_{0I}(\xi) = \text{pre}(\xi) \times \sigma_\xi$ and $\delta_{1I}(\xi) = \text{post}(\xi) \times \sigma_\xi$, where $\sigma_\xi$ is the mgce for $\xi$. (We recall that $\Xi_B$ is the set of all connected transactions of $B$).

From the above definition, each transition in the abstract net corresponds to a connected transaction, i.e., to an equivalence class of isomorphic processes. Therefore, any $\xi$ is a pattern representing several computations that differ only on the colour of consumed / produced tokens. Note also that any transition is defined as $\text{pre}(\xi) \times \sigma_\xi \Rightarrow \text{post}(\xi) \times \sigma_\xi$, where $\sigma_\xi$ is the mgce of $\xi$. The application of the mgce $\sigma_\xi$ to the initial and final marking of $\xi$ makes explicit on the markings the constraints on the colours of consumed / produced. Since we use the mgce, transitions $\text{pre}(\xi) \times \sigma_\xi \Rightarrow \text{post}(\xi) \times \sigma_\xi$ may contain variables, and hence the obtained net is a coloured net.

Example 4.7 (Abstract net for the mobile lessees problem). Figure 4.6 shows a partial view of the abstract net corresponding to the mobile lessees example. Transition \texttt{wants&free} corresponds to the atomic step in which a person who is searching for an apartment rents a free apartment that she/he likes. For simplicity, in the graphical representation we use bidirectional arrows instead of drawing two arrows consuming and producing tokens in state \texttt{pref}. Transition \texttt{n changes} corresponds to the case in which $n$ people interchange their apartments. Note there are infinitely many transitions of this kind, one for any $n \geq 1$ ($n$ can take the value 1 since nothing prevents a person to take the apartment she rents). We remark that for any $n$ there is one and only one transition, i.e., formal parameters allow any combination of people and apartments. Similarly, the transition \texttt{n changes&wants&free} stands for the atomic step in which $n$ people change their apartments, but one of them takes a free apartment and one person looking for an apartment participates in the exchange. Note that the abstract net does not contain transitions like \texttt{n changes&k wants&k free}, because this kind of behaviour corresponds to the execution of several disconnected transactions, and hence it can be obtained by combining previous transitions.

Observe that this infinite abstract net is modelled with a finite concrete zs net.

Finally, the correspondence between the different views given by the concrete zs net and the abstract net is guaranteed by the following result.

Theorem 4.10. Let $B$ be a c-zs net and $I_B$ its abstract net. Then $m \rightarrow_{T_B} m'$ iff $m \Rightarrow_{T_B} m'$.

Proof.

$\Rightarrow$ By induction on the structure of the proof $m \rightarrow_{T_B} m'$.

- Rule (\textsc{coloured-firing}). There is a transition $m_1 \parallel m'_1$ in $I_B$ s.t. $m_1 \times \sigma \oplus m'' = m$ and $m'_1 \times \sigma \oplus m'' = m'$, i.e., $m_1$ is consumed, $m'_1$ is produced, and $m''$ denotes idle
resources. Consequently, by the construction of the abstract net there is a connected
transaction $\xi$ with a mgce $\sigma_\xi$ s.t. $\text{pre}(\xi) \times \sigma_\xi = m_1$ and $\text{post}(\xi) \times \sigma_\xi = m'_1$. By
Lemma 4.6 there exists a proof for $m_1 \times \sigma \rightarrow_{I_B} m'_1 \times \sigma$. Since in a proof it is always
possible to add idle resources (it is easy to show by proof induction that if $m \rightarrow_T m'$
then $m \oplus m'' \rightarrow_T m' \oplus m''$), we have $m_1 \times \sigma \oplus m'' \rightarrow_{I_B} m'_1 \times \sigma \oplus m''$, and hence
$m \rightarrow_{I_B} m'$.

- **Rule (step).** The proof is immediate by using inductive hypothesis on premises.

$\Leftarrow$) If $m \rightarrow_B m'$, then by Lemma 4.8 there exists a set of connected transactions $\{\xi_i\}$ s.t.

- $m = \bigoplus_i \text{pre}(\xi_i) \times \rho \oplus m''$ and $m' = \bigoplus_i \text{post}(\xi_i) \times \rho \oplus m''$;

By construction of the abstract net, there exists a transition $t_{\xi_i}$ in $I_B$ for any connected
transaction $\xi_i$ of $N_B$ s.t. $\cdot t_{\xi_i} = \text{pre}(\xi_i) \times \sigma_{\xi_i}$ and $\cdot t_{\xi_i}^* = \text{post}(\xi_i) \times \sigma_{\xi_i}$. Since $\sigma_{\xi_i}$ is the mgce
of $\xi_i$ and $\rho$ is a compatible execution of every $\xi_i$, there exists $\gamma_i$ s.t. $\rho = \sigma_{\xi_i} \mid \gamma_i$. Then, it
is possible to build a proof for

$$\text{pre}(\xi_i) \times \rho = \text{pre}(\xi_i) \times (\sigma_{\xi_i} \mid \gamma_i) = (\text{pre}(\xi_i) \times \sigma_{\xi_i}) \times \gamma_i \rightarrow_{I_B}$$

$$\text{post}(\xi_i) \times \sigma_{\xi_i} \times \gamma_i = \text{post}(\xi_i) \times (\sigma_{\xi_i} \mid \gamma_i) = \text{post}(\xi_i) \times \rho$$

by using rule (coloured-firing) for any $t_{\xi_i}$. Finally, all proofs can be combined by
using repeatedly rule (step) and by noting that the idle resources can always be added
to a proof. \qed
4.6 ZS nets as C-ZS nets

P/T (and ZS) nets can be seen as a particular case of C-P/T (resp., C-ZS) nets where tokens are coloured with the empty sequence $\bullet$.

**Definition 4.23** (Coloured version of a P/T net). Let $N = (S_N,T_N,\delta_0N,\delta_1N, m_0N)$ be a P/T net. The **coloured version** of $N$ is the C-P/T net $C_N = (S_N,\emptyset, T_N, \delta_{0CN}, \delta_{1CN}, m_{0CN})$, where $\delta_{iCN}(t)(a,\bullet) = \delta_{iN}(t)(a)$ for $i = 0,1$ and $m_{0CN}(a,\bullet) = m_{0N}(a)$. The C-ZS net $C_B$ is the coloured version of the ZS net $B$ if the underlying C-P/T net $N_{C_B}$ is the coloured version of the underlying P/T net $N_B$ of $B$, and $Z_B = Z_{C_B}$.

In what follows we show that the construction of the abstract net under these two different views is consistent. That is, for every ordinary ZS net $B$, the result of constructing the abstract net of the coloured version of $B$ is essentially the same as the net obtained by colouring the abstract net of $B$. The following result makes the lower tile of the hierarchy of transactional nets (shown in Figure 4.7) to commute.

**Theorem 4.11.** Let $B$ be a ZS net, $I_B$ its abstract net, $C_B$ and $I_{C_B}$ their coloured versions, and $I_{C_B}$ the abstract net of $C_B$ (i.e., the coloured version of $B$). Then $C_{I_B} \approx I_{C_B}$.

**Proof.** The proof follows by noting that any transaction of $B$ uniquely corresponds to a transaction of $C_B$ and vice versa.

- **Transactions of $B$ correspond to transactions of $C_B$.** We show that for any connected transaction $\xi : K \to N_B$ of $B$ there exists a connected transaction $\xi' : C_K \to C_{N_B}$ of $C_B$ (we recall that $N_B$ is the underlying net of $B$). If $K$ is a causal net, then $C_K$ is a coloured causal net. Moreover $C_K$ is a plain net, and by Lemma 4.1 $C_K$ is compatible. Define $\xi'$ in a way s.t. $\forall a \in S_{C_K} : \xi'(a) = \xi(a)$ and $\forall t \in T_{C_K} : \xi'(t) = \xi(t)$ and $\sigma(t) = \emptyset$. It is easy to notice that $\xi'$ is a process of $C_{N_B}$ since $\xi$ is a process. Moreover $\xi'$ is a connected transaction of $C_{N_B}$, since it has the same structure of $\xi$ (i.e., it has the same initial, evolution, and final places, and it is full and connected), and the mapping is the same as in $\xi$ (therefore initial and final places are stable, and evolution places are zero-safe). Note that for all $\xi$ there exists $\xi'$. Hence, we can define a function $f$ such that for all connected transaction $\xi$, then $f(\xi) = \xi'$, where $\xi'$ is the coloured connected transaction defined as before.
4.7. Related works

This section compares the work presented in this chapter against different approaches in literature. In particular, we focus on the model of coloured nets, on the notion of processes for coloured nets, and on the encoding of ZS nets as coloured nets.

**Coloured nets.** Different variants of coloured nets (known also known as *High-level Petri nets*) have been proposed in literature (see for instance [73, 72, 93, 12]). The model we use here is a simplified version of the general cases, which is enough to account for name passing. In general, the values carried by tokens belongs to a given type, which can be arbitrarily complex, instead of just tuples of names as we use here. Note that the presentation of coloured ZS nets we gave can be extended to consider typed tokens, since the ZS approach is somehow orthogonal to the data type of the values carried by tokens. Clearly, the expressions that characterise the colours of fetched and produced tokens of transitions are typed expressions that may contain variables. In addition, it is possible to attach a boolean expression (with variables), called the guard, to each transition. A guard specifies that only binding elements for which the boolean expression evaluates to true are acceptable (e.g., it is possible to write that a token should have a colour different from a particular constant). This feature is not included in our model, which only accounts for simple pattern matching among variables and constants. On the other hand, we enriched the non-linear pattern matching in [4, 31] with constants. Nevertheless, we remark that non-linear pattern matching is regarded as not suitable for mobile calculi and it is explicitly excluded by some process calculi (i.e., Join), because it is difficult to implement in an efficient way (for a discussion see [31]).
Coloured Processes. Our proposal for processes of coloured nets is similar to the approach presented in [5], that avoids the usual flattening of High-Level Petri nets. By flattening a high-level Petri net $N$ an equivalent ordinary net $\text{flat}(N)$ is obtained. Then, the processes of $N$ are those of $\text{flat}(N)$. Instead, in [5], a high-level process is intended to capture a set of different concurrent computations corresponding to different input parameters. In particular, this is achieved (analogously to our approach) by defining suitable high-level causal nets. The feature that distinguishes our approach from [5] is that we are able to take into account the constraints that relate the values that are carried on by tokens. Such constraints are yield by means unification over the variables used by transitions. When no such unification exists, then the causal net does not describe a valid concurrent computation of the net.

ZS nets can be encoded as C-P/T nets. ZS nets have been encoded in [20] into a fragment of the join calculus corresponding to the coloured nets. In such encoding (which is called flat) the transactional mechanism of ZS nets is implemented through a centralised coordinator, which is aware of the zero tokens present in the net. Roughly, stable tokens produced during a transaction are kept frozen by the coordinator, which will release them when no zero token is left in the net.
Chapter 5

Reconfigurable zs nets

The idea behind reconfigurable nets (R-P/T nets) is that the data carried by tokens (i.e., their colours) are the names of places in the net, and transitions can use the colours of fetched tokens (received values) as places in their postsets. Consequently, the postset of a transition is not static as in P/T nets and C-P/T nets but it depends on the colours of the consumed tokens. For instance, a transition \( t = a(x) j x(a) \) denotes a pattern that consumes a token from \( a \) and generates a token in the place corresponding to the colour \( x \) of the consumed token, which is instantiated at runtime. For example, when \( t \) is applied to \( m_1 = a(b) \) it produces \( m_2 = b(a) \). Instead, when applied to \( m'_1 = a(c) \) it generates \( m'_2 = c(a) \).

Consequently, the definitions for nets and place/transitions nets are extended to allow received names to appear as places in the postsets of transitions.

5.1 Background: Reconfigurable nets

As for coloured nets, we consider an infinite set of variable names \( \mathcal{X} \), ranged over by \( x, y, \ldots \). We require also variable names to be different from place names, i.e., \( \mathcal{X} \cap \mathcal{P} = \emptyset \). Moreover, the colours carried on by tokens are the names of places, hence \( \mathcal{V} = \mathcal{P} \). Note that constant colours (as used in C-P/T) can be modelled in reconfigurable nets as isolated places.

In this section we omit the definition of reconfigurable nets, markings, initial and final places since they are analogous to those in previous chapters, and we start by defining reconfigurable marked place/transformation net.

Definition 5.1 (R-P/T net). A Reconfigurable marked place / transition net is a 5-tuple \( N = (S_N, T_N, \delta_0N, \delta_1N, m_0N) \) where \( S_N \subseteq \mathcal{P} \) is a set of places, \( T_N \) is a set of transitions, the functions \( \delta_0N : T_N \rightarrow \mathcal{M}_{S_N,S_N \cup \mathcal{X}} \) and \( \delta_1N : T_N \rightarrow \mathcal{M}_{S_N \cup \mathcal{X}, S_N \cup \mathcal{X}} \) assign respectively source and target to each transition, and \( m_0N \in \mathcal{M}_{S_N,S_N} \) is the initial marking. Moreover, for every \( t \) in \( T_N \) we require \( \delta_1N(t) \subseteq \mathcal{M}_{S_N \cup m(t), S_N \cup m(t)} \), i.e., variables occurring in the postset of a transition are received names.
Note that variables can appear in the preset of transitions only as colours (similar to coloured transitions), but they can occur also as places in the preset. The usage of variables is analogous to the coloured model, i.e., they are formal parameters that are instantiated when a transition is fired. The main difference is that variables appearing in the preset of a transition as places have to be substituted also by received names. The following definition introduces the substitution of names occurring both as colours and places.

**Definition 5.2** (Substitution). Let \( \sigma : \mathcal{X} \to \mathcal{X} \cup \mathcal{P} \) be a partial function. The substitution \( \sigma \) on a multiset \( m \in \mathcal{M}_{\mathcal{X} \cup \mathcal{P}, \mathcal{X} \cup \mathcal{P}} \) is given by

\[
(m\sigma)(s_1)(c_1) = \sum_{s \in \{s' | s' \sigma = s_1 \}} \sum_{c \in \{c' | c' \sigma = c_1 \}} m(s)(c)
\]

The composed substitution \( (\sigma_1; \sigma_2) \) applied over a multiset is defined as \( m(\sigma_1; \sigma_2) = (m\sigma_1)\sigma_2 \).

We remark that the difference w.r.t. Definition 4.6 is that \( \sigma \) is applied also to place names and not just to colours.

As in coloured nets, variables used for defining a transition are local to “that” transition, and hence they can be renamed. The \( \alpha \)-conversion on received colours for reconfigurable transitions is defined as follows.

**Definition 5.3** (\( \alpha \)-equivalence of transitions). Two reconfigurable transitions \( t_1 = m_1 \mid m'_1 \) and \( t_2 = m_2 \mid m'_2 \) are \( \alpha \)-convertible if there exists an injective substitution \( \sigma : \mathcal{X} \to \mathcal{X} \), where \( \text{rn}(t_1) \subseteq \text{dom}(\sigma) \), such that \( m_1 \star \sigma = m_2 \) and \( m'_1 \sigma = m'_2 \). The \( \alpha \)-conversion is an equivalence relation, which is denoted by \( \equiv_\alpha \). (Note that since \( \sigma \) is injective, then \( \sigma^{-1} \) exists and \( m_2 \star \sigma^{-1} = m_1 \) and \( m'_2 \sigma^{-1} = m'_1 \) hold). Generally, we refer to transitions up-to \( \alpha \)-equivalence.

**Notation 5.1.** Since in reconfigurable nets \( \mathcal{V} = \mathcal{P} \), we will use \( \text{col}_\mathcal{P} \) instead of \( \text{col}_\mathcal{V} \) to denote the constant colours of a marking.

### 5.1.1 Operational semantics

The operational semantics for reconfigurable nets is defined by replacing the rule (Firing) in Figure 3.1 by the following rule:

\[
\begin{array}{c}
\text{(Reconfig-Firing)} \\
\frac{t = m \mid m'_1 \in T \quad m'' \in \mathcal{M}_{S,S}}{m \star \sigma \oplus m'' \xrightarrow{T} m' \sigma \oplus m''} \quad \text{rn}(t) \subseteq \text{dom}(\sigma) \text{ and range}(\sigma) \subseteq S
\end{array}
\]

Comparing this rule against the rule (Coloured-Firing) of C-P/T nets, it should be clear that in both cases a transition \( t = m \mid m' \) can be fired on a marking \( m_1 \) only when it contains an instance of the preset \( m \) obtained by substituting colour
5.2. RECONFIGURABLE DETERMINISTIC PROCESSES

Figure 5.1: A simple R-P/T net.

\[
\begin{align*}
\text{a}(x, y) & \land x(\bullet) \oplus y(x) \in T \quad \sigma = \{ a/x, b/y \} \\
\text{b}(v) & \land v(\bullet) \in T \quad \sigma = \{ a/v \}
\end{align*}
\]

\[
\begin{align*}
\text{a}(a, b) & \rightarrow_T \text{a}(\bullet) \oplus \text{b}(a) \\
\text{b}(a) & \rightarrow_T \text{a}(\bullet)
\end{align*}
\]

\[\text{STEP}\]

\[
\text{a}(a, \cdot) \oplus \text{b}(a) \rightarrow_T \text{a}(\bullet) \oplus \text{a}(\bullet) \oplus \text{b}(a)
\]

Figure 5.2: A computation on a reconfigurable net.

variables (received names) by constants (i.e., \(m \star \sigma\)). Hence, transitions in both C-P/T nets and R-P/T nets consume messages from a fixed set of places. Differently, the firing of \(t\) in a C-P/T produces tokens in a fixed set of places (although their colour can differ from firing to firing), while in R-P/T nets two different firings of the same transition can produce tokens in different places, i.e. the postset of a transition changes dynamically depending on the colours of the consumed tokens.

Example 5.1. A simple reconfigurable net is depicted in Figure 5.1. A proof for a computation that concurrently fires transitions \(t_1\) and \(t_2\) is in Figure 5.2 (RF stands for RECONF-FIRING).

5.2 Reconfigurable Deterministic Processes

In this section we revise the notions of morphism, causal net, and process to take into account reconfigurable transitions.

Before defining the notion of morphisms over reconfigurable nets, we describe the application of a mapping over a multiset in the reconfigurable setting.

Definition 5.4. Let \(S\) be a set of names, i.e. \(S \subseteq \mathcal{P}\). Then a mapping \(f : S \rightarrow \mathcal{P}\) applied to a colour sequence \(c \in \mathcal{P}^*\) is the sequence \(d \in (\mathcal{P}^*)^*\) obtained by applying \(f'(x)\) to every element of \(c\), where \(f'(x) = f(x)\) if \(x \in S\), otherwise \(f'(x) = x\) (i.e., \(f'(x)\) is the identity over names not in \(S\)). Given a multiset \(m \in \mathcal{M}_{S,P}\) and a partial function \(f : S \rightarrow S'\) s.t. \((\text{dom}(m) \cup \text{col}_P(m)) \subseteq \text{dom}(f)\), \(f\) applied to \(m\) is the multiset \(f(m) \in \mathcal{M}_{S',P}\) s.t. \(f(m)(s_1)(c_1) = \sum_{s \in \{s' | f(s') = s_1\}} \sum_{c \in \{c' | f(c') = c_1\}} m(c)(s)\).
As usual, a morphism between reconfigurable nets maps places into places and transitions into transitions. As done for coloured nets, we also require transitions in one net to be a particular case of the transitions in the other, i.e., received names have been renamed or substituted by constants.

**Definition 5.5** (Instance of a transition). Let \( t = m \mapsto m' \) be a transition. Given a substitution \( \sigma \) s.t. \( rn(t) \subseteq dom(\sigma) \), a transition \( i \) is said an *instance of* \( t \) for \( \sigma \) if \( i = m \ast \sigma \mapsto m' \sigma \).

Then the notion of morphism between reconfigurable nets requires transitions in one net to correspond with instances of transitions in the other.

**Definition 5.6** (Reconfigurable net morphism). Let \( N, N' \) be R-P/T nets. A tuple \( f = (f_S : S_N \rightarrow S_{N'}, f_T : T_N \rightarrow T_{N'}, \sigma = \{ \sigma_t \}_{t \in T_N} \) \), where \( \sigma \) is a family of substitutions (one for each \( t \in T_N \) s.t. \( \sigma_t : rn(f_T(t)) \rightarrow col(\bullet t) \), is said a *reconfigurable net morphism* from \( N \) to \( N' \) (written \( f : N \rightarrow_{\sigma} N' \)) if for any \( t \in T_N \) \( f_S(\bullet t) \mapsto f_S(t') \) is an instance of \( f_T(t) \) for \( \sigma_t \).

For the particular case of coloured transitions (i.e., transitions without reconfigurable capabilities), the condition required on the mapping is analogous to that on coloured net morphisms (Definition 4.8). In fact, the substitution on the postset of transitions has effect only on colours, and therefore \( f_T(t') \ast \sigma_t = f_T(t') \ast \sigma_t \).

The composition of reconfigurable net morphisms and isomorphic R-P/T are defined analogously to coloured model (Definitions 4.9 and 4.10). Also in this case, reconfigurable nets and reconfigurable net morphisms form a category.

**Example 5.2.** Figure 5.3 shows a morphism from \( K \) to \( N \). Note that the received name \( y \) of \( t_1 \) has been instantiated with the constant \( c \), while the variable \( x \) has been renamed as \( z \). In fact

\[
\begin{align*}
    f_S(a_1(z, c_1)) \mapsto f_S(z(x) \oplus c_1(z)) &= a(z, c) \mapsto z(x) \oplus c(z) \\
    &= a(x, y) \ast \{ z/x, c/y \}(x(x) \oplus y(x))\{ z/x, c/y \} \\
    &= a(x, y) \ast \sigma_{t_1} (x(x) \oplus y(x))\sigma_{t_1}.
\end{align*}
\]

Similarly, \( t_2' \) corresponds to an instance of \( t_2 \) where \( v \) is named \( w \).

Note that any instantiation \( \sigma_t \) can be written as the union of two disjoint substitutions: (i) \( \iota : \mathcal{X} \rightarrow \mathcal{P} \) substituting variables with constants; and (ii) \( \alpha : \mathcal{X} \rightarrow \mathcal{X} \) renaming variables. We use this distinction when defining the notion of reconfigurable processes.

**Definition 5.7** (Proper instantiation). Given an instantiation \( \sigma_t = \iota_t \uplus \alpha_t \) such that \( \iota_t : \mathcal{X} \rightarrow \mathcal{P} \) and \( \alpha_t : \mathcal{X} \rightarrow \mathcal{X} \), we call \( \iota_t \) a proper instantiation of \( t \). Usually we write a reconfigurable morphism \( P : K \rightarrow (\alpha_t) N \) as \( P : K \rightarrow (\iota_t, \alpha_t) N \).
5.2. RECONFIGURABLE DETERMINISTIC PROCESSES

![Diagram](image)

**Figure 5.3:** Reconfigurable net morphism.

The proper instantiations used by the morphism shown in Figure 5.3 are $\iota_{t_1} = \{c/y\}$ and $\iota_{t_2} = \emptyset$.

The following definitions for causal nets, compatible execution and compatible causal nets are analogous to the coloured case.

**Definition 5.8** (Deterministic reconfigurable causal net). A reconfigurable net $K = (S_K, T_K, \delta_{0K}, \delta_{1K})$ is a **deterministic reconfigurable causal net** if it is acyclic and transitions do not share:

- places in their pre- and postsets, i.e. if $a \in \delta_i (t_0)$ and $a \in \delta_i (t_1)$ then $t_0 = t_1$ for $i = 0, 1$; nor

- colour variables, i.e. $\forall t_1, t_2 \in T_K : t_1 \neq t_2$ implies $rn(t_1) \cap rn(t_2) = \emptyset$.

Nets $N$ and $K$ in Figure 5.3 are deterministic reconfigurable causal nets.

**Definition 5.9** (Compatible execution of $K$). Let $K$ be a reconfigurable causal net. A substitution $\rho$ is said a **compatible execution** of $K$ if $\forall a \in S_K$ if $\exists t_1, t_2 \in T_K$ s.t. $(a, c_1) \in t_1^*$ and $(a, c_2) \in t_2^*$ then $c_1\rho = c_2\rho$. If such $\rho$ exists, then we call $K$ compatible.

Net $K$ in Figure 5.3 is compatible. In fact the substitution $\rho = \{w/z\}$ unifies the colours of tokens produced in and consumed from $c_1$, which is the only place of $K$ shared by two transitions.
As for the coloured case, we like each process to capture the most general pattern for executions. Hence we will use the notion of general compatible executions. Nevertheless, in the reconfigurable case, there are other cases in which a causal net over-instantiates variables. Consider, for instance the morphism shown in Figure 5.4. Clearly, the causal net \( K' \) describes a more specific computation on \( N \) that the net \( K \) in Figure 5.3. In \( K' \) the colour variable \( x \) has been instantiated to \( a \) while in \( K \) the computation is described for any instantiation of \( x \) (and in particular when \( x \) takes the value \( a \)). For this reason, when defining the notion of processes for reconfigurable nets we require the instantiations of the morphism to be minimal, in the sense that variables substituted by constants (i) occur as places in the postset of transitions, and (ii) at least one place corresponding to the instantiated variable is consumed by the process. These conditions are satisfied by the morphism shown in Figure 5.3. Note that the only proper instantiation is \( \iota'_t = \{ c/y \} \). In fact the instantiated variable \( y \) (i) appears as a place name in the postset of \( t_1 \), and (ii) the coresponding place \( c_t \) is a place consumed by the process.

**Definition 5.10** (Process of a reconfigurable net). A deterministic causal process for a R-P/T net \( N \) is a coloured net morphism \( P : K \to \iota' \alpha N \) from a compatible reconfigurable causal net \( K \) to \( N \), if \( \forall \iota_t \in \iota', \) for all \( \{ a/x \} \) s.t. \( \iota_t = \{ a/x \} \cup \iota'_t \) then:

1. \( x \in \text{dom}(P(t')) \), i.e. substituted variables occur as places in the postset of the instantiated transition; and
2. The instantiation is minimal in the sense that at least one token generated in
the variable place denoted by \( x \) is consumed by the process, i.e.
\[
\{[P(i(i_e))] \mid i(i_e) \in t^* \land P(i) = a \land |i^*| = 1\} \not\subset \{a(i'_{i_e}) \in (P(t)^* (i'_t \cup \alpha_t)) \ast \{a/x\}\}
\]

The second condition in the above definition is interpreted as follow. Consider a transition \( t \) that is instantiated with \( \iota_t = \{a/x\} \cup i'_t \). The multiset \( \{[P(i(i_e))] \mid i(i_e) \in t^* \land P(i) = a \land |i^*| = 1\} \) denotes the marking associated to the tokens of the postset of \( t \) (condition \( i(i_e) \in t^* \)) that are mapped to \( a \) \( (P(i) = a) \) and are consumed by \( P \) \( (|i^*| = 1) \). Instead, the multiset \( \{a(i'_{i_e}) \in (P(t)^* (i'_t \cup \alpha_t)) \ast \{a/x\}\} \) stands for the marking that can be obtained from the postset of \( P(t) \) without instantiating \( x \). By requiring the first multiset not to be included in the second, we state the fact that it is necessary to instantiate \( x \).

**Example 5.3.** The morphism shown in Figure 5.3 is a process of \( N \). In fact, the only non-empty proper instantiation is \( \iota_{t'_1} = \{c/y\} \), which has one pair that satisfies the two conditions:

1. \( y \in P(t'_1)^* = t'_1 \); and
2. \( \{[P(i(i_e))] \mid i(i_e) \in t'_1^* \land P(i) = c \land |i^*| = 1\} = \{\{P(c_1, z)\} = \{(c, z)\} \not\subset \{c(i'_e) \in (P(t'_1)^* (z/x)) \ast \{c/y\}\} = \{c(i'_e) \in (z(\bullet) \oplus y(z))\} = \emptyset\}
\]

Instead, the morphism in Figure 5.4 is not a process. In fact the the pair \( \{a/x\} \) in the instantiation \( \iota_{t'_1} = \{a/x, c/y\} \) does not satisfy the condition (2). Note that the first multiset \( \{[P(i(i_e))] \mid i(i_e) \in t'_1^* \land P(i) = a \land |i^*| = 1\} \) is empty, and hence included in any other set.

As usual, the set of origins and destinations of a process \( P : K \rightarrow_{\iota, \alpha} N \) are denoted by \( O(P) = ^\circ K \) and \( D(P) = K^\circ \cap S_K \), respectively. The set of evolution places of \( P \) is the set \( E_P = \{P(a) \mid a \in K, |a^*| = 1\} \).

Reconfigurable process isomorphism could be defined analogously to coloured process isomorphism, nevertheless such definition would be so narrow, because it would imply constant colours to be used in the same way by isomorphic processes. Consider the processes in Figure 5.5. Clearly, there is no isomorphism between \( K_1 \) and \( K_2 \), because the colours of \( t_1 \) and \( t'_1 \) are different constants. Nevertheless, both processes describe the same computation in \( N \), they differ only on the names used by transitions to denote a colour of the original net \( N \), i.e. \( K_1 \) uses \( a_1 \) to denote a while \( K_2 \) uses \( a_2 \). Hence, we would like such processes to be identified as isomorphic. The following definition allows different constants mapped to the same place in \( N \) to be used interchangeably by isomorphic processes.

**Definition 5.11 (Process Isomorphism).** Let \( P_1 : K_1 \rightarrow_{\sigma_1} N \) and \( P_2 : K_2 \rightarrow_{\sigma_2} N \) two reconfigurable processes. The triple \( (f_S : S_{K_1} \rightarrow S_{K_2}, f_T : T_{K_1} \rightarrow T_{K_2}, \{\sigma_t\}_{t \in T_{K_1}}) \), where \( f_S \) and \( f_T \) are injective mappings and all \( \sigma_t \) are injective variable renamings, is an isomorphism from \( P_1 \) to \( P_2 \) if \( f_S(\ast t) \ast f_S(\ast t) \ast f_S(\ast t) \ast f_S(\ast t) \ast f_S(\ast t) = \ast f_T(\ast t) \ast f_S(\ast t) \ast f_T(\ast t) \ast f_S(\ast t) \ast f_S(\ast t) \) (where \( m \ast f_{S_1} \) stands for the application of \( f_{S_1} \) just on the colours of \( m \)).
As in the previous cases, it is easy to show that process isomorphism is an equivalence relation over processes. We denote with $[P]_\sim$ the equivalence class of $P$.

### 5.3 Correspondence between processes and firings

As done for coloured processes, in this section we show the relation between reconfigurable processes and firings in R-P/T nets. The results are analogous to those presented in § 4.3 for coloured nets. Also the proofs follow similarly, hence we will omit the detailed description of similar parts and proceed by highlighting the main differences.

Constant colours in reconfigurable nets correspond to places defined by the net, hence reconfigurable processes should include a place for any constant colour that appears in it. Differently from coloured processes, where isolated places can be always removed to obtain another process, it is not always possible to remove isolated places from reconfigurable processes. Consider the process depicted in Figure 5.6. Clearly, it is not possible to obtain a new process by removing the place $c_1$ from $K$ since $K$ needs to define a place to talk about the colour $c$ in the pattern matching expression of transition $t'_1$. In general, it is not always possible to define a process without isolated places for a computation in a reconfigurable net. Nevertheless, it is possible to think about processes up-to places denoting constant colours in transitions.

**Definition 5.12** (Process up-to constant colours). Let $P : K \rightarrow_{\sigma} N$ be a reconfigurable process, $P$ can be considered up-to constant colours if $\forall a \in S_K \cap S_N : \bullet a = a^* = \emptyset$ and $P(a) = a$. 
5.3. CORRESPONDENCE BETWEEN PROCESSES AND FIRINGS

Figure 5.6: A R-P/T morphism using pattern matching.

For simplicity we prefer to name constants identically in the causal net $K$ and the net $N$ instead of handling explicitly the places that are used just as colours. Nevertheless, this choice does not compromise the results in this section. In what follows we refer to process implicitly as up-to constants. In particular, the definition of isolated places of a process does not consider places that are constants.

**Definition 5.13** (Isolated places). Let $P : K \rightarrow_c N$ be a reconfigurable process, the set $I(P)$ of isolated places of $P$ is defined as $I(P) = (O(P) \cap D(P)) \setminus S_N$, where $O(P)$ and $D(P)$ are defined as usual.

The definitions for coloured initial places, colour assignments and coloured isolated processes are analogous to those for the coloured model. The only difference is for coloured final places, where variables have to be taken into account. In particular, the multiset coloured final places are obtained by colouring final places and final variables.

**Definition 5.14** (Coloured final places). Let $P$ be a coloured process. The multiset of coloured connected final places is defined as follow:

$$post_c(P) = \{((x, c) | \exists t \in T_K, (x, c) \in t^* \wedge x \in D_c(P) \cup X]$$

In order to build the processes associated to a sequence of steps we will use the auxiliary relations of suitable mappings $F$, pre-markings $G$ and suitable colour assignment $A$ defined in Section 4.3.
The first result shows that any step \( m_1 \rightarrow_{T_N} m_2 \) in a \( R\)-\( P/T \) \( N \) it is possible to build a reconfigurable process whose initial and final markings coincide resp. with \( m_1 \) and \( m_2 \).

**Lemma 5.1.** Let \( N = (S_N, T_N, \delta_{0N}, \delta_{1N}) \) be a \( R\)-\( P/T \) net. If \( m_1 \rightarrow_{T_N} m_2 \), then there exist a process (up-to constants) \( P : K \rightarrow_{G} N \), a compatible execution \( \rho : \mathcal{X} \rightarrow \mathcal{P} \) of \( K \), and a colour assignment \( a \) s.t.

- \( m_1 = P(\text{pre}_c(P)) \ast \rho \oplus P(I(P, a)) \)
- \( m_2 = P(\text{post}_c(P)) \rho \oplus P(I(P, a)) \)

**Proof.** By rule induction.

- **Rule (RECONF-Firing):** \( \exists t = m| m' \in T_N \) s.t. \( m_1 = m \ast \sigma_1 \oplus m''_1 \) and \( m_2 = m'_1 \sigma_1 \oplus m''_1 \), where \( \text{range}(\sigma_1) \subseteq \mathcal{P} \). In particular, \( m'_1 \) can be written as two multisets \( m'_1 = m'_{\mathcal{P}} \oplus m'_{\mathcal{X}} \), where \( m'_{\mathcal{P}} \) is the multiset of constant places, i.e. \( \text{dom}(m'_{\mathcal{P}}) \subseteq \mathcal{P} \), and \( m'_{\mathcal{X}} \) is the multiset of variable places, i.e. \( \text{dom}(m'_{\mathcal{X}}) \subseteq \mathcal{X} \).

Then, define a reconfigurable causal net \( K = (S_K, T_K, \delta_{0K}, \delta_{1K}) \) as follow:

- \( S_K = S_N \cup S_{K_1} \cup S_{K_{\mathcal{P}}} \cup S_{K_{\mathcal{X}}} \), where all sets are pairwise disjoint and \( |S_{K_1}| = |m|, |S_{K_{\mathcal{P}}}| = |m'_1|, \) and \( |S_{K_{\mathcal{X}}}| = |m'_{\mathcal{X}}| \). Different from coloured nets, we put on \( S_{K_{\mathcal{X}}} \) as many places as the number of tokens generated by \( t \) on constant places. Note also that all places of \( S_N \) are also places of \( S_K \) (they will be used as constant colours).

- \( T_K = \{ t' \} \)

- Then define \( f_S(S_K) = f_{S_{K_1}} \cup f_{S_{K_{\mathcal{P}}}} \cup f_{S_{K_{\mathcal{X}}}} \cup id_{S_N} \) by choosing any \( f_{S_{K_1}}, f_{S_{K_{\mathcal{P}}}}, f_{S_{K_{\mathcal{X}}}} \) such that \( \mathcal{F}(S_{K_1}, m, f_{S_{K_1}}), \mathcal{F}(S_{K_{\mathcal{P}}}, m'_{\mathcal{P}}, f_{S_{K_{\mathcal{P}}}}), \mathcal{F}(S_{K_{\mathcal{X}}}, m'_{\mathcal{X}}, f_{S_{K_{\mathcal{X}}}}) \) and \( \mathcal{F}(S_{K_{\mathcal{P}}}, m''_{\mathcal{P}}, f_{S_{K_{\mathcal{X}}}}) \). Let \( id_{S_K} \) be the identity function on places belonging to \( S_N \) (we recall that we are constructing a process up-to constants, which are isolated places in \( K \) that are mapped to homonymous places in \( N \)). Finally, take any injective function \( \sigma' : \text{col}(X(t)) \rightarrow \mathcal{X} \) (i.e., a variable renaming) and define

\[
\delta_{0K}(t') = n_1 \ast \sigma' \text{ for any } n_1 \text{ s.t. } \mathcal{G}(S_{K_1}, m, f_{S_{K_1}}, n_1)
\]

\[
\delta_{1K}(t') = n_2 \ast \sigma' \oplus m'_{\mathcal{X}} \sigma' \text{ for any } n_2 \text{ s.t. } \mathcal{G}(S_{K_{\mathcal{X}}}, m'_{\mathcal{P}}, f_{S_{K_{\mathcal{X}}}}, n_2)
\]

Following the same reasoning as in the proof of Lemma 4.2, the construction of \( K \) yields a compatible causal net, where any substitution is a compatible execution. Clearly, any place \( a \in S_N \) s.t. \( a \in S_K \) (i.e., the constants) are mapped by \( f_S \) to the homonymous places on \( N \).

It remains to prove that the triple \( (f_S, f_T, \{ \sigma' \}) \) is a process, where \( f_S \) and \( \sigma' \) are the functions chosen previously, and \( f_T \) mapping \( t' \) into \( t \), i.e. \( f_T(t') = t \). In fact
\[ f_s(t') = f_s(n_1 \ast \sigma_v) = f_s(n_2 \ast \sigma_v + m'_\chi \sigma_v) \]

By def. of \( K \)
\[ = f_s(n_1 \ast \sigma_v) = f_s(n_2 \ast \sigma_v \oplus f_s(m'_\chi \sigma_v) \]

By distributivity of \( f_s \)
\[ = m \ast \sigma_v \mid m'_p \ast \sigma_v \oplus f_s(m'_\chi \sigma_v) \]

By Remark 4.2(2)
\[ = m \ast \sigma_v \mid m'_p \sigma_v \oplus f_s(m'_\chi \sigma_v) \]

Since \( \text{dom}(m'_p) \subseteq P \),
\[ = m \ast \sigma_v \mid m'_p \sigma_v \oplus m'_\chi \sigma_v \]

Since \( \text{dom}(m'_\chi) \subseteq \chi \) and
\[ = m \ast \sigma_v \mid m'_p \oplus m'_\chi \sigma_v \]

By distrib. of subst.
\[ = t \ast \sigma_v \mid t \ast \sigma_v \]

By def. of \( N \)
\[ = t \ast \sigma_v \mid t \ast \sigma_v \ast \sigma_v \]

By def. of \( f_T \)

As done for coloured net, by considering the substitution \( \rho = \sigma_v^{-1} \cdot \sigma_1 \), it can be proved that \( P(\text{post}_c(P)) \ast \rho = m \sigma_1 \). While for the final elements, we have

\[ P(\text{post}_c(P)) \rho = P(\text{post}_c(P)) \sigma_v^{-1} \cdot \sigma_1 \]

By definition of \( \rho \)
\[ = P(t') \sigma_v^{-1} \cdot \sigma_1 \]

By definition of \( P \)
\[ = P(n_2 \ast \sigma_v \oplus m'_\chi \sigma_v) \sigma_v^{-1} \cdot \sigma_1 \]

By definition of \( \delta_{1K}(t) \)
\[ = f_s(n_2 \ast \sigma_v \oplus m'_\chi \sigma_v) \sigma_v^{-1} \cdot \sigma_1 \]

By definition of \( P \)
\[ = (f_s(n_2 \ast \sigma_v \oplus f_s(m'_\chi \sigma_v)) \sigma_v^{-1} \cdot \sigma_1 \]

By distributivity of \( f_s \)
\[ = (m'_p \ast \sigma_v \oplus f_s(m'_\chi \sigma_v)) \sigma_v^{-1} \cdot \sigma_1 \]

By Remark 4.2(2)
\[ = (m'_p \sigma_v \oplus f_s(m'_\chi \sigma_v)) \sigma_v^{-1} \cdot \sigma_1 \]

Since \( \text{dom}(m'_p) \subseteq P \),
\[ = (m'_p \sigma_v \oplus m'_\chi \sigma_v) \sigma_v^{-1} \cdot \sigma_1 \]

Since \( \text{dom}(m'_\chi) \subseteq \chi \) and
\[ = (m'_p \oplus m'_\chi) \sigma_v \sigma_v^{-1} \cdot \sigma_1 \]

By distrib. of subst.
\[ = (m'_p \oplus m'_\chi)(\sigma_v^{-1} \cdot \sigma_1) \]

By composition of subst.
\[ = (m'_p \oplus m'_\chi) \sigma_1 \]

By composition of inverses.
\[ = m'_p \sigma_1 \]

By def. of \( m' \).

As for coloured nets, we can prove \( P(I(P, a)) = m'' \). Note that the isolated places of \( P \) (up-to constants) are exactly \( S_{K_m''} \).

- **Rule (step):** The proof of this case is analogous to that in proof of Lemma 4.2. The only difference is to require nets \( K_1 \) and \( K_2 \) to share only the names of constants, i.e. \( S_{K_1} \cap S_{K_2} = S_N \).

\( \square \)

Next Lemma states the correspondence among computations \( m_1 \rightarrow T_N \ast m_2 \) in a \( \mathcal{R} \mathbf{-P/T} \) \( N \) and reconfigurable processes.

**Lemma 5.2.** Let \( N = (S_N, C_N, T_N, \delta_{0N}, \delta_{1N}) \) be a \( \mathcal{R} \mathbf{-P/T} \) net. If \( m_1 \rightarrow T_N \ast m_2 \), then there exist a process (up-to constants) \( P : K \rightarrow \sigma N \), a compatible execution \( \rho : \chi \rightarrow \mathcal{P} \) of \( K \), and a colour assignment \( a \) s.t.
• \( m_1 = P\left(\text{pre}_c(P)\right) \ast \rho \oplus P(I(P, a)) \)

• \( m_2 = P\left(\text{post}_c(P)\right) \rho \oplus P(I(P, a)) \)

Proof. By induction on the length \( n \) of the derivation.

• **Base case** \( n = 0 \): This case follows as in proof of Lemma 4.3.

• **Inductive Step**: \( m_1 \rightarrow_{T_N} m_3 \rightarrow_{T_N}^* m_2 \). Since \( m_1 \rightarrow_{T_N} m_3 \), by Lemma 4.2, there exists a process \( P_1 \) with a compatible execution \( \rho_1 \) and a colour assignment \( a_1 \) s.t.

\[
- m_1 = P_1(\text{pre}_c(P_1)) \ast \rho_1 \oplus P_1(I(P_1, a_1))
- m_3 = P_1(\text{post}_c(P_1)) \rho_1 \oplus P_1(I(P_1, a_1))
\]

By inductive hypothesis applied on \( m_3 \rightarrow_{T_N}^* m_2 \), there exists a process \( P_2 \) with a compatible execution \( \rho_2 \) and a colour assignment \( a_2 \) s.t.

\[
- m_3 = P_2(\text{pre}_c(P_2)) \ast \rho_2 \oplus P_2(I(P_2, a_2))
- m_2 = P_2(\text{post}_c(P_2)) \rho_2 \oplus P_2(I(P_2, a_2))
\]

The main difference with the proof of Lemma 4.3 is that the condition on \( m_3 = P_1(\text{post}_c(P_1)) \rho_1 \oplus P_1(I(P_1, a_1)) = P_2(\text{pre}_c(P_2)) \ast \rho_2 \oplus P_2(I(P_2, a_2)) \) does not assure that the number of final places of \( P_1 \) is the same as the number of initial places of \( P_2 \) (in fact there are can be variable places in \( \text{post}_c(P_1) \)). Writing the coloured final multiset of \( P_1 \) as two different multiset, one where places are constant and other where places are variable, we have

\[
m_3 = P_1(\text{post}_c(P_1)|_{S_{K_1'}})\rho_1 \oplus P_1(\text{post}_c(P_1)|_{\neq})\rho_1 \oplus P_1(I(P_1, a_1))
\]

In order to be able to concatenate \( P_1 \) and \( P_2 \), the following condition should hold \( P_2(\text{pre}_c(P_2)) \ast \rho_2 \subseteq P_1(\text{post}_c(P_1)|_{S_{K_1'}})\rho_1 \oplus P_1(I(P_1, a_1)) \), i.e., \( P_1 \) should contain enough places to attach the minimal places of \( P_2 \). If this is not the case, define the net \( K_1' \) as an extension of \( K \) where some final variables are instantiated as places. First, select a minimal instantiation \( \iota \) s.t. \( \rho_1 = \iota \cup \rho_1' \) and

\[
P_2(\text{pre}_c(P_2)) \ast \rho_2 \subseteq P_1(\text{post}_c(P_1)|_{S_{K_1'}})\rho_1 \oplus (P_1(\text{post}_c(P_1)|_{\text{dom}(\iota)})\rho_1 \oplus P_1(I(P_1, a_1))
\]

Then, \( K_1' \) is defined in the following way

- \( S_{K_1'} = S_K \cup S_{\text{new}} \) s.t. \( |S_{\text{new}}| = |P_1(\text{post}_c(P_1)|_{\text{dom}(\iota)})\iota| \), i.e. the places that are instantiated by \( \iota \). Select also a mapping \( f_{S_{\text{new}}} \) s.t. \( F(S_{\text{new}}, P_1(\text{post}_c(P_1)|_{\text{dom}(\iota)})\iota, f_{S_{\text{new}}}) \).
- \( T_{K_1'} = T_{K_1} \)
\[ \delta_{IK^i}(t) = \delta_{IK}(t) \text{ for } i = 0, 1, \text{ if } r_n(t) \cap \text{dom}(\iota) = \emptyset \text{ (i.e., transitions not instantiated by } t \text{ remain unchanged), otherwise} \]
\[ \begin{align*}
\delta_{IK^i}(t) & = \delta_{IK}(t) \ast t \\
\delta_{IK^i}(t) & = (\delta_{IK}(t) \ast \delta_{IK}(t)|_{\text{dom}(\iota)})t \oplus n_{2i}
\end{align*} \]

where \( n_{2i} \) is a multiset s.t. \( \mathcal{g}(S_{K_{new}^i},(\delta_{IK}(t)|_{\text{dom}(\iota)})t, f_{S_{new}^i}S_{K_{new}^i}, n_{2i}) \) and \( \{S_{K_{new}^i}\}_{t \in T_K} \) form a partition of \( S_{new}^i \), i.e., the flow relation for transitions that are instantiated is changed in order to substitute instantiated variables by new places in the postset.

It is easy to notice that \( K_1' \) is still a causal net, the new flow relation neither adds transitions to places in \( K \) nor shares new places. Moreover, \( K_1' \) is still compatible, since instantiated transitions are compatible with \( \rho \), which is also a compatible execution of \( K_1' \). It is possible to defined a morphism \( f_1' \) from \( K_1' \) to \( N \) starting from \( P_1 \), as follow

\[ \begin{align*}
f_{1S}^1 & = f_{S1} \uplus f_{S_{new}^i} \\
f_{1T}^1 & = f_{T1} \\
\sigma_{1T}^1 & = \sigma_{IT} t \end{align*} \]

It is easy to show that \( f_1' \) defined above is a morphism from \( K_1' \) to \( N \). Note that \( \sigma_{IT} = \sigma_{IT} \) for all transitions that are note instantiated. For modified transitions, we have that

\[ f_{1S}^1(\ast t)| f_{1S}^1(t) = \]

By def. of \( K \)

\[ f_{1S}^1(\delta_{IK}(t) \ast t)| f_{1S}^1((\delta_{IK}(t) \ast \delta_{IK}(t)|_{\text{dom}(\iota)})t \oplus n_{2i}) = \]

By distributivity of \( f_{1S}^1 \)

\[ f_{1S}^1(\delta_{IK}(t) \ast t)| f_{1S}^1(\delta_{IK}(t))f_{1S}^1(t) \oplus f_{1S}^1(\delta_{IK}(t)|_{\text{dom}(\iota)})f_{1S}^1(t) \oplus f_{1S}^1(n_{2i}) = \]

Since \( P_1 \) is up-to constants, w.l.o.g. we assume \( f_{1S}(\iota) = t \)

\[ f_{1S}^1(\delta_{IK}(t) \ast t)| f_{1S}^1(\delta_{IK}(t))t \oplus f_{1S}^1(\delta_{IK}(t)|_{\text{dom}(\iota)})t \oplus f_{1S}^1(n_{2i}) = \]

By def. of \( f_{1S}^1 \)

\[ f_{1S}(\delta_{IK}(t)) t \oplus f_{1S}(\delta_{IK}(t)|_{\text{dom}(\iota)})t \oplus f_{1S}(n_{2i}) = \]

Since \( P_1 \) is a morphism

\[ \begin{align*}
(\ast f_{IT}(t) \ast \sigma_{IT}) t | f_{IT}(t) \bullet \sigma_{IT} t \odot ((f_{IT}(t) \ast \sigma_{IT})|_{\text{dom}(\iota)})t \oplus f_{1S}(n_{2i}) = \\
\text{By def. of } n_{2i} \text{ and Remark 4.2(2)}
\end{align*} \]

\[ \begin{align*}
(\ast f_{IT}(t) \ast \sigma_{IT}) t | f_{IT}(t) \bullet \sigma_{IT} t \odot ((f_{IT}(t) \ast \sigma_{IT})|_{\text{dom}(\iota)})t \oplus (\delta_{IK}(t)|_{\text{dom}(\iota)})t = \\
\text{By def. of } K_1
\end{align*} \]

By subtraction and addition of the same multiset \( (f_{IT}(t) \ast \sigma_{IT})|_{\text{dom}(\iota)})t \)

\[ \begin{align*}
(\ast f_{IT}(t) \ast \sigma_{IT}) t | f_{IT}(t) \bullet \sigma_{IT} t = \\
\text{By def. of composition of substitutions}
\end{align*} \]

\[ \begin{align*}
\ast f_{IT}(t) \ast (\sigma_{IT} t) & | f_{IT}(t) \bullet (\sigma_{IT} t) = \\
\ast f_{IT}(t) \ast \sigma_{IT} | f_{IT}(t) \bullet \sigma_{IT}
\end{align*} \]

By def. of \( f_{IT} \) and \( \sigma_{IT} \)
Before defining the union of both nets, we remove from the isolated places of \( P_2 \) those corresponding to final variables in \( K_1' \), that is we define \( K_2' \) by removing from \( S_{K_2} \) some isolated places of \( P_2 \) in order to satisfy the following equality

\[
P_1(\text{post}_c(P_1)|_{S_{K_1'}}) \rho_1 \oplus (P_1(\text{post}_c(P_1)|_{\text{dom}(\rho)})) \rho_1 \oplus P_1(I(P_1, a_1)) = P_2(\text{pre}_c(P_2)) \star \rho_2 \oplus P_2(I(P_2, a_2)|_{S_{K_2'}})
\]

Note that a process \( P'_2 : K'_2 \rightarrow N \) can be defined by restricting \( P_2 \) to the elements of \( K'_2 \). Then, the net \( K \) obtained as the union of \( K_1' \) and \( K'_2 \) can be defined as in proof of Lemma 4.3. The process \( P \) can be defined as the union of \( f'_1 \) and \( P'_2 \). \( P \) is a process because \( P'_2 \) is a process, and \( f'_1 \) contains minimal instantiations when considering the whole causal net \( K \).

Reasoning as in proof of Lemma 4.3, it can be shown that

\[
P(\text{pre}_c(P)) \star \rho \oplus P(I(P, a)) = m_1
\]

\[
P(\text{post}_c(P)) \rho \oplus P(I(P, a)) = m_2
\]

\[\square\]

The following Lemma assures that any reconfigurable process describes a computation in a R-P/T net.

**Lemma 5.3.** Let \( N = (S_N, C_N, T_N, \delta_{0N}, \delta_{1N}) \) be a R-P/T net and \( P : K \rightarrow_{\sigma} N \) a compatible process of \( N \). For every compatible execution \( \rho : X \rightarrow \mathcal{P} \) substituting all colour variables in \( \text{pre}_c(P) \), i.e. \( \text{col}_X(\text{pre}_c(P)) \subseteq \text{dom}(\rho) \), and for any colour assignment \( a \), then there exist \( m_1 \) and \( m_2 \) s.t.

- \( m_1 \rightarrow_{T_N}^* m_2 \)
- \( m_1 = P(\text{pre}_c(P)) \star \rho \oplus P(I(P, a)) \)
- \( m_2 = P(\text{post}_c(P)) \rho \oplus P(I(P, a)) \)

**Proof.** By induction on the number of transitions \( n = |T_K| \) (the proof is analogous to proof of Lemma 4.4).

\[\square\]

Finally, next theorem states the relation between reconfigurable processes and step sequences of a R-P/T net, assuring that for any processes there exists a computation in the net, and vice versa.

**Theorem 5.4.** Let \( N = (S_N, C_N, T_N, \delta_{0N}, \delta_{1N}) \) be a R-P/T net. Then, \( m_1 \rightarrow_{T_N}^* m_2 \) iff there exists a process \( P : K \rightarrow_{\sigma} N \), a compatible execution \( \rho : X \rightarrow \mathcal{P} \) substituting all colour variables in \( \text{pre}_c(P) \), i.e. \( \text{col}_X(\text{pre}_c(P)) \subseteq \text{dom}(\rho) \), and a colour assignment \( a \), s.t.

- \( m_1 = P(\text{pre}_c(P)) \star \rho \oplus P(I(P, a)) \)
- \( m_2 = P(\text{post}_c(P)) \rho \oplus P(I(P, a)) \)

**Proof.** Immediate from Lemmata 5.2 and 5.3.

\[\square\]
5.4 Reconfigurable ZS nets

Since reconfigurable nets do not distinguish places from colours, reconfigurable zero-safe nets should consider both types of colours, i.e. stable and zero-safe. Hence, the distinction between stable and zero markings should take into account the types of the colours carried by tokens. Actually, a stable marking should contain only stable names, and hence, we require a multiset \( s \) denoting a stable marking to consist of tokens in stable places coloured with stable names, i.e. \( s \in \mathcal{M}_{L,L} \). (We recall that \( L \) is the set of stable places, \( Z \) are the zero-safe places, and \( S = L \cup Z \)). At a first glance, it could appear that any other marking denotes a zero marking. This is not the case for a non-empty marking \( m \in \mathcal{M}_{L,Z} \), in which tokens in stable places are coloured with zero names. Markings of this kind are somehow contrary to the ZS approach. In fact, tokens in stable places produced by a transaction are released only at commit, when all zero tokens have been consumed. Consequently, we will restrict zero markings to \( z \in \mathcal{M}_{Z,S} \). We denote the set of well-defined markings as \( \mathcal{W}_{L,Z} = \mathcal{M}_{L,L} \cup \mathcal{M}_{Z,LZ} \). Additionally, we consider the set of variable names \( \mathcal{X} \) as partitioned in two sets: \( \mathcal{X}_L \), the set of stable variables \( X, Y, \ldots \), and \( \mathcal{X}_Z \) the set of zero variables \( x, y, \ldots \).

**Definition 5.15** (r-ZS net). A **reconfigurable ZS net** is a 6-tuple \( B = (S_B, T_B, \delta_{0B}, \delta_{1B}, m_{0B}, Z_B) \) where \( N_B = (S_B, T_B, \delta_{0B}, \delta_{1B}, m_{0B}) \) is the underlying r-P/T net and the set \( Z_B \subseteq S_B \) is the set of zero places. The places in \( S_B \setminus Z_B \) (denoted by \( L_B \)) are called stable places. A **stable marking** \( m \) is a coloured multiset of stable places (i.e., \( m \in \mathcal{M}_{L_B,L_B} \)), and the initial marking \( m_{0B} \) must be stable. Moreover, we impose the pre- and postset functions to be defined over well-defined markings, i.e., \( \forall t \in T_B : \delta_{iN}(t) \in \mathcal{W}_{L_B \cup \mathcal{X}_L, Z_B \cup \mathcal{X}_Z} \), for \( i = 0, 1 \).

We require transitions to be fired with appropriate names, that is, zero variables are substituted by zero names and stable variables by stable names.

**Definition 5.16** (Type preserving substitution). A substitution \( \sigma \) is **type preserving** if \( \forall X \in \mathcal{X}_L : \sigma(X) \in (L_B \cup \mathcal{X}_L) \) and \( \forall x \in \mathcal{X}_Z : \sigma(x) \in (Z_B \cup \mathcal{X}_Z) \).

In what follows we assume all substitutions to preserve types. Then, the firing rule for r-ZS nets can be written as follows, while rules (STEP), (CONCATENATION) and (CLOSE) remain unchanged (see Figure 3.1).

\[
\begin{align*}
\text{RECONF-FIRING} & \quad \text{t} = s \oplus z \mid s' \oplus z' \in T \quad s'' \in \mathcal{M}_{L,L} \quad z'' \in \mathcal{M}_{Z,S} \quad \text{rn}(t) \subseteq \text{dom}(\sigma) \quad \text{range}(\sigma) \subseteq S \\
& \quad (s \oplus \sigma \oplus s'', z \oplus \sigma \oplus z'') \rightarrow_T (s' \oplus \sigma \oplus s'', z' \oplus \sigma \oplus z'')
\end{align*}
\]

**Example 5.4** (Mailing list). Consider a data structure that allows to send atomically a message to a list of subscribers (in the sense that the message is either sent to all or to none). Figure 5.7 shows a ZS net corresponding to such structure. Name
nil] is a stable constant colour, all other colours used for labelling arcs are stable variables.

Tokens in the stable place newSubs correspond to the agents that want to be subscribed to the list. Their colours are the places in which they expect to receive a new message. Place top contains the element on top of the list (the latest subscriber). We assume that an empty list is denoted with a token in the place top coloured with the constant colour nil. A list is encoded with several tokens in place subscList. Each token in subscList is coloured with a pair: the first colour carries the information of a subscriber, while the second has the identity of the next subscriber in the list.

A new subscriber N is added on top of the list by firing the transition add. The token in top corresponding to the current top of the list (whose colour is T) is replaced with a new token of colour N, i.e., the new subscriber becomes the top of the list. Also, a new token is produced in subscList whose colour is (N,T), meaning that the subscriber that follows N in the list is the previous top of the list, namely T.

Transition tell allows to send a message M to every subscriber in the list. By firing tell, a new transaction is initiated, i.e. a new token is generated in the zero place sending. Additionally, the top of the list is consumed, and a new token with the same colour is produced in top, which will be released only when the transaction finishes. Consequently, transitions add and tell (on that list) will not be enabled until the current transaction finishes.

A token present in the zero place sending contains the information of the current subscriber to be notified (i.e., the first colour of the pair), and the message to be sent (i.e., the second colour). Transition notify is a reconfigurable transition. In fact, it consumes from sending the token (T,M) and sends M to the subscriber T (nevertheless this token will be available actually when the transaction finishes). Additionally, notify takes from subscList the subscriber F that follows T in the list, and update the state of the transaction by putting in the zero place sending a token (F,M), meaning that M has to be sent to next F.

The transaction finishes when the end of the list is reached. That is, when the token in sending is addressed to the receiver nil. At this point, the transition end is fired, which will consume the zero token present in the net. This will release all stable tokens produced during the transaction, i.e., all subscribers atomically will receive the message M, and the top of the list will be available for executing new activities.

5.5 Abstract Semantics

The definition for connected transactions is analogous to Definition 4.21 but considers isolated places taken up-to constants colours.
5.5. ABSTRACT SEMANTICS

\[ \text{newSubs} \rightarrow \text{top} \rightarrow \text{message} \]
\[ \text{subscList} \rightarrow \text{add} \rightarrow \text{tell} \rightarrow \text{notify} \rightarrow \text{end} \]
\[ \text{nil} \rightarrow \text{sending} \]

\textbf{Figure 5.7: A r-zs net for the mailing list example.}

\textbf{Definition 5.17} (Reconfigurable connected transaction). Given a r-zs net $B$, and a coloured deterministic process $P$ of the underlying C-p/t net $N_B$. The equivalence class $\xi = \left[ P \right]_{\equiv}^B$ of all processes isomorphic to $P$ (see Definition 2.6) is a \textit{reconfigurable connected transaction} of $B$ if:

1. $\text{pre}_c(P)$ and $\text{post}_c(P)$ are stable markings, i.e., the process starts by consuming stable tokens and produces only stable tokens;

2. $E_P \subseteq Z_B$, i.e. the evolution places of the process (see Notation 2.1) are zero places, this means that stable tokens produced during the transaction cannot be consumed during in the same transaction;

3. $P$ is connected (see Definition 2.8); and

4. $P$ (considered up-to constants) is full, i.e., it does not contain idle (i.e., isolated) places other than constants ($I(P) = \emptyset$).

We write $\text{pre}(\xi)$ and $\text{post}(\xi)$ for $P(\text{pre}_c(P))$ and $P(\text{post}_c(P))$ respectively.

\textbf{Example 5.5.} Figure 5.8 shows the causal net corresponding to the connected transaction $\xi_1$ for the r-zs net of the mailing list (Figure 5.7). The depicted morphism maps places and transitions of the causal net to the homonymous elements of the r-zs net in Figure 5.7. The substitutions $\sigma_1$ of the morphisms are shown next to the corresponding transitions. Note that

\[
\begin{align*}
\text{pre}(\xi_1) &= \text{top}(T) \oplus \text{message}(M) \oplus \text{subscList}(T', F') \oplus \text{subscList}(T'', F'') \\
\text{post}(\xi_1) &= \text{top}(T) \oplus \text{subscList}(T', F') \oplus \text{subscList}(T'', F'') \oplus T'(M') \oplus T''(M'')
\end{align*}
\]
Figure 5.8: A reconfigurable connected transaction $\xi_1$ for the mailing list example.

Then, by taking the mgce $\rho = \{T/T', M/M', F'/T'', M/M'', \text{nil}/F'', M/M''''\}$, we have

$$\begin{align*}
\text{pre}(\xi_1) \ast \rho &= \text{top}(T) \oplus \text{subscList}(T, F') \oplus \text{subscList}(F', \text{nil}) \oplus \text{message}(M) \\
\text{post}(\xi_1) \rho &= \text{top}(T) \oplus \text{subscList}(T, F') \oplus \text{subscList}(F', \text{nil}) \oplus T(M) \oplus F'(M)
\end{align*}$$

From the above equalities, it is clear that the computation described by $\xi_1$ is a transaction that atomically sends the message $M$ to a mailing list, that has two subscribers, namely $T$ and $F'$. (We do not draw the place nil, which is used as a constant colour, since we are considering processes up-to constants). Moreover note that the transaction is a transition of the reconfigurable net because of the variables $T''$ and $T'''$, which appear as places.

Similarly to coloured nets, the correspondence results are still valid for reconfigurable nets. The proofs follow similarly to proofs in Section 4.3 and proceed as in Lemma 5.2 when concatenating processes.

**Theorem 5.5.** Let $B$ be a r-ZS net and $\xi$ a connected transaction of $B$, for any compatible execution $\rho : X \rightarrow P$ substituting all colour variables in $\text{pre}(\xi)$ (i.e. $\text{col}_X(\text{pre}(\xi)) \subseteq \text{dom}(\rho)$), $\text{pre}(\xi) \ast \rho \Rightarrow_{T_B} \text{post}(\xi) \rho$. 
Figure 5.9: A partial view of the abstract net for the mailing list example.

**Theorem 5.6.** Let $B$ be a r-zs net. If $m \Rightarrow_T m'$, then there exists a set of connected transactions $\{\xi_i\}$, and a substitution $\rho$ that is a compatible execution of all $\xi_i$ s.t.

- $m = \bigoplus_i \text{pre}(\xi_i) \ast \rho \oplus m''$ and $m' = \bigoplus_i \text{post}(\xi_i) \rho \oplus m''$.

As for the coloured case, the reconfigurable causal abstract net of a r-zs net is a r-p/T net defined in terms of the connected transactions and their mgce.

**Definition 5.18** (Reconfigurable causal abstract net). Let $B = (S_B, T_B, \delta_{0B}, \delta_{1B}, m_{0B}, Z_B)$ be a r-zs net. The reconfigurable causal abstract net of $B$ is defined as $I_B = (S_B \setminus Z_B, C_B, \Xi_B, \delta_{0i}, \delta_{1i}, m_{0B})$, with $\delta_{0i}(\xi) = \text{pre}(\xi) \ast \sigma_\xi$ and $\delta_{0i}(\xi) = \text{post}(\xi) \sigma_\xi$, where $\sigma_\xi$ is the mgce for $\xi$. (We recall that $\Xi_B$ is the set of all connected transactions of $B$).

**Example 5.6** (Abstract net for the mailing list example). In Figure 5.9 we (partially) show the abstract net corresponding to the r-zs net in Figure 5.7. Transition add is identical to transition add in Figure 5.7. The transition tell n sends atomically a message $M$ to $n$ subscribers in the list whose top is $N_1$ and finishes in nil. It is worth noting that there is one and only one transition tell n for any $n \geq 1$, which is parametric w.r.t. the message to be sent and the subscribers in the list. The transition drop handles the case in which the list is empty. In such situations the message sent is simply lost (consumed).

Also for the reconfigurable case, the following theorem assures the correspondence between the abstract and the concrete view.

**Theorem 5.7.** Let $B$ be a r-zs net and $I_B$ its abstract net. Then $m \Rightarrow_{T_B} m'$ iff $m \Rightarrow_{T_B} m'$.

*Proof.* The proof follows as in Theorem 4.10. \qed
5.6 C-ZS nets as R-ZS nets

C-ZS nets are a particular case of R-ZS net, where no transition uses received names as places in its postset. Consequently, a coloured net can be seen as a reconfigurable net, where constant colours are places.

**Definition 5.19** (Reconfigurable version of a C-P/T net). Let \( N = (S_N, C_N, T_N, \delta_{0N}, \delta_{1N}, m_{0N}) \) be a C-P/T net. The reconfigurable version of \( N \) is the R-P/T net \( R_N = (S_N \cup (C_N \cap P), T_N, \delta_{0N}, \delta_{1N}, m_{0N}) \). The r-ZS net \( R_B \) is the coloured version of the zs net \( B \) if the underlying R-P/T net \( N_{R_B} \) is the reconfigurable version of the underlying C-P/T net \( N_B \) of \( B \), and \( Z_B = Z_{C_B} \).

Thus, given a C-ZS net \( B \), it is possible to construct its abstract C-P/T net \( C_B \) and its abstract R-P/T net \( R_B \). The following result assures that both constructions are isomorphic, and makes the diagram in Figure 5.10 to commute.

**Proposition 5.8.** Let \( N \) be a coloured net. If \( P \) is a coloured process of \( N \), then \( P \) is a reconfigurable process of \( R_N \).

**Proof.** Since \( P \) is a coloured process, all substitutions in the morphism are renamings (i.e., proper instantiations are empty for all transitions), and hence, \( P \) is a reconfigurable process of \( R_N \). \( \square \)

**Theorem 5.9.** Let \( B \) be a C-ZS net, \( I_B \) its coloured abstract net, \( R_B \) and \( R_{I_B} \) their reconfigurable versions, and \( I_{C_B} \) the abstract net of \( R_B \) (i.e., of the reconfigurable version of \( B \)). Then \( C_{I_B} \approx I_{C_B} \).

**Proof.** The proof follows from Proposition 5.8. First, by noting that equivalence classes are the same under both views. Moreover, the causal net associated to each process is the same under both views. Hence, the mgc for each connected transaction is the same. \( \square \)

The Theorem above guarantees that both the concrete and abstract views of coloured (and consequently ordinary) zs nets are fully preserved when adding reconfigurable capabilities.
Chapter 6

Dynamic ZS NETS

While in reconfigurable nets the sets of states and transitions remain unchanged during computations, dynamic nets can create new components while executing: new places and transitions may be added to the net when a transition is fired. Nevertheless, it is not possible to modify existing transitions: they always consume tokens from a fixed multiset of places and the postset is always the same expression (multiset of places or nets) parametric on the received values. Moreover, after a place is created, no new transitions can fetch tokens from it (i.e., there is no input capability). We remark that reconfigurable nets are a special case of dynamic nets, whose transitions do not add new components.

6.1 Background: Dynamic nets

Different formulations for dynamic nets have been proposed in literature [4, 31]. The definition we give here follows the presentation in [4].

As for reconfigurable nets, we consider an infinite set of place names $\mathcal{P}$, and infinite set of variable names $\mathcal{X}$, ranged over by $x, y, \ldots$. We require also variable names to be different from place names, i.e., $\mathcal{X} \cap \mathcal{P} = \emptyset$. We remark that the colours carried on by tokens are the names of places. We will use $\mathcal{C} = \mathcal{P} \cup \mathcal{X}$ to refer both places and variable names.

**Definition 6.1 (DN).** The set $\text{DN}$ is the least set satisfying the following equation:

\[
\mathcal{N} = \{(S_N, T_N, \delta_0, \delta_1, m_0) \mid S_N \subseteq \mathcal{P} \land \delta_0 : T_N \rightarrow \mathcal{M}_{S_N, c} \land \delta_1 : T_N \rightarrow \mathcal{N} \land m_0 \in \mathcal{M}_{T, c}\}
\]

Note that $\mathcal{N}$ is a domain equation [62] defining the recursive type of dynamic nets. The simplest elements in $\mathcal{N}$ are markings, i.e. the tuples $(S, \emptyset, \emptyset, \emptyset, m)$ with $m \in \mathcal{M}_{T, c}$. Then, nets are defined recursively, since the postset of any transition is another element of $\mathcal{N}$. The set $\text{DN}$ is defined as the least fixed point of $\mathcal{N}$.

For $(S_N, T_N, \delta_0, \delta_1, m_0) \in \text{DN}$, $S_N$ is the set of places, $T_N$ is the set of the transitions, $\delta_0$ and $\delta_1$ are the functions assigning the pre- and postset to every
transition, and \( m_{0N} \) is the initial marking. As usual, we denote \( S_N \cup T_N \) by \( N \), and omit subscript \( N \) whenever no confusion arises. Moreover, we abbreviate a transition \( t \in T \) such that \( \delta_0(t) = m \) and \( \delta_1(t) = N \) as \( m \vdash N \), and refer to \( m \) as the preset of \( t \) (written \( \mathbf{t} \)) and \( N \) as the postset of \( t \) (written \( \mathbf{t}^* \)).

Note that, differently from previous models, the initial marking \( m_{0N} \) is not required to be a multiset over the places of the net, i.e., the initial marking can put tokens in places that are not defined by the net. In fact, the initial marking \( m_{0N} \) is a multiset over \( \mathcal{P} \) (i.e., \( m_{0N} \in \mathcal{M}_{\mathcal{P}C} \)) and not over the places of the net \( S_N \) (\( \mathcal{M}_{SNC} \)). A trivial example is to consider a coloured transition \( a(x)|\emptyset b(\bullet) \), which is written: \( a(x)|(\emptyset, \emptyset, \emptyset, \emptyset, b(\bullet)) \), where \( b \) clearly does not belong to the places the subnet \( (\emptyset, \emptyset, \emptyset, \emptyset, b(\bullet)) \). We usually abbreviate transitions as the previous one, where the postset does not define new places, by writing just the initial marking fixed by the postset, i.e. \( a(\bullet)|\emptyset b(\bullet) \).

Names defined in \( S_N \) act as binders on \( N \). Therefore, nets are considered up-to \( \alpha \)-conversion on \( S_N \). For instance, the nets \((\{a\}, \emptyset, \emptyset, a(\bullet)) \) and \((\{b\}, \emptyset, \emptyset, b(\bullet)) \) are \( \alpha \)-equivalent, while \((\emptyset, \emptyset, \emptyset, a(\bullet)) \) and \((\emptyset, \emptyset, \emptyset, b(\bullet)) \) are not, since \( a \) and \( b \) are free names. Analogously, the names of transitions in \( T_N \) are binders, and hence we consider nets up-to \( \alpha \)-conversion on transition names. In particular, the creation of a new subnet \( N_1 \) in \( N \) means the addition of a fresh net \( N'_1 \) s.t. \( N'_1 \) is \( \alpha \)-equivalent to \( N_1 \) and all names in \( S_{N'_1} \) and \( T_{N'_1} \) are guaranteed to be different from any place and transition in \( N \) (i.e., they are fresh names).

**Example 6.1** (A simple dynamic net). Consider the net \( N \) represented in Figure 6.1(a). The double-lined arrow indicates the dynamic transition \( t = a(\bullet)|\emptyset N_1 \), which creates an instance of the subnet \( N_1 \) when fired. In order to distinguish graphically names of places defined by a net from those that are free names, we write \( (\nu a) \) to denote that \( a \) is a bound name of the depicted net. We allow the initial marking of \( N_1 \) and the postset of transitions in \( T_{N_1} \) to generate tokens in \( a \). Therefore, the following is a valid definition for \( N_1 \): \( S_{N_1} = \{d\} \), \( T_{N_1} = \{t_1\} \), \( m_1 = a(\bullet) \oplus d(\bullet) \) and \( t = d(\bullet)|a(\bullet) \). A firing of \( t \) will lead to (a net isomorphic to) the net shown in 6.1(b). A new place \( d \) and a transition \( t_1 \) (whose pre- and postset are \( d(\bullet) \) and \( a(\bullet) \), resp.) have been added to the net. Also two tokens have been added: one in \( a \) and the other in \( d \), accordingly to the initial marking of \( N_1 \). In this marking \( t \) is enabled and can be fired again. The intended meaning of the new activation of \( t \) is to create a new subnet: a new place and a new transition whose names are different from others already present in the net (Figure 6.1(c)).

As in the previous models, the received names \( rn(t) \) of a transition \( t = m \vdash N \) denotes the set of variables occurring as colours in the preset of \( t \), i.e. \( rn(t) = col_X(m) \). In addition, given a dynamic net or a dynamic transition, we will talk about defined and free names.

**Definition 6.2** (Defined and Free names). The set of **defined names of a marking** \( m \) is \( dn(m) = \{a| (a, c) \in m \} \), i.e. names appearing in place position. Given
\[
N = (S_N, T_N, \delta_0, \delta_1, m_0) \in \text{DN}, \text{ the set of defined (dn) and free (fn) names of transitions, sets of transitions, and nets are defined as follow:}
\]

\[
dn(m_1 \upharpoonright N_1) = dn(m_1)
\]
\[
dn(T_N) = \bigcup_{t \in T_N} dn(t)
\]
\[
dn(N) = S_N
\]
\[
fn(t \leftarrow m_1 \upharpoonright N_1) = dn(m_1) \cup \text{col}_P(m_1) \cup (fn(N_1) \setminus rn(t))
\]
\[
fn(T_N) = \bigcup_{t \in T_N} fn(t) \setminus dn(T_N)
\]
\[
fn(N) = fn(T_N) \setminus S_N
\]

Note that \(dn(T_N) \subseteq dn(N)\), since places on the preset of transitions are places defined by the net.

**Definition 6.3 (Dynamic Net).** \(N \in \text{dn}\) is a dynamic net if \(fn(N) = \emptyset\).

The definition above states that a dynamic net is closed, i.e. it does not generate tokens in places that do not belong to it. The condition \(fn(N) = \emptyset\) assures tokens to be generated always in places of the net, since markings are bound to places defined in the net, which are guaranteed to be different from places in other nets.

The net presented in Example 6.1 is closed, and hence dynamic, even though \(N_1\) is not. In fact, the postset of transition \(t_1\) contains the place \(a\) which is not a defined name of \(N_1\).

**Remark 6.1.** As variables in a transition are used to describe parameters, we consider only coloured nets whose transitions \(t = \leftarrow m \upharpoonright N\) satisfy the condition \((fn(N) \cap \mathcal{X}) \subseteq rn(t)\). This restriction requires all variables occurring in the postset of a transition to be bound to some variable in the preset. Note that this is always the case when a net is closed.

As for coloured and reconfigurable nets, the firing of a transition \(t\) requires the postset to be instantiated with the received colours of \(t\), i.e., the parameters \(rn(t)\) of the transition \(t\). Hence, we need a suitable notion of substitution on nets.
Definition 6.4 (Instantiation of transitions and nets). Let $\sigma : \mathcal{X} \to \mathcal{P} \cup \mathcal{X}$ be a substitution. The instantiation of a transition $t = m_1 \parallel N_1$ with $\sigma$ is the transition $t\sigma = m_1 \parallel \sigma(N_1\sigma)$.

Given a dynamic net $N = (S_N, T_N, \delta_{0N}, \delta_{1N}, m_{0N})$, the instantiation of $N$ with $\sigma$ s.t. $(\text{dom}(\sigma) \cup \text{range}(\sigma)) \cap S_N = \emptyset$ is defined as $N\sigma = (S_N, T_N, \delta_{0N}, \delta_{1N}\sigma, m_{0N}\sigma)$, where $\delta_{1N}\sigma(t) = (\delta_{1N}(t))\sigma$ if $\text{rn}(t) \cap (\text{dom}(\sigma) \cup \text{range}(\sigma)) = \emptyset$.

Remark 6.2. (i) The recursive definition given above is well-founded because it is recursive on the structure of a net $N \in \text{dn}$, which is well-founded. (ii) The side condition required on the domain and range of the substitution used to instantiate a net (or a transition) avoids name clashes. When such condition is not satisfied, an $\alpha$-conversion on the places of the net (or on the received names of the transition) can be applied before instantiation.

Definition 6.5 ($\alpha$-equivalence of transitions). Two transitions $t_1$ and $t_2$ are $\alpha$-convertible if there exists an injective substitution $\sigma : \mathcal{X} \to \mathcal{X}$, where $\text{rn}(t_1) \subseteq \text{dom}(\sigma)$, such that $t_1\sigma = t_2$. As usual, $\alpha$-conversion is an equivalence relation, which is denoted by $\equiv_\alpha$. We shall always consider transitions up-to $\alpha$-equivalence.

Definition 6.6 (Composition of nets). Let $N_1$ and $N_2$ be dynamic nets s.t. $N_1 \cap N_2 = \emptyset$ and $\text{fn}(N_1) \cap S_{N_2} = \emptyset$. Then, the addition of $N_2$ to $N_1$ (written $N_1 \oplus N_2$) is defined as $N_1 \oplus N_2 = (S_{N_1} \cup S_{N_2}, T_{N_1} \cup T_{N_2}, \delta_{0N_1} \cup \delta_{0N_2}, \delta_{1N_1} \cup \delta_{1N_2}, m_{0N_1} \oplus m_{0N_2})$. The addition $N_1 \oplus N_2$ is said the parallel composition of $N_1$ and $N_2$ (written $N_1 \oplus N_2$) if also $\text{fn}(N_2) \cap S_{N_1} = \emptyset$.

Observe that the side conditions required by parallel composition avoid free names of one net to be captured by the transitions defined in the other. Nevertheless, when a subnet $N_2$ is added to a net $N_1$ ($N_1 \oplus N_2$) we allow the free names of $N_2$ to be captured by the definitions in $N_1$. We remind that we are considering nets up-to $\alpha$-conversion on the name of places. Hence, it is always possible to choose $N'_2$ $\alpha$-equivalent to $N_2$ s.t. $N_1 \cap N_2 = \emptyset$. 

Figure 6.2: Operational semantics of dynamic nets.
In order to provide the operational semantics for dynamic nets, we remark that the state of a net is not given just in terms of the markings, but also in the structure of the net. The operational semantics is presented in Figure 6.2. For simplicity we write \((S, T, m)\) as a shorthand for \((S, T, \delta_0, \delta_1, m)\). Rule \((\text{DYN-FIRING})\) stands for the firing of \(t\) when the marking contains an instance of the preset of \(t\) (for a suitable substitution on colours \(\sigma\)). The resulting net consists of the original net, where the consumed tokens have been removed, and a new instance of \(N_1\) (i.e., the postset of \(t\)). Note that the composition \(\otimes\) of nets assures that the names of the added components are fresh. Rule \((\text{DYN-STEP})\) stands for the parallel composition of computations when the initial marking contains enough tokens to execute them independently. Note that both concurrent steps operate over the same net structure, in fact both start from a net whose places and transitions are \(S\) and \(T\). Those steps can add new elements (i.e., \(N_1\) and \(N_2\)), which by definition are fresh. Moreover, the new components can be chosen to assure new elements to be disjoint, i.e. such that \((N_1 \oplus N_2)\) is defined.

It is worth noting that reconfigurable nets are a particular case of dynamic nets. In fact, when \(t\) is a reconfigurable transition, i.e. \(N_1 = (\emptyset, \emptyset, m_1)\), the expression \((S, T, m) \otimes N_1 = (S, T, m \oplus m_1)\) corresponds to the rule \((\text{RECONF-FIRING})\) given in Section 4.1.1.

**Example 6.2.** Consider the dynamic net presented in Example 6.1 (see Figure 6.1(a)) but with the initial marking \(m' = a(\bullet) \oplus a(\bullet)\). In what follows we show a computation that fires concurrently two instances of \(t\). We have

\[
\begin{align*}
(a(\bullet))^{N_1} &\rightarrow_{(\text{DF})} (a(\bullet))^N_1 \\
\{(a), \{t\}, a(\bullet)) &\rightarrow (\{(a), \{t\}, \emptyset) \oplus N_1' \\
\{(a), \{t\}, a(\bullet) \oplus a(\bullet)) &\rightarrow (\{(a), \{t\}, \emptyset) \oplus (N_1' \oplus N_1'')
\end{align*}
\]

where \(N_1' = (\{d\}, \{t_1\}, a(\bullet) \oplus d(\bullet))\) and \(N_1'' = (\{d'\}, \{t_1'\}, a(\bullet) \oplus d'(\bullet))\). Note that \(N_1' \oplus N_1'' = (\{d, d'\}, \{t_1, t_1'\}, a(\bullet) \oplus d(\bullet) \oplus a(\bullet) \oplus d'(\bullet))\).

And finally,

\[
\{(a), \{t\}, \emptyset) \oplus (N_1' \oplus N_1'') = (\{a, d, d'\}, \{t,t_1, t_1'\}, a(\bullet) \oplus d(\bullet) \oplus a(\bullet) \oplus d'(\bullet)).
\]

**Processes for Dynamic nets.** A process, viewed as morphism from a causal net into a \(P/T\) net, identifies elements of the causal net as particular instances of elements in \(P/T\) net. For \(P/T\), c\(-P/T\), and r\(-P/T\) nets, where the elements of the net are fixed, the correspondence between instances and general elements is quite clear. In particular, states are mapped into states and transitions (instances of some patterns) to transitions (representing general patterns). Instead, when describing the execution of a dynamic net \(D\) it could be necessary to talk about states and transitions that are not present in \(D\) (although \(D\) describes how to create them). Consider the net \(N\) introduced in Example 6.1, which has one place
6.2 Flat Dynamic zs nets

The evolving structure of dynamic nets opens several possibilities when applying the zs approach. The more obvious option is to provide transactions by allowing any net to define stable and zero places, as done for the other kind of nets. Nevertheless, other options can take advantage of the possibility of creating subnets to specify sub-activities that should be executed atomically, providing in this way a hierarchy of atomic activities, i.e., nested transactions. In the rest of this section we describe the operational semantics for the first case, called flat dynamic zs nets. We explore nested transaction in a mobile calculus in the Part II, where we present an extension of the Join calculus with nested transactions.

As mentioned above, flat dynamic zs nets correspond to a direct application of the zs approach where the places of a net $B$ are either stable, i.e. in $L_B = S_B \setminus Z_B$, or zero, i.e., in $Z_B$. As for reconfigurable nets, we rely on two disjoint set of variables: $X_{L_B}$, ranged over by $X, Y, \ldots$ for stable variables, and $X_{Z_B}$ for zero variables $x, y, \ldots$. Similarly, we use $s \in M_{L,L}$ for denoting a stable marking, and $z \in M_{Z,S}$ for zero markings. Moreover $W_{L,Z} = M_{L,L} \cup M_{Z,L \cup Z}$ stands for the set of well-defined markings.

As done for dynamic nets, we give a domain equation for the set of dynamic zs nets.

**Definition 6.7 (dzn).** The set $\text{dzn}$ is the least set satisfying the following equation:

$$
\begin{align*}
B &= \{ (S_B, T_B, \delta_{0B}, \delta_{1B}, m_{0B}, Z_B) \mid \\
S_B &\subseteq \mathcal{P} \land \delta_{0B} : T_B \rightarrow M_{S_B,C} \land \delta_{1B} : T_B \rightarrow B \land m_{0B} \in M_{P,C} \land Z_B \subseteq S_B \} \\
\end{align*}
$$

The received, defined and free names of the elements in dzn is analogous to the case of dynamic nets (see Definition 6.2). Additional, we require every well-defined flat dynamic zs net to be closed and to have a stable initial marking.

**Definition 6.8 (Flat d-zs net).** $B = (S_B, T_B, \delta_{0B}, \delta_{1B}, m_{0B}, Z_B) \in \text{dzn}$ is a dynamic zero-safe net (d-zs net) if $fn(B) = \emptyset$. The places in $S_B \setminus Z_B$ (denoted by $L_B$) are called stable places. A stable marking $m$ is a coloured multiset of stable places (i.e., $m \in M_{L_B, L_B}$), and the initial marking $m_{0B}$ must be stable. Moreover, we impose the pre- and postset functions to be defined over well-defined markings.
Figure 6.3: Operational semantics of flat dynamic sz nets.

We extend the definition of composition of nets (operators $\circ$ and $\oplus$) to consider d-zs net.

**Definition 6.9** (Composition of d-zs nets). Let $B_1$ and $B_2$ be dynamic nets s.t. $B_1 \cap B_2 = \emptyset$ and $fn(B_1) \cap S_{B_2} = \emptyset$. Then, the **addition** of $B_2$ to $B_1$ (written $B_1 \oplus B_2$) is defined as $B_1 \oplus B_2 = (S_{B_1} \uplus S_{B_2}, T_{B_1} \uplus T_{B_2}, \delta_{0B_1} \uplus \delta_{0B_2}, \delta_{1B_1} \uplus \delta_{1B_2}, m_{0B_1} \oplus m_{0B_2}, Z_{B_1} \uplus Z_{B_2})$. The addition $B_1 \oplus B_2$ is said the **parallel composition** of $B_1$ and $B_2$ (written $B_1 \oplus B_2$) if also $fn(B_2) \cap S_{B_1} = \emptyset$.

Rules in Figure 6.3 shows the operational semantics of flat d-zs nets. The rules are the straightforward extension of those corresponding to r-zs nets. The states of the nets are given by tuples $(S, T, m, Z)$ that represent the structure and the marking of the net. As usual, we write a marking $m = s \oplus z = (s, z)$ when $s$ is a stable marking and $z$ is zero-safe. Rules (DYN-FIRING) and (DYN-STEP) are analogous to those of dynamic nets, shown in Figure 6.2. Rule (DYN-CONCATENATION) allows the sequential composition of computations when the stable tokens produced by the first computation (i.e., $s'_j$) are not used by the second. Finally, rule (DYN-STEP) states that the atomic movement of the net are the computations that start and end without zero-safe tokens.

**Example 6.3** (Private Mailing Lists). Consider the mailing list problem presented in Example 5.4. Suppose now there are $n$ users $u_n$, each of them wants to send atomically messages present in $m_4$ to listeners whose names are in $s_3$. (Every user has its own list of subscribers and messages). The system can be modelled as a flat dynamic zs net by reusing the mailing list structure in Figure 5.7. Consider the dynamic net in Figure 6.4 for the case of two users. The net $N_1$ (appearing in the
postset of \( t = \text{new}(X,Y)|N_1 \) corresponds exactly to the net in Figure 5.7 plus the initial marking \( m_{0N_1} = X(\text{newSubs}) \oplus Y(\text{message}) \oplus \text{top}(\text{nil}) \).

There are \( n \) transitions \( \text{subsc}_1 \) and \( \text{dist}_1 \), i.e., a pair for each user \( u_i \). The listeners for the user \( u_i \) are in place \( s_i \), while the messages are in \( m_i \). For instance, the listeners for \( u_1 \) are \( j_1 \) and \( j_2 \), while the message to be sent to them is \( l_1 \). The user \( u_1 \) can create a list by putting a token on \( \text{new} \) with colour \( p_1 = (a_{s_1}, d_{m_1}) \) (see the initial marking of the net in Figure 6.4). The net obtained after firing \( \text{new} \) with colours \( (a_{s_1}, d_{m_1}) \) is depicted in Figure 6.5. A new instance of \( N_1 \) has been added to the net. For convenience in the graphical representation, we rename local places \( \text{newSubs} \) with \( 1_s \) and \( \text{message} \) with \( 1_m \) because they are used to colour tokens. Also the tokens corresponding to the initial marking of \( N_1 \) are produced: token with colour \( \text{nil} \) in place \( \text{top} \), token \( 1_s \) in \( a_{s_1} \), and \( 1_m \) in \( d_{m_1} \). After the mailing list is created, the listeners in \( s_1 \) can be added to the list by firing the reconfigurable transition \( \text{subsc}_1 \). Note that any coloured token \( S \) in \( s_1 \) is forwarded to the place \( 1_s \), which will enable the transition \( \text{add} \) of the mailing list structure. Tokens in \( m_1 \) are forwarded by \( \text{dist}_1 \) to the place \( 1_m \).

Suppose that \( t \) is fired again for the token \( p_2 = (a_{s_2}, d_{m_2}) \) in \( \text{new} \). In this case, a new mailing list structure is created, which is guaranteed to be independent of the first structure (see Figure 6.6).

### 6.3 No Abstract View for Flat Dynamic zs nets

In what follows, we show, by using a simple example, that it is not always possible to build a dynamic net that represents the abstract semantics of a flat dynamic zs net.

**Example 6.4** (A simple flat d-zs net). Consider the following d-zs net \( B \), where

- \( S_B = \{a, b, z_1, z_2\} \)
- \( T_B = \{t_1, t_2, t_3\} \)
Figure 6.5: Private Mailing Lists after firing t with colours \( p_1 = (a_{s_1}, d_{m_1}) \)
Figure 6.6: Private Mailing Lists after firing $t$ with colours $p_2 = (a_s, d_m)$
6.3. NO ABSTRACT VIEW FOR FLAT DYNAMIC ZS NETS

- \( \delta_{IB} \) are defined s.t. \( t_1 = z_1(X) \oplus z_2(Y) \mid X(a) \oplus Y(b) \), \( t_2 = a(\bullet) \mid z_1(a) \), \( t_3 = b(\bullet) \mid B_1 \), where \( B_1 = \{ \{c\}, \{t\} \} \), \( \delta_{0B_1}(t) = c(\bullet) \), \( \delta_{1B_1}(t) = z_2(c) \), \( z_2(c) \), \( \emptyset \)
- \( m_{0B} = a(\bullet) \oplus b(\bullet) \)
- \( Z_B = \{ z_1, z_2 \} \)

The initial state is shown in Figure 6.7(a). Note that both \( t_2 \) and \( t_3 \) are enabled. First, the transition \( t_2 \) is fired, which initiates a transaction and produces the net in Figure 6.7(b). Then \( t_3 \) is fired, obtaining the net in Figure 6.7(c). The new stable place \( c \) (i.e. a renaming of \( c \)) and the transition \( t \) have been created by the firing of \( t_3 \). Moreover, the token \( z_2(c) \) is generated. Finally, the transaction can commit by firing transition \( t_1 \) that will produce the state shown in Figure 6.7(d). In this state both \( t_2 \) and \( t \) are enabled and can start a new transaction, that will enable transition \( t_1 \), which will commit the transaction.

Note that the computation presented in the previous example corresponds to the movement \( (S_B, T_B, a(\bullet) \oplus c(\bullet), Z_B) \rightarrow (S_B \uplus S_{B_1}, T_B \uplus T_{B_1}, a(\bullet) \oplus c(\bullet), Z_B \uplus Z_{B_1}) \). In order to be able to describe such transaction as a single transition it should be possible to write a transition \( a(\bullet) \oplus b(\bullet) \mid N_t \). It is clear that \( N_t \) adds only one new stable place, namely \( c \), and therefore \( S_{N_t} = \{ c \} \). Moreover, the initial marking of \( N_t \) is \( a(\bullet) \oplus c(\bullet) \), i.e. the tokens produced at the end of the transaction. The problem is when defining \( T_{N_t} \). Note that when the transaction finished, it adds one atomic computation (Figure 6.7(d)): the concurrent firing of \( t_2 \) and \( t \), followed by the execution of \( t_1 \). This derivation can be though as a transition \( a(\bullet) \oplus c(\bullet) \mid a(\bullet) \oplus c(\bullet) \). The main problem is that this is not valid transition for \( N_t \) because its preset fetches tokens from places not in \( S_{N_t} \). Consequently, the abstract view cannot be built for dynamic nets. This example shows that the ZS approach provides dynamic nets with the capability of attaching new transitions to already existent places, a behaviour forbidden for dynamic nets. Consequently (and differently from ordinary, coloured and reconfigurable nets), the abstract view cannot be defined for D-ZS nets.
Figure 6.7: A simple d-zS net.
Part II

Committed Join
Roadmap to the PART II

In this part we introduce a linguistic extension of the Join calculus (described in Section 2.3) tailored to model long-running transactions (also called negotiations, contracts or agreements). The proposed model, named Committed Join (cJoin), accounts for negotiations that are: (1) mobile (communication links may change during a computation) (2) multiway (they have multiple entry and exit points), (3) nested (transactions can be described in term of sub-transactions), and (4) compensable (user-programmed procedures are activated on abort). Moreover, serializability is guaranteed for processes that can be typed as shallow, and a distributed implementation of (flat) cJoin can be programmed in Join by exploiting a modified version of the d2pc protocol (summarised in Section 2.4)

Contracts are decision processes distributed on several nodes, each with the possibility of consulting both local and global resources and generating local subcontracts (e.g. modelling decisions internal to an organisation). The design of cJoin has been inspired by the requirements (i)-(vi) below: (i) each internal sub-decision should be stored locally and not made public before a common agreement is achieved; (ii) global resources might be made available to partners upon commits, marking the conclusion of some contract; (iii) decision processes can be aborted, in which case all participants should be informed and suitable compensation procedures activated (e.g., upon abort, new attempts for finding an agreement can be initiated); (iv) divergence is possible, but well designed GC applications should guarantee that each contract eventually leads to an abort or a commit; (v) when two processes involved in separate negotiations exchange some information, then their contracts should be merged into a unique one; finally (vi) it should be possible to have nested contracts. Though an internal abort can be compensated in such a way that the main contract can still be successfully completed, a failure of the main contract should cause the abort of all ongoing internal activities.

Content of PART II We introduce cJoin in Chapter 7 and give its operational semantics following the reflexive cham style. Then, we discuss its expressiveness on the basis of a few examples and encodings. Finally, we provide a big-step semantics for a sub-calculus of cJoin, called shallow processes, and we show that shallow processes are serializable.

In Chapter 8 we study the primitives of cJoin from the point of view of its implementability. In particular we show that the sub-calculus of processes without sub-negotiations, called flat cJoin, can be encoded in the ordinary Join calculus in a fully distributed way. We first define a type system that singles out flat processes and prove subject reduction for it. Then, we show that all flat cJoin processes can be written in an equivalent canonical form, where a few elementary definition patterns are used. For canonical flat processes we give an encoding into Join. Finally, we present an extension of JoCaml (i.e. a running implementation of Join) with flat cJoin processes.
Chapter 7

Committed Join

7.1 Motivations

In this chapter we present Committed Join, which extends Join with commit primitives. The selection of Join has several different motivations. First, it is a well-known calculus, with an assessed formal theory and, it has distributed implementations, e.g. JoCaml [42] and Polyphonic C² [8]. Second, Join terms can be viewed as dynamic Petri nets [31], i.e., coloured Petri nets with reconfiguration capabilities (i.e., each firing of a transition may deliver tokens to a different set of output places) and the possibility of dynamic creation of components (i.e., the firing of a transition may allocate a new net, parametric in the values of the consumed tokens). As the commitment scheme we introduce has the flavour of multiway transactions in ZS nets, the close relation between Join terms and Petri nets allows us to start from dynamic ZS nets proposed in the first part. In fact, cJoin is mainly based on a distinction between local and global resources (i.e., names) borrowing ideas from the ZS approach. A negotiation can freely compute using internal names, denoting internal states of the negotiation. Usually, during its evolution, a negotiation produces global resources which will be available to the environment only upon commit. Commit is defined as a clean termination, similar to the commit in ZS, where the computed restrictions are made available when the goal is successfully proved. That is, commit can only be executed when all internal computations end, i.e., all participants have done their tasks reaching a (local) state that does not contain locally defined names (the reached state is consistent). The multiway property of ZS transactions is achieved in cJoin by using new kind of ports that allow independent agreements to interact and, at the same time, combine them in a new larger transaction.

In order to simplify the task of writing programs, cJoin provides explicit primitives to denote transactional processes, instead of relying on a typing scheme (i.e., zero-safe or stable ports), although the typing discipline will emerge back in the Join encoding presented in Section 8.4. Moreover, the atomicity of transactions in ZS nets, which is achieved by hiding computations that do not commit, is relaxed in
cJoin. In fact, the cJoin transactions should be explicitly aborted and rolled-back by activating user-programmed processes (i.e., compensations). This property makes cJoin transactions more suitable for handling agreements that can need long periods of time in order to complete. Finally, cJoin negotiations can be nested. Nesting is desirable at least because (i) it provides different levels of abstractions for describing computations, and (ii) it is a basic mechanism for handling local failures, i.e. it is possible for a transaction to commit even though some internal processes have failed.

In a few words, cJoin negotiations become decision processes distributed on several nodes, each with the possibility of manipulating local resources and generating local sub-contracts (e.g. modelling decisions internal to an organisation). Each internal sub-decision is not made public before a common agreement is achieved, and therefore global resources produced during a negotiation are available to partners upon commits, marking the conclusion of some contract. Negotiations can be aborted, in which case the associated compensations must be performed. Moreover, contracts initiated independently can interact exchanging some information and, in this way, becoming part of a larger negotiation. This capability provides a mechanism for allowing new participants to dynamically join an ongoing contract, taking part in the final decision. If a participant $s$ already involved in some negotiation joins a new negotiation, then the two distinct negotiations are merged, as $s$ could freely establish causal dependencies among its activities in the two negotiations. A participant interested in carrying out two separate contracts can emit two independent agents, carrying out the two negotiations separately in a concurrent way.

## 7.2 Committed Join

In this section we present the syntax and semantics of cJoin, which extend the definitions given in Section 2.3 for the Join calculus.

### 7.2.1 Syntax

In order to represent long-running negotiations, we extend the syntax of the Join calculus as shown in Figure 7.1. We add terms to denote negotiation processes. They are represented by $[P : Q]$, where $P$ is the normal execution of the activity and $Q$ is its compensation, i.e., the process that should be executed in case $P$ aborts. A negotiation process is executed in isolation until reaching either a commit state or an abort condition. If $P$ commits, the obtained result is delivered to the outside of the negotiation. Instead, $Q$ is activated when $P$ aborts. The abort decision is caused by the presence of the special basic process *abort*.

A new kind of definitions $J \triangleright P$, called *merge definitions*, is introduced to describe the interactions among negotiations. Merge definitions allow the consumption of messages produced in the scope of different contracts by joining all participants
\[ M, N ::= 0 \mid x\langle y \rangle \mid M \upharpoonright N \]
\[ P, Q ::= M \mid \text{abort} \mid \text{def } D \text{ in } P \mid [P : Q] \mid P \upharpoonright Q \]
\[ D, E ::= J \upharpoonright P \mid J \triangleright P \mid D \upharpoonright E \]
\[ J, K ::= x\langle y \rangle \mid J \upharpoonright K \]

Figure 7.1: Syntax of cJoin.

\[
\begin{align*}
\text{rn}(x\langle y \rangle) &= \{y\} \\
\text{rn}(J \upharpoonright K) &= \text{rn}(J) \cup \text{rn}(K) \\
\text{fn}(D \upharpoonright E) &= \text{fn}(D) \cup \text{fn}(E) \\
\text{fn}(J \triangleright P) &= \text{dn}(J) \cup (\text{fn}(P) \setminus \text{rn}(J)) \\
\text{fn}(0) &= \emptyset \\
\text{fn}(P \upharpoonright Q) &= \text{fn}(P) \cup \text{fn}(Q) \\
\text{fn}(\text{abort}) &= \emptyset \\
\text{fn}(\text{def } D \text{ in } P) &= \{x\} \cup \{y\} \\
\text{fn}([P : Q]) &= \text{fn}(P) \cup \text{fn}(Q) \\
\text{dn}(P) &= \text{dn}_o(D) \cup \text{dn}_m(D) \\
\text{dn}_{o}(D \upharpoonright E) &= \text{dn}_o(D) \cup \text{dn}_o(E) \\
\text{dn}_{o}(J \triangleright P) &= \emptyset \\
\text{dn}_{m}(J \triangleright P) &= \emptyset \\
\text{dn}_{o}(J \upharpoonright P) &= \text{dn}(J) \\
\text{dn}_{m}(J \upharpoonright P) &= \text{dn}(J) \\
\text{dn}_{o}(D \upharpoonright E) &= \text{dn}_o(D) \cup \text{dn}_o(E) \\
\text{dn}_{m}(D \upharpoonright E) &= \text{dn}_m(D) \cup \text{dn}_m(E) \\
\text{dn}_{m}(J \upharpoonright P) &= \text{dn}(J) \\
\end{align*}
\]

Figure 7.2: Defined, received, and free names for cJoin processes.

In a unique larger negotiation. Moreover, usual definitions can be used to create negotiations dynamically. For instance, by firing \( J \upharpoonright [P : Q] \) a new instance of the negotiation \( P \) with compensation \( Q \) is activated.

For convenience we introduce the syntactical category \( M \) of processes without definitions, i.e. a parallel composition of messages.

The detailed definition of free, defined and received names are in Figure 7.2. Note that the set of received names \( \text{rn} \) for a pattern remains unchanged w.r.t. Join, while \( \text{fn} \) is the obvious extension to consider the new operators (i.e. \( \text{abort} \), \( \triangleright \) and \( \upharpoonright \)). For a definition \( D \), we redefine \( \text{dn}(D) \) as the union of two sets: \( \text{dn}_o(D) \) denotes the defined ordinary names, while \( \text{dn}_m(D) \) are the defined merge names.

**Remark 7.1.** Since ordinary and merge names have a different behaviour, we assume \( \text{dn}_o(D) \cap \text{dn}_m(D) = \emptyset \) for every definition \( D \).

### 7.2.2 Operational Semantics

The operational semantics of cJoin is defined in the reflexive CHAM style (described in Section 2.3.2). Molecules \( m \) and solutions \( S \) are defined below.

\[ m ::= P \mid D \mid \_P \_ \mid \{[S]\} \quad S ::= m \mid m, S \]

As in ordinary Join, processes and definitions are molecules. Additionally, a molecule \( \_P \_ \) denotes a compensation that is frozen inside a solution. The chemical rules are in Figure 7.3. The first five rules are the ordinary ones for Join.
\[(\text{STR-NULL})\]
\[0 \Rightarrow \]
\[(\text{STR-JOIN})\]
\[P \mid Q \Rightarrow P, Q\]
\[(\text{STR-AND})\]
\[D \land E \equiv D, E\]
\[(\text{STR-DEF})\]
\[\text{def } D \text{ in } P \equiv D\sigma_{dn(D)}, P\sigma_{dn(D)} \text{ (range}(\sigma_{dn(D)}) \text{ globally fresh)}\]
\[(\text{RED})\]
\[J \triangleright P, J\sigma \rightarrow J \triangleright P, P\sigma\]
\[(\text{STR-CONT})\]
\[[P : Q] \equiv \{P, \downarrow Q\}\]
\[(\text{COMMIT})\]
\[\{M|\text{def } D \text{ in } 0, \downarrow Q\} \rightarrow M\]
\[(\text{ABORT})\]
\[\{\text{abort}|P, \downarrow Q\} \rightarrow Q\]
\[(\text{MERGE})\]
\[\Pi_j J_j \triangleright P, \bigotimes_i \{J_i\sigma, S_i, \downarrow Q_i\} \rightarrow \Pi_j J_j \triangleright P, \{\bigotimes_i S_i, P\sigma, \downarrow \Pi_i Q_i\}\]

Figure 7.3: Operational semantics of cJoin.

Rule STR-CONT describes how a negotiation corresponds to a sub-solution of two molecules: the process \(P\) and its compensation \(Q\), which is frozen (because the operator \(\downarrow\) forbids the enclosed process to react).

COMMIT can be executed only when all activities in the negotiation have been done, i.e. when the (local) state (i.e., the messages contained in the negotiation) does not contain locally defined names. This way, a commit means clean termination where all names denoting internal states of contracts have been consumed. Note that all definitions belonging to a contract are discarded at commit time because the messages that are being released do not contain those names (we recall that local names cannot be extruded). Similarly, the compensation is discarded at commit. Moreover, a negotiation cannot commit when abort is within the solution because abort is not a message. The abort is handled by rule ABORT, which activates the compensation procedure while discarding all terms in the solution. Compensations are not predefined to execute atomically, but they can be explicitly programmed as negotiations.

Interactions among contracts are specified by rule MERGE, which consumes messages \(J_i\sigma\) from different contracts and creates a new larger negotiation by combining the existing contracts together with the new instance \(P\sigma\), where \(\text{dom}(\sigma) = \text{rn}(\Pi_j J_j)\). The compensation for the joint negotiation is the parallel composition of all the original compensations. When merging negotiations, clashes of locally defined names should be avoided by imposing the side condition \(\text{dn}(S_i) \cap (\text{dn}(S_j) \cup \text{fn}(S_j)) = \emptyset\) for \(i \neq j\). However, if we are guaranteed that STR-DEF generates globally fresh names (and not just locally fresh names) then this side condition can be safely omitted, as it is trivially satisfied.

**Notation 7.1.** For simplicity in notation, we will use also \(\rightarrow\) to denote sequences of derivations where possible many applications of heating/cooling rules occur before/after the reduction, i.e. \(\Rightarrow^* \rightarrow \Rightarrow^*\). Moreover we write \(P \rightarrow P'\) for \(\{P\} \rightarrow \{P'\}\).

**Example 7.1.** The following simple examples illustrate the main features of cJoin.
First consider the following process
\[
R_1 = \text{def } D \land J_1 | J_2 | P \triangleright P \text{ in } [\text{def } D_1 \text{ in } J_1 \sigma | P_1 : Q_1 | [\text{def } D_2 \text{ in } J_2 \sigma | P_2 : Q_2]
\]
which defines one merge rule and has two parallel negotiations. Assuming \(dn(J_1 | J_2)\), \(dn(D_1)\) and \(dn(D_2)\) mutually disjoint, the following reduction can take place
\[
\{R_1\} \Rightarrow* \{[D \land J_1 | J_2 | P, \{D_1, J_1 \sigma, P_1, \perp Q_1 -_\} \{D_2, J_2 \sigma, P_2, \perp Q_2 -_\}]\}
\rightarrow \{[D, J_1 | J_2 | P, \{P \sigma, D_1, D_2, P_1, P_2, \perp Q_1 | Q_2 -_\}]\}
\Rightarrow* \{[\text{def } D \land J_1 | J_2 | P \text{ in } [\text{def } D_1 \land D_2 \text{ in } P \sigma | P_1 | P_2 : Q_1 | Q_2]] = \{R_1\}\}

In this case, two independent negotiations have communicated through merge ports and, as a consequence, they are bound together into a larger negotiation.

Suppose now that \(\{[\text{def } D_1 \land D_2 \text{ in } P \sigma | P_1 | P_2]\} \Rightarrow* \{[M|\text{def } D' \text{ in } 0]\}\), in such case, after reducing inside the transaction, we can use the rule (COMMIT), i.e.,
\[
\{R_1\} \rightarrow* \{[\text{def } D \land J_1 | J_2 | P \text{ in } [M|\text{def } D' \text{ in } 0 : Q_1 | Q_2]\}
\rightarrow \{[\text{def } D \land J_1 | J_2 | P \text{ in } M]\}
\]

It should be noticed that \(M\) stands for the messages sent by a transaction to non local ports (i.e. free names or channels defined by \(D\)). This kind of messages are maintained inside a transaction until the commit, when they are finally released.

Instead, suppose that \(\{[\text{def } D_1 \land D_2 \text{ in } P \sigma | P_1 | P_2]\} \Rightarrow* \{[\text{abort} | M | \text{def } D' \text{ in } P']\}\).

Therefore, the whole transaction can be aborted by using rule (ABORT), i.e.,
\[
\{R_1\} \rightarrow* \{[\text{def } D \land J_1 | J_2 | P \text{ in } [\text{abort} | M | \text{def } D' \text{ in } P' : Q_1 | Q_2]\}
\rightarrow \{[\text{def } D \land J_1 | J_2 | P \text{ in } Q_1 | Q_2]\}
\]

In this case, the compensation is released. Actually, the released compensation corresponds to the compensations of the original independent negotiations.

In what follows we illustrate the behaviour of nested transactions. Consider the following process, which has two nested negotiations
\[
R_2 = [\text{def } D \text{ in } P | [\text{def } D_1 \text{ in } P_1 : Q_1] | [\text{def } D_2 \text{ in } P_2 : Q_2] : Q]
\]

Suppose now that \(P = \text{abort} | P'\), then the whole transaction can be aborted by releasing the compensation \(Q\). That is
\[
\{R_2\} = \{[\text{def } D \text{ in } \text{abort} | P' | [\text{def } D_1 \text{ in } P_1 : Q_1] | [\text{def } D_2 \text{ in } P_2 : Q_2] : Q]\}
\rightarrow \{Q\}
\]

Suppose instead, that \(P = 0\) and \(P_1 = \text{abort} | P'_1\). In this case the abort affects only the corresponding sub-transaction, i.e.
\[
\{R_2\} = \{[\text{def } D \text{ in } [\text{def } D_1 \text{ in } \text{abort} | P'_1 : Q_1]| [\text{def } D_2 \text{ in } P_2 : Q_2] : Q]\}
\rightarrow \{[\text{def } D \text{ in } Q_1 | [\text{def } D_2 \text{ in } P_2 : Q_2] : Q]\} = \{R_2'\}
\]
Note that the abort is hidden to sibling and parent negotiations, and only the compensation of the aborted negotiation is activated. After that, the second sub-negotiation can commit, e.g. $[[\text{def } D_2 \text{ in } P_2 : Q_2]] \rightarrow^* \{M\}$. Then, we would have

$$[[R_2]] \rightarrow^* \{[\text{def } D \text{ in } Q_1 | M : Q]\}$$

Finally, the top-level negotiation can commit, e.g. when $Q_1 | M = 0$. In this case the top-level transaction commits even though some sub-negotiations has aborted.

**Proposition 7.1.** cJoin is a conservative extension of Join.

**Proof.** We want to prove that cJoin is a conservative extension of Join. It is obvious from the syntax that any Join process is also a cJoin process. It remains to show that: (1) for any Join processes $P$ and $Q$, if $P \rightarrow Q$ in Join, then $P \rightarrow Q$ in cJoin; and (2) for any Join processes $P$, if $P \rightarrow Q$ in cJoin, then $Q$ is a Join process and $P \rightarrow Q$ in Join. Both implications follow straightforwardly from the fact that the chemical rules of cJoin can be partitioned in two sets: one consisting exactly of the chemical rules of Join, and the other containing structural rules involving non-Join operators on both sides and reaction rules involving non-Join operators in the left-hand side. \(\square\)

**Discussion** Sibling contracts can be merged only by using merge definitions introduced by their parent. In practice, it might be useful to apply a merge definition provided by any ancestor. To this aim, the rule STR-Move below might be added, so that merge definitions could float across contract boundaries.

**STR-Move** \(J \triangleright P, [S] \Rightarrow \{J \triangleright P, S\}\)

Regarding deadlocks, note that stall negotiations are not discarded. For instance, the process $[\text{def } x \langle y \rangle \triangleright 0 \text{ in } x \langle \rangle : Q]$ cannot compute. Neither it can commit, because there is a message on the local port $x$. In this situation the contract is blocked and should be aborted. Some of these situations can be recognised and handled locally to promote the abort (i.e., when no local rules can be applied). These situations can be represented by ad hoc rules or by a general rule to generate nondeterministically the abort (situations that real implementations typically handle with timeouts). Nevertheless, we cannot expect to axiomatise stall situations because it would mean to write axioms recognising non-termination, which is an undecidable problem. On the other hand, we do not want to limit the expressiveness of the language.

With respect to the requirements discussed in the Introduction, we have that, membranes are exploited to define the boundaries of negotiations. Process like $[P_1 \mid [P_2 : Q_2] \mid [P_3 : Q_3] : Q_1]$ straightforwardly model sub-negotiations. The decisions taken internally by $P_2$ can influence $P_3$ only if some merge definition is available at the level of $P_1$. In absence of merge definitions, global and local resources are kept separate in each sub-negotiation. The commit of $P_2$ can only happen when
the internal state contains only global resources. At commit time, the result of
the negotiation is made available to \( P_1 \). An abort generated in \( P_2 \) activates the
compensation \( Q_2 \) at the level of \( P_1 \), neither forcing the abort of any other sub-
negotiations (\( P_3 \)) nor the abort of the main contract (\( P_1 \)). Note that if \( [P_2 : Q_2] \)
was the result of the merging of several negotiations, then \( Q_2 \) is the union of all the
compensations of the participants.

An important restriction is that local resources can neither cross negotiations
boundaries nor be extruded to siblings negotiations. The only way to exchange
information between siblings negotiations is by merging all participants into a unique
negotiation that must then commit, or abort, or diverge as such.

### 7.2.3 Abstract Semantics

The notion of process equivalence we use in this thesis relies on barbs, and the def-
nitions given for \( \text{Join} \) (see Section 2.3.4) remains unchanged for \( \text{cJoin} \). In particular,
note that the set of strong barbs of a process is given by the following predicate

\[
P \downarrow_x \iff \exists P', \bar{u} : P \equiv \text{def } D \text{ in } P' | x \langle \bar{u} \rangle \text{ and } x \not\in \text{dn}(D)
\]

Consequently, the processes introduced by \( \text{cJoin} \), i.e. \( \text{abort} \) and \( \_ : \_ \), have no
strong barbs. As far as merge definitions is concerned, merge names are part of the
defined names of a process, and hence not observable.

### 7.3 Examples

In this section we present some basic examples in order to show the main features
of commited actions.

**Example 7.2** (Mailing list). Consider a data structure that allows to send atomi-
cally a message to a list of subscribers (in the sense that the same message is either
sent to all or to none). Such structure can be defined as \( \text{ML} = \text{MailingList}(k) \triangleright \text{MLDef} \), where:

\[
\text{MLDef} = \text{def List in } k\langle \text{add, tell, close} \rangle | l\langle \text{nil} \rangle
\]

\[
\text{List} = \begin{cases}
\text{nil} \langle v, w \rangle \triangleright \langle w \rangle \\
\text{add} \langle x \rangle \triangleright \text{def } z \langle v, w \rangle \triangleright x \langle v \rangle | y \langle v, w \rangle \text{ in } l \langle z \rangle \\
\text{tell} \langle v \rangle \triangleright \text{def } z \langle \rangle \triangleright 0 \text{ in } y \langle v, z \rangle | l \langle y \rangle : l \langle y \rangle \\
\text{close} \langle \rangle \triangleright 0
\end{cases}
\]

A new mailing list is created by sending a message to the port \( \text{MailingList} \). Since
\( \text{cJoin} \) adheres to the "continuation passing" style of programming, the content of
the message sent to \( \text{MailingList} \) is a continuation port \( k \), which expects information
about the newly created mailing list. The creation of a new list defines five fresh
ports nil, l, add, tell and close: three of them (namely add, tell, and close) will be used to interact with the list from “outside” and will be sent to the port k as the outcome of the creation. The remaining two ports will never be extruded. They denote the empty list (nil) and the actual state of the list (l).

Once a list is created, a new subscriber can be added by sending a message add with the name x of the port where it will be listening to for new messages. In this case, the list is modified by installing z (on top of it), a forwarder of messages to x.

The port tell is used to send a message v to the list. When tell is received a new negotiation identified by a fresh name z is generated, and the state of the structure is put inside the negotiation, therefore all other activities, such as adding or closing are blocked until the negotiation ends. Inside the negotiation, the message v is sent to the forwarder at the top of the list y with the identifier of the negotiation z. Note that each forwarder sends the message to the corresponding subscriber and to the following forwarder in the list. This is repeated until nil is reached, when a message to the identifier of the transaction is sent. The firing rule z(1) ≫ 0 consumes the last local name and the contract commits by releasing all the messages addressed to the subscribers and the state of the list. Then the list is ready to serve new requests.

The following process Sys subscribes two users, Alice and Bob to the mailing list, and sends the message News.

\[
\text{Emp} = employees(a, t, c) \triangleright a\langle Alice \rangle | a\langle Bob \rangle | t\langle News \rangle
\]

\[
\text{Sys} = \text{def ML } \land \text{Emp in MailingList(employees)}
\]

A possible computation of the process Sys is shown in Figure 7.4. In this particular computation, both subscriptions take place before the emission of the message, nevertheless the process does not fix this priority and consequently messages could be consumed in a different order. For simplicity, we abbreviate chemical soups by omitting definitions present in successive soups, though we usually write only those definitions involved in the reduction step. Inside soups, we underline the fired reaction rule and the consumed messages that matched the corresponding pattern.

The phase LIST CREATION instantiates a new mailing list by defining the fresh ports add, tell and close, which are sent to the port employee. The second phase (SUBSCRIPTIONS) adds the names Alice and Bob to the created mailing list. Phase Distribution of News generates a new transaction that produces (in a sequential way) the copies of the name News to be sent to any subscriber of the list. Nevertheless those messages are not released until the phase COMMIT takes place. Only when the transaction commits, the generated messages are atomically sent to subscribers.

Example 7.3 (Trip booking). Figure 7.5 shows the encoding of the application Trip that allows a user to book flights and accommodations. Trip is defined in terms of three components: the hotel H, the airline A and the customer C. The component H is a process that activates (by firing the definition for WaitBooking) a negotiation to serve customer requests. Such negotiation starts by publishing on the merge port offerRoom (defined in Trip) the names of the services a client should use to reserve a room: request to ask for a quote; and confirm to accept an offer. The component
7.3. EXAMPLES

Initial Soup: \( \llbracket \text{Sys} \rrbracket \Rightarrow \llbracket \text{MailingList}\langle k \rangle \vee \text{MLDef, Emp, MailingList}\langle \text{employees} \rangle \rrbracket \)

LIST CREATION:
\[ \llbracket \text{MailingList}\langle k \rangle \vee \text{MLDef, Emp, MailingList}\langle \text{employees} \rangle \rrbracket \rightarrow \]
\[ \llbracket \text{MailingList}\langle k \rangle \vee \text{MLDef, Emp, MLDef}\langle \text{employees/}k \rangle \rrbracket \Rightarrow \]
\[ \llbracket \text{MailingList}\langle k \rangle \vee \text{MLDef, Emp, List}\langle \text{employees/}k \rangle, \text{employee}\langle \text{add, tell, close} \rangle, l\langle \text{nil} \rangle \rrbracket \rightarrow \]
\[ \{ l\langle y \rangle \mid \text{add}\langle x \rangle \triangleright \ldots, \ldots, l\langle \text{nil} \rangle, \text{add}\langle \text{Alice} \rangle, \text{add}\langle \text{Bob} \rangle, \text{tell}\langle \text{News} \rangle \} \]

SUBSCRIPTIONS:
\[ \{ l\langle y \rangle \mid \text{add}\langle x \rangle \triangleright \ldots, \ldots, l\langle \text{nil} \rangle, \text{add}\langle \text{Alice} \rangle, \text{add}\langle \text{Bob} \rangle, \text{tell}\langle \text{News} \rangle \} \rightarrow \]
\[ \{ l\langle y \rangle \mid \text{add}\langle x \rangle \triangleright \ldots, \ldots, z_A\langle v, w \rangle \triangleright \text{Alice}\langle v \rangle|\text{nil}\langle v, w \rangle, l\langle z_A \rangle, \text{add}\langle \text{Bob} \rangle, \text{tell}\langle \text{News} \rangle \} \rightarrow \]
\[ \{ l\langle y \rangle \mid \text{tell}\langle v \rangle \triangleright \ldots, \ldots, z_B\langle v, w \rangle \triangleright \text{Bob}\langle v \rangle|z_A\langle v, w \rangle, l\langle z_B \rangle, \text{tell}\langle \text{News} \rangle \} \]

DISTRIBUTION OF News:
\[ \{ l\langle y \rangle \mid \text{tell}\langle v \rangle \triangleright \ldots, \ldots, z_B\langle v, w \rangle \triangleright \text{Bob}\langle v \rangle|z_A\langle v, w \rangle, l\langle z_B \rangle, \text{tell}\langle \text{News} \rangle \} \rightarrow \]
\[ \{ z_B\langle v, w \rangle \triangleright \text{Bob}\langle v \rangle|z_A\langle v, w \rangle, \ldots, \{ z\} \triangleright 0, z_B\langle \text{News}, z \rangle, l\langle z_B \rangle, l\langle z_B \rangle \} \rightarrow \]
\[ \{ z_A\langle v, w \rangle \triangleright \text{Alice}\langle v \rangle|\text{nil}\langle v, w \rangle, \ldots, \{ z\} \triangleright 0, z_B\langle \text{News}, z \rangle, l\langle z_B \rangle, l\langle z_B \rangle \} \rightarrow \]
\[ \{ \text{nil}\langle v, w \rangle \triangleright w\langle \rangle, \ldots, \{ z\} \triangleright 0, z_B\langle \text{News}, \text{Alice}\langle v \rangle, z\langle \rangle, l\langle z_B \rangle, l\langle z_B \rangle \} \rightarrow \]
\[ \{ \ldots, \{ z\} \triangleright 0, z_B\langle \text{News}, \text{Alice}\langle v \rangle, z\langle \rangle, l\langle z_B \rangle, l\langle z_B \rangle \} \rightarrow \]
\[ \{ \ldots, \{ z\} \triangleright 0, z_B\langle \text{News}, \text{Alice}\langle v \rangle, z\langle \rangle, l\langle z_B \rangle, l\langle z_B \rangle \} \]}

COMMIT:
\[ \{ \ldots, \{ z\} \triangleright 0, z_B\langle \text{News}, \text{Alice}\langle v \rangle, z\langle \rangle, l\langle z_B \rangle, l\langle z_B \rangle \} \} \Rightarrow \]
\[ \{ \ldots, \{ z_B\langle \text{News}, \text{Alice}\langle v \rangle, z\langle \rangle, l\langle z_B \rangle, l\langle z_B \rangle \} \} \rightarrow \]
\[ \{ \ldots, z_B\langle \text{News}, \text{Alice}\langle v \rangle, l\langle z_B \rangle, l\langle z_B \rangle \} \} \]

Figure 7.4: A possible computation of the Mailing list example
\[ H \equiv \text{def} \quad \text{WaitBooking}(\cdot) \triangleright [\text{def} \quad \text{request}(o) \triangleright o(\$) \mid \text{price}(\$) \\
\land \quad \text{price}(\$) \mid \text{confirm}(v) \triangleright \text{BookedRoom}(v) \\
\land \quad \text{price}(\$) \triangleright \text{abort} \\
\text{in} \quad \text{offerRoom}(\text{request}, \text{confirm}) : Q] \\
\land \quad \text{BookedRoom}(v) \triangleright R \\
\text{in} \quad \text{WaitBooking}(\cdot) | \ldots \\
C \equiv \text{def} \quad \text{BookHotel}(\cdot) \triangleright [\text{def} \quad \text{HotelMsg}(r, c) \triangleright \\
\text{def} \quad \text{offer}(\$) \triangleright c(\text{visa}) \mid \text{hotelOK}(h\text{Conf}) \\
\land \quad \text{offer}(\$) \triangleright \text{abort} \\
\land \quad h\text{Conf}(\cdot) \triangleright \text{HotelFound}(\cdot) \\
\text{in} \quad r(\text{offer}) \quad \text{in} \quad \text{searchRoom}(\text{HotelMsg}) : Q'] \\
\land \quad \text{BookFlight}(\cdot) \triangleright [\text{def} \quad \text{FlightMsg}(r, c) \triangleright \\
\text{def} \quad \text{offer}(\$) \triangleright c(\text{visa}) \mid \text{flightOK}(f\text{Conf}) \\
\land \quad \text{offer}(\$) \triangleright \text{abort} \\
\land \quad f\text{Conf}(\cdot) \triangleright \text{FlightFound}(\cdot) \\
\text{in} \quad r(\text{offer}) \quad \text{in} \quad \text{searchFlight}(\text{FlightMsg}) : Q''] \\
\land \quad \text{flightOK}(fc) \mid \text{hotelOK}(rc) \quad \triangleright fc(\cdot) \mid rc(\cdot) \\
\text{in} \quad \text{BookHotel}(\cdot) \mid \text{BookFlight}(\cdot) \mid \ldots \\
\text{Trip} \equiv \text{def} \quad \text{searchRoom}(hm) \mid \text{offerRoom}(r, c) \triangleright hm(r, c) \\
\land \quad \text{searchFlight}(fm) \mid \text{offerFlight}(r, c) \triangleright fm(r, c) \quad \text{in} \quad H \mid A \mid C \]

Figure 7.5: Trip booking example.

A (omitted in Figure 7.5) is defined analogously, but it publishes services on port \textit{offerFlight} instead of \textit{offerRoom}.

The component \(C\) defines two rules for creating negotiations: one for booking rooms and the other for buying flight tickets. Both contracts are quite similar. In particular, the negotiation for booking a room starts by sending a message to the merge port \textit{searchRoom} (defined in Trip) to obtain the names for interacting with a hotel. The first merge rule in \textit{Trip} will associate an offer \textit{offerRoom}(r, c) from a hotel with a request \textit{searchRoom}(r, c) from a client by sending the names \(r\) and \(c\) to the corresponding port \(hm\). Once received \(r\) and \(c\) on \textit{HotelMsg}, \(C\) uses \(r\) to send a message to \(H\) for asking for a quote. Then, the hotel will answer with an offer on port \textit{offer}. Whether the customer accepts or not a particular quote is modelled by the multiple definitions for the pattern \textit{offer}(\$) in \(C\). If the offer is not adequate then \(C\) can abort the negotiation, which will activate the compensation \(Q \mid Q'\) (analogously for \(H\) and \(A\)). \(C\) can accept the offer by sending a confirmation message on port \(c\). In this case, \(C\) also generates a message to the local port \textit{hotelOK}(h\text{Conf}). This message will be managed by the local merge rule defined by \(C\). The contract will be blocked until a running negotiation for buying flight tickets generates a message on \textit{flightOK}. At this time, the local merge definition can be fired and both contracts merged. Eventually the negotiation will commit by releasing the messages on \textit{HotelFound} and \textit{FlightFound}. Moreover, messages \textit{BookedRoom}(v) and \textit{SoldFlight}(v) generated by \(H\) and \(A\) to change their local states are released only at
this time, when all participants have committed.

**Example 7.4** (Andorra Kernel Language). As a third example, we sketch how merge definitions and nesting can be used to model the committed choice of logical and concurrent constraint programming languages discussed in Section 1.2.3. As an example we will consider a simplified version of the Andorra Kernel Language (AKL) [63], which is a concurrent logic programming language. We describe the computation model of AKL [63], following the presentation in [63], but considering only `commit('!')` as guard operator.

**Syntax of Programs and Goals** An AKL program is a list of guarded clauses. In particular we consider rules with the following shape

\[< \text{head} >: - < \text{guard} > | < \text{body} >\]

where the head is a program atom, i.e. an atomic formula \( p(x_1, \ldots, x_n) \) with \( x_1, \ldots, x_n \) are variables, and the guard and the body are sequences of program atoms. Moreover, all variables \( x_i \) occurring in the head are different, but they may be repeated in the guard and the body.

**Computation Model** The computational states for the execution of (our simplified version) of AKL programs are given by the following grammar

\[
\langle \text{configuration} \rangle ::= \langle \text{and-box} \rangle \\
\langle \text{and-box} \rangle ::= \text{and}(\langle \text{sequence of local goals} \rangle; \langle \text{constraint} \rangle) \\
\langle \text{local goal} \rangle ::= \langle \text{atomic goal} \rangle \mid \langle \text{choice-box} \rangle \\
\langle \text{choice-box} \rangle ::= \text{choice}(\langle \text{sequence of guarded goals} \rangle) \\
\langle \text{guarded goal} \rangle ::= \langle \text{and-box} \rangle \langle \text{sequence of atomic goals} \rangle
\]

Note that a configuration given by an and-box \( \text{and}(P_1, \ldots, P_n; \sigma) \), where all \( P_i \) are local goals, represents the proof of \( P_1 \land \ldots \land P_n \) with the associated satisfiable constraint \( \sigma \). Note that any \( P_i \) may be either an atomic goal \( p(x_1, \ldots, x_n) \) (i.e. the usual goals of logic programming) or a choice-box \( \text{choice}(S_1, \ldots, S_m) \) where

\[
S_j = \text{and}(P_{j1}, \ldots, P_{jk}; \sigma_j)|B_j.
\]

Such a choice-box is proved by trying in parallel all guards \( \text{and}(P_{j1}, \ldots, P_{jk}; \sigma_j) \). If a guard \( \text{and}(P_{j1}, \ldots, P_{jk}; \sigma_j) \) is successfully proved, then the corresponding body \( B_j \) is selected and all other goals in the choice box are discarded, i.e. a committed choice takes place.

Rewriting rules are summarised in Table 7.1. The letters \( C \) and \( D \) stand for sequences of local goals, \( \sigma \) and \( \theta \) for constraints; \( B \) for a sequence of atomic goals; \( G \) for a goal; \( P, Q \) for sequences of goals; \( S \) and \( T \) for sequences of guarded goals; \( V \) and \( W \) for set of variables; and \( \beta \) for a single goal. And-boxes of the form \( \text{and}(); \sigma \rangle \) are written as \( \sigma \). The term \( \text{tt} \) denotes a variable-free constraint formula that is true in the constraint theory. The symbol \( \text{fail} \) will be used to denote failed goals. Each subgoal has an environment, defined as the conjunction of the constraints of
(C1) \( \text{and}(C, \text{op}(\sigma), D; \theta) \Rightarrow \text{and}(C, D; \theta \land \sigma) \) if \( \theta \land \sigma \) is satisfiable and \( \text{op}(\sigma) \) is satisfied

(Cf) \( \text{and}(C, \text{op}(\sigma), D; \theta) \Rightarrow \text{fail} \) if \( \sigma \) is inconsistent with the environment of \( \text{op}(\sigma) \)

(FK) \( A \Rightarrow \text{choice}(|G_1; \text{tt}|B_1, \ldots, \text{and}(|G_n; \text{tt}|B_n)| \) where \( A \) unifies with \( H_i \) for rules \( H_i : -G_i; B_i \)

(PD) \( \text{and}(C, \text{choice}(\theta_B; B), D; \theta) \Rightarrow \text{and}(C, B, D; \theta \land \sigma) \) if \( \theta \land \sigma \) is satisfiable

(I) \( \text{choice}(P, \sigma; B, Q) \Rightarrow \text{choice}(\sigma; B) \)

(SC) \( \text{and}(B; \sigma) \Rightarrow \text{fail} \) if \( \sigma \) is unsatisfiable with the environment

(F1) \( \text{and}(B, \text{fail}, C; \sigma) \Rightarrow \text{fail} \)

(F2) \( \text{choice}(S, (\text{fail}; B); T) \Rightarrow \text{choice}(S, T) \)

Table 7.1: Operational semantics of AKL.

all the and-boxes in which the goal occurs. The notation \( \text{op}(\sigma) \) denotes a constraint operation \( \text{op} \) (e.g., \( \text{ask} \) or \( \text{tell} \)) applied to the primitive constraint \( \sigma \).

Rules (C1) and (CF) denote the effects of a goal that has been proved by producing the constraint \( \sigma \). When \( \sigma \) is consistent with the enclosing environment, the goal is reduced and the constraint of the enclosing and-box is updated, otherwise the whole goal fails. Local forking (FK) on an atomic goal initiates the parallel proof of all definitions. The deterministic promotion (DP) communicates the results of a local computation to the siblings of the goal that started the local computation. Rule (I) describes the pruning to be applied when the guard of a commit rule is proved. Note that the actual promotion is performed by the deterministic promotion rule. The last three rules are normalisation rules that describe the propagation of failures (rules (F1) and (F2)), and the synchronisation with the environment (SC).

An execution of \( \mathcal{P} \) is initiated by providing a goal \( \mathcal{G} \), which is a conjunction of atoms.

A \( \text{cJoin} \) process that simulates the execution of a program \( \mathcal{P} \) queried with \( \mathcal{G} \) is:

\[
\text{def } D \land [\text{defs}(\mathcal{P})] \land [\text{undef}(\mathcal{P})] \text{ in } [\text{and}(\mathcal{G})]_{\text{true} \mathcal{G}, \text{false} \mathcal{G}} | \text{unif}(\text{final}, \text{tt})
\]

The definitions in \( D \) are needed to promote constraints computed locally. The rules in \( \mathcal{P} \) are translated separately by grouping all rules defining the same atom \( A \). Such partition is denoted by \( \text{defs}(\mathcal{P}) \), while \( \text{undef}(\mathcal{P}) \) denotes all atoms in \( \mathcal{P} \) without defining rules, i.e., atoms whose proofs will always fail. Constraints \( \theta \) are encoded conveniently as \( \text{cJoin} \) processes (\( \text{tt} \) is the empty constraint). In general, the term \( \text{unif}(\text{final}, \theta) \) stores the computed constraints of a running proof. At the end, a message on either port \( \text{true} \mathcal{G} \) or \( \text{false} \mathcal{G} \) will inform the environment about the outcome of the computation. The encoding of clauses and goals is in Figure 7.6.

An atom \( A \) is encoded as a merge rule that substitutes a message \( A(t, f) \) by the proof of its defining rules (rule \( \text{DEF} \)). An undefined atom is encoded as a
rule that always fails (UNDEF). A conjunction (AND) corresponds to a process that activates a new negotiation containing the atoms \( A_i \langle t_i, f_i \rangle \) to be proved and the initial local constraints \( \text{unif} \langle w, \text{tt} \rangle \). Every sub-proof initiated by \( A_i \langle t_i, f_i \rangle \) will notify its termination by using ports \( t_i \) (success) and \( f_i \) (failure). If all sub-proofs end successfully, the second definition in the negotiation can be fired producing \( t \rangle \) (i.e., the signal of the successful proof of the conjunction) and \( \text{done} \langle w \rangle \) that activates the promotion of computed constraints (omitted for simplicity). Note that \( t \rangle \) will be released only at commit, after constraints have been promoted. Instead, if a sub-atom fails, all running sub-contracts are aborted and the contract commits by releasing the activation of a new proof.

Rule CHOICE opens a negotiation for proving one of the guards of a multiple choice goal. When a guard is successful the negotiation can commit by releasing the message \( i \rangle \), which activates the body \( B_i \) of the chosen goal.

If there is an AKL refutation for the goal \( G \) with computed constraints \( \text{and}(\text{tt}, \theta) \) then the messages \( \text{final}(\theta) \) and \( \text{true}G \rangle \) can be released. On the other hand, \( \text{false}G \rangle \) is generated only if \( G \) cannot be proved.

### 7.4 Serializability and Big-Step Semantics

The semantics of cJoin given in Figure 7.3 allows the cooperation among several negotiations. Nevertheless, we would like to reason about a process by analysing interacting negotiations independently from the rest of the system. A concurrent execution \( T_1 \parallel \ldots \parallel T_n \) of several transactions is said serializable if there exists a sequence \( T_{i_1}; T_{i_2}; \ldots; T_{i_n} \) that executes all transactions one at a time (without inter-
leaving their steps) and produces the same result [10]. Serializability is important because it allows to reason about the behaviour of a system by considering one transaction at a time. In this section we introduce a syntactical restriction on processes, called shalowness, and we show that it guarantees serializability.

The idea is to describe multi-party negotiations as abstract transitions that fetch the messages needed to initiate all sub-negotiations separately and produce the processes released at commit or abort. Consequently, serializable negotiations can postpone the activation of each sub-negotiation until all other cooperating sub-negotiations needed to commit can be activated.

**Definition 7.1 (Shalowness).** The *nesting* \( \text{nest}(P) \) of \( P \) is defined by:

\[
\text{nest}(0) = \text{nest}(\text{abort}) = \text{nest}(x(y)) = 0 \\
\text{nest}(\text{def } D \text{ in } P) = \text{nest}(P) \\
\text{nest}(\text{def } P | Q) = \max\{\text{nest}(P), \text{nest}(Q)\}
\]

A basic definition \( D \) is a shallow definition if it has one of the following forms:

1. \( D = J \triangleright P \), where \( \text{nest}(P) = 0 \) or \( P = [R : Q] \) and \( \text{nest}(R|Q) = 0 \)
2. \( D = J \triangleright P \) and \( \text{nest}(P) = 0 \)

A process \( P \) is shallow if any basic definition in \( P \) is shallow. Moreover, we call a process a shallow process \( P \) stable iff \( \text{nest}(P) = 0 \).

The shallow property imposes a discipline for activating negotiations. In particular, condition 1 assures that the firing of an ordinary rule increases the height of the nesting by at most one level (i.e., a definition produces either a stable process or one negotiation without nested sub-contracts). Condition 2 forbids the creation of sub-negotiations while merging. The absence of condition 2 would prevent the possibility of postponing the activation of some negotiations until all cooperating sub-negotiations can be activated inside the same negotiation. Consider the following process:

\[
P = \text{def } x\langle u \rangle | y\langle w \rangle \triangleright [\text{def } z\langle \rangle \triangleright 0 \text{ in } v\langle z \rangle : 0] \\
in \quad \quad \quad \quad \text{def } z_0\langle a \rangle | z_1\langle b \rangle \triangleright 0 \text{ in } x\langle z_0 \rangle | x\langle z_1 \rangle : Q_1 \\
\quad \quad \quad \quad \quad \quad \quad \text{def } z_2\langle \rangle \triangleright 0 \text{ in } y\langle z_2 \rangle : Q_2 \\
\quad \quad \quad \quad \quad \quad \quad \quad \text{def } z_3\langle \rangle \triangleright 0 \text{ in } y\langle z_3 \rangle : Q_3
\]

where the definition \( x\langle u \rangle | y\langle z \rangle \triangleright [\text{def } z\langle \rangle \triangleright 0 \text{ in } v\langle w \rangle : Q] \) is not shallow, because the nesting of the guarded process is 1. A possible reduction of \( P \) is as follow.
(1) \( P \rightarrow \text{def } x\langle v \rangle|y\langle w \rangle \gg [\text{def } z\langle \rangle \gg 0 \text{ in } v\langle z \rangle : 0] \)
\[\text{in } [\text{def } z_0\langle a \rangle|z_1\langle b \rangle \gg 0 \land z_2\langle \rangle \gg 0 \text{ in } [\text{def } z\langle \rangle \gg 0 \text{ in } z_0\langle z \rangle : 0] | x\langle z_1 \rangle : Q_1|Q_2] \]
\[| [\text{def } z_3\langle \rangle \gg 0 \text{ in } y\langle z_3 \rangle : Q_3] \]

(2) \( \rightarrow \text{def } x\langle v \rangle|y\langle w \rangle \gg [\text{def } z\langle \rangle \gg 0 \text{ in } v\langle z \rangle : 0] \)
\[\text{in } [\text{def } z_1\langle a \rangle|z_0\langle b \rangle \gg 0 \land z_2\langle \rangle \gg 0 \land z_3\langle \rangle \gg 0 \text{ in } [\text{def } z\langle \rangle \gg 0 \text{ in } z_0\langle z \rangle : 0] | [\text{def } z\langle z'_\rangle \gg 0 \text{ in } z_1\langle z'_\rangle : 0] : Q_1|Q_2|Q_3] \]

(3) \( \rightarrow \text{def } x\langle v \rangle|y\langle w \rangle \gg [\text{def } z\langle \rangle \gg 0 \text{ in } v\langle z \rangle : 0] \)
\[\text{in } [\text{def } z_1\langle a \rangle|z_0\langle b \rangle \gg 0 \land z_2\langle \rangle \gg 0 \land z_3\langle \rangle \gg 0 \text{ in } 0 : 0] : Q_1|Q_2|Q_3 \]

(4) \( \rightarrow \text{def } x\langle v \rangle|y\langle w \rangle \gg [\text{def } z\langle \rangle \gg 0 \text{ in } v\langle z \rangle : 0] \)
\[\text{in } [\text{def } z_1\langle a \rangle|z_0\langle b \rangle \gg 0 \land z_2\langle \rangle \gg 0 \land z_3\langle \rangle \gg 0 \text{ in } 0 : Q_1|Q_2|Q_3] \]

(5) \( \rightarrow \text{def } x\langle v \rangle|y\langle w \rangle \gg [\text{def } z\langle \rangle \gg 0 \text{ in } v\langle z \rangle : 0] \)
\[\text{in } 0 \]

The reduction (1) merges the first two negotiations into a larger one, and at the same time generates a sub-negotiation. Step (2) merges the two top-level transactions and produces a new sub-transaction. Note that in this reduction one of the merged transactions have already an active sub-negotiation. The computation follows by merging in step (3) the two sub-negotiations into a unique negotiation, which commits in step (4). The top-level transaction commits in step (5).

The main problem with the previous reduction is that step (2) merges a transaction that has an active sub-negotiation. Moreover, it is not possible for \( P \) to reduce to the same process in a different way, i.e. by merging only negotiations without sub-negotiations. Consequently, it is not possible to describe the behaviour of this process by considering just one level of nesting.

It should be noticed that although shallowness forbids rules such as \( J_1 \gg P \mid [P_1 : Q] \) and \( J_2 \gg [[P_1 : Q_1] : Q] \), they however can be encoded as shallow definitions by using new local parts, e.g. as \( D_1 = J_1 \gg x\langle \rangle |P \land x\langle \rangle \gg [P_1 : Q] \) and \( D_2 = J_2 \gg [\text{def } x\langle \rangle \gg P_1 : Q_1] \text{ in } x\langle \rangle : Q \), respectively. Note that \( \text{def } D_2 \text{ in } J_2\sigma \) reduces in two steps to \( \text{def } D_2 \text{ in } [\text{def } x\langle \rangle \gg P_1\sigma : Q_1\sigma] \text{ in } [P_1\sigma : Q_1\sigma] : Q\sigma] \), which has nested negotiations and is shallow.

In the following \( \mathcal{P} \) and \( Q \) will denote shallow processes, \( \mathcal{D} \) a shallow definition, \( \mathcal{S} \) a stable process, and \( \mathcal{B} \) a shallow definition containing just merge rules. We abbreviate \( \text{def } \mathcal{D} \text{ in } \mathcal{P} \) as \( \mathcal{D} \vdash \mathcal{P} \), and \( \vdash \mathcal{P} \) as \( \mathcal{P} \). Terms are considered up-to structural equivalence generated by closure w.r.t. the equations for the associativity and commutativity of | and \( \land \), 0 the unit for |, and

\[ \mathcal{D} \vdash \mathcal{P} \mid \text{def } \mathcal{D}' \text{ in } Q = \mathcal{D} \land \mathcal{D}'\sigma_{dn} \vdash \mathcal{P} \mid Q\sigma_{dn} \]
\[\text{range}(\sigma_{dn}) \cap (fn(\mathcal{D}) \cup fn(\mathcal{P}) \cup fn(\text{def } \mathcal{D}' \text{ in } Q)) = \emptyset \]
We characterise serializability by the big-step reduction relation between shallow processes presented in Figure 7.7.

Steps can be composed in parallel (PAR) and sequentially (SEQ), even with idle transitions (IDLE). Rule GLOBAL FIRING corresponds to the firing of an ordinary definition in a top-level process. Instead LOCAL FIRING states possible internal transitions of a running contract. LOCAL FIRING represents suitable sub-negotiations as ordinary transitions at an abstract level. In fact, the computations occurring at a lower level in the nesting hierarchy (premise of LOCAL FIRING) that are relevant to its containing negotiation are those relating stable processes, i.e., $\mathcal{S}$ and $\mathcal{S}^\prime$. A negotiation has available, in addition to its own definitions, the merge definitions introduced by its parent. In fact, a merge definition applied on a single contract behaves as an ordinary rule but defined in a global scope. The operator $\widetilde{\cup}$ transforms merge definitions in ordinary ones: $J \triangleright \mathcal{P} = J \triangleright \mathcal{P}$ and $\widetilde{\mathcal{B}} \cup \mathcal{B}^\prime = \widetilde{\mathcal{B}} \cup \widetilde{\mathcal{B}}^\prime$. If the rule STR-MOVE is also considered for cJoin, then the premise of LOCAL FIRING must be $\widetilde{\mathcal{B}} \cup \mathcal{B} \triangleright \mathcal{S} \rightarrow \widetilde{\mathcal{B}} \cup \mathcal{B} \triangleright \mathcal{S}^\prime$.

Rules LOCAL COMMIT and ABORT handle the termination of a negotiation, whereas MERGE describes the interaction among sibling negotiations. This time, negotiations can be joined only if they do not contain running contracts.

The big-step relation enforces serializability. In fact, the completed negotiations at a particular level become ordinary transitions at the upper level and all interacting transactions can be analysed independently from the rest of the system. The following results states the correspondence between both semantics for shallow processes, proving that shallow processes are serializable.

**Proposition 7.2.** If $\mathcal{D}, \mathcal{S}, \otimes\overset{\kappa}{\mathcal{S}}_k[S_k^\prime : S_k^\prime] \rightarrow^* \mathcal{D}, \otimes\overset{\kappa}{\mathcal{P}}_\kappa : S_\kappa] \rightarrow \mathcal{D}, [\mathcal{P} : \Pi_\kappa S_\kappa]$, then there exists a derivation $\mathcal{D}, \mathcal{S}, \otimes\overset{\kappa}{\mathcal{S}}_k[S_k^\prime : S_k^\prime] \rightarrow^* \mathcal{D}, \otimes\overset{\kappa}{\mathcal{P}}_\kappa : S_\kappa^\prime] \rightarrow \mathcal{D}, [\mathcal{S}^\prime : \Pi_\kappa S_\kappa] \rightarrow^* \mathcal{D}, [\mathcal{P} : \Pi_\kappa S_\kappa]$. 

---

**Figure 7.7:** Big-step semantics of cJoin.
7.5. Encoding Zero-Safe Nets

In this section we show that ZS nets (described in Chapter 3) can be straightforwardly encoded in cJoin. A distributed implementation of ZS nets in Join has been presented in [21], but there the encoding is complicated by the need of attaching a local transaction manager to each transition.

As done in [21], and without loss of generality, we restrict to ZS nets whose transitions have the basic shapes in Figure 7.9.a, for E any stable place and e, e1, e2
any zero places. Moreover we write \texttt{Open} \texttt{e} for \texttt{E}\texttt{}\texttt{e}, \texttt{Fork} \texttt{e}_1, \texttt{e}_2 for \texttt{E}\texttt{e}_1 + \texttt{e}_2, and so on. The translation in Figure 7.9.b associates a \texttt{cJoin} definition to each basic shape. Places are seen as ports and tokens as messages. Tokens in stable places carry no value, while tokens in zero places carry the identifier of the transaction they belong to. The \texttt{cJoin} process associated to the ZS net \((T, S)\) is \texttt{def} \([T]\) \texttt{in} \([S]\), where \([T] = \Lambda_{t \in T}[t]\).

A transaction can be opened by firing a transition of the form \texttt{E open e}. In \texttt{cJoin}, this means opening a new negotiation whose internal state contains the definition of a fresh name \(z\), the message \(e(z)\), and whose compensation, by default, gives back the stable resources \texttt{E}(). The dummy definition \(z \rightarrow 0\) is the \texttt{cJoin} way of declaring a fresh identifier \(z\) for the transaction. When two transactions are merged by applying \(e_1(z_1) | e_2(z_2) \rightarrow e(z_1)\), then \(z_1\) and \(z_2\) become equivalent identifiers for the same larger negotiation. When computing inside a negotiation, each zero token carries one of the possibly many equivalent identifiers for that negotiation (e.g., \(z_1\)). If stable messages \texttt{E}() are released inside the negotiation, e.g., by firing \texttt{e close E}, then they are frozen until commit, because the only rules that can fetch them are outside the negotiation boundaries, in the top chemical soup. The commit can happen if and only if the negotiation reaches a local state containing only stable messages (and dummy definitions). Then, the reaction \texttt{COMMIT} can close the negotiations and release all stable tokens to the environment. Theorems 7.7 and 7.9 state the correctness and completeness of \([\_\_]\). We start by proving some auxiliary Lemmata, which relate ZS movements involving non stable configurations, with reductions in
cJoin.

**Lemma 7.5.** Let \( [z]_w = z \langle w \rangle \) and \( [Z_1 \oplus Z_2]_w = [Z_1]_w \mid [Z_2]_w \). If \((S, Z) \rightarrow_T (S', Z')\) then

\[
\text{def } [T] \text{ in } [S] \mid \Pi_i \text{def } \land_k w_k(\rho) \triangleright 0 \text{ in } [Z_k^i]_{w_k} : \ldots \rightarrow^* \\
\text{def } [T] \text{ in } [S'''] \mid \Pi_j \text{def } \land_j w_j(\rho) \triangleright 0 \text{ in } [Z_j^j]_{w_j} : [S_j'] : \ldots \\
\]

where \( Z = \bigcup_i \bigcup_k Z_i^k \), \( Z' = \bigcup_j \bigcup_j Z_j^j \), \( S' = \bigcup_j S_j' \cup S'' \).

**Proof.** By induction on the structure of the proof. The base case corresponds to rule firing and follows by analysing the shape of the fired transition (according to Figure 7.9.a.). The inductive step follows by rule case analysis. Rule step corresponds to parallel computations in CHAM. The proof for CONCATENATION follows by applying inductive hypothesis on both premises obtaining two derivations and noting that, by CHAM semantics, a derivation is also valid when processes are added into a solution, i.e., if

\[
\text{def } [T] \text{ in } [S_1] \mid \Pi_i \text{def } \land_k w_k(\rho) \triangleright 0 \text{ in } [Z_k^i]_{w_k} : \ldots \rightarrow^* \\
\text{def } [T] \text{ in } [S''''] \mid \Pi_j \text{def } \land_j w_j(\rho) \triangleright 0 \text{ in } [Z_j^j]_{w_j} : [S_j'] : \ldots \\
\]

then also the following derivation is possible:

\[
\text{def } [T] \text{ in } [S_2] \mid [S_1] \mid \Pi_i \text{def } \land_k w_k(\rho) \triangleright 0 \text{ in } [Z_k^i]_{w_k} : \ldots \rightarrow^* \\
\text{def } [T] \text{ in } [S_2] \mid [S_1] \mid \Pi_j \text{def } \land_j w_j(\rho) \triangleright 0 \text{ in } [Z_j^j]_{w_j} : [S_j'] : \ldots \\
\]

Analogously, \( S_j' \) and \( S_1''' \) can be added to the derivation associated to the second premise. Finally, the derivation corresponding to the whole proof can be constructed by combining sequentially the two extended derivations. \( \square \)

**Corollary 7.6.** If \((S, \emptyset) \rightarrow_T (S', \emptyset)\) then \( \text{def } [T] \text{ in } [S] \rightarrow^* \text{ def } [T] \text{ in } [S'] \).

**Proof.** By Lemma 7.5, there exists a derivation for \((S, \emptyset) \rightarrow_T (S', \emptyset)\) containing negotiations. Nevertheless, those negotiations do not contain messages to ports encoding zero-safe places, and therefore they can commit and release all messages to ports denoting stable places. Consequently, rule COMMIT can be applied repeatedly to finish all running negotiations. \( \square \)

**Lemma 7.7.** If there exists \( \text{def } [T] \text{ in } [S''''] \mid \Pi_i \text{def } \land_k w_k(\rho) \triangleright 0 \text{ in } [Z_k^i]_{w_k} : [S_i] : \ldots \rightarrow^* \\
\text{def } [T] \text{ in } [S''''] \mid \Pi_j \text{def } \land_j w_j(\rho) \triangleright 0 \text{ in } [Z_j^j]_{w_j} : [S_j'] : \ldots \) without using rule (COMMIT)

then \((S, Z) \rightarrow_T (S', Z')\), where \( Z = \bigcup_i \bigcup_k Z_i^k \), \( Z' = \bigcup_j \bigcup_j Z_j^j \), \( S = \bigcup_i S_i \cup S'' \) and \( S' = \bigcup_j S_j' \cup S''' \).

**Proof.** By induction on the length of the derivation. The base case corresponds to an idle movement of the net. The inductive step follows by analysing the last applied rule in the reduction. For RED and MERGE, the proof is constructed by using inductive hypothesis on the initial derivation until the last step, and also for the final step. Then by analysing the different shapes of the definition used for the last reduction (Figure 7.9) conclude that the only possible cases correspond to rule CONCATENATION. Note that, there are no ABORT steps, because the encoding cannot generate process abort. \( \square \)
Theorem 7.8. If \( \text{def} \ [T] \text{ in } [S] \rightarrow^* \text{def} \ [T] \text{ in } [S'] \) then \((S, \emptyset) \rightarrow_T^* (S', \emptyset)\).

Proof. The proof follows by using the results presented in Section 7.4. Note that all definitions generated by the encoding are shallow, and consequently negotiations are serializable by Theorem 7.4. Therefore, it is enough to consider reductions where steps from different transactions are not interleaved. Consequently, the whole reduction is equivalent to a sequence that computes transactions one at a time, i.e., it can be decomposed in sub-sequences of the form \( \text{def} \ [T] \text{ in } [S_i] \rightarrow^* \text{def} \ [T] \text{ in } [S_{i+1}] \), where COMMIT can be applied only as the last step. By Lemma 7.7, any sub-reduction without considering the application of COMMIT corresponds to a movement \((S_i, \emptyset) \rightarrow_T (S_{i+1}, \emptyset)\) on the zs net. Hence, any of this sub-reductions corresponds to a transaction \((S_i, \emptyset) \rightarrow_T (S_{i+1}, \emptyset)\). Consequently, the whole reduction is a sequence of transactions on the zs net.

\[ \square \]

Theorem 7.9. \((S, \emptyset) \rightarrow_T^* (S', \emptyset)\) iff \( \text{def} \ [T] \text{ in } [S] \rightarrow^* \text{def} \ [T] \text{ in } [S'] \).

Proof. If follows from Corollary 7.6 and Lemma 7.8

\[ \square \]

The main behavioural difference between the cJoin encoding in Figure 7.9 and the Join encoding in [21] relies on the treatment of failures, as here no abort can be generated (and consequently compensations cannot be activated). A possible solution would be to add a timeout component each time a new negotiation is open, which is able to produce the abort via the rule \( A = \text{timeout}() \triangleright \text{abort} \). In this case, we should let \( [E \text{ open } e] = E() \triangleright [\text{def } z() \triangleright 0 \text{ in } e(z) | \text{timeout } : E()] \) and encode the net \( N = (T, S) \) as \( \text{def } A \land [T] \text{ in } [S] \).

### 7.6 Related works

In this section we compare the primitives of cJoin with other approaches in the literature that extend process calculi with commit mechanisms. We organise the presentation by taking into account the model of transactions that each extension provides.

**Atomic transactions.** Several proposals extends Linda [56] with atomic transactions. Process in Linda communicates (asynchronously) by inserting, consuming, and testing for the presence of data in a shared repository, called the tuple space. Basically, Linda agents are built on the following primitives: (1) \( \text{out} \) for the output of the message \( \langle a \rangle \); (2) \( \text{rd} \) for the reading of a message from the tuple space (but without consuming it); and (3) \( \text{in} \) for the fetching of a message, after which the message is no longer available in the tuple space. Two additional predicates \( \text{rdp} \) and \( \text{inp} \) allow, respectively, for (4) checking for the presence of a message without consuming it; and (5) atomically testing and consume the message if present. Any Linda program is written by composing in sequence and/or parallel the above primitives and a tuple space. For instance, the program

\[ \text{in}(a). \text{in}(a). \text{out}(a).0 \mid \text{in}(a). \text{out}(a). \text{out}(a).0 \mid \langle a \rangle \]
stands for two parallel agents: one reads $a$ twice from the tuple space and then writes $a$, and the other reads $a$ and then write two copies in the tuple space. The term $\langle a \rangle$ stands for a tuple space that contains only $a$. Note that if the second agent starts by consuming $a$, then it produces two copies of $\langle a \rangle$ and terminates. Therefore the first agent may consume the two produced messages and then it also terminates. Differently, if the initial message $a$ is fetched by the first agent, both agents are blocked. This simple example shows that agents are non-atomic (just the basic primitives (1)–(5) are atomics). Moreover, the core language does not provide any primitive for combining actions that have to be executed atomically.

In what follows, we report on four different extensions that add transaction primitives to Linda.

The first extension has been proposed in [2], and is largely inspired by the traditional way of demarcating transactions in databases, i.e. by using pairs begin / commit. The proposed calculus, called Persistent Linda (PLinda), relies on making atomic the operation eval of Linda. The prefix eval($f(x)$) forks a new thread to evaluate $f$ for the value $x$. When the calculation of $f$ terminates, the value $f(x)$ is added to the tuple space. PLinda, defines the corresponding prefix xeval($f(x)$), meaning that the evaluation of $f$ on $x$ is transactional, i.e., isolated from others and have to terminate. If the evaluation aborts, either because it executes the spontaneous operation xabort, or because it is deadlocked, any modification is rolled back. Operations xeval may have a second argument, called the on – abort, which is the process to be run if the xeval operation fails. PLinda provides a second kind of transactions, that begin when a xstart command is executed and end with a xcommit or xabort command. If the transaction ends with xcommit, then all modifications to the tuple space are made durable, while if the transaction ends with xabort all modifications are rolled back.

In spirit, transactions in PLinda are cJoin negotiations that do not use merge capabilities neither nesting. That is processes that run in isolation until completing. If they commit, changes are made available to the system. The difference is when a transaction abort, since in PLinda there exists an automatic roll-back mechanism, which is not in cJoin. Note that on-abort programs are analogous to compensations in cJoin.

The main drawback of this proposal is that primitives are informally discussed, and no formal definition of the locking and roll-back mechanisms used to achieve the behaviour of transactional primitives is presented. The following extensions, although similar in spirit, are aimed at providing a formal account for such mechanisms. First of all, [33, 34] enriches Linda with two new primitives create($x$) and commit($x$). When a process create($x$).$P$ is executed, it creates a new transaction uniquely characterised by a fresh transaction identifier $y$, and becomes $P[y/x]$ executed under the control of a transaction monitor (note that the name $x$ in $P$ is substituted by the fresh identifier $y$). A transaction named $y$ can commit when all participating threads are messages $M$ or have the form commit($y$).$P$. At commit time all prefixes commit($y$) are consumed and all produced processes are released
together with pending messages.

The proposed primitives are aimed at forbidding steps of one agent to interact with the steps of others. Consider the following program (taken from [33]),

\[ (a) \mid create(x).rd(a).in(b).commit(x).0 \mid create(y).in(a).out(b).commit(y).P \]

containing a datum \( a \), a transaction \( x \) which reads \( a \) and then consumes \( b \), and a transaction \( y \) which removes \( a \) and then produces \( b \). Since both transactions are intended to be atomic, none computation can allow \( x \) to complete; if \( x \) executes first it blocks after reading \( a \) because \( b \) is not available, differently if \( y \) executes first, then \( x \) is blocked because \( a \) has been consumed by \( y \). The pair \( create / commit \) forbids the following interleaving steps from \( x \) and \( y \): the datum \( a \) is first read by \( x \), and then consumed by \( y \); after, the datum \( b \) is first produced by \( y \) and then consumed by \( x \).

Differently from cJoin, transactions in [33] have just one entry point, but they can have several exit points, which are coordinated by the transaction monitor. In particular, all exit threads should arrive to the commit decision, i.e., they should terminate. Nesting, although being allowed syntactically, is forbidden by the semantics. In fact, it is not possible to open a new transaction inside a running transaction. Similarly to cJoin, data (i.e., messages) that are not consumed internally by a transaction are made available to the environment only at commit time. Nevertheless there are several differences among the rules that discipline the access to local and global data. In particular, messages produced by a transaction can be consumed only by processes participating in the same transaction, i.e., there is no merge capability. Additionally, a transaction can read or input from the environment during its execution. This behaviour is not allowed in cJoin, since global resources are part of the stable state, and cannot be manipulated by an executing transaction, i.e., all global tokens needed by a negotiation are consumed at the beginning, or alternatively, a transaction must be defined as several transactions that start independently and then are merged.

The model in [33] has been proposed in order to investigate the serializability of transactions in JavaSpaces [103, 32], a coordination middleware produced by Sun Microsystems, which provides the generative communication operations of Linda plus event notification mechanisms of control-driven models [3]. As a main contribution, [103] guarantees that JavaSpaces transactions are serializable when processes are built-up only by using Linda primitives \( out(a) \), \( rd(a) \) and \( in(a) \). Nevertheless, if predicates \( rdp(a) \) and \( inp(a) \) are also considered, serializability does not hold in JavaSpaces. The work proposes also alternative semantics for serializable transactions in JavaSpaces.

Also aimed at modelling JavaSpaces transactions, the third extension proposes nested transactions and an optimistic transactional facilities [71]. In this case, every transaction conceptually operates over its own local copy of the tuple space and performs actions restricted to this copy. When a transaction is ready to commit,
the local state of the tuple space is compared with the global space. If serializability invariants have been violated, the transaction aborts and its local copy is discarded. Otherwise, the changes can be propagated to the global space. At the syntactical level two new primitives are introduced, \texttt{start()} to begin a transaction and \texttt{commit()} to terminate the transaction. The operational semantics of the language makes explicit the association of a transactions with a \textit{log}. In particular, when a transaction \( t \) is started a new log \( \ell \) is created for that transaction. After that, any time \( t \) accesses the tuple space, the log \( \ell \) is updated with the operation done over the tuple space. When \( t \) commits, the log \( \ell \) is used to update the global space. Therefore, the transactional mechanism relies more on log management than on coordination primitives.

The proposal in \cite{71} is aimed at providing an optimistic implementation for (a nested version of) \cite{33}. Consider, for instance, the following process

\[ \texttt{start().out}(a).P \mid \texttt{start().in}(a).Q \]

The semantics proposed in \cite{33}, makes the data \( a \) produced by the first transaction to be unavailable to the second until \( P \) commits. The optimistic version of this model allows the second transaction to use \( a \) when \( P \) is still executing, but the second transaction can commit only if the first one commits. Clearly, the transactional model (when nesting is not considered) is the same on both proposals. A subtle difference is that in \cite{33} a transaction commits only when all agents in it commit, while in \cite{71} a transaction can commit as soon as one agent commits. In fact, the rule that defines the semantics of commit for a transaction \( t \) is

\[ t[\text{commit()}]\mid P \rightarrow 0 \]

Moreover, this rule applies also when \( P \) contains transactions. This is a non-usual behaviour for nesting, because a committing parent preempts the execution of its children. This is a clear distinction with cJoin. This model has been generalised in \cite{69} by presenting a framework, called tfj, for specifying the semantics of transactional mechanisms. The different flavours of transaction implementations rely on defining a different semantics for the operations that handle the logs and the global space updating.

The last extension of Linda we present is called Tralinda \cite{29,30} and has been inspired by the ZS approach (summarised in Chapter 3). In fact, Tralinda relies on a typing mechanism for partitioning messages into two classes, low-level and high-level. Low-level messages are introduced to coordinate the exchange of information between agents during a transaction. Low-level messages can exist only inside a transaction and their lifetime lasts as long as necessary for their producers and consumers to agree on some decision. On the contrary, high-level messages (also called \textit{stable}) represent the observable state of the system. Moreover, Tralinda provides a new kind of prefixes, called \textit{atomic}, that allows for grouping sequences of actions to be performed inside a transaction. Consider the following example (borrowed from \cite{29}).
An agent $P$ can activate the program $Q$ after reading all the data $a_1, \ldots, a_n$, regardless of the order in which they are consumed. $P$ can be written as the parallel composition of $n+1$ processes $\text{in}(a_1) \cdot \text{put}(z) \cdot 0, \text{in}(a_2) \cdot \text{put}(z) \cdot 0, \ldots, \text{in}(a_n) \cdot \text{put}(z) \cdot 0, \text{in}(z) \ldots \text{in}(z) \cdot Q$, where all $a_i$'s are stable, $z$ is low-level, and $\underline{\_}$ denotes the atomic prefixing, with $Q$ prefixed $n$ times by $\text{in}(z)$. Then, the transaction starts as soon as any $a_i$ is consumed and the atomic prefixing forces the output of a low-level message $z$. Moreover, the transaction can commit only when $n$ messages $z$ have been produced, which the atomic prefix of $Q$ can consume, otherwise it will fail. Hence $Q$ cannot be activated unless all the $a_i$'s have been retrieved.

The transaction model of Traínda is close to that of cJoin, in the sense that transactions are multiway, commit means consumption of low-level tokens, while high-level data produced inside a transaction is available at commit time. The main difference relies on the fact that, although both models rely on a distinction among transactional and stable resources, transactions Traínda are defined by typing resources, while in cJoin they are syntactic entities. Moreover, cJoin accounts for name mobility, nesting and programmed compensations, all features not present in Traínda.

A different line of work about atomic commit is presented in [15], where the concept of synchronous rendezvous in process calculi is characterised as a special case of atomic commit. This connection induces a generalisation of the rendezvous mechanism where multiple parties (instead of just sender and receiver) can participate. The proposed calculus is basically the synchronous $\pi$-calculus [88] where input prefixes are join-patterns.

A different perspective on transactional process is given by the work on Reversible Communicating Systems (rCCS) [45], where a notion of distributed backtracking is built on top of CCS [87]. The main idea is that processes behaves as usual when computing forward, but any time a thread synchronises with other threads or when it forks, then it records this information in a local memory. If later the thread decides to roll-back a step, it has to synchronise with the corresponding partner that has also to roll-back. Moreover, rCCS is equipped with a basic commit mechanism that allows to insert check points in the computation, in order to stop roll-backs. This simple mechanism can be though as a basic model of transactions that takes place between two check points.

**Long-running transactions.** Several proposals extends the $\pi$-calculus [88] with compensable processes. The first proposal we discuss is called $\pi t$ [14]. The transaction model of $\pi t$ has been largely inspired by the long running transactions of XLANG [104] (the underlying model of BizTalk [80], which is a commercial workflow system — see Section 1.2.2). The $\pi t$ extends the asynchronous version of $\pi$ [68] with nested transactional contexts (i.e., nested long running transactions). A long running transaction is represented by a process context $(P, P_1, P_0)$, where $P$ denotes the activity to executed until commit, $P_1$ is exception to be activated if
$P$ fails, and $P_c$ is the compensation, which has to be executed if $P$ completes and the parent transaction aborts. Additionally, *done* and *abort* denote respectively a successful and faulty termination of the normal process $P$ of a transactional context $\text{context}(P, R, P_c)$. Basically, $\pi t$ is a calculus for compensations. In fact, it mainly aims at providing an accurate semantics for the dynamic construction of exception handlers to be executed when an abort condition is raised in a nested hierarchy of transactions.

Similarly to $\text{cJoin}$, $\pi t$ transactions commit only when all branches have terminated. Differently, in $\pi t$ transactions have only one entry point (although they may have several exit points). Also, transactional contexts are not isolated. In fact, messages produced in a transaction are made immediately available to the outside. The model does not impose constraint on the commit dependency of interacting contexts, i.e., it could be the case that the sender of a message aborts and the consumer commits.

A timed version of $\pi t$ transactions, called $\text{web}\pi$, has been proposed in [79]. The main contribution of this approach is the formalisation of transactions that are aborted when they are unable to complete within a period of time. Clearly, this is an aspect not present in $\text{cJoin}$.

A flexible scheme of compensable transactions is formalised in [13], which is based on the *cohesor* model [44]. Transactions in the cohesor model are nested and compensable (i.e., they have an associated ad hoc processes that reverse the effects of the transaction). A parent transaction maintains a list of sub-transactions that necessary should commit. That is, a parent can commit only if all specified children have committed. Moreover, transactions can selectively decide to compensate some sub-transactions. This work does not provide a formal high-level description of the semantics of cohesors. Instead it describes the commit protocol that coordinates the execution of cohesors.

We remark that cohesors are a more flexible scheme for handling nesting that the uniform treatment provided by $\text{cJoin}$ negotiations. Nevertheless, since the work does not provide a language, it is difficult to figure out how a programming language with cohesors would look like.

**Frameworks of multiple models of transactions** The first proposals that fall into this category are the pik and pike calculi [46]. Those calculi show how various transaction facilities can be built up by adding a notion of storage to a process calculus like $\pi$. In particular, the several flavours of transactions relies on the operations that handle log entries (both queries and updates). The negative aspect of these approaches is that they move the problem of coordination of distributed process execution to the coordination of atomic update of logs. Another approach in this line is [69], which has been mentioned before.
Chapter 8

Flat cJoin in Join

In this chapter we focus on the implementation of cJoin primitives. The ultimate goal of this activity is to have a running implementation of the commit primitives. The two crucial points when implementing cJoin are that: (1) the commit/abort of a transaction that is the result of merging several independent negotiations is a global decision, and (2) the number of participants (i.e., agents) and their identities are not known statically. We show that, for a significant fragment of cJoin (i.e., where processes have no nested transactions), global decisions can be implemented in a fully distributed way by using a modified version of distributed two phase commit protocol (d2PC) proposed in [21] for implementing zero-safe nets [27], and recalled in Section 2.4.

First of all we characterise the sub-calculus of processes without nested transactions, called flat cJoin, through a type system. Moreover, to facilitate the definition of the implementation, we show that flat processes can be written in a suitable canonical form, where only a few elementary definition patterns are allowed. This can be done without loss of generality, as we show that any flat process can be transformed in an equivalent process in canonical form. The elementary definition patterns we consider are inspired by the basic shapes of transitions in zero-safe nets: they are obtained by imposing a strict bound on the number of messages that can be consumed / produced within a single reduction.

Finally, we show that canonical cJoin processes can be written as equivalent Join processes. Although, we show that Join is expressive enough to encode flat cJoin, i.e., that the new primitives for flat negotiations do not increase the expressivity of the language, we argue that the syntax of cJoin yields a separation of concerns that is difficult to achieve at the level of Join, thus cJoin facilitates programming and reasoning about distributed contracts. Therefore, we show in Section 8.5 an extension of the JoCaml language, that allows us to write programs with flat cJoin transactions.
8.1 A type system for flat cJoin

We single out flat processes of cJoin with a type system. We consider the set
\( T = \{ \Box_0, \Box_1, \Box_2 \} \) of types and use the following type judgements:
\[ \vdash P : \Box_0 \quad \text{The constructor of negotiations} \quad [\cdot] \text{ does not appear at all in } P. \]
\[ \vdash P : \Box_1 \quad \text{P does not contain active negotiations but can activate flat contracts.} \]
\[ \vdash P : \Box_2 \quad \text{P can have or generate flat negotiations but not nested ones.} \]
\[ \vdash D : \Box_0 \quad \text{D does not contain constructors for negotiations.} \]
\[ \vdash D : \Box_1 \quad \text{D can contain or initiate flat negotiations but not nested ones.} \]

**Definition 8.1** (Flat (or well-typed) definitions and processes). A definition \( D \) is said flat or well-typed if \( \vdash D : \Box_1 \) in the type system shown in Figure 8.1. Similarly, a process \( P \) is said flat or well-typed if \( \vdash P : \Box_2 \).

We comment the typing rules in Figure 8.1. Rules (SUB-P) and (SUB-D) stand for the sub-type order \( \Box_0 < \Box_1 < \Box_2 \). Clearly, the inert process \( 0 \), the emission of a message \( x\langle y \rangle \) and the constant abort do not contain constructors for negotiations, and are typed \( \Box_0 \) (Rules ZERO, MESS, ABORT). By rule (PAR), the parallel composition \( P|Q \) can be typed \( \Box_i \) if both \( P \) and \( Q \) type \( \Box_i \). Consequently, the type of \( P|Q \) corresponds to the greatest of the lower types that can be assigned to \( P \) and \( Q \). In fact, considering \( P \) and \( Q \) well-typed, if \( P \) contains an active negotiation (i.e., \( \vdash P : \Box_2 \)), independently of the structure of \( Q \), the process \( P|Q \) contains an active contract (i.e., \( \vdash P|Q : \Box_2 \)). Rule (NEGOT) prevents nesting by stating that \( [P : Q] \) can be typed \( \Box_2 \) only when \( P \) does not have negotiations (i.e., \( \vdash P : \Box_0 \)). Instead, the compensation \( Q \) can use negotiations in definitions. This will not compromise flat condition because compensations execute at the top-level and not inside the negotiations they are originated from. Rule (DEF) combines the typing of definitions and processes. Note that def \( D \) in \( P \) can be typed \( \Box_0 \) only if neither \( D \) nor \( P \) use constructors for negotiations, i.e., if both have type \( \Box_0 \). Instead, it can be typed \( \Box_1 \) when negotiations appear only in definitions (\( D \) or those contained in \( P \)). Finally, if def \( D \) in \( P \) types \( \Box_2 \), its active negotiations appear in \( P \), which therefore types \( \Box_2 \).

By rule (CONJ), a conjunction of definitions is typed \( \Box_i \) only when both sub-terms type \( \Box_i \). By rules (ORD) and (ORD-0), an ordinary definition \( J \triangleright P \) is well-typed when its guarded processes \( P \) is well-typed. Moreover, it has type \( \Box_0 \) if \( P \) does not contain constructors for negotiations (i.e., \( \vdash P : \Box_0 \)). Differently, a merge rule is well-typed only if \( P \) has type \( \Box_0 \) (rule (MERGE)). This is required in order to avoid nesting, because the instances of \( P \) will execute inside a negotiation.

**Example 8.1. Well-typed terms.** Consider the mailing list process introduced in Example 7.2. Several sub-terms and their types are below:

\[
\begin{align*}
P_1 &= \text{def } x\langle \rangle \triangleright 0 \text{ in } y\langle v, z \rangle \mid l\langle y \rangle \\
D_1 &= l\langle y \rangle \mid \text{tell}(v) \triangleright P_2 \\
\vdash P_1 : \Box_0 & \quad \vdash P_2 : \Box_2 \\
\vdash D_1 : \Box_1 & \\
D_2 &= l\langle y \rangle \mid \text{close}(\) \triangleright 0 \\
\vdash P_2 : \Box_2 & \\
\vdash D_2 : \Box_0 & \quad \vdash D_1 \land D_2 : \Box_1
\end{align*}
\]
Moreover, \( \vdash \text{MLDef} : \square_1 \) (it does not have active negotiations but can initiate them), and also \( \vdash \text{ML} : \square_1 \).

**Example 8.2. Counterexample.** Process \( \text{def } \chi() \triangleright [P : 0] \text{ in } [\text{def } D \text{ in } \chi()] : 0 \) is not well-typed because it has a merge definition whose guarded process is a negotiation (rule \( \text{(MERGE)} \)) cannot be applied because \( \not\vdash \chi() \triangleright [P : 0] : \square_0 \). In fact, it can reduce in one step to \( \text{def } \chi() \triangleright [P : 0] \text{ in } [\text{def } D \text{ in } [P : 0] : 0] \) when \( x \notin dn(D) \), which has nested contracts.

### 8.1.1 Properties of the type system

In this section we show several interesting properties of well-typed terms that will be used in Section 8.2. The main results are that \( \text{Join} \) processes type \( \square_0 \) (Proposition 8.1) and subject reduction holds for \( \square_1 \) (Lemma 8.5) and \( \square_2 \) (Lemma 8.6).

**Proposition 8.1 (Join processes type \( \square_0 \)).** Let \( P \) be a \( \text{Join} \) process, then \( \vdash P : \square_0 \).

**Proof.** The proof follows by showing (by proof induction) that for any \( \text{Join} \) definition \( D \) in which all guarded processes has type \( \square_0 \), then \( \vdash D : \text{Join} \) holds. Then proof follows by induction on the structure of the proof \( \vdash P : \square_0 \).

In order to prove subject reduction we need some technical preliminaries. In particular, we extend the typing reduction from processes to solutions.

**Definition 8.2 (Type of a solution).** The type \( \tau \) of a solution \( S \), noted as \( S : \tau \), is defined by rules in Figure 8.2. Moreover, \( S \) is flat iff \( S : \square_2 \).

We start by proving that Definition 8.2 is consistent w.r.t. structural congruence of solutions, i.e. all types are preserved by \( \alpha \)-conversion and heating / cooling; and that the type of a solution reflects on the type of its molecules.

**Proposition 8.2.** Let \( \sigma \) be a renaming substitution. If \( \vdash \tau \) then \( \vdash P\sigma : \tau \).
Figure 8.2: Flat Solution Typing.

**Proof.** Immediate by the fact that typing does not take into account names, but just the structure of terms, which cannot be changed by renaming substitutions.

**Lemma 8.3.** Let $S : \square_j$. If $\{S\} \Rightarrow \{S'\}$ then $S' : \square_j$.

**Proof.** By straightforward case analysis on the applied cooling/heating rule. When the applied rule is \textsc{(Str-Def)}, then the Proposition 8.2 is used.

**Corollary 8.4.** Let $\{S\} \Rightarrow^* \{\otimes_i m_i\}$. Then $S : \square_j \iff \forall i : m_i : \square_{j'}$ and $j' \leq j$.

We are now ready to prove subject reduction for $\square_0$.

**Lemma 8.5** (Subject Reduction for $\square_0$). Let $P : \square_0$. If $P \Rightarrow^* P'$ then $P' : \square_0$.

**Proof.** The proof follows by induction on the length of the derivation.

- **Base case:** Trivial, because $P' \equiv P$.

- **Inductive Step:** Suppose $P \Rightarrow P'' \Rightarrow^* P'$. The proof follows by case analysis of the first applied rule and inductive hypothesis. Note that if $P \Rightarrow P''$ then $\{P\} \Rightarrow^* \{S\} \Rightarrow \{S''\} \Rightarrow^* \{P''\}$. From Corollary 8.4, $S \equiv \otimes_i m_i$, with $m_i : \square_0$. Hence the only possible rule that can be applied is \textsc{(Red)}, because any other rule requires at least a molecule composed by a membrane, which cannot be typed $\square_0$. Consequently, $S \equiv J \triangleright Q, J \sigma, S''$ and $S' \equiv J \triangleright Q, Q \sigma, S''$, where $S'' : \square_0$. As $Q : \square_0$, by Proposition 8.2, $Q \sigma : \square_0$. Hence $S' : \square_0$ and therefore (by Corollary 8.4) $P'' \Rightarrow^* P'$.

The following result assures that flat processes do not introduce nesting.

**Theorem 8.6** (Subject Reduction for $\square_2$). Let $P : \square_2$. If $P \Rightarrow^* P'$ then $P' : \square_2$.

**Proof.** The proof follows by induction on the length of the derivation.

- **Base case:** Trivial, because $P' \equiv P$.

- **Inductive Step:** Suppose $P \Rightarrow P'' \Rightarrow^* P'$. The proof follows by case analysis of the first applied rule and inductive hypothesis. Note that if $P \Rightarrow P''$ then $\{P\} \Rightarrow^* \{S\} \Rightarrow \{S''\} \Rightarrow^* \{P''\}$. By Corollary 8.4 $S \equiv \otimes_i m_i$ и $m_i : \square_j$ where $j \leq 2$. There are four cases:
8.2. A CANONICAL FORM FOR FLAT PROCESSES

\[ \text{count}(0) = 1 \quad \text{count}(\text{abort}) = 1 \quad \text{count}(x(\bar{u})) = \text{count}(\text{def } D \text{ in } P) = \text{count}(P) \]

\[ \text{count}(P | Q) = \text{count}(P) + \text{count}(Q) \]

\[ \text{count}([P : Q]) = \text{count}(P) \]

Figure 8.3: Definition of \text{count}(P).

- Rule (\text{RED}): When the reduction occurs at top-level, i.e. \( S \equiv J \triangleright Q, J \sigma, S'' \), \( S' \equiv J \triangleright Q, Q \sigma, S'' \), and \( S'' : \Box_2 \), the proof is similar to Lemma 8.5. The other possibility is that the reduction occurs inside a transaction, e.g. \( S \equiv [S_1], S'' \) and \( S' \equiv [S'_1], S'' \), where \([S_1] \rightarrow [S'_1]\) by applying (\text{RED}) and \( S'' : \Box_2 \). Note that \([S_1] : \Box_2\), and therefore \( S_1 : \Box_0 \). By Lemma 8.5, \( S'_1 : \Box_0 \) and hence \( S' : \Box_2 \).

(The cases below occur at top-level, because negotiations cannot be nested in \( P \).)

- Rule (\text{COMMIT}): \( S \equiv [[M|\text{def } D \text{ in } 0], Q], S'' \), and \( S \equiv M, S'' \), with \( S'' : \Box_2 \) (by Corollary 8.4). As \( M \) is the parallel composition of messages, it can be typed \( \Box_0 \) and therefore \( S' : \Box_2 \).

- Rule (\text{ABORT}): \( S \equiv [[\text{abort}|P], Q], S'' \) and \( S \equiv Q, S'' \), with \( S'' : \Box_2 \) (by Corollary 8.4). As \( \rightarrow Q : \Box_0 \), it must be \( Q : \Box_1 \) and therefore \( S' : \Box_2 \).

- Rule (\text{MERGE}): \( S \equiv J_1, \ldots, J_n \triangleright R, \otimes_i[[J_i|S_i, Q_i], S'' \) and \( S' \equiv J_1, \ldots |J_n \triangleright R, \otimes_i[S_i, R_i, Q_i|Q_i], S'' \), with \( S'' : \Box_2 \) (by Corollary 8.4). As \( R : \Box_0 \), and all \( S_i : \Box_0 \), \( Q_i : \Box_1 \), then \( S' : \Box_2 \).

\[ \square \]

Remark 8.1. Subject reduction does not hold for \( \Box_1 \). Consider \( P = \text{def } x(\bar{u}) \triangleright [Q : Q'] \text{ in } x(\bar{u}) \), where \( \vdash Q : \Box_0 \) and \( \vdash Q' : \Box_1 \). Although \( \vdash P : \Box_1 \), \( P \) reduces to \( P' = \text{def } x(\bar{u}) \triangleright [Q : Q'] \text{ in } [Q : Q'] \), which can be typed \( \Box_2 \) but not \( \Box_1 \).

Definition 8.3 (Flat cJoin). Flat cJoin is the sub-calculus of all flat processes.

8.2 A canonical form for flat processes

To simplify the definition of the encoding of flat cJoin into Join, we will restrict our attention to processes built with some basic shapes. In particular, we forbid definitions to consume and produce messages freely. The auxiliary function \text{count} in Figure 8.3 counts the atomic agents present in a process. Note that \text{count}(P) > 0, even though \( P = 0 \). This is because the canonical form concerns to the syntax of processes, and hence we distinguish at the syntactical level \( P \) from \( P|0 \).

Definition 8.4 (Canonical Form). The set \( \mathcal{C}_P \) of canonical flat process is defined inductively by the rules in Figure 8.4. We will say \( P \) is in canonical form if \( P \in \mathcal{C}_P \).

Note that 0 and \( x(\bar{u}) \) and \text{abort} are in canonical form (rules CF-ZERO, CF-MESS and CF-ABT). A negotiation \([P : Q]\) is in canonical form if both \( P \) and \( Q \) are
canonical and $P$ has only one active agent (rule CF-NEGOT). By rule (CF-DEF), the process $\textbf{def } D \textbf{ in } P$ is in canonical if $D$ is a canonical definition (the set $C_P$ stands for all definitions in canonical form) and $P$ is a canonical process. The parallel composition $P|Q$ is in canonical form if $P$ and $Q$ are canonical and both have exactly one active agent (rule CF-PAR). As far as definitions are concerned, it is worth noting that the conditions match with the basic shapes of $\alpha$ nets (Figure 7.9). By (CF-OPEN), a canonical reaction that creates a new negotiation consumes exactly one message and produces only one agent inside the new negotiation. Rule (CF-ORD-JOIN) assures that a synchronisation consumes two messages and produces exactly a new agent. Differently, rule (CF-MERGE-JOIN) allows to join several negotiations simultaneously. Moreover, a join cannot spawn directly a new negotiation (a task left to (CF-OPEN)). Finally, rules (CF-ORD-MOV) and (CF-MERGE-MOV) are instances of transitions calc, fork, and close (with drop as a particular case) of $\alpha$ nets.

The following propositions states some interesting properties about canonical processes.

**Proposition 8.7** (Canonical processes are flat). If $P \in C_P$ then $\vdash P : \Box_2$.

**Proof.** By straightforward induction on the proof of $P \in C_P$. □

**Proposition 8.8** (Degree of concurrency). If $P \in C_P$ then $\text{count}(P) \leq 2$.

**Proof.** By straightforward induction on the proof of $P \in C_P$. □
8.3 Writing flat cJoin process in canonical form

In this section we show that any flat process can be written as an equivalent flat process in canonical form. We start by giving a translation \( \llbracket \cdot \rrbracket \) of cJoin processes in canonical processes (Definition 8.5), then we prove that for any cJoin process \( P \), the corresponding \( \llbracket P \rrbracket \) is in canonical form (Lemma 8.15). Finally we show that \( P \approx \llbracket P \rrbracket \) (Theorem 8.25), where \( \approx \) denotes weak barbed bisimilarity (see Definition 2.11).

Definition 8.5 (Canonical transformation). Let \( P \) be a flat process. The canonical process \( \llbracket P \rrbracket \) is given by rules in Figure 8.5.

Clearly, a process in canonical form needs no transformation. The rule for transforming a negotiation \( [P : Q] \) assures that the content of the negotiation is encoded in canonical form and its \textit{count} is equal to 1. This is achieved by generating a private
forwarder \( x \) that activates \( \{P\} \). Similarly, the encoding for the parallel composition \( P \mid Q \) generates two forwarders that activate respectively \( \{P\} \) and \( \{Q\} \). This assures that \( \text{count}(\{P \mid Q\}) = 2 \). When a process is equipped with local definitions both the process and the definitions are transformed. The translation of a definition \( D \) may require the generation of new ports that should be different from other names used in the process. For this reason, a definition \( D \) in \textbf{def} \( D \) in \( P \) is encoded by taking into account the set of names used by the whole process, i.e. \( N = fn(D, P) \cup dn(D) \). The canonical form of \( D \) is defined by using the auxiliary function \( \{D\}_K \) with \( K \) any set of new auxiliary ports (disjoint from \( N \)). Note that this condition is enough to assure that names are fresh in any context, because of the static scoping discipline of cJoin.

The remaining rules assure that definitions are built with the elementary shapes in Figure 8.4. The key aspect is the transformation of general join patterns as a combination of two-way join patterns, a problem already studied in [51] (we use here a slightly different approach). The main idea when translating a general join pattern is that partial joins can be buffered in such a way that roll-back is always possible until the pattern is complete and the guarded process is fired. To this aim, the function \( \{D\}_K \) is used.

The basic definition \( D = x\langle y\rangle | z\rangle \triangleright P \) can be translated as follow by taking \( \{w, z\} \not\subseteq fn(P) \) and different from \( x \) and \( y \) :

\[
\{D\}_{\{w\}} = (x\langle y\rangle | w\rangle \triangleright P)_{\emptyset} \land \{y\langle z\rangle | w\rangle \triangleright \emptyset\}_{\emptyset} \\
= x\langle y\rangle | w\rangle \triangleright \textbf{def} \ z\rangle \triangleright \{P\} \textbf{ in} \ z\rangle \land \ y\langle z\rangle | w\rangle \land \ w\rangle | y\rangle | z\rangle
\]

Note that the internal decision of buffering, i.e. consuming \( y \) and \( z \) and generating \( w \), can always be undone by the rule that regenerates \( y \) and \( z \) from \( w \). Moreover, the guarded process \( 0 \) is activated only when \( x\langle \rangle \), \( y\langle \rangle \) and \( z\rangle \) are present in the solution.

On the contrary, it is not possible to use this approach for encoding merge definitions, because there is no way to roll-back the fusion of negotiations. For this reason we allow multiway patterns for merge definitions.

The encoding presented before is not deterministic and depends on the names selected for forwarders and the set \( K \) used to encode definitions. Nevertheless, all of them are equivalent up-to renaming of defined names (\( \alpha \)-conversion denoted as \( \equiv_{\alpha} \)).

**Proposition 8.10** (Uniqueness). If \( P_1 = \{P\} \) and \( P_2 = \{P\} \) then \( P_1 \equiv_{\alpha} P_2 \).

*Proof.* The proof follows by induction on the applied rule for encoding \( P \). \( \square \)

**Proposition 8.11.** \( \{P_1\}_{\sigma} \equiv_{\alpha} \{P_1\}_{\sigma} \).

*Proof.* The proof follows by straightforward induction on the structure of \( P \). \( \square \)

An interesting property of the encoding is that it preserves types.
Proposition 8.12 (Type preservation). If \( \vdash P : \square_i \) then \( \langle P \rangle : \square_i \).

Proof. By induction on the structure of \( P \). The proof is straightforward since the encoding introduces the operator \( \ulcorner \cdot \urcorner \) neither in processes nor in definitions.

The remain of this section is devoted to show that \( \langle P \rangle \) is in canonical form. We start by introducing some auxiliary results that will be used to prove the Lemma 8.15, which assures \( \langle P \rangle \in \mathcal{C}_P \). Propositions 8.13 and 8.14 state that the encoding of definitions produces canonical definitions provided with the fact that guarded processes are canonical.

Proposition 8.13. Given a definition \( D = J \triangleright z \langle \vec{w} \rangle \), then \( \forall K \text{ s.t. } \langle D \rangle^K \) is defined and \( K \cap dn(J) = \emptyset \) we have \( \langle D \rangle^K \in \mathcal{C}_D \).

Proof. The proof follows by straightforward induction on the length of \( J \).

Proposition 8.14. Let \( D \) be a flat definition (i.e., \( \vdash D : \square_i \)). If any guarded process \( P \) in \( D \) satisfies \( \langle P \rangle \in \mathcal{C}_P \), then \( \forall K \text{ s.t. } \langle D \rangle^K \) is defined: \( \langle D \rangle^K \in \mathcal{C}_D \).

Proof. The proof follows by induction on the structure of \( D \).

- \( D = D_1 \land D_2 \): By inductive hypothesis on both \( D_1 \) and \( D_2 \).

- \( D = J \triangleright P \): By hypothesis \( \langle P \rangle \in \mathcal{C}_P \), then by Proposition 8.7 \( \vdash \langle P \rangle : \square_2 \). For the join pattern \( J \), there are several cases:

  (i) \( J = x \langle \vec{u} \rangle \): If \( \forall \langle P \rangle : \square_1 \), then by Proposition 8.9, \( count(\langle P \rangle) = 1 \). Hence, by rule (CF-OPEN), \( D \in \mathcal{C}_D \). Otherwise, if \( \vdash \langle P \rangle : \square_1 \), then by Proposition 8.8, \( count(\langle P \rangle) \leq 2 \). Hence, by rule (CF-ORD-MOV), \( D \in \mathcal{C}_D \).

  (ii) \( J = x_1 \langle \vec{u}_1 \rangle \mid x_2 \langle \vec{u}_2 \rangle \): As in (i), it can be proved that \( x \langle \rangle \triangleright \langle P \rangle \in \mathcal{C}_D \). Clearly \( x \langle \rangle \in \mathcal{C}_P \) and \( count(x \langle \rangle) = 1 \leq 2 \). Then, by rule (CF-DEF), \( P' = \text{def} \ z \langle \rangle \triangleright \langle P \rangle \in z \langle \rangle \in \mathcal{C}_P \). Clearly, \( \vdash P' : \square_1 \), and \( count(P') = 1 \leq 2 \). Then, the proof \( \langle J \triangleright P \rangle_0 = J \triangleright P' \in \mathcal{C}_D \) is completed by using rule (CF-ORD-JOIN).

  (iii) \( J = x_1 \langle \vec{u}_1 \rangle \mid x_2 \langle \vec{u}_2 \rangle \mid J' \): As in (ii), it can be shown that \( \langle x_1 \langle \vec{u}_1 \rangle \mid x_2 \langle \vec{w} \rangle \triangleright \langle P \rangle \rangle_0 \in \mathcal{C}_D \). By Proposition 8.13, \( \langle x_2 \langle \vec{u}_2 \rangle \mid J \triangleright \langle \vec{w} \rangle \rangle^K \in \mathcal{C}_D \). The proof is completed by using rule (CF-CONJ).

- \( D = J \triangleright P \): By hypothesis \( \langle P \rangle \in \mathcal{C}_P \), then, by Proposition 8.7, \( \vdash \langle P \rangle : \square_2 \). As in previous cases (i.e., (ii)) it can be proved that \( P' = \text{def} \ z \langle \rangle \triangleright \langle P \rangle \) in \( z \langle \rangle \in \mathcal{C}_P \). Moreover, since \( J \triangleright P \) is a flat definition, \( \vdash P : \square_0 \). Hence, by Proposition 8.12, \( \vdash \langle P \rangle : \square_0 \), and consequently \( \vdash P' : \square_0 \). Finally, by rule (CF-MERGE-JOIN), \( J \triangleright P' \in \mathcal{C}_D \).

Finally, the following Lemma states that the encoding produces only canonical processes.

Lemma 8.15. Let \( P \) be a flat process, then \( \langle P \rangle \in \mathcal{C}_P \).

- $P = 0$, $P = x(y)$, $P = \text{abort}$: Immediate because they are in canonical form, and hence $\langle P \rangle = P$.
- $P = P_1 | P_2$: if $P \in \mathcal{C}_P$ then immediate. Otherwise, by inductive hypothesis $\langle P_1 \rangle, \langle P_2 \rangle \in \mathcal{C}_P$. As done in proof of Proposition 8.14, it is easy to show that $x(\langle P_1 \rangle) \triangleright \langle P_2 \rangle \in \mathcal{C}_P$. It can also be proved that $y(\langle P_1 \rangle) \triangleright \langle P_2 \rangle \in \mathcal{C}_P$. Hence $\langle P \rangle \in \mathcal{C}_P$.
- $P = [P_1 : Q]$: By inductive hypothesis $\langle P_1 \rangle, \langle Q \rangle \in \mathcal{C}_P$. As done in proof of Proposition 8.14 it can be shown that $P_1' = \text{def } x(\langle P_1 \rangle) \triangleright \langle Q \rangle \text{ in } x(\langle P_1 \rangle) \in \mathcal{C}_P$. Clearly, $\text{count}(P_1') = 1$. Hence $\langle P \rangle \in \mathcal{C}_P$.
- $P = \text{def } D \text{ in } P_1$: By inductive hypothesis, for all guarded processes $Q$ in $D$ we have $\langle Q \rangle \in \mathcal{C}_P$. Then, by Proposition 8.14, $\langle D \rangle_K \in \mathcal{C}_P$. By inductive hypothesis, $\langle P_1 \rangle \in \mathcal{C}_P$, and finally, by rule (cf-def), $\langle P \rangle \in \mathcal{C}_P$.

\[ \square \]

Remark 8.2. From Lemma 8.15 we have that $\langle \langle P \rangle \rangle = \langle P \rangle$.

### 8.3.1 The canonical form of $P$ is bisimilar to $P$

This section is devoted to prove that the canonical form of a process behaves like the original process. We will use barbed bisimilarity (defined in Section 2.3.4) as the notion of behavioural equivalence.

We start by noting that the encoding in Figure 8.5 introduces several internal steps that are not present in the original process. We will relate a flat process $P$ with its canonical form $\langle P \rangle$ and all processes that can be reached from $\langle P \rangle$ by using only these internal steps. To facilitate the notation of such processes we will annotate basic definitions generated by the encoding.

**Definition 8.6** (Annotated process). An annotated process is a cJoin process, where basic definitions may be underlined as follow $J \triangleright P$.

Note the annotation is done just for simplifying the following definitions, but they do not modify the encoding neither the language.

**Definition 8.7** (Annotated canonical form). Let $P$ be a flat cJoin process. Then, its annotated canonical form $\langle P \rangle$ is recursively defined by rules in Figure 8.6.

Note that the encoding in Figure 8.6 is the same as that in Figure 8.5. The only difference is that the internal definitions added by the encoding are annotated.

**Remark 8.3.** In what follows we assume all canonical forms to be annotated.

**Definition 8.8** (Annotated reduction). The annotated reduction $\rightsquigarrow$ among annotated processes is defined as follow: $P \rightsquigarrow P'$ if either:
8.3. WRITING FLAT CJOIN PROCESS IN CANONICAL FORM

\[ \text{Proceses} \]
\[ \{P\} = P \quad \text{if } P \in C_P \]
\[ \text{Otherwise} \]
\[ \{[P : Q]\} = [\text{def } x(\cdot) \triangleright \{P\} \text{ in } x(\cdot) : \{Q\}] \quad \text{if } x \notin \text{fn}(P) \]
\[ \{P | Q\} = \text{def } x(\cdot) \triangleright \{P\} \land y(\cdot) \triangleright \{Q\} \text{ in } x(\cdot) | y(\cdot) \quad \text{where } \{x, y\} \cap \text{fn}(P|Q) = \emptyset \]
\[ \{\text{def } D \text{ in } P\} = \text{def } \{D\}_K \text{ in } \{P\} \quad \text{where } K \cap (\text{fn}(D, P) \cup \text{dn}(D)) = \emptyset \]

\[ \text{Definitions} \]
\[ \{D \land E\}_{K \lor L} = \{D\}_K \land \{E\}_L \]
\[ \langle J \triangleright P \rangle_0 = J \triangleright \text{def } z(\cdot) \triangleright \{P\} \text{ in } z(\cdot) \quad \text{if } z \notin \text{fn}(P) \]
\[ \langle x(\bar{u}) \triangleright P \rangle_0 = x(\bar{u}) \triangleright \{P\} \]
\[ \langle x(\bar{u}_1) | x_2(\bar{u}_2) \triangleright P \rangle_0 = x(\bar{u}_1) | x_2(\bar{u}_2) \triangleright \text{def } x(\cdot) \triangleright \{P\} \text{ in } x(\cdot) \quad \text{if } x \notin \text{fn}(P) \]
\[ \langle x(\bar{u}_1) | x_2(\bar{u}_2) \rangle_{z; \bar{u}\in K} = \langle x(\bar{u}_1) | x_2(\bar{u}_2) \rangle_{z; \bar{u}\in K} \quad \bar{w} = \bar{u}_2, \text{rn}(J) \]
\[ \langle x(\bar{u}_1) | x(\bar{u}_2) \rangle_{z; \bar{u}\in K} = x(\bar{u}_1) | x(\bar{u}_2) \rangle_{z; \bar{u}\in K} \quad \bar{w} = \bar{u}_1, \bar{u}_2 \]

Figure 8.6: Annotated translation.

(i) \[ P \Rightarrow^* \{[J \triangleright P, J, \sigma, S]\} \rightarrow \{[J \triangleright P, P, \sigma, S]\} \Rightarrow^* P' \]

(ii) If \[ \overset{*}{P} \Rightarrow \{[P_1, S]\}, P_1 \Rightarrow P_1', \text{ and } P' \Rightarrow \{[P_1', S]\}. \]

It should be clear that \( \Rightarrow \subseteq \Rightarrow^* \).

The main result of this section is Theorem 8.25, which states that \( \Rightarrow \) is weak bisimilar to \( \{P\} \). This result is achieved by showing that the relations \( \mathcal{R} = \{(P, Q) | \{P\} \Rightarrow^* Q\} \) and \( \mathcal{R}^- \) are a weak barbed simulations (Lemma 8.23 and 8.24). In order to be able to prove such results, we first introduce the following auxiliary propositions (Propositions 8.16—8.22). First of all we show that all annotated reductions from the encoding of messages \( M \) reduces to \( M \), i.e., \( \{M\} \) is strongly confluent to \( M \).

**Proposition 8.16.** For all \( Q \) s.t. \( \{M\} \Rightarrow^* Q \), then \( Q \Rightarrow^* M \)

**Proof.** By induction on the structure of \( M \). \( \square \)

Then we show that annotated reductions do not execute any rule of the original process. We state this property by assuring that if a process \( P \) contains a set of
messages \( M \), then any process \( Q \) reachable from \( \langle P \rangle \) with annotated reductions (i.e., \( \langle P \rangle \sim^* Q \)) can always reduce to a process that contains also \( M \).

**Proposition 8.17.** Let \( P \equiv_e \text{def } D \in M|P_1 \) be a flat cJoin process and \( \langle P \rangle \equiv_e \text{def } \langle D \rangle_K \wedge x \rangle \triangleright \langle M \rangle \wedge y \rangle \triangleright \langle P_1 \rangle \) in \( x \rangle |y \rangle \) (assuming \( x \) and \( y \) fresh). Then, for all \( Q \) s.t. \( \langle P \rangle \sim^* Q \) all the following conditions hold:

(i) \( Q \equiv_e \text{def } \langle D \rangle_K \wedge D_1 \wedge D_2 \text{ in } M_1|R_1|R_2 \), where \( fn(M_1) \cap dn(D_1 \wedge D_2) = \emptyset \), \( \forall z \in dn(\langle D \rangle_K) : R_1 \vdash z \wedge R_2 \vdash z \).

(ii) \( \text{def } x \rangle \triangleright \langle M \rangle \text{ in } x \rangle \sim^* \text{def } D_1 \text{ in } R_1|M_a \).

(iii) \( \text{def } y \rangle \triangleright \langle P_1 \rangle \text{ in } y \rangle \sim^* \text{def } D_2 \text{ in } R_2|M_b \).

(iv) \( \text{def } \langle D \rangle_K \text{ in } M_a|M_b \sim^* \text{def } \langle D \rangle_K \text{ in } M_1 \).

(v) \( Q \sim^* \text{def } \langle D \rangle_K \text{ in } M|\text{def } D_1 \wedge D_2 \text{ in } R_2|M_b \).

Note that the proposition gives also a characterisation of the structure of \( Q \) (conditions (i)—(iv)) and a way in which it can be obtained. In particular, by reducing the encoding of \( M \) to obtain \( \text{def } D_1 \text{ in } R_1|M_a \), and the encoding of \( P_1 \) to generate \( \text{def } D_2 \text{ in } R_2|M_b \). Finally, \( Q \) is reached by reducing with the encoding of the original definitions \( D \) the messages \( M_a|M_b \). Condition (v) states the confluence of \( Q \) to a process that still has \( M \). Definitions \( D_1 \) and \( D_2 \) are the steps introduced by the encoding.

**Proof.** The proof follows by induction on the length of the derivation.

- **Base case:** \( \langle P \rangle \sim^0 Q \), hence \( Q \equiv_e \langle P \rangle \equiv_e \text{def } \langle D \rangle_K \wedge x \rangle \triangleright \langle M \rangle \wedge y \rangle \triangleright \langle P_1 \rangle \text{ in } x \rangle |y \rangle \).

  It is enough to take \( M_1 \equiv_e M_a \equiv_e M_b \equiv_e 0 \), \( R_1 \equiv_e x \rangle \), \( R_2 \equiv_e y \rangle \), \( D_1 \equiv_e x \rangle \triangleright \langle M \rangle \) and \( D_2 \equiv_e y \rangle \triangleright \langle P_1 \rangle \). Then (i), (ii), (iii) and (iv) immediately follow.

- **Inductive step:** \( \langle P \rangle \sim^* Q' \sim^* Q \). By inductive hypothesis on \( \langle P \rangle \sim^* Q' \), the following conditions hold:

  (i') \( Q' \equiv_e \text{def } \langle D \rangle_K \wedge D_1 \wedge D_2 \text{ in } M'_1|R'_1|R'_2 \), where \( fn(M'_1) \cap dn(D'_1 \wedge D'_2) = \emptyset \), \( \forall z \in dn(\langle D \rangle_K) : R'_1 \vdash z \wedge R'_2 \vdash z \).

  (ii') \( \text{def } x \rangle \triangleright \langle M \rangle \text{ in } x \rangle \sim^* \text{def } D'_1 \text{ in } R'_1|M'_a \).
Then by analysing the last reduction $Q' \rightsquigarrow Q$:

- If the applied rule is in $\langle D' \rangle_K$, then the only possibility is to consume messages in $M_1'$ (because $\forall z \in dn(\langle D' \rangle_K) : R_1 \nvdash R_2 \nvdash z$). Hence,

$$Q \equiv_e \text{def } \langle D' \rangle_K \land D' \land D_2 \text{ in } M_1R_1|R_2$$

where $\text{def } \langle D' \rangle_K \text{ in } M_1'$, which satisfies condition (ii) and (iv). Conditions (ii)–(iii) follow from (ii)'–(iii)'. Condition (v) is immediate by noting that all annotated rules in $\langle D' \rangle_K$ are reversible, in the sense that both $x_1(z_1)z_2(z_2) = x_2(z)$ and $z_1(z_2) = x_1(z_1)z_2$ are in $\langle D' \rangle_K$. Hence $Q' \rightsquigarrow Q \rightsquigarrow Q'$ and then, by (iv'), $Q \rightsquigarrow Q' \rightsquigarrow \text{def } \langle D' \rangle_K \text{ in } M_1 \text{def } D_1 \land D_2 \text{ in } R_1|R_2$.

- If the applied rule is in $D_1'$, then the only possibility is to reduce $R_1'$. Then

$$Q' \rightsquigarrow Q \equiv_e \text{def } \langle D \rangle_K \land D_1 \land D_2 \text{ in } M_1|M_a''|R_1|R_2'$$

where

$$\text{def } x() \lor \langle M \rangle \text{ in } x() \rightsquigarrow \text{def } D_1' \text{ in } R_1'|M_a'' \rightsquigarrow \text{def } D_1 \text{ in } R_1|M_a''|M_a''$$

that satisfies condition (ii). For condition (i), it is enough to consider $M_1 = M_1'|M_a''$. Conditions (iii) and (iv) are obtained from (iii)' and (iv)'. As far as condition (iv) is concerned, note that $\text{def } \langle D' \rangle_K \text{ in } M_1'|M_a'' \rightsquigarrow \text{def } \langle D \rangle_K \text{ in } M_1'$, and hence (since annotated reductions in $\langle D \rangle_K$ are reversible) $\text{def } \langle D \rangle_K \text{ in } M_1' \rightsquigarrow \text{def } \langle D \rangle_K \text{ in } M_1'|M_a''$.

Then

$$Q \equiv_e \text{def } \langle D \rangle_K \land D_1 \land D_2 \text{ in } M_1'|M_a''|R_1|R_2' \rightsquigarrow \text{def } \langle D \rangle_K \land D_1 \land D_2 \text{ in } M_1'|M_a''|M_a''|R_1|R_2' \rightsquigarrow$$

Since $\text{def } x() \lor \langle M \rangle \text{ in } x() \rightsquigarrow \text{def } D_1 \text{ in } R_1|M_a''|M_a''$, from Proposition 8.16 we have that $\text{def } D_1 \text{ in } R_1|M_a''|M_a'' \rightsquigarrow \text{def } D_1 \text{ in } M$. Finally

$$Q \equiv_e \text{def } \langle D \rangle_K \land D_1 \land D_2 \text{ in } M_1'|M_a''|M_a''|R_1|R_2' \rightsquigarrow \text{def } \langle D \rangle_K \land D_1 \land D_2 \text{ in } M|R_2'|M_a.$$
Remark 8.4. Proposition 8.17 assures that for all \( Q \), such that \( \langle P \rangle \leadsto^* Q \), if \( P = \text{def } D \text{ in } M|P' \) then \( Q \leadsto^* \text{def } \langle D \rangle_K \text{ in } M|Q' \) and \( \text{def } \langle D \rangle_K \text{ in } \langle P' \rangle \leadsto^* \text{def } \langle D \rangle_K \text{ in } Q' \) (for a suitable \( K \)).

The following proposition states that a canonical form \( \langle P \rangle \) of \( P \) can reach \textit{abort} with annotated reductions only if \textit{abort} is already present in \( P \).

\textbf{Proposition 8.18.} If \( \langle P \rangle \leadsto^* Q \equiv_e \text{abort}|Q' \), then \( P \equiv_e \text{abort}|P' \), and \( \langle P' \rangle \leadsto^* Q' \).

\textit{Proof.} The proof follows by induction on the structure of \( P \).

- \( P \equiv_e 0 \) and \( P \equiv_e x\langle y \rangle \): Since \( P \in C_P \), \( \langle P \rangle \equiv_e P \) and hence there is no \( Q \).
- \( P \equiv_e \text{abort} \): Since \( \langle P \rangle \equiv_e P \) has no reductions, \( Q \equiv_e \langle P \rangle \equiv_e P \equiv_e \text{abort}|0 \).
- \( P \equiv_e P_1|P_2 \): If \( P \) is already a canonical process, then \( \langle P \rangle \equiv_e P \), which has no annotated definitions and, consequently, no annotated reductions. Hence, the only possibility is \( Q \equiv_e \langle P \rangle \equiv_e P \), and hence \( Q \equiv_e P \equiv_e \text{abort}|Q' \). Otherwise, i.e. \( P \) is not canonical, we have

\[
\langle P \rangle \equiv_e \text{def } x\langle \rangle \triangleright \langle P_1 \rangle \wedge y\langle \rangle \triangleright \langle P_2 \rangle \text{ in } x\langle \rangle y\langle \rangle
\]

Hence all possible \( R \) s.t. \( \langle P \rangle \leadsto^* R \) are as follow:

1. \( R \equiv_e \langle P \rangle \): In this case \( R \not\equiv_e \text{abort}|Q' \). Hence \( R \) is not a possible \( Q \).
2. \( R \equiv_e \text{def } x\langle \rangle \triangleright \langle P_1 \rangle \text{ in } x\langle \rangle |Q_2 \), with \( \langle P_2 \rangle \leadsto^* Q_2 \): If \( R \equiv_e \text{abort}|Q' \), then the only possibility is \( Q_2 \equiv_e \text{abort}|Q_2 ' \). Hence, by inductive hypothesis on \( \langle P_2 \rangle \leadsto^* Q_2 ', P_2 \equiv_e \text{abort}|P_2 ' \) and \( \langle P_2 ' \rangle \leadsto^* Q_2 ' \). Then,

\[
P \equiv_e P_1|P_2 \equiv_e P_1|(\text{abort}|P_2 ') \equiv_e \text{abort}|(P_1|P_2 ')
\]

and, clearly, \( \langle P_1|P_2 ' \rangle \leadsto^* \text{def } x\langle \rangle \triangleright \langle P_1 \rangle \text{ in } x\langle \rangle Q_2 ' \).
3. \( R \equiv_e \text{def } y\langle \rangle \triangleright \langle P_2 \rangle \text{ in } Q_1 |x\langle \rangle \), with \( \langle P_1 \rangle \leadsto^* Q_1 \): The proof of this case is analogous to the previous one.
4. \( R \equiv_e Q_1'|Q_2 \), with \( \langle P_1 \rangle \leadsto^* Q_1 \) and \( \langle P_2 \rangle \leadsto^* Q_2 \): There are two cases \( Q_1 \equiv_e \text{abort}|Q_1 ' \) or \( Q_2 \equiv_e \text{abort}|Q_2 ' \). Each of them follow as in case 2.

- \( P \equiv_e [P_1 : P_2] \): The canonical form of \( P \) (even when it is already a canonical processes) is a negotiation, \( \langle P \rangle \equiv_e [Q_1 : Q_2] \). Note that all processes \( Q \) derivable from \( \langle P \rangle \) with annotated steps (i.e., \( \leadsto^* \)) are negotiations, since operator \( [\_ : \_] \) is removed only by rules (\textit{commit}) or (\textit{abort}), which are not annotated reductions. Consequently, for this case there is no \( Q \) s.t. \( \langle P \rangle \leadsto^* Q \) and \( Q \equiv_e \text{abort}|Q' \).
8.3. WRITING FLAT CJOIN PROCESS IN CANONICAL FORM

- \( P \equiv_e \text{def } D \text{ in } P_1 \): If \( P \) is already in canonical form then \( \{ P \} \equiv_e P \), and since \( P \) has no annotations, the only possibility is \( Q \equiv_e \{ P \} \). Then \( P \equiv_e \{ P \} \equiv_e Q \equiv_e \text{abort} \mid P' \). Otherwise, i.e. \( P \) is not canonical, then \( \{ P \} \equiv_e \text{def } D \mid K \text{ in } \{ P_1 \} \). Note that the annotated rules generated by \( \{ D \} \mid K \) do not introduce \text{abort}, and they do not generate messages to ports not defined by \( \{ D \} \mid K \). Hence, if the process \text{abort} is reached by \( \{ P \} \), then the only possibility is \( \{ P_1 \} \sim^* \text{abort} \mid Q'_1 \) and \( Q \equiv_e \text{abort} \mid Q' \). By inductive hypothesis on \( \{ P_1 \} \sim^* \text{abort} \mid Q'_1 \), \( P_1 \equiv_e \text{abort} \mid P'_1 \) and \( \{ P_1 \} \sim^* Q'_1 \). Hence

\[
P \equiv_e \text{def } D \text{ in } P_1 \equiv_e \text{def } D \text{ in } \text{abort} \mid P'_1 \equiv_e \text{abort} \mid \text{def } D \text{ in } P'_1
\]

By taking \( P' \equiv_e \text{def } D \text{ in } P'_1 \), it is easy to see that \( \{ P' \} \equiv_e \{ \text{def } D \text{ in } P'_1 \} \sim^* \text{def } D \mid K \text{ in } Q'_1 \sim^* Q' \).

\( \square \)

The following proposition states that a canonical form can reduce with annotated steps to a process that contains \( M \) only if the original process has \( M \).

**Proposition 8.19.** Let \( P \) be a flat cJoin process. If \( \{ P \} \sim^* Q \equiv_e M \mid Q' \), then \( P \equiv_e M \mid P' \) and \( \{ P \} \sim^* Q' \).

**Proof.** The proof follows by induction on the structure of \( P \).

- \( P \equiv_e x( y ) \): Immediate, because \( P \in \mathcal{C}_P \) and \( \{ P \} \equiv_e P \), which has no annotated definitions.

- \( P \equiv_e 0 \) and \( P \equiv_e \text{abort} \): There is no \( Q \) since \( P \in \mathcal{C}_P \) and \( \{ P \} \equiv_e P \).

- \( P \equiv_e P_1 \mid P_2 \): If \( P \) is already a canonical process, then \( \{ P \} \equiv_e P \), which has no annotated definitions and, consequently, no annotated reductions. Hence, the only possibility is \( Q \equiv_e \{ P \} \equiv_e P \), and hence \( P \equiv_e Q \equiv_e M \mid Q' \). Otherwise, i.e. \( P \) is not canonical, we have

\[
\{ P \} \equiv_e \text{def } x( ) \triangleright \{ P_1 \} \land y( ) \triangleright \{ P_2 \} \text{ in } x( ) \mid y( )
\]

Hence all possible \( R \) s.t. \( \{ P \} \sim^* R \) are as follow:

1. \( R \equiv_e \{ P \} \): In this case \( R \not\equiv_e M \mid Q' \). Hence \( R \) is not a possible \( Q \).

2. \( R \equiv_e \text{def } x( ) \triangleright \{ P_1 \} \text{ in } x( ) \mid Q_2 \equiv_e Q_2 \mid \text{def } x( ) \triangleright \{ P_1 \} \text{ in } x( ) \), with \( \{ P_2 \} \sim^* Q_2 \):

   The only possibility is \( Q_2 \equiv_e M \mid Q'_2 \) and \( Q' \equiv_e Q'_2 \mid \text{def } x( ) \triangleright \{ P_1 \} \text{ in } x( ) \). Then by inductive hypothesis on \( \{ P_2 \} \sim^* Q_2 \), we have \( P_2 \equiv_e M \mid P'_2 \) and \( \{ P'_2 \} \sim^* Q'_2 \). Consequently, \( P \equiv_e P_1 \mid P_2 \equiv_e M \mid P_1 \mid P'_2 \), and then by taking \( P' \equiv_e P_1 \mid P'_2 \), it is easy to notice that \( \{ P_1 \mid P'_2 \} \sim^* Q' \equiv_e Q'_2 \mid \text{def } x( ) \triangleright \{ P_1 \} \text{ in } x( ) \).
3. $R \equiv_e \text{def } y(\cdot) \triangleright (P_2) \text{ in } Q_1|x(\cdot)$, with $(P_1) \sim^* Q_1$: The proof is analogous to the previous case.

4. $R \equiv_e Q_1|Q_2$, with $(P_1) \sim^* Q_1$ and $(P_2) \sim^* Q_2$: If $R \equiv_e M|Q'$, then we can write $M \equiv_e M_1|M_2$, $Q' \equiv_e Q_1|Q_2$; $Q_1 \equiv_e M_1|Q_1'$ and $Q_2 \equiv_e M_2|Q_2'$. By applying inductive hypothesis on both $(P_1) \sim^* Q_1$ and $(P_2) \sim^* Q_2$, we have $P_1 \equiv_e M_1|P_1'$ and $P_2 \equiv_e M_2|P_2'$. And finally, $P \equiv_e P_1|P_2 \equiv_e M_1|M_2|P_1'|P_2' \equiv_e M|P'$. Moreover, it is easy to notice that $(P') \equiv_e (P_1'|P_2') \sim^* Q_1'|Q_2'$.

- $P \equiv_e [P_1 : P_2]$: The canonical form of $P$ (even when it is already a canonical processes) is a negotiation, $(P) \equiv_e [Q_1 : Q_2]$. Note that all processes $Q$ derivable from $(P)$ with annotated steps (i.e., $\sim^*$) are negotiations, since operator $\cdot : \cdot$ is removed only by rules (COMMIT) or (ABORT), which are not annotated reductions. Consequently, for this case there is no $Q$ s.t. $(P) \sim^* Q$ and $Q \equiv_e M|Q'$.

- $P \equiv_e \text{def } D$ in $P_1$: If $P$ is already in canonical form then $(P) \equiv_e P$, and since $P$ has no annotations, the only possibility is $Q \equiv_e (P)$. Then $P \equiv_e (P) \equiv_e Q \equiv_e M|Q'$. Clearly $(Q') \equiv_e Q' \sim^* Q'$.

Otherwise, i.e. $P$ is not canonical, then $(P) \equiv_e \text{def } (D)_{K}$ in $(P_1)$. Note that the annotated rules generated by $(D)_{K}$ send messages only to ports in $[D]_{K}$. Hence, all messages on free names must be generated by $(P_1)$. Hence, then the only possibility is $(P_1) \sim^* M|Q_1'$, where $fn(M) \cap dn((D)_{K}) = \emptyset$, and $Q \equiv_e M|Q'$, where $\text{def } (D)_{K}$ in $Q_1' \sim^* Q'$. By inductive hypothesis, $P_1 \equiv_e M|P_1'$ and $P_1' \sim^* Q_1'$, and hence $P \equiv_e \text{def } D$ in $P_1 \equiv_e \text{def } D$ in $M|P_1'$. Since $dn(D) \subseteq dn((D)_{K})$, i.e. $D$ does not capture any name of $M$, $P \equiv_e M|\text{def } D$ in $P_1'$. Moreover by taking $P' \equiv_e \text{def } D$ in $P_1'$, it is easy to notice that $(P') \equiv_e \text{def } (D)$ in $P_1' \sim^* \text{def } (D)_{K}$ in $Q_1' \sim^* Q'$.

\[\square\]

The following proposition is a generalisation of the previous one, where messages may appear under a definition. We present both results because Proposition 8.19 is used to prove 8.20.

**Proposition 8.20.** Let $P$ be a flat cJoin process. If $(P) \sim^* Q \equiv_e \text{def } (D)_{K}$ in $M|Q'$ s.t. $dn(M) \cap (dn(D) \cup K) = \emptyset$, then $P \equiv_e \text{def } D$ in $M|P'$ and $\text{def } (D)_{K}$ in $(P') \sim^*$

**Proof.** The proof follows by induction on the structure of $P$.

- $P \equiv_e x(y)$, $P \equiv_e 0$ and $P \equiv_e \text{abort}$: Immediate, because $P \in C_P$ and $(P) \equiv_e P$, which has no annotated definitions, hence the only possible $Q$ is $(P) = P$.

- $P \equiv_e P_1|P_2$: This case follows as in proof of Proposition 8.19, by analysing the different cases, where $(P) \sim^* R$, $(P_1) \sim^* Q_1$ and $(P_2) \sim^* Q_2$:

1. $R \equiv_e (P)$: In this case $R$ is not a possible $Q$. 

\[\square\]
2. \( R \equiv_e Q_2 \text{def } x() \triangleright (P_1) \text{ in } x() \): Hence \( Q_2 \equiv_e \text{def } (D)_K \text{ in } M|Q_2 \) and, by assuming \((D)_K\) not to capture names in \((P_1), Q' \equiv_e \text{def } (D)_K \text{ in } M|Q_2\) \(\text{def } x() \triangleright (P_1) \text{ in } x()\). Then, by inductive hypothesis on \((P_2) \sim^* Q_2\), we have \(P_2 \equiv_e \text{def } D \text{ in } M|P_2\) and \(\text{def } (D) \text{ in } (P_2') \sim^* Q_2\). Consequently, \(P \equiv_e P_1|P_2 \equiv_e P_1|\text{def } D \text{ in } M|P'_2\). Since \((D)_K\) does not capture names in \((P_1)\), then \(D\) does not capture names in \(P_1\). Hence, \(P \equiv_e \text{def } D \text{ in } M|P_1|P_2\).

   It is easy to notice that \(\text{def } (D) \text{ in } P_1|P'_2 \sim^* \text{def } (D)_K \text{ in } Q_1|Q'_2\).

3. \( R \equiv_e Q_1|Q_2\): The proof is analogous to the previous case.

4. \( R \equiv_e Q_1|Q_2\): The proof proceeds as in Proposition 8.19.

   • \( P \equiv_e [P_1 : P_2]\): This case is not possible, like in Proposition 8.19.

   • \( P \equiv_e \text{def } D \text{ in } P_1\): If \( P \) is already in canonical form then \((P) \equiv_e P\), and since \( P \) has no annotations, the only possibility is \( Q \equiv_e \{P\} \). Then \( P \equiv_e \{P\} \equiv_e Q \equiv_e \{Q\\}. Elsewise, i.e. \( P \) is not canonical, \((P) \equiv_e \text{def } (D)_K \text{ in } \{P_1\}\). Note that the annotated rules generated by \((D)_K\) send messages only to ordinary ports in \((D)_K\). Hence, all messages sent on free ports or merge names are generated by \((P_1)\). Then, the only possibility is \((P_1) \sim^* M|Q_1\) and \(\text{def } (D)_K \text{ in } Q_1 \sim^* \text{def } (D)_K \text{ in } Q'\).

   By Proposition 8.19, \( P_1 \equiv_e M|P'_1\) and \((P'_1) \sim^* Q'_1\). Hence \( P \equiv_e \text{def } D \text{ in } M|P'_1\). Finally, by taking \(P' \equiv_e P'_1\), it is easy to note that \(\text{def } D \text{ in } P'_1 \sim^* \text{def } (D)_K \text{ in } Q_1 \sim^* \text{def } (D)_K \text{ in } Q'\).

\[ \square \]

Next Proposition states that if a process \( Q \) derivable from \((P)\) with annotated reductions, i.e. \((P) \sim^* Q\), is able to fire a non-annotated definition, e.g. \((x_1|u_1)|z(w) \triangleright P'_0\), then \( P \) can fire the definition \( x_1|u_1|z(w)|J \triangleright P'\), whose encoding has produce \(\{x_1|u_1|z(w) \triangleright P'_0\}\).

**Proposition 8.21.** Let \( P \) be a flat cJoin process. For all \( Q \) s.t. \((P) \sim^* Q\) and \( Q \equiv_e \text{def } (x_1|u_1)|z(w) \triangleright P'_0\) \(\wedge (x_2|w_2)|J \triangleright z(w)|K_1 \wedge \{D\}_K \text{ in } \{x_1|u_1|z(w)\} K_2 \wedge (D)_K \text{ in } Q_2\), then \( P \equiv_e \text{def } x_1|u_1|z(w)|J \triangleright P' \wedge D \text{ in } (x_1|u_1|z(w)|J) \sigma|P_3\), where

\[ \text{def } (x_1|u_1|z(w)|J \triangleright P'_0)\{z\} \equiv K_1 \wedge \{D\}_K \text{ in } \{P_3\}\]

\[ \sim^* \text{def } (x_1|u_1|z(w)|J \triangleright P'_0)\{z\} \equiv K_1 \wedge \{D\}_K \text{ in } Q_2\]

(for suitable \( z, K_1 \) and \( K_2 \)).

**Proof.** By inductive hypothesis on \( P \).

   • \( P \equiv_e x(y), P \equiv_e 0 \) and \( P \equiv_e \text{abort}\): Immediate, because \( P \in \mathcal{C}_P \) and \((P) \equiv_e P\), which has no annotated definitions, hence there is no possible \( Q \).

   • \( P \equiv_e P_1|P_2\): This case follows as in proof of Proposition 8.19, by analysing the different cases for \((P) \sim^* R\), and by using inductive hypothesis both on \((P)_1 \sim^* Q_1\) and \((P)_2 \sim^* Q_2\).
\[ P \equiv_e [P_1 : P_2]: \text{This case is not possible, as in Proposition 8.19.} \]

\[ P \equiv_e \text{def } D \text{ in } P_1: \text{If } P \text{ is already in canonical form, then } \langle P \rangle \equiv_e P \text{ and, since } P \text{ has no annotations, the only possibility is } Q \equiv_e \langle P \rangle. \text{ Then, } P \equiv_e \langle P \rangle \equiv_e Q \equiv_e \text{def } D \text{ in } M|P_1'. \text{ Clearly } \langle Q' \rangle \equiv_e Q' \sim^* Q'. \]

The last possibility is \( D = x_1(\bar{u}_1)|x_2(\bar{u}_2)|J \triangleright P' \land D' \), hence for \( K = \{ z \} \cup K_1 \cup K_2, \langle D \rangle|_K = \{ x_1(\bar{u}_1)|z(\bar{w}) \triangleright P'_0 \land \{ x_2(\bar{u}_2)|J \triangleright z(\bar{w}) \rangle \}^+_K \land \{ D' \}|_{K_2}. \) Moreover, \( P_1 \equiv_e (x_1(\bar{u}_1)|z(\bar{w}))\sigma|Q_2. \) Since \( z \) is a name added by the encoding, then the only possible way to generate \( z(\bar{w})\sigma \) is by firing the rules in \( \langle x_2(\bar{u}_2)|J \triangleright z(\bar{w}) \rangle \}^+_K_1 \). It is easy to notice that this is the case only when all messages \( (x_2(\bar{u}_2)|J)\sigma \) are in \( Q \). Finally by proposition 8.20, messages \( (x_1(\bar{u}_1)|x_2(\bar{u}_2)|J)\sigma \) are in \( P \), i.e. \( P \equiv_e \text{def } x_1(\bar{u}_1)|x_2(\bar{u}_2)|J \triangleright P' \land D' \) in \( (x_1(\bar{u}_1)|x_2(\bar{u}_2)|J)\sigma|P_3 \), where \( \text{def } \langle D \rangle|_K \text{ in } \langle P_3 \rangle \sim^* \text{def } \langle D \rangle|_K \text{ in } Q_2. \]

Last proposition assures that the firing of a definition, can be simulated in the canonical form by performing several annotated reductions and one non annotated step.

**Proposition 8.22.** \( \langle J \triangleright P_2 \rangle|_K \land D \text{ in } J\sigma|P_2 \sim^* \rightarrow^* \sim^* \text{def } \langle J \triangleright P_3 \rangle|_K \land D \text{ in } (\langle P \rangle)|\sigma|P_2 \)

**Proof.** By induction on the structure of \( J \). For \( J \) consisting of more than two messages, i.e. \( J = x_1(\bar{u}_1)|x_2(\bar{u}_2)|J' \), then by induction on the length of \( J' \) it is proved that \( \text{def } \langle x_2(\bar{u}_1)|x_2(\bar{u}_2)|J' \triangleright z(\bar{w}) \rangle \}^+_K \land D \text{ in } J\sigma|P \sim^* \text{def } \langle x_2(\bar{u}_2)|J' \triangleright z(\bar{w}) \rangle \}^+_K \land D \text{ in } z(\bar{w})\sigma|P. \)

Now we are ready to prove the equivalence among processes and their canonical forms. To simplify the writing of proof in the next lemmata we use the following diagrams.

**Notation 8.1.** In order to facilitate the presentation of the following proofs, we use diagrams, where nodes are terms and arrows stand for relations. Moreover, solid arrows are universally quantified, while dotted arrows are existentially quantified. For instance, the following diagram

\[
\begin{array}{ccc}
P & \equiv_e & Q \\
\downarrow & * & \downarrow \\
P' & \equiv_e & Q'
\end{array}
\]

stands for \( \forall P,Q,P' \text{ s.t. } P \equiv_e Q \text{ and } P \sim P', \text{ then } \exists Q' \text{ s.t. } Q \rightarrow^* Q' \text{ and } P' \equiv_e Q' \)

**Lemma 8.23.** The relation \( \mathcal{R} = \{(P,Q)|\langle P \rangle \sim^* Q\} \) is a weak barbed simulation.

**Proof.** By coinduction we prove that for all \( P, Q \) then:

\begin{enumerate}
\item \( \forall x: P \downarrow x \Rightarrow Q \downarrow x. \)
\item \( \forall P' \text{ s.t. } P \rightarrow P', \exists Q': Q \rightarrow^* Q' \text{ and } P' \sim Q'. \)
\end{enumerate}
Actually the proof is up-to strong bisimilarity [53], since we consider reductions up-to \( \equiv_e \), which is a strong bisimulation (see Notation 2.2). Nevertheless, for simplicity, we write \( \rightarrow (\sim) \) also for \( \rightarrow_{\equiv_e} \) (resp. \( \sim_{\equiv_{eqv}} \)).

- \( P \equiv_e 0, P \equiv_e x<\bar{u}> \) or \( P \equiv_e \) abort:
  1. As \( P \) is canonical, \( \{P\} = P \). Additionally, since \( \{P\} = P \) has not annotated definitions, if \( \{P\} \sim^* Q \), then \( Q \equiv_e \{P\} \equiv_e P \). Clearly, \( \forall x : P \downarrow_x \Rightarrow Q \downarrow_x \).
  2. It is trivially satisfied since there are no possible reductions for \( P \).

- \( P \equiv_e P_1|P_2 \): By definition of \( \mathcal{R} \), if \( P \mathcal{R} Q \), then \( \{P\} \sim^* Q \). Then, \( Q \) has one of the following form, where \( \{P_1\} \sim Q_1 \) and \( \{P_2\} \sim Q_2 \):
  1. \( Q \equiv_e \{P\} \)
  2. \( Q \equiv_e \text{ def } x() \triangleright \{P_1\} \text{ in } x()|Q_2 \)
  3. \( Q \equiv_e \text{ def } y() \triangleright \{P_2\} \text{ in } Q_1|x() \)
  4. \( Q \equiv_e Q_1|Q_2 \)

Note that above cases have the following reductions

- \( Q \equiv_e \{P\} \sim^2 \{P_1\}|\{P_2\} \)
- \( Q \equiv_e \text{ def } x() \triangleright \{P_1\} \text{ in } x()|Q_2 \sim \{P_1\}|Q_2 \)
- \( Q \equiv_e \text{ def } y() \triangleright \{P_2\} \text{ in } Q_1|x() \sim Q_1|\{P_2\} \)
- \( Q \equiv_e Q_1|Q_2 \)

Hence, we can assume that \( Q \sim^* Q_1|Q_2 \), where \( \{P_1\} \sim Q_1 \) and \( \{P_2\} \sim Q_2 \).

(i) If \( P \downarrow_x \) then \( P_1 \downarrow_x \lor P_2 \downarrow_x \). By inductive hypothesis on both \( P_1 \) and \( P_2 \), we have that for all \( Q_1, Q_2 \) s.t. \( \{P_1\} \sim^* Q_1 \) and \( \{P_2\} \sim^* Q_2 \), \( Q_1 \downarrow_x \lor Q_2 \downarrow_x \), and hence \( (Q_1|Q_2) \downarrow_x \). Therefore, for all four cases listed above, \( Q \downarrow_x \).

(ii) There are two possible reductions on \( P \). Since both cases follow analogously, we detail here the proof for the first case.

* \( P' = P_1'|P_2 \) and \( P_1 \rightarrow P_1' \): The proof has the following shape

\[
\begin{array}{c}
P \xrightarrow{\mathcal{R}} Q \\
(P_1'|P_2) \downarrow_x \\
Q_1|Q_2 \\
P' \equiv_e P_1'|P_2 \xrightarrow{\mathcal{R}} Q' \equiv_e Q_1'|Q_2
\end{array}
\]
The top-left side of the diagram, i.e. $P \mathcal{R} Q$ and $P \rightarrow P'$, follows by hypothesis. The right side is obtained by case analysis on the possible forms for $Q$ (cases 1-4 listed above), and by noting that for any $Q$ the following hold: $Q \leadsto^* Q_1 | Q_2$, where $P_1 \mathcal{R} Q_1$ and $P_2 \mathcal{R} Q_2$. By inductive hypothesis on $P_1 \rightarrow P'_1$, for all $Q_1$ s.t. $P_1 \mathcal{R} Q_1$, there exists $Q'_1$ s.t. $Q_1 \rightarrow^* Q'_1$ and $P'_1 \mathcal{R} Q'_1$. Hence $Q_1 | Q_2 \rightarrow^* Q'_1 | Q_2$.

The central part of the diagram follows by the definition of $\mathcal{R}$. In fact, $(P_1' | P_2) \mathcal{R} (P_1' | P_2)$, which reduces in two steps (by firing the definitions of the forwarders) to $(P'_1) | (P_2)$. Finally, since $(P'_1) \leadsto^* Q'_1$ and $(P_2) \leadsto Q_2$, we have that $(P'_1) | (P_2) \leadsto Q'_1 | Q_2$. Consequently, $P' \mathcal{R} (Q'_1 | Q_2)$.

* $P' = P_1' | P_2'$ and $P_2 \rightarrow P_2'$: Proof analogous to the previous case.

- $P \equiv_e [P_1 : P_2]$: 
  
  (i) Immediate because $\not\exists x$ s.t. $P \downarrow_x$.
  
  (ii) There are three different possibilities for $P \rightarrow P'$:

  1. **Internal**: $P' \equiv_e [P'_1 : P_2]$, where $P_1 \rightarrow P'_1$. The proof has the following shape

     \[
     \begin{array}{ccc}
     P & \xrightarrow{\mathcal{R}} & Q \\
     P' \equiv_e [P'_1 : P_2] & \xrightarrow{\mathcal{R}} & Q' \equiv_e [Q'_1 : (P_2)]
     \end{array}
     \]

     On top of the diagram, $P \mathcal{R} Q$ and $P \rightarrow P'$ follow by hypothesis. The right side is obtained by analysing the possible forms that $Q$ can take (i.e., $(P) \leadsto^* Q$), and by noting that for any $Q$, it is always the case that $Q \leadsto^* [Q_1 : (P_2)]$ and $P_1 \mathcal{R} Q_1$. By inductive hypothesis on $P_1 \rightarrow P'_1$, for any $Q_1$ s.t. $P_1 \mathcal{R} Q_1$, then there exists $Q'_1$ s.t. $Q_1 \rightarrow^* Q'_1$ and $P'_1 \mathcal{R} Q'_1$. Consequently $[Q_1 : (P_2)] \rightarrow^* [Q'_1 : (P_2)]$.

     The central part of the diagram follows by the definition of $\mathcal{R}$. In fact, $(P'_1 : P_2) \mathcal{R} ([P'_1 : P_2])$, which reduces in one step (by firing the definition of the forwarder) to $([P'_1] : (P_2)]$. Since $P'_1 \mathcal{R} Q'_1$, then $([P'_1] : (P_2)] \leadsto^* [Q'_1 : (P_2)]$. Consequently $P' \mathcal{R} Q' \equiv_e [Q'_1 : (P_2)]$.

  2. **Commit**: $P_1 \equiv_e M | \text{def } D$ in 0 and $P' \equiv_e M$. The proof is obtained as follow.
The main point in the above diagram is that since $P \stackrel{\mathcal{R}}{\rightarrow} Q$, and by Proposition 8.17, $Q \rightarrow^* [M] \text{def } \{D\}_K \text{ in } 0 : \{P_2\}$, which reduces (by committing) to $Q' \equiv_e M$. The remaining arrows follow by definition of $\mathcal{R}$. In particular $Q \rightarrow^* M$ by Proposition 8.16.

3. **Abort**: $P_1 \equiv_e \text{abort}|R$ and $P' \equiv_e P_2$. This case follows as below

The main point here is that $P_1$ contains the process *abort*, which is preserved by the translation (i.e., $\{P\}$ contains a forwarder to *abort*). Since there is no way to consume *abort*, then any process $Q_1$ derivable from $\{P_1\}$ has *abort*, or a forwarder that activates *abort*. Hence, for any possible $Q_1$, there exists a derivation $Q_1 \rightarrow^* \text{abort}|R'$.

- $P \equiv_e \text{def } D$ in $P_1$:
  
  (i) If $P \downarrow_x$ then $P_1 \downarrow_x$ and $x \not\in \text{dn}(D)$. By inductive hypothesis on $P_1$, $\{P_1\} \downarrow_x$. We can assume w.l.o.g. that $\{D\}_K$ does not capture $x$ (because $x \in \text{fn}(P)$, hence $x \not\in \text{dn}(D)$ and $x \not\in K$), and hence $\{P\} \downarrow_x$. Then, $\{P\} \equiv_e \text{def } D' \text{ in } x(u)|P'$. By Proposition 8.17, for any $Q$ derivable from $\{P\}$ with annotated steps, i.e. $P \mathcal{R} Q$, $Q \rightarrow^* \text{def } D' \text{ in } x(u)|Q'$. Hence, for all $Q$ s.t. $P \mathcal{R} Q$ we have $Q \downarrow_x$.

  (ii) There are several possibilities

  1. $P' \equiv_e \text{def } D$ in $P'_1$ and $P_1 \rightarrow P'_1$. The proof corresponding to this case is below.
where \( P_1 \mathcal{R} Q_1 \) (note that if \( Q \) has been obtained by using annotated definitions of \( \langle D \rangle_K \), such definitions are reversible, and hence \( Q \) can reach a process that can be obtained without applying rules of \( \langle D \rangle_K \)). By inductive hypothesis applied on \( P_1 \rightarrow P'_1 \) we have that there exists \( Q'_1 \) s.t.
(a) \( P'_1 \mathcal{R} Q'_1 \), and (b) \( Q_1 \rightarrow^* Q'_1 \). By (a) \( \text{def} \ \langle D \rangle_K \text{ in } Q_1 \rightarrow^* Q' \), while (b) allows the derivation \( \text{def} \ D \text{ in } P'_1 = \text{def} \ \langle D \rangle_K \text{ in } \langle P' \rangle \rightarrow^* Q' \).

2. \( P \equiv_e \text{def} \ J_1 | \ldots | J_n \triangleright P_1 \land D \in \Pi^n_k = \text{def} \ D_k \text{ in } J_k \sigma | P_k : Q_k | P_2 \) and \( P' \equiv_e \text{def} \ J_1 | \ldots | J_n \triangleright P_1 \land D \in \text{def} \ \wedge_k^n D_k \text{ in } P_1 \sigma | P_k : \Pi^n_k = Q_k | P_2 \). The proof for this case has the following shape.

\[
\begin{array}{c}
P \quad \mathcal{R} \quad Q \\
\mathcal{R} \quad \langle P' \rangle \\
\mathcal{R} \quad Q'' \\
\implies \quad Q' \\
\end{array}
\]

The right side of the diagram is explained as follow. Since negotiations cannot be removed by annotated steps, for any \( \langle P \rangle \rightarrow^* Q \) it is easy to see that \( Q \rightarrow^* Q'' \equiv_e \text{def} \ J_1 | \ldots | J_n \triangleright \langle P_1 \rangle \land \langle D \rangle_K \text{ in } \Pi^n_k = R_k : \langle Q_k \rangle | P_2 \), where \( \text{def} \ D_k \text{ in } J_k \sigma | P_k \rightarrow^* R_k \), and \( \langle P_2 \rangle \rightarrow^* P_2 \). Then by Proposition 8.17, \( R_k \equiv_e \text{def} \ \langle D_k \rangle_{K_k} \text{ in } J_k \sigma | P_k \), where \( \text{def} \ \langle D_k \rangle_{K_k} \text{ in } \langle P_k \rangle \rightarrow^* \text{def} \ \langle D_k \rangle_{K_k} \text{ in } P'_k \), for suitable \( K_k \). Then, by assuming w.l.o.g that names in \( \langle D_k \rangle_{K_k} \) do not clash, and by using the merge definition we have

\[
Q'' \rightarrow Q \equiv_e \text{def} \ J_1 | \ldots | J_n \triangleright \langle P_1 \rangle \land \langle D \rangle_K \text{ in } \text{def} \ \wedge_k^n \langle D_k \rangle_{K_k} \text{ in } \langle P_1 \rangle \sigma | P_k : \Pi^n_k = \langle Q_k \rangle | P_2
\]

The central part of the diagram is built by noting that \( \langle P'_1 \rangle \) can reduce with annotated steps to \( Q''' \) s.t.

\[
\langle P' \rangle \rightarrow^* Q''' \equiv_e \text{def} \ J_1 | \ldots | J_n \triangleright \langle P_1 \rangle \land \langle D \rangle_K \text{ in } \text{def} \ \wedge_k^n \langle D_k \rangle_{K_k} \text{ in } \langle P_1 \rangle \sigma | P_k : \Pi^n_k = \langle Q_k \rangle | P_2
\]

Finally, by Proposition 8.11, \( Q''' \equiv_e Q' \), which allows to close the diagram with \( P' \mathcal{R} Q' \).

3. \( P \equiv_e \text{def} \ J \triangleright P_1 \land D \text{ in } J \sigma | P_2 \) and \( P' \equiv_e \text{def} \ J \triangleright P_1 \land D \text{ in } P_1 \sigma | P_2 \). The proof is as follow.

\[
\begin{array}{c}
P \quad \mathcal{R} \quad Q \\
\mathcal{R} \quad \langle P' \rangle \\
\mathcal{R} \quad Q'' \equiv_e \text{def} \ \langle J \triangleright P_1 \rangle_{K_1} \land \langle D \rangle_{K_2} \text{ in } J \sigma | Q_2 \\
\implies \quad Q' \equiv_e \text{def} \ \langle J \triangleright P_1 \rangle_{K_1} \land \langle D \rangle_{K_2} \text{ in } \langle P_1 \rangle \sigma | Q_2 \\
\end{array}
\]
The arrow \( Q \rightsquigarrow^* Q'' \equiv_e \text{def} \ J \triangleright (P_1) \land (D) \in \ K \text{in} \ J_{\sigma}Q_2 \) follows by Proposition 8.17. The arrow \( Q'' \rightarrow^* Q' \) follows from Proposition 8.22. The remaining part is completed as follow. By definition of \( \mathcal{R} \), \( P' \mathcal{R} (P') \), which reduces to \( Q'' \equiv_e \text{def} \ \langle J \triangleright P_1 \rangle \in \ K_1 \land (D) \in \ K_2 \text{in} \ \langle P'_1 \rangle Q_2 \). Then, by Proposition 8.11, \( Q'' \equiv_e Q' \). Hence \( P' \mathcal{R} Q' \).

\[ \square \]

**Lemma 8.24.** The relation \( \mathcal{R}^{-1} = \{(Q, P) | (P) \rightsquigarrow^* Q\} \) is a weak barbed simulation.

**Proof.** By coinduction we prove that for all \( Q \mathcal{R}^{-1} P \) then:

(i) \( \forall x : Q \downarrow_x \Rightarrow P \downarrow_x \).

(ii) \( \forall Q' \text{ s.t. } Q \rightarrow Q', \exists P' : P \rightarrow P' \text{ and } Q' \mathcal{R}^{-1} P' \).

The proof is up-to strong bisimilarity \[53\], since we consider reductions up-to \( \equiv_e \), which is a strong bisimulation (see Notation 2.2). Nevertheless, for simplicity, we write \( \rightarrow (\rightsquigarrow) \) also for \( \rightarrow_{\equiv_e} \) (resp. \( \rightsquigarrow_{\equiv_{\text{equiv}}}) \).

- \( P \equiv_e 0, P \equiv_e x(\overline{a}) \) or \( P \equiv_e \text{ abort} \): since \( P \) is canonical, \( \langle P \rangle = P \). Additionally, \( \langle P \rangle \) has not annotated definitions, and hence the only possible \( Q \) s.t. \( \langle P \rangle \rightsquigarrow^* Q \) is \( Q \equiv_e \langle P \rangle \equiv_e P \).

  (i) Since \( Q \equiv_e P \), \( \forall x : Q \downarrow_x \Rightarrow P \downarrow_x \).

  (ii) It is trivially satisfied since there are no possible reductions for \( Q \).

- \( P \equiv_e P_1 | P_2 \): If \( P \) is in canonical form the proof is immediate since \( \langle P \rangle = P \), which has no annotated definitions, and consequently the only possibility is \( Q \equiv_e P \).

  Otherwise, if \( P \) is not canonical, then \( Q \) has one of the following forms, where \( \langle P_1 \rangle \rightsquigarrow Q_1 \) and \( \langle P_2 \rangle \rightsquigarrow Q_2 \):

  1. \( Q \equiv_e \langle P \rangle = \text{def} x() \triangleright (P_1) \land y() \triangleright (P_2) \text{ in } x()|y() \): In this case \( \exists z : Q \downarrow_z \), hence condition (i) is trivially satisfied. As far as condition (ii) is concerned, note that the only possibility for \( Q \rightarrow Q' \) is to reduce by using one annotated definition, hence \( Q \rightsquigarrow Q' \), and therefore by definition of \( \mathcal{R}^{-1}, Q' \mathcal{R}^{-1} P \). Clearly \( P \rightarrow^* P' \).

  2. \( Q \equiv_e \text{def} x() \triangleright (P_1) \text{ in } x()|Q_2 \): (i) In this case, \( \forall z : Q \downarrow_z \Rightarrow Q_2 \downarrow_z \). By inductive hypothesis, if \( Q_2 \downarrow_z \Rightarrow P_2 \downarrow_z \), and hence \( P \downarrow_z \). For condition (ii) there are two cases: the first one is when the forwarder \( x \) is fired, and the proof follows as in the previous case. Otherwise, i.e. \( Q_2 \rightarrow Q'_2 \), the proof has the following shape

\[
Q \equiv_e \text{def} x() \triangleright (P_1) \text{ in } x()|Q_2 \mathcal{R}^{-1} P \equiv_e P_1 | P_2
\]

\[
Q' \equiv_e \text{def} x() \triangleright (P_1) \text{ in } x()|Q_2 \mathcal{R}^{-1} P' \equiv_e P_1 | P_2'
\]
Since $Q_2 \mathcal{R}^{-1} P_2$ and $Q_2 \rightarrow Q_2'$, by inductive hypothesis there exists $P_2'$, s.t. $P_2 \rightarrow^* P_2'$ and $Q_2' \mathcal{R}^{-1} P_2'$, hence $\{P_2\} \rightarrow^* \{Q_2\}$. The bottom part of the diagram follows by definition of $\mathcal{R}^{-1}$. Since $\{P_1 \vert P_2\} \mathcal{R}^{-1} P'$ and $\{P_1 \vert P_2\} \rightarrow^* Q'$, we have that $Q' \mathcal{R}^{-1} P'$.

3. $Q \equiv_{e} \text{def } y() \triangleright \{P_2\}$ in $Q_1[x()]$. The proof is analogous to the previous case.

4. $Q \equiv_{e} Q_1 \vert Q_2$ There are two possibilities: $Q_1 \rightarrow Q'_1$ and $Q_2 \rightarrow Q'_2$. Both of them follows as in the previous cases.

- $P \equiv_{e} [P_1 : P_2]$: If $P$ is in canonical form the proof is immediate since $\{P\} = P$, which has no annotated definitions, and consequently the only possibility is $Q \equiv_{e} P$. Otherwise, if $P$ is not canonical, then $\{P\} \rightarrow^* Q$.

  (i) Since $\{P\}$ is a negotiation and annotated reductions cannot remove negotiation boundaries $\begin{array}{|c|}
\hline
\end{array}$, then $Q$ has no bars.

  (ii) There are four different cases for $\{P\} \equiv [\text{def } x() \triangleright \{P_2\}]$ in $x() : \{P_2\}$: $\rightarrow^*$ $Q$:

1. $Q \equiv_{e} \{P\}$: This case is straightforward because the only possibility for $Q \rightarrow Q'$ is by firing the forwarder $x$, and hence $Q \rightarrow Q'$. Consequently $\{P\} \rightarrow^* Q$ and by definition of $\mathcal{R}^{-1}$, $Q' \mathcal{R}^{-1} P$.

2. $Q \equiv_{e} [Q_1 : \{P_2\}]$, where $\{P_1\} \rightarrow^* Q_1$, and $Q_1 \rightarrow Q'_1$. Then, by inductive hypothesis, there exists $P_1'$ s.t. $P_1 \rightarrow^* P_1'$ and $Q'_1 \mathcal{R}^{-1} P_1'$. Consequently, the proof is built as follow

$$Q \equiv_{e} [Q_1 : \{P_2\}] \mathcal{R}^{-1} P \equiv_{e} [P_1 : P_2] \rightarrow^* \{P_1\} \mathcal{R}^{-1} Q' \equiv_{e} [Q'_1 : \{P_2\}] \mathcal{R}^{-1} P' \equiv_{e} [P'_1 : P_2]$$

3. $Q \equiv_{e} [M \text{def } D \text{ in } 0 : \{P_2\}]$: The proof is obtained by using Proposition 8.19 that assures $P_1 \equiv_{e} M \equiv_{e} M \text{def } D \text{ in } 0$. Hence the proof is as follow

$$Q \equiv_{e} [M \text{def } D \text{ in } 0 : \{P_2\}] \mathcal{R}^{-1} P \equiv_{e} [M \text{def } D \text{ in } 0 : P_2]$$

$$\{M\} \mathcal{R}^{-1} Q' \equiv_{e} M \mathcal{R}^{-1} P' \equiv_{e} M$$

Arrow $\{M\} \rightarrow^* M$ follows by Proposition 8.16.
4. $Q \equiv_e \text{[abort]} Q_1 : \{ P_2 \}$ and $Q' \equiv_e \{ P_2 \}$: this case follows by using Proposition 8.18 that assures that $P_1 \equiv_e \text{[abort]} P'_1$. Then the proof is as follows

$$Q \equiv_e \text{[abort]} Q'_1 : \{ P_2 \} \xrightarrow{\mathcal{R}^{-1}} P \equiv_e \text{[abort]} P'_1 : P_2$$

$$\xrightarrow{\vdash} Q' \equiv_e \{ P_2 \} \xrightarrow{\mathcal{R}^{-1}} P' \equiv_e P_2$$

- $P \equiv_e \text{def } D \text{ in } M|P_1$ (we do not take $P \equiv_e \text{def } D \text{ in } M$ since $P$ may have negotiations, which are not messages): Then, all possible $Q$ are such that $\{ P \} \equiv_e \text{def } \{ D \}_K \text{ in } \{ M \}|P_1 \xrightarrow{\ast} Q \equiv_e \text{def } \{ D \}_K \text{ in } Q_1$.

(i) If $Q \not\vdash x$ then $Q \equiv_e \text{def } \{ D \}_K \text{ in } x(\overline{a})|Q'_1$, where $x \notin dn(\{ D \}_K)$. By Proposition 8.19, $P \equiv_e \text{def } D \text{ in } x(\overline{a})|P_1$. Hence $P \not\vdash x$.

(ii) There are several possibilities

- $Q \not\equiv Q'$: The proof is immediate since $Q' \mathcal{R}^{-1} P$ and $P \rightarrow^* P'$.
- $Q \rightarrow Q'$ but not $Q \not\equiv Q'$: Assuming $x$ and $y$ to be fresh in $P$, then $\{ P \} \equiv_e \text{def } \{ D \}_K \text{ in } x(\overline{a}) \land y(\overline{a}) \rightarrow^* \{ P \}_x \text{ in } x(\overline{a})|y(\overline{a})$, for some $K$.

Since $\{ P \} \equiv_e Q$, then by proposition 8.17, the following conditions hold:

(a) $Q \equiv_e \text{def } \{ D \}_K \land D_1 \land D_2 \text{ in } M_1|R_1|R_2$, where $fn(M_1) \cap dn(D_1 \land D_2) = \emptyset$, $fn(R_1) \cap dn(D_2) = 0$, $fn(R_2) \cap dn(D_1) = 0$, $\forall z \in dn(\{ D \}_K) : R_1 \not\vdash z \land R_2 \not\vdash z$,

(b) $\text{def } x(\overline{a}) \rightarrow^* \text{def } D_1 \text{ in } R_1|M_a$,

(c) $\text{def } y(\overline{a}) \rightarrow^* \text{def } D_2 \text{ in } R_2|M_b$,

(d) $\text{def } \{ D \}_K \text{ in } M_a|M_b \rightarrow^* \text{def } \{ D \}_K \text{ in } M_1$.

Since $Q \rightarrow Q'$ but not $Q \not\equiv Q'$, there are three different possibilities:

1. The reduction occurs in $R_2$. Since $\forall z \in dn(\{ D \}_K) : R_2 \not\vdash z$, then the only possibility is $\text{def } D_2 \text{ in } R_2 \rightarrow \text{def } D_2 \text{ in } Q'_2$. Hence, $Q' \equiv_e \text{def } \{ D \}_K \land D_1 \land D_2 \text{ in } M_1|R_1|Q'_2$. From (b) we have that

$$\text{def } y(\overline{a}) \rightarrow^* \{ P \}_x \text{ in } y(\overline{a}) \rightarrow^* Q'' \equiv_e \text{def } D_2 \text{ in } R_2|M_b$$

$$\rightarrow Q'' \equiv_e \text{def } D_2 \text{ in } Q'_2|M_b$$

(8.1)

Since $Q''$ has (at least) one non-annotated definition, the forwarder $y(\overline{a}) \rightarrow^* \{ P \}_x$ has been fired, hence $\{ P \}_x \rightarrow^* Q'' \equiv_e \text{def } D_2 \text{ in } R_2|M_b$. Therefore $Q'' \mathcal{R}^{-1} P'_1$, and by inductive hypothesis, the following hold

$$\text{def } D_2 \text{ in } R_2|M_b \xrightarrow{\mathcal{R}^{-1}} P_1$$

(8.2)

$$\xrightarrow{\vdash} \text{def } D_2 \text{ in } Q'_2|M_b \xrightarrow{\mathcal{R}^{-1}} P'_2$$
Then, the whole proof is built as follow

$$Q \xrightarrow{\kappa^{-1}} \text{def } D \text{ in } M[P_1]$$

$$\langle \text{def } D \text{ in } M[P_2] \rangle$$

$$\text{def } \langle D \rangle_K \land D_1 \land y \rangle \triangleright \langle P_2 \rangle \text{ in } R_1[M_a]$$

$$\langle \text{def } \langle D \rangle_K \land D_1 \land D_2 \text{ in } R_1[M_a]Q_2[M_b] \rangle$$

$$Q' \xrightarrow{\kappa^{-1}} \cdots \text{def } D \text{ in } M[P_2']$$

In the above diagram \text{def } D \text{ in } M[P_1] \rightarrow^* \text{def } D \text{ in } M[P_2] \text{ by Equation (8.2). The central part follow by definition of } \kappa^{-1}. \text{ In fact } \langle \text{def } D \text{ in } M[P_2] \rangle \xrightarrow{\kappa^{-1}} \text{def } D \text{ in } M[P_2], \text{ which reduces first by condition (b), and then by Equation (8.1). The last reduction is obtained by condition (d), which allows to close the diagram with } Q' \xrightarrow{\kappa^{-1}} P'. \text{ }

2. The reduction occurs in $M_1$ by using a merge definition $J_1 \ldots J_2 \triangleright \langle P \rangle_2$ of $\langle D \rangle_K$. Then

$$Q \equiv_\varepsilon \text{def } J_1 \ldots J_2 \triangleright \langle P \rangle_2 \land \langle D \rangle_K \land D_1 \land D_2$$

in $M_1[\Pi_{k=1}^n] \text{ def } \langle D_k \rangle_{K_k}$ in $J_k \sigma[P_k] : \langle Q_k \rangle |R_1'|R_2'$

and

$$Q' \equiv_\varepsilon \text{def } J_1 \ldots J_2 \triangleright \langle P \rangle_2 \land \langle D \rangle_K \land D_1 \land D_2$$

in $M_1[\text{ def } \land_{k=1}^n \langle D_k \rangle_{K_k} \text{ in } \langle P \rangle_2 \sigma[\Pi_{k=1}^n P_k'] : \Pi_{k=1}^n \langle Q_k \rangle] |R_1'|R_2'$

Then, the proof has the following shape

$$Q \xrightarrow{\kappa^{-1}} P$$

$$\text{def } \langle P' \rangle$$

$$\text{def } Q'$$

$$Q$$

Since, negotiations cannot be added by annotated steps, for any $\langle P \rangle \rightsquigarrow^* Q$ it is easy to see that
\[ P \equiv_e \text{def } J_1 \ldots J_n \uparrow P_2 \wedge D \text{ in } M[\prod_{k=1}^{n} [R_k : Q_k]]P_1 \]

where \( \langle R_k \rangle \equiv^* \text{def } \langle D_k \rangle_{K_k} \) in \( J_k \sigma[P'_k] \) and \( \text{def } J_1 \ldots J_n \uparrow \langle P_2 \rangle \wedge \langle D \rangle_K \) in \( \langle P_1 \rangle \equiv^* R_2 \). Then, by Proposition 8.20, for all \( R_k \), we have \( R_k \equiv_e \text{def } D_k \) in \( J_k \sigma[P_k] \) and \( \text{def } \langle D_k \rangle_{K_k} \) in \( P_k \equiv^* P'_k \).

Then, by using the merge definition

\[ P \rightarrow P' \equiv_e \text{def } J_1 \ldots J_n \uparrow P_2 \wedge D \]

\[ \text{in } M[\text{def } \land_{k=1}^{n} D_k \text{ in } P_2 \sigma[\prod_{k=1}^{n} P'_k : \prod_{k=1}^{n} (Q_k)]]P_1 \]

Then it is easy to show that \( \langle P' \rangle \equiv^* Q' \), where

\[ Q' \equiv_e \text{def } J_1 \ldots J_n \uparrow \langle P_2 \rangle \wedge \langle D' \rangle_K \wedge D_1 \wedge D_2 \]

\[ \text{in } M_1[\text{def } \land_{k=1}^{n} \langle D_k \rangle_{K_k} \text{ in } \langle P_2 \rangle \sigma[\prod_{k=1}^{n} P'_k : \prod_{k=1}^{n} (Q_k)]]R_1 | R_2' \]

By Proposition 8.11, \( Q' \equiv E Q'' \), and hence \( Q' \mathcal{R}^{-1} P' \).

3. The reduction occurs in \( M_1 \) by using an ordinary definition \( J \uparrow \langle P_2 \rangle \)

\( \) of \( \langle D \rangle_K \). There are two cases:

- \( J \uparrow P_2 \) is a definition of \( D \): The proof follows as before.
- \( J \uparrow P_2 \) is not a definition of \( D \), then \( D \equiv J \uparrow P_2 \wedge D' \) s.t. \( \langle J \uparrow P_2 \rangle_{K_1} = \langle J \uparrow P_2 \rangle \wedge D'' \).

\[ Q \equiv_e \text{def } J \uparrow \langle P_2 \rangle \wedge D_1 \]

\[ \text{in } J \sigma[Q_1] \]

\[ \overline{\mathcal{R}^{-1}} \]

\[ P \equiv_e \text{def } J' \uparrow P_2 \wedge D' \]

\[ \text{in } J' \sigma[Q_3] \]

\[ \langle P' \rangle \]

\[ \vdash \]

\[ Q'' \equiv_e Q' \]

\[ \langle Q'' \rangle \equiv^* Q'' \]

\[ \mathcal{R}^{-1} \]

\[ P' \equiv_e \text{def } J' \uparrow P_2 \wedge D' \]

\[ \text{in } P_2 \sigma[Q_3] \]

The diagram is explained as follow. By Proposition 8.21, for all \( P \) s.t. \( Q \mathcal{R}^{-1} P \) the following conditions hold: (a) \( P \equiv_e \text{def } J' \uparrow P_2 \wedge D' \) in \( J' \sigma[P_3] \) and (b) \( \text{def } \langle J' \uparrow P_2 \wedge D' \rangle_K \) in \( \langle P_3 \rangle \equiv^* \text{def } \langle J' \uparrow P_2 \wedge D' \rangle K \) in \( Q_1 \). Clearly, \( P \) reduces to \( P' \) by firing the definition \( J' \uparrow P_2 \).

The remaining part follows from the definition of \( \mathcal{R}^{-1} \). In particular, \( \langle P' \rangle \mathcal{R}^{-1} P' \) by definition. It is easy to notice that, \( \langle P' \rangle \equiv^* Q'' \equiv_e \text{def } J \uparrow \langle P_2 \rangle \wedge D_1 \) in \( \langle P_2 \rangle \sigma[Q_1] \) by using condition (b) above. Finally, by Proposition 8.11, \( Q' \equiv_e Q'' \), and hence \( Q' \mathcal{R}^{-1} P' \).
Next theorem is the main result of this section, and states that canonical flat processes are enough to encode flat cJoin (up-to weak barbed bisimilarity).

**Theorem 8.25** (Expressiveness of canonical flat cJoin). \( P \approx \langle P \rangle \).

*Proof.* Immediate, since \( R \) and \( R^{-1} \) are weak barbed simulations, and both \( P R \langle P \rangle \) and \( \langle P \rangle R^{-1} P \) hold by Lemmata 8.23 and 8.24.

### 8.4 Compiling flat cJoin into Join.

In this section we describe the encoding in Join of flat cJoin processes. The results in previous section (Theorem 8.25) allows us to consider only canonical processes to prove that flat processes are implementable. In particular, we will show that flat negotiations of cJoin corresponds to the execution of distributed agreements in Join, i.e. processes running inside a negotiation are translated as participants of a distributed commit protocol. In order to coordinate such agreements we will use a variant of the commit protocol d2PC described in Section 2.4. We start by describing the changes to the d2PC (Section 8.4.1) and showing that the obtained protocol, called d2PC with Compensations (d2PCC) is still correct (Section 8.4.2). Then, we provide an encoding of canonical cJoin processes into Join (Section 8.4.3).

For simplicity, the following sections relies on Join calculus extended with the data type \( SET \), for finite sets and the standard operations of empty set \( \emptyset \), union \( \cup \), and difference \( \setminus \). We also allow pattern matching expressions on received names, which can be non-linear for the received names of a single port. Note that this extensions do not add expressivity to the language. First of all, sets can be encoded in lambda calculus, which in terms can be encoded in Join [52]. Moreover, in [84] it has been shown how non-linear pattern matching of the d2PC can be avoided by using linear pattern matching, which can be easily encoded in Join [6]).

#### 8.4.1 d2PC with Compensations (d2PCC)

There are two main differences between the original d2PC and the version we introduce here:

(i) **Handling of abortion:** A coordinator of the original d2PC in Figure 2.11 handles the non-deterministic behaviour of zS nets by initiating in a non-deterministic way the execution of protocol with abort, i.e., without receiving any message (rule \( state(\alpha) \triangleright failed() \mid release(\alpha) \)). This behaviour is not allowed in our modified version, in which coordinators can initiate the execution of the protocol only when they receive either a message on the local port put (for voting commit) or abort (for voting abort). Moreover, differently from original coordinators, which can release the compensation \( \alpha \) as soon as they decide
to abort —either autonomously (rule \( \text{state}(\alpha) \triangleright \text{failed}() \mid \text{release}(\alpha) \)) or by receiving an abort vote (rule \( \text{commit}(\emptyset, \ell', \ell'', \kappa, \alpha) \mid \text{fail}() \triangleright \text{failed}() \mid \text{release}(\alpha) \))—our modified coordinators activate the compensation only when they are sure that all other participants in the same agreement are aborting. The main point is that a coordinator is not allowed to release a compensation until the whole set of participants has been discovered, because otherwise the compensation process could interact with the pending participants, producing an unwanted computation. Therefore, we handle the abort phase analogously to the commit phase.

(ii) **Updating the initial state of coordinators:** We enrich coordinators with the capability of updating their states before the beginning of the execution of the protocol. Since in the original \( \text{d2PC} \) the synchronisation set \( \ell \) is required only when the coordinator commits, \( \ell \) is passed to the coordinator together with the commit vote (i.e., as a parameter of the message \text{put}). Differently, a coordinator in the \( \text{d2PCC} \) requires the synchronisation set also during abort. As the abort message could be received from parties, the state of a coordinator should know at any time its synchronisation set. Therefore, the state of coordinator is updated any time it joins with other coordinator.

Before giving the full \( \text{Join} \) code for coordinators, we describe intuitively their behaviour with the transition state diagram in Figure 8.7. The initial state is called \text{state}. While in the initial state, a coordinator may accept requests for being joined (event \text{mayjoin}) with another participants. Any request is confirmed either with \text{join} or \text{nojoin}. In both cases the coordinator returns to the initial state. In the initial state the coordinator can also receive the message to start the execution of the
protocol, either with put (i.e., commit) or abort (i.e., abort). After receiving put the coordinator goes to the state commit. While in state commit, a coordinator behaves like in the original protocol, i.e. by notifying all known parties and by receiving commit confirmation until all parties commit. In such case, the coordinator reaches the state finished. Instead, if the coordinator receives the message abort when being in state or commit, it goes to state abort. While in abort, coordinators notify all known parties and discover the whole set of participants (analogously to commit). When all abort confirmations are received, the coordinator reaches the final state finished.

The Join code for modified coordinators D is presented in Figure 8.8. The first three rules handle the update of of the initial state of the coordinator. When the coordinator is in the initial state state(α, β) and received a request mayjoin(t, f) for updating the state, it may accept the request (rule (14)) by passing to the state waitjoin and sends on t the private ports on which it expects the update confirmation (i.e., message join) or the cancellation (i.e., message nojoin). Rule (2) handles the reception of a join confirmation, which updates the set of known parties, while rule (3) deals with the cancellation. In both cases the coordinator transits to the initial state (possibly updating it).

Rule (4) starts the protocol with the commit vote, while rules (5)–(7) handle committing phase, and are analogous to those in the original protocol (Figure 2.11). There are two subtle differences: (i) channels state and commit have the extra parameter β, which is a list of the ports abort_i of known participants to be used only if the state abort is reached; and (ii) coordinators goes to state finished after commit (rule (7)). Nevertheless, the behaviour for committing coordinators are as in the original proposal (Section 2.4), so we omit its description here.

The behaviour for the aborting phase is given by rules (8)–(13). Rules (8) and (9) start the aborting phase when the coordinator receives a message on channel abort and it is either in the initial state (rule (8)) or in the commit phase (rule (9)). In both cases the coordinator triggers a message abort(β, β', β'', α), which carries the following values:

- β records the set of abort ports of known participants that must still be contacted (analogous to ℓ);
- β' stores the list of ports abort_i of known participants involved in the same transaction, which is typically augmented during the D2PC with the sets sent by other participants (analogous to ℓ');
- β'' records the parties who have already sent their consensus for abort (analogous to ℓ'');
- α store the messages to be released when aborting, i.e., the activation of the compensation.

Note that the behaviour for the aborting phase (rules (10)–(13)) is analogous to the committing phase, and it can be described as follow:
8.4. Compiling Flat CJOIN INTO JOIN.

(1) \[ D \equiv \text{state}(\alpha, \beta) | \text{mayjoin}(t, f) \triangleright t(\text{join}, \text{nojoin}) | \text{waitjoin}(\alpha, \beta) \]

(2) \[ \land \quad \text{waitjoin}(\alpha, \beta) | \text{join}(\beta') \triangleright \text{state}(\alpha, \beta \cup \beta') \]

(3) \[ \land \quad \text{waitjoin}(\alpha, \beta) | \text{nojoin}() \triangleright \text{state}(\alpha, \beta) \]

(4) \[ \land \quad \text{state}(\alpha, \beta) | \text{put}(\ell, \kappa) \triangleright \text{commit}(\ell \setminus \{\text{lock}\}, \ell, \{\text{lock}\}, \alpha, \kappa, \beta) \]

(5) \[ \land \quad \text{commit}([l] \cup \ell, \ell', \alpha, \kappa, \beta) \triangleright \text{commit}(\ell, \ell', \alpha, \kappa, \beta) | l(\ell', \text{lock}, \text{abt}) \]

(6) \[ \land \quad \text{commit}(\ell, \ell', \ell'', \alpha, \kappa, \beta) | \text{lock}(\ell'', l, a) \triangleright \text{commit}(\ell \cup (\ell'' \setminus \ell'), \ell \cup \ell'', \ell' \cup \{l\}, \alpha, \kappa, \beta \cup \{a\}) \]

(7) \[ \land \quad \text{commit}(\emptyset, \ell, \ell', \alpha, \kappa, \beta) \triangleright \text{release}(\kappa) | \text{finished}() \]

(8) \[ \land \quad \text{state}(\alpha, \beta) | \text{abt}(\beta', a) \triangleright \text{abt}(((\beta \cup \beta') \setminus \{\text{abt}\}, \beta \cup \beta', \{\text{abt}, a\}, \alpha) \]

(9) \[ \land \quad \text{commit}(\emptyset, \ell', \alpha, \kappa, \beta) | \text{abt}(\beta', a) \triangleright \text{abt}(((\beta \cup \beta') \setminus \{\text{abt}\}, \beta \cup \beta', \{\text{abt}, a\}, \alpha) \]

(10) \[ \land \quad \text{abt}(\{a\} \cup \beta, \beta', \beta'', \alpha) \triangleright \text{abt}(\beta, \beta', \beta'', \alpha) | a(\beta, \text{abt}) \]

(11) \[ \land \quad \text{abt}(\beta, \beta', \beta'', \alpha) | \text{lock}(\ell'' \setminus \ell', l, a) \triangleright \text{abt}(\beta \cup (\{a\} \setminus \beta'), \beta'' \cup \{a\}, \beta',\alpha) \]

(12) \[ \land \quad \text{abt}(\beta, \beta', \beta'', \alpha) | \text{abt}(\beta', a) \triangleright \text{abt}(\beta \cup (\beta'' \setminus (\beta' \cup \{a\})), \beta \cup \beta'', \beta'' \cup \{a\}, \alpha) \]

(13) \[ \land \quad \text{abt}(\emptyset, \beta, \beta, \alpha) \triangleright \alpha() | \text{finished}() \]

(14) \[ \land \quad \text{finished}() | \text{put}(\ell, \text{cnt}) \triangleright \text{finished}() \]

(15) \[ \land \quad \text{finished}() | \text{lock}(\ell, l, a) \triangleright \text{finished}() \]

(16) \[ \land \quad \text{finished}() | \text{abt}(\beta, a) \triangleright \text{finished}() \]

(17) \[ \land \quad \text{finished}() | \text{mayjoin}(t, f) \triangleright f() | \text{finished}() \]

(18) \[ \land \quad \text{commit}(\ell, \ell', \ell'', \alpha, \kappa, \beta) | \text{mayjoin}(t, f) \triangleright f() | \text{commit}(\ell, \ell', \ell'', \alpha, \kappa, \beta) \]

(19) \[ \land \quad \text{abt}(\beta, \beta', \beta'', \alpha) | \text{mayjoin}(t, f) \triangleright f() | \text{abt}(\beta, \beta', \beta'', \alpha) \]

Figure 8.8: Join code of coordinators.
1. **first phase.** The participant sends the abort vote to every known thread in \( \beta \) (rule (10)). The message contains the list \( \beta' \) of all known participants, and the sender identification \( abt \).

2. **second phase.** The participant collects the messages sent by other parties and updates its own synchronisation set (rule (11) and (12)). A request will be also sent to the new items in the synchronisation set (by repeating the first phase for them).

3. When the set of aborting parties is transitive closed, the protocol terminates locally and the coordinator transits to the state **finished** and releases the compensation \( \alpha \) (rule (13)).

Rules (14)–(16) are for collecting garbage, and state that messages received when the protocol has finished are ignored. Moreover rules (17)-(19) state that the state of a coordinator cannot be updated when the protocol has begun.

### 8.4.2 Correctness of the d2PCC

The proof of correctness of the d2PCC is follows analogously to proof of correctness of the d2PC (Section 2.4). It is split in three steps: (part 1) shows that if all coordinators are ready to commit, then all continuations will be released (assuming fairness); (part 2) handles the case in which all coordinators are in the abort phase, and hence all compensations are released; (part 3) strengthens the results by considering the initial states.

In the following theorems, let \( \sigma_i \) for \( i \in \mathbb{N} \) be the renaming that indexes with \( i \) all the defined names in \( D_i \), i.e. \( dn(D_i) = \{ state_i, mayjoin_i, waitjoin_i, join_i, nojoin_i, put_i, abt_i, commit_i, lock_i, abort_i, finished_i \} \). Also, we write \( D_i \) for \( D\sigma_i \). We write \( A' \), when \( A \) is an empty set or a singleton \( \{ a \} \): in the former case it means 0, in the latter case it represents \( a' \). Moreover, we let a symmetric **lock covering** be a finite family \( \{ \ell_i \}_{i \in I} \) such that \( \ell_i \subseteq \{ lock_j | j \in I \} \), with \( lock_j \in \ell_i \) if and only if \( lock_i \in \ell_j \) for all \( i, j \in I \). Similarly, we let a symmetric **abort covering** be a finite family \( \{ \beta_i \}_{i \in I} \) such that \( \beta_i \subseteq \{ abort_j | j \in I \} \), with \( abort_j \in \beta_i \) if and only if \( abort_i \in \beta_j \) for all \( i, j \in I \). We say a covering to be **full** if the transitive closure of the symmetric relation relates all sets, e.g. by defining \( \ell_i \ell_j \) if \( lock_i \in \ell_j \), then \( \ell_i \ell_j \) for all \( i, j \in I \).

The following results state the correctness of the modified version of the commit protocol, and are adaptations of the results given for the original d2PC algorithm presented in Section 2.4. The first result stands for the case in which all coordinators are in the commit phase.

**Theorem 8.26** (Correctness of the d2PCC, part 1). Let \( P = \Pi_{i \in I} commit_i \{ \ell_i \} \{ lock_i \}, \ell_i, \{ lock_i \}, \alpha_i, \kappa_i, \beta_i \) where \( \{ \ell_i \}_{i \in I} \) is a full symmetric lock covering, and let \( n \) be the cardinality of \( I \). The process \( \text{def } \bigwedge_{i \in I} D_i \ in P \) is strongly confluent, in the
sense that it always converges after a finite number of steps bound by $O(n^2)$ to the configuration
\[
\text{def } \bigwedge_{i \in I} D_i \text{ in } \Pi_{i \in I} \kappa_i \langle \text{finished}_i \rangle
\]

The second result guarantees the termination of the protocol when all coordinators are in the abort phase.

**Theorem 8.27** (Correctness of the d2PCC, part 2). Let $P = \Pi_{i \in I} \text{abort}_i \langle \beta_i, \beta_i \rangle \{\text{abt}_i\}$, \{abt$_i$\} $\cup \{a_i\}, \alpha_i\}$, where $\{\beta_i\}_{i \in I}$ is a full symmetric abt covering and $a_i \in \beta_i$; and let $n$ be the cardinality of $I$. The process $\text{def } \bigwedge_{i \in I} D_i \text{ in } P$ is strongly confluent, in the sense that it always converges after a finite number of steps bound by $O(n^2)$ to the configuration
\[
\text{def } \bigwedge_{i \in I} D_i \text{ in } \Pi_{i \in I} \alpha_i \langle \text{finished}_i \rangle
\]

Finally, the next theorem states the possible computations that can occur from the initial state of the coordinators.

**Theorem 8.28** (Correctness of the d2PCC, part 3). Let $\{\ell_i\}_{i \in I}$ and $\{\beta_i \cup \{a_i\}\}_{i \in I}$ be respectively full symmetric lock and abt covering, and let $A \cup C \cup B$ a partition of $I$, such that

- $\forall i \in A : P_i = \text{state}_i \langle \alpha_i, \beta_i \rangle | \text{abt}_i \langle \{a_i\}, a_i \rangle$, i.e. coordinators that should propose abort;
- $\forall i \in C : P_i = \text{state}_i \langle \alpha_i, \beta_i \rangle | \text{put}_i \langle \ell_i, \kappa_i \rangle$; i.e. coordinators that should vote commit;
- $\forall i \in B : P_i = \text{state}_i \langle \alpha_i, \beta_i \rangle$, i.e. participants that still have no vote to propose.

Then, the process $\text{def } \bigwedge_{i \in I} D_i \text{ in } \Pi_{i \in I} P$ is strongly confluent, in the sense that it always converges after a finite number of steps bound by $O(n^2)$ to

1. $\text{def } \bigwedge_{i \in I} D_i \text{ in } \Pi_{i \in I} \alpha_i \langle \text{finished}_i \rangle$, if $A \neq \emptyset$;
2. $\text{def } \bigwedge_{i \in I} D_i \text{ in } \Pi_{i \in I} \kappa_i \langle \text{finished}_i \rangle$, if $C = I$;
3. $\text{def } \bigwedge_{i \in I} D_i \text{ in } P$, where $P \neq \text{finished}_i \langle \rangle P'$, $P \neq \alpha_i \langle \rangle P'$, $P \neq \kappa_i \langle \rangle P'$; otherwise.

The above result assures that the commit protocol releases all compensations if at least one coordinator is required to abort (i.e., $A \neq \emptyset$), while all continuations are released only if all coordinators are required to commit (i.e., $C = I$). Otherwise, none coordinator finishes, and none compensations nor continuations are released. Note that, differently from the d2PC, in the d2PCC it is not possible to make blocked transactions to abort.

These results will be used when proving the correctness and completeness of the translation of cJoin in Join proposed in the next section.
8.4.3 Translating flat cJoin in Join

As we are interested in computations that start from and lead to consistent states, we restrict our attention to processes that start without active negotiations, that is canonical flat cJoin processes that additionally type □₁. In order to define the encoding of a process we distinguish five types of names:

- f for free names,
- o for top level ordinary names, i.e., names introduced by ordinary definitions outside negotiations,
- m for top level merge names, i.e., names introduced by merge definitions outside negotiations,
- z₀ for names introduced by ordinary definitions local to negotiations,
- zₘ for names introduced by merge definitions of negotiations.

We denote with \( \mathcal{T} \) the set \{f, o, m, z₀, zₘ\}, and let \( t \) range over \( \mathcal{T} \). The encoding of a process \( P \) is defined by taking into account an environment \( \xi \) that associates a type to any free name in \( P \). Formally, \( \xi \) is a partial function over the set of names \( \mathcal{N} \), i.e. \( \xi : \mathcal{N} \to \mathcal{T} \). We denote a particular \( \xi \) with \( \text{dom}(\xi) = \{x_1, \ldots, x_n\} \) as \( x_1 : t_1, \ldots, x_n : t_n \). Also, assuming \( x \notin \text{dom}(\xi') \), we write \( \xi = x : t, \xi' \) for the environment where \( \xi(x) = t \) and \( \xi(y) = \xi'(y) \) for \( y \neq x \). For simplicity, we will abbreviate \( \bar{u} : t \), meaning that every name in the tuple \( \bar{u} \) is assigned with the type \( t \). Similarly, the environment \( \xi = \xi_1 \triangleleft \xi_2 \) is such that \( \xi(x) = \xi_2(x) \) if \( x \in \text{dom}(\xi_2) \) and \( \xi(x) = \xi_1(x) \) otherwise.

When encoding \( P \), the environment \( \xi \) is used to decide whether a free name in \( P \) is an ordinary or a merge one. Therefore, the encoding is well-defined only when \( \text{fn}(P) \subseteq \text{dom}(\xi) \).

**Definition 8.9** (Encoding). The Join process associated to a canonical flat cJoin process \( P \) with type □₁ is \([P]_{\xi} [x \cdot t | x \in \text{fn}(P)]\) (see Figure 8.9).

**Top-level processes.** The function \([P]_{\xi} \) defines the encoding for top-level processes. The encoding is the identity over the empty process. The translation of the emission of a message \( x\langle \bar{u} \rangle \) depends on the type of \( x \) and the names in \( \bar{u} \). For simplicity we assume all names in \( \bar{u} \) being of the same type, but the presentation can be extended by considering all possible combinations of the types of names in \( \bar{u} \). Note that the encoding for the emission on a free name is the identity. Differently, the emission of a message in a top-level ordinary name \( x \) is translated as a message on \( x' \), depending on the type \( t \) of the parameter \( \bar{u} \). Ports \( x^f \), \( x^o \) and \( x^m \) are introduced by the encoding of definitions presented below.

A top-level message \( x\langle \bar{u} \rangle \) on a merge name \( (x : m) \) lives outside a negotiation and cannot be consumed. Moreover, since \( x \) is a defined name it is not observable,
8.4. Compiling flat cjoin into join.

Top-level processes

\[
\begin{align*}
[0]_{\xi} & = 0 \\
[x(\vec{u})]_{x.f, \xi} & = x(\vec{u}) \\
[x(\vec{u})]_{x.o, t, \xi} & = x^t(\vec{u}) \\
[x(\vec{u})]_{x.m, \xi} & = 0 \\
[P|Q]_{\xi} & = [P]_{\xi} | [Q]_{\xi} \\
[\text{abort}]_{\xi} & = 0 \\
[\text{def } D \text{ in } P]_{\xi} = \text{def } [D]_{\xi'} \text{ in } [P]_{\xi'}, \xi' = \xi \cup \{x : o|x \in dn_o(D)\} \cup \{x : m|x \in dn_m(D)\} \\
\end{align*}
\]

Processes in a negotiation

\[
\begin{align*}
[0]_{\xi}^{p.a,j,\ell} & = p(\ell, 0) \\
[x(\vec{u})]_{x.f, u, t, \xi}^{p.a,j,\ell} & = p(\ell, \{x'(\vec{u})\}) \quad \text{for } t \in \{f, o, m\} \\
[x(\vec{u})]_{x.o, \xi}^{p.a,j,\ell} & = p(\ell, \{x'(\vec{u})\}) \quad \text{for } t \in \{f, o, m\} \\
[x(\vec{u})]_{x.t_1, u.t_2, \xi}^{p.a,j,\ell} & = 0 \quad \text{for } t_1 \in \{f, o\} \text{ and } t_2 \in \{z_0, z_m\} \\
[x(\vec{u})]_{x.m, u.t, \xi}^{p.a,j,\ell} & = x^t(\vec{u}, p, a, j, \ell) \quad \text{for } t \in \{f, o, m\} \\
[x(\vec{u})]_{x.z_0, u.t, \xi}^{p.a,j,\ell} & = x^t(\vec{u}, p, a, j, \ell) \quad \text{for } t \in \{z_0, z_m\} \\
[x(\vec{u})]_{x.z, u.t, \xi}^{p.a,j,\ell} & = x^t(\vec{u}, p, a, j, \ell) \quad \text{for } t \in T \\
[x(\vec{u})]_{x.z, m, u.t, \xi}^{p.a,j,\ell} & = 0 \\
[\text{abort}]_{\xi}^{p.a,j,\ell} & = a(a, a) \\
[\text{def } D \text{ in } P]_{\xi}^{p_1,p_2, (a_1,a_2), (j_1,j_2), \ell} = \text{def } [D]_{\xi'}^{p_1,p_2, (a_1,a_2), (j_1,j_2), \ell} \\
\quad \text{if } count(P) = 1 \text{ and } \xi' = \xi \cup \{x : z_0|x \in dn_o(D)\} \cup \{x : z_m|x \in dn_m(D)\} \\
[\text{def } D \text{ in } P]_{\xi}^{p_1,p_2, (a_1,a_2), (j_1,j_2), \ell} = \text{def } [D]_{\xi'}^{p_1,p_2, (a_1,a_2), (j_1,j_2), \ell} \\
\quad \text{if } count(P) = 2 \text{ and } \xi' = \xi \cup \{x : z_0|x \in dn_o(D)\} \cup \{x : z_m|x \in dn_m(D)\} \\
[P|Q]_{\xi}^{p_1,p_2, (a_1,a_2), (j_1,j_2), \ell} = [P]_{\xi}^{p_1,a_1,j_1,\ell} | [Q]_{\xi}^{p_2,a_2,j_2,\ell} \\
\quad \text{if } count(P) = count(Q) = 1
\end{align*}
\]

Figure 8.9: Encoding of canonical flat processes.
and hence it is encoded as the inert process 0. Analogously for abort, which is meaningless outside contracts. The encoding of a parallel composition $P|Q$ is the parallel composition of encoding of $P$ and $Q$.

Note that the environment $\xi$ is updated when encoding a top-level process with local definitions. That is, when defining $[\text{def } D \text{ in } P]_{\xi}$, both $D$ and $P$ are encoded by taking into account also $dn_{\alpha}(D)$ and $dn_{m}(D)$.

When a negotiation is translated into Join, it is associated with a new coordinator $D$ (Figure 8.8), which will monitor the execution of the contract. As $P$ will run as part of a negotiation, it is encoded as $[P]_{\xi}^{\text{put, abort, magjoin, lock}}$, where put, abort, lock and mayjoin are ports in $dn_{\alpha}(D)$. We can safely assume that $P$ initiates with a unique thread because we are translating canonical processes with type $\square_1$, and therefore negotiations $[P : Q]$ appear in definitions with $\text{count}(P) = 1$ (Proposition 8.9). The compensation $Q$ is encoded as a top-level process, which is activated with a message on the local port $\text{cmp}$. As $\text{cmp}$ is used only to initialize the state of the coordinator ($\text{state}(\{\text{cmp}\}, \{\text{abort}\})$), the message $\text{cmp}()$ is emitted only when the coordinator (and consequently the contract) aborts. Also note that in the initial state the coordinator knows only itself (i.e., parameter $\{\text{abort}\}$).

**Processes in negotiations.** The auxiliary encoding $[\_]_{\xi}^{p, a, j, t}$ describes the implementation of a thread being monitored by a manager $D$, which defines channels $p$ and $a$ for receiving commit or abort confirmations, and $j$ for updating the state of the coordinator. The set $\ell$ collects the references to known parties in the same negotiation (called *synchronisation set*).

The inert process 0 in a negotiation means thread completion, hence it is translated as $p(\ell, \emptyset)$ to notify the coordinator that the thread is ready to commit. The message $p$ contains $\ell$ to inform $D$ about the lock ports of known parties. Note that no continuation is set for this coordinator.

The encoding of a message $x(\bar{u})$ requires a case analysis on the different kinds of names involved in it. When the message is sent to a free name or to an ordinary name defined at the top-level (i.e. $x : f$ or $x : m$) there are two different cases. If the arguments $\bar{u}$ are not local names (i.e. $\bar{u} : f$, $\bar{u} : o$, or $\bar{u} : m$), then the thread is attempting to close the negotiation by releasing $x(\bar{u})$. Hence, it is encoded as a commit notification, i.e. $p(\ell, \{x(\bar{u})\})$ when $x : f$, and $p(\ell, \{x^t(\bar{u})\})$ when $x : o$, where $t$ is the type of $\bar{u}$. Note that $x(\bar{u})$ and $x^t(\bar{u})$ are the continuation for the coordinator, which will be released only if the negotiation finally commits.

Instead, when the arguments are names defined in a contract (i.e. $\bar{u} : z_o$ or $\bar{u} : z_m$), then the negotiation can enter in a stall situation unless other participants abort the whole contract. Consider the negotiation $P = [\text{def } x \triangleright P' \text{ in } y(x)|P : Q]$ that cannot commit (even when $P' = 0$) since $y(x)$ cannot be consumed, which is required to enable the commit of the negotiation (COMMIT requires all local names not to appear in messages). The only possibility for $P$ to conclude is when $P'$ aborts.

This possible stall situation is simulated by encoding this kind of messages as 0, in this way the thread finishes without notifying its coordinator neither commit nor
8.4. Compiling Flat CJOIN Into Join.

Abort, and the coordinator will be blocked (unless one of its parties aborts).

Similarly, merge names are encoded by taking into account the type of their parameters. Nevertheless, messages to merge ports have a different behaviour from messages sent to ordinary ports. Consider the following process

\[ P = \text{def } x(y) \triangleright P_1 \text{ in } [x(x) | \text{def } z(x) \triangleright P_2 \text{ in } 0 : Q] \]

This process has two possible reductions, one by firing the definition \( x(y) \triangleright P_1 \), which produces \( P = \text{def } x(y) \triangleright P_1 \text{ in } [P_1[x/y]|\text{def } z(x) \triangleright P_2 \text{ in } 0] \), and the other by applying the rule (COMMIT), obtaining \( \text{def } x(y) \triangleright P_1 \text{ in } x(x) \). To model this non-deterministic behaviour we introduce two kind of ports: \( x_s^t \) and \( x_a^t \) where \( t \) is the type of the parameter and the subscripts \( s \) and \( z \) are associated to one of the two different behaviours. In particular, port \( x_s^t \) (with \( t \in T \)) is used to encode the behaviour of a merge name that receives names of type \( t \) and continues the execution of the negotiation. Instead, port \( x_a^t \) allows also the possibility of committing a contract even when the message is not consumed. Note that the emission on \( x \) is translated as a message that carries the values \( p, a, j \) and \( \ell \) for interacting with the manager. (A detailed discussion about encoding merge definitions is below).

A name \( x \) defined in a negotiation is encoded by using five different ports (one port \( x^t \) for any possible type \( t \) in \( T \)). Then a message \( x(u) \) is translated as a message on \( x^t \), where \( t \) is the type of \( u \). Note also that the encoded version of a local message carries the ports \( p, a \) and \( j \), and the list of known parties \( \ell \) that identify the thread.

The constant process \( \text{abort} \) is translated into a message \( a [\{ a \}, a] \) that informs the manager about the abort. The parameters are the list \( \{ a \} \) of the known parties and the coordinator that reaches the abort (i.e., \( a \)).

We give two different rules for encoding \( \text{def } D \text{ in } P \), depending on whether \( \text{count}(P) = 1 \) or \( \text{count}(P) = 2 \). If \( \text{count}(P) = 1 \), then just one coordinator is required. Instead, when \( \text{count}(P) = 2 \), the encoding requires two coordinators: two ports \( p_1 \) and \( p_2 \) for notifying the commit, two ports \( a_1 \) and \( a_2 \) for aborting, and two ports \( j_1 \) and \( j_2 \) for updating the state of two coordinators. The generation of different coordinators is due to the encoding of fork definitions described below. Also note that the translation \( D \) and \( P \) considers the updated environment \( \xi \), which takes into account the names introduced by \( D \).

The encoding of the parallel composition \( P \parallel Q \) requires information about two different coordinators. In this case, \( P \) is encoded by using \( p_1, a_1, j_1 \) and \( Q \) by using \( p_2, a_2, j_2 \).

Definitions. The encoding of definitions is in Figure 8.10. We recall that \( [\_]^0_\xi \) is for top-level definitions, while \( [\_]^1_\xi \) is for definitions inside negotiations. In both cases the encoding of a conjunction \( D \land E \) is the obvious one.

The translation of a top-level definition of the port \( x \) creates three new ports \( x^f \), \( x^o \) and \( x^m \), which handle free, ordinary and merge parameters respectively. Such ports are associated with different translations of the guarded process \( P \). Note that for the port \( x^t(u) \), \( P \) is encoded by considering the environment \( \xi \triangleleft \{ x : o, u : t \} \),
DEFINITIONS

\[[D \land E]_\xi^i = [D]_\xi^i \land [E]_\xi^j\]

\[[x(\bar{u}) \triangleright P]_\xi^0 = \bigwedge_{\ell \in \{f, o, m\}} x^f(\bar{u}) \triangleright [P]_{\xi \in \{x, o, u, t\}}\]

\[[x(\bar{u}) \triangleright P]_\xi^1 = \bigwedge_{\ell \in \mathcal{T}} x^f(\bar{u}, p, a, j, \ell) \triangleright \bigwedge_{\ell \in \{x \in [x, z, o, u, t]\}} [P]_{\xi \in \{x, p, a, j, \ell\}} \quad \text{if } \text{count}(P) = 1\]

\[[x(\bar{u}) \triangleright P]_\xi^1 = \bigwedge_{\ell \in \mathcal{T}} x^f(\bar{u}, p, a, j, \ell) \triangleright \bigwedge_{\ell \in \{x \in [x, z, o, u, t]\}} [P]_{\xi \in \{x, p, a, j, \ell\}} \quad \text{if } \text{count}(P) = 2\]

\[\text{def } f(\bar{u}) \triangleright 0 \land t(\bar{u}, \bar{a}) \triangleright \text{def } D \in [P]_{\xi \in \{x, z, o, u, t\}} \mid \text{state}(\emptyset, \{a, b\}) \mid \text{ok}(\{a, b\})\]

\[\text{in } j(f, t)\]

Figure 8.10: Encoding of canonical flat definitions.
8.4. Compiling Flat CJoin into Join.

in which \( \bar{u} \) is taken as being of type \( t \). We recall that, for simplicity, we assume all names in \( \bar{u} \) being of the same kind. Note also that, when encoding \( P \), the defined name \( x \) is an ordinary top-level name (i.e., \( x : o \)).

Similarly, when encoding a top-level join rule \( x_1 \langle \bar{u}_1 \rangle x_2 \langle \bar{u}_2 \rangle \triangleright P \) both \( x_1 \) and \( x_2 \) are associated with three ports, and hence the encoding produces nine different definitions, one for each possible combination on the types of \( \bar{u}_1 \) and \( \bar{u}_2 \).

Ports defined locally by negotiations are encoded with five different ports, one for any kind of parameter (i.e., \( u : t \) for \( t \in T \)). Then, a rule \( x \langle \bar{u} \rangle \triangleright P \) where \( \text{count}(P) = 1 \) is translated into five rules. Each rule introduces a new port \( x^t \) \( (t \in T) \) to handle a particular kind of received names \( \bar{u} \). Additionally, the new ports \( x^t \) have the extra parameters \( p, a, j \) and \( \ell \), which identify the thread in which \( x \) runs. In fact, the guarded process \( P \) is encoded w.r.t. the values \( p, a, j, \ell \) of the manager of the fetched message on \( x^t \).

Similarly, a \texttt{fork} (i.e., \( x \langle \bar{u} \rangle \triangleright P \) where \( \text{count}(P) = 2 \)) is encoded with five rules. In order to translate \( P \), two coordinators are needed, hence the activation of this rule creates a new coordinator that will join the manager of the thread that forks. Therefore, when the translated definition is fired, a request for join is sent to the manager of the thread that forks (\( j \langle t, f \rangle \)). If this request is answered with a message on \( f \), then the thread finishes because the manager is already executing the commit protocol (this is the case when some thread in the same negotiation has already reached abort, hence it is useless to continue the computation). Otherwise, if the manager can be joined, then a message on \( t \) is received. In this case, a new coordinator \( D \) is created, which defines the ports \texttt{put}, \texttt{abt}, \texttt{mayjoin} and \texttt{lock}. Note that \( P \) is encoded by considering the ports of the new coordinator \( D \), and channels \( p, a, \) and \( j \) associated to the thread that forks (they are retrieved from the message on \( x \)). The channel \texttt{lock} is added to the participant list \( \ell \), which will be common to both new threads. Additionally, the initial state of the new coordinator is set to contain a null compensation and the \texttt{abt} ports of joined threads (i.e., \texttt{state}(\emptyset, \{\texttt{abt}, a\})). Finally, the fork is confirmed to the original coordinator by sending the identity of the new coordinator \( \texttt{ok}\{\texttt{abt}\} \).

The remaining shape for ordinary definitions is a \texttt{join} \( x \langle \bar{u} \rangle \mid y \langle \bar{v} \rangle \triangleright P \) where two different threads are synchronised and only one of them remains active (\( \text{count}(P) = 1 \)). The translation states that the execution of a join must query involved coordinators whether they are able to be joined or not (messages \( j_1 \langle t_1, f_1 \rangle \) and \( j_2 \langle t_2, f_2 \rangle \)). The join can take place only when both coordinators agree to join by sending the corresponding messages \( t_1 \) and \( t_2 \). If at least one of them does not agree, then the join cannot take place (see definitions handling \( f_1 \) and \( f_2 \)). If coordinators can be joined, then \( P \) is activated on the first thread (note \( P \) is encoded by using the ports \( p_1, a_1 \) and \( j_1 \)) and the second is committed (\( p_2 \langle \ell_1 \cup \ell_2, \emptyset \rangle \)). The synchronisation set for both coordinators is updated to \( \ell_1 \cup \ell_2 \cup \{\text{lock}\} \). Finally, the join operation is confirmed to both coordinators by sending the messages \( \text{ok}_1 \{\{a_2\}\} \mid \text{ok}_2 \{\{a_1\}\} \).

The last rules encode merge definitions, whose basic shapes are similar to ordinary ones, consequently they are translated analogously. The main difference is
that merge names have a non-deterministic behaviour, because a negotiation can commit also when it contains messages addressed to a merge name or it can wait until those messages are consumed. Therefore, a merge name \( x \) is encoded with two different kinds of ports: \( x' \) encoding the waiting behaviour (i.e., the negotiation will not commit until the message is consumed), and \( x'' \) allowing both behaviours because they can choose non-deterministically either to commit or to wait. Note that messages sent to merge names that are not used inside a negotiation are discarded when the thread commits, because they are not observed (since they are defined names) and cannot be used by any reduction.

Finally, when a merge name \( x \) is defined more than once in a conjunction, redundant definitions for \( x^k \) are introduced. However, redundant definitions do not change the behaviour of a process. Additionally, merge definitions are useless when appearing inside negotiations, because for flat processes no sub-negotiations exist that can be merged. Hence, we omit their translation (the special symbol \( \top \) denotes this fact).

In the rest of this section we discuss the correctness and completeness of our encoding. We will use \( \rightarrow J \) and \( \rightarrow_{cJ} \) to distinguish reductions in Join from those in cJoin. Firstly, we introduce some definitions

**Definition 8.10 (Standard cJoin negotiation).** A negotiation \( N_i \) is a standard cJoin negotiation if it occurs under a definition \( D \), i.e. \( P = \text{def} \ D \in N_i \mid P' \), and has the following shape

\[
N_i = [\Pi_{j \in \text{G}_i} \text{AB}_i \text{ abort} \\
\mid \text{def} \ D_i \in (\Pi_{j \in \text{G}_i} x_{G_i,j} \langle u_{G_i,j} \rangle)[\Pi_{j \in \text{M}_i} x_{M_i,j} \langle u_{M_i,j} \rangle][\Pi_{j \in \text{L}_i} x_{L_i,j} \langle u_{L_i,j} \rangle) : Q_i]
\]

where:

- \( \Pi_{j \in \text{G}_i} \text{AB}_i \text{ abort} \), stands for all agents \( \text{abort} \) produced inside a negotiation;
- \( \Pi_{j \in \text{G}_i} x_{G_i,j} \langle u_{G_i,j} \rangle \) are the messages sent to the free or ordinary ports \( x_{G_i,j} \), which are kept inside a negotiation until commit, i.e. \( \forall j \in \text{G}_i : x_{G_i,j} \in (\text{dn}_\alpha(D) \cup \text{fn}(P)) \);
- \( \Pi_{j \in \text{M}_i} x_{M_i,j} \langle u_{M_i,j} \rangle \) are the messages sent to merge names \( x_{M_i,j} \), i.e. \( \forall j \in \text{M}_i : x_{M_i,j} \in \text{dn}_m(D) \); and
- \( \Pi_{j \in \text{L}_i} x_{L_i,j} \langle u_{L_i,j} \rangle \) are messages to local ports \( x_{L_i,j} \), i.e. \( \forall j \in \text{M}_i : x_{L_i,j} \in \text{dn}(D_i) \).

Note that the number of agents in \( N_i \), i.e. \( \text{count}(N_i) \) (see Figure 8.3), is given by the expression \( \text{AB}_i + \text{G}_i + \text{M}_i + \text{L}_i \).

Moreover we say a standard cJoin negotiation \( N_i \) is not finished if \( \text{AB}_i = 0 \) (i.e., there is no abort \( N_i \)), and either \( \text{L}_i \neq \emptyset \) or some name in one \( u_{G_i,j} \) or \( u_{M_i,j} \) is a local name (i.e. there is a local name).

The Join counterpart of a standard cJoin negotiation is defined as follow.
Definition 8.11 (Standard Join negotiation). A join process \( R_i \) is said the standard Join negotiation associated to \( \mathcal{N}_i \) if it has the following form

\[
R_i = \text{def } \left[ D_{i,j} \right]_{j \in \mathcal{C}_i} \land \left( \bigwedge_{j \in \mathcal{C}_i} D_{i,j} \land \text{cmp}_{i,j} \right) \triangleright Q_{i,j} \]

\[
\text{in } \Pi_{j \in \mathcal{C}_i} \text{ state}_{i,j} \langle \alpha_{i,j}, \beta_{i,j} \rangle
\]

\[
| \Pi_{k \in C_{A_i}} \text{ abt}_{i,k} \langle \beta_{i,k}^l, s_{i,k} \rangle |
\]

\[
| \Pi_{k \in C_{G_i}} [x_{G_i,k} \langle u_{G_i,k} \rangle]_{\xi_i} \text{ put}_{i,k} \langle \beta_{i,k}^l, s_{i,k} \rangle \text{ may join}_{i,k} \langle \ell_{i,k} \rangle |
\]

\[
| \Pi_{k \in C_{P_i}} \text{ put}_{i,k} \langle \ell_{i,k}, \emptyset \rangle |
\]

where

- \( C_i \geq \text{count}(\mathcal{N}_i) \), that is the number \( C_i \) of coordinators involved in the negotiation is at least the number of agents in \( \mathcal{N}_i \) (we recall that the encoding generates a coordinator for any agent in \( \mathcal{N}_i \), \( C_i > \text{count}(\mathcal{N}_i) \) when some thread in the negotiation has finished (reducing to 0).

- The set of coordinator \( D_{i,j} \) for \( j \in 0..C_i \), can be partitioned as follow \( 0..C_i = C_{G_i} \cup C_{M_i} \cup C_{L_i} \cup C_{A_i} \cup C_{P_i} \) s.t.:
  - \( |C_{G_i}| = |G_i| \) are the coordinators associated to threads that carry on global messages,
  - \( |C_{M_i}| = |M_i| \) are the coordinators of threads that has messages to merge ports
  - \( |C_{L_i}| = |L_i| \) are the coordinators for messages to local ports,
  - \( |C_{A_i}| = AB_t \) are coordinators of aborted threads, and
  - \( C_{P_i} \) are the threads that have already finished.

- \( \{ \beta_{i,j} \}_{j \in \mathcal{C}_i} \) and \( \{ \ell_{i,j} \}_{j \in \mathcal{C}_i} \) are respectively symmetric abt and lock covering.

- Additionally, \( \Pi_{j \in \mathcal{C}_i} Q_{i,j} = Q_i \) and \( \alpha_{i,j} = \{ \text{cmp}_{i,j} \} \).

Note that the standard Join negotiation \( R_i \) is such that all coordinators are in the initial state (i.e., \( \text{state}_{i,j} \langle \alpha_{i,j}, \beta_{i,j} \rangle \)), all coordinators that are associated to aborted agents have available an abort message (i.e., \( \Pi_{k \in C_{A_i}} \text{ abt}_{i,k} \langle \beta_{i,k}^l, s_{i,k} \rangle \)), all finished agents have vote commit (\( \Pi_{k \in C_{P_i}} \text{ put}_{i,k} \langle \ell_{i,k}, \emptyset \rangle \)), the remaining coordinators are associated with the encoding of the corresponding messages. In addition, note that the parallel composition of the compensations assigned to all coordinators are equivalent to the compensation of \( \mathcal{N}_i \).

The next result states the correspondence among reductions in cJoin and Join.

Lemma 8.29. Let \( P \equiv \text{def } D \text{ in } M \) be a canonical flat process. If \( P \rightarrow^*_{cJ} P' \) then:
\[ P' \equiv_e \text{def } D' \text{ in } M' | \Pi_{i \in 1..N} N_i, \text{ where } N_i \text{ are standard } \text{cJoin} \text{ negotiations}, \]

\[ \exists Q \text{ s.t. } [P]_{\xi} \rightarrow Q \text{ and } Q \equiv_e \text{def } [D']_{\xi} \text{ in } [M']_{\xi} | \Pi_i R_i, \text{ where each } R_i \text{ is the standard } \text{Join} \text{ negotiation associated to } N_i, \text{ for suitable } \xi \text{ and } \xi'. \]

**Proof.** The proof follows by induction on the length of the derivation \( P \rightarrow^{*}_{cJ} P' \).

- **Base case** \( P \rightarrow^{0}_{cJ} P' \). Then, \( P' = \text{def } D \text{ in } M \). Moreover, \([P]_{\xi} = \text{def } [D']_{\xi} \text{ in } [M]_{\xi}\), for a suitable \( \xi \). Then it is enough to take \( Q = [P]_{\xi} \).

- **Inductive Step** \( P \rightarrow^{*}_{cJ} P'' \rightarrow^{*}_{cJ} P' \). By inductive hypothesis on \( P \rightarrow^{*}_{cJ} P'' \)

  \[ P'' \equiv_e \text{def } D'' \text{ in } M'' | \Pi_{i \in 1..N} N_i', \text{ where } N_i' \text{ are standard } \text{cJoin} \text{ negotiations}, \]

  \[ \exists Q' \text{ s.t. } [P]_{\xi} \rightarrow Q \text{ and } Q' \equiv_e \text{def } [D''']_{\xi'} | \Pi_i R_i', \text{ where each } R_i' \text{ is the standard } \text{Join} \text{ negotiation associated to } N_i', \text{ for suitable } \xi \text{ and } \xi'. \]

Then, the proof follows by straightforward analysis of applied rule for \( P'' \rightarrow^{*}_{cJ} P' \). The interesting cases are when a negotiation is finished either by committing or aborting.

- **Rule (commit).** If the last applied rule is (commit), then there is a standard negotiation \( N_h \) s.t.

  \[ N_h' = [\Pi_{j \in 0..A B} \text{abort}] \text{def } D_h \text{ in } (\Pi_{j \in G_h} x_{G_h,j} \langle \bar{u}_{G_h,j} \rangle | \Pi_{j \in M_h} x_{M_h,j} \langle \bar{u}_{M_h,j} \rangle | \Pi_{j \in L_h} x_{L_h,j} \langle \bar{u}_{L_h,j} \rangle : Q_h], \]

  and \( AB_h = 0, L_h = 0 \), for all \( j \in (G_h \cup M_h) : \bar{u}_j \not\in dn(D_h) \). Then

  \[ P' \equiv_e \text{def } D'' \text{ in } M'' | M'_h | \Pi_{i \in 1..N} N'_i \]

where \( M'_h = \Pi_{j \in G_h} x_{G_h,j} \langle \bar{u}_{G_h,j} \rangle | \Pi_{j \in M_h} x_{M_h,j} \langle \bar{u}_{M_h,j} \rangle \) and \( N'_i \) are standard cJoin negotiations. Since \( AB_h = 0, L_h = 0 \), and for all \( j \in (G_h \cup M_h) : \bar{u}_j \not\in dn(D_h) \), by using heating/cooling rules \( R_h \) can be written as follow

\[ N_h' = [(\Pi_{j \in G_h} x_{G_h,j} \langle \bar{u}_{G_h,j} \rangle | \Pi_{j \in M_h} x_{M_h,j} \langle \bar{u}_{M_h,j} \rangle)] \text{def } D_h \text{ in } 0 : Q_h] \]

Therefore, the corresponding standard Join negotiation is as follow

\[ R_h = \text{def } [D_h]_{\bar{u}_{G_h,k}} \wedge (\bigwedge_{j \in 1..C_h} D_{h,j} \wedge \text{comp}_{h,j}() \triangleright Q_{h,j}) \]

in \( \Pi_{j \in 1..C_h} state_{h,j}(\alpha_{h,j}, \beta_{h,j}) \)

\[ \Pi_{k \in C_G_h} \text{put}_{h,k}(\bar{u}_{G_h,k}, abt_{h,k}, mayjoin_{h,k}, \ell_{h,k}) \]

\[ \Pi_{k \in C_M_h} \text{put}_{h,k}(\bar{u}_{M_h,k}, abt_{h,k}, mayjoin_{h,k}, \ell_{h,k}) \]

\[ \Pi_{k \in C_P_h} \text{put}_{h,k}(\ell_{h,k}, 0) \]

Since all parameter in messages (i.e., \( \bar{u}_{G_h,k} \) and \( \bar{u}_{M_h,k} \)) are not local, we have

\[ R'_h = \text{def } [D_h]_{\bar{u}_{G_h,k}} \wedge (\bigwedge_{j \in 1..C_h} D_{h,j} \wedge \text{comp}_{h,j}() \triangleright Q_{h,j}) \]

in \( \Pi_{j \in 1..C_h} state_{h,j}(\alpha_{h,j}, \beta_{h,j}) \)

\[ \Pi_{k \in C_G_h} \text{put}_{h,k}(\ell_{h,k}, x_{G_h,k}(\bar{u}_{G_h,k})) \]

\[ \Pi_{k \in C_M_h} \text{put}_{h,k}(\bar{u}_{M_h,k}, abt_{h,k}, mayjoin_{h,k}, \ell_{h,k}) \]

\[ \Pi_{k \in C_P_h} \text{put}_{h,k}(\ell_{h,k}, 0) \]
Moreover, since for any merge name \( x \) with content \( \bar{u} \) of type \( t \), there exist a reduction rule \( x^h_{\bar{u},p,a,j,\ell} \triangleright p(\ell,\emptyset) \), then

\[
R'_h \rightarrow^* R''_h \equiv_e \text{def } [D_h]_{\bar{x}_h} \land (\bigwedge_{j \in 1..C_h} D_{h,j} \land \text{cmp}_{h,j} ) \triangleright Q'_h \]

\[
\in \Pi_{j \in 1..C_h} \text{state}_{h,j}(\alpha_{h,j}, \beta_{h,j}) \quad \Pi_{k \in C_{G_h}} \text{put}_{h,k}(\ell_{h,k}, \{x^h_{\bar{u}_{G_h,k}}(\bar{u}_{G_h,k})\}) \quad \Pi_{k \in C_{M_h}} \text{put}_{h,k}(\ell_{h,k}, \emptyset) \quad \Pi_{k \in C_{R_h}} \text{put}_{h,k}(\ell_{h,k}, \emptyset)
\]

By Theorem 8.28, \( R''_h \rightarrow^* \Pi_{k \in C_{G_h}} x^h_{\bar{u}_{G_h,k}}(\bar{u}_{G_h,k}) = [M_h]_{\xi} \). Hence, \( Q'' \rightarrow^* Q \equiv_e \text{def } [D]_{\xi} \in [M'']_{\xi} \mid [M_h]_{\xi} \mid \Pi_{i \in 1..N \setminus \{h\}} R'_i \).

- **Rule (ABORT).** If the last reduction is by using (ABORT), then the proof follows analogously to the previous case, but noting that at least one coordinator aborts.

\[ \square \]

**Theorem 8.30 (Correctness).** Let \( P \) be a canonical flat process and \( \vdash P : \Box_1 \). If \( P \rightarrow^* P' \) and \( \vdash P' : \Box_1 \), then \( \exists Q \text{ s.t. } [P]_{\{x : f(x \in \text{fn}(P))\}} \rightarrow^* Q \) and \( \forall x : P' \downarrow_x \Rightarrow Q \downarrow_x \)

**Proof.** If \( P \equiv \text{def } D \in M \) the proof follows by noting that \( P' \equiv_e \text{def } D' \in M' \). By Lemma 8.29, \( \exists Q \text{ s.t. } [P]_{\{x : f(x \in \text{fn}(P))\}} \rightarrow^* Q \) and \( Q \equiv_e \text{def } [D']_{\xi} \in [M']_{\xi} \). It is easy to notice that \( \forall x : P' \downarrow_x \Rightarrow Q \downarrow_x \). Otherwise, \( P \) has no local definitions, then it is the parallel composition of messages on free ports, the inert process 0 and abort. Therefore, the only possibility if \( P' = P \), and hence condition \( \forall x : P' \downarrow_x \Rightarrow Q \downarrow_x \) immediately follows. \( \square \)

In order to state the completeness result, we introduce the normalisation procedure on Join processes. Given a Join process \( P \), \( norm(P) \) denotes the process obtained by the repeated application (until termination) of definitions in coordinators \( D \) (definitions in Figure 8.8) and rules for joining coordinators introduced by the encoding (ports \( f_i \) and \( t_i \) in Figure 8.10). That is, the normalisation procedure ends the execution of all instances of the \( d2\text{PCC} \). It is worth noting that \( norm(P) \) is defined for all \( P \) since the \( d2\text{PCC} \) algorithm always terminates by Theorem 8.28. Given a standard Join negotiation \( R_i \) for \( R'_i \) derivable from \( R_i \) (i.e., \( R_i \rightarrow^* R'_i \)) we say \( R_i \) finished if \( \text{norm}(R'_i) \equiv_e \text{def } D_i \in \Pi_{j \in 1..C_i} \text{finished}_j \mid R''_i \), i.e., all coordinators reach the state \( \text{finished} \) during normalisation. Differently, \( R'_i \) is not finished if \( \text{norm}(R'_i) \equiv_e \text{def } D_i \in \Pi_{j \in 1..C_i} \text{state}_j(\alpha_j, \beta_j)|R'_i \), i.e., no coordinator has decided. From Theorem 8.28, each \( R'_i \) is either finished or not finished.

**Lemma 8.31.** Let \( P \equiv \text{def } D \in M \) be a canonical flat process. If \( [P]_{\xi} \rightarrow^* Q \), then

- \( Q \equiv_e \text{def } [D]_{\xi} \in [M']_{\xi} \mid \Pi_{i \in 1..N} R'_i \mid \Pi_{k \in 1..F} T'_k \), where \( R'_i \) are not finished standard Join negotiations, while \( T'_k \) are finished. Moreover \( \text{norm}(\Pi_{k \in 1..F} T'_k) \equiv [M_2]_{\xi} | R' \), and
\( P \rightarrow_c^* P' \equiv c \text{ def } D' \text{ in } M_1' \mid M_2 \mid \Pi_{i \in 1..N} N_i \text{ where } N_i \text{ are not finished standard cJoin negotiations (see Definition 8.10) corresponding to the not finished standard Join negotiations } R_i'. \)

\( \text{norm}(Q) \equiv c \text{ def } [D]^0_{\xi} \text{ in } [M_1'|M_2]_{\xi'} \mid \text{norm}(\Pi_{i \in 1..N} R_i') \mid R' \)

**Proof.** By induction on the length of the derivation \([P]_\xi \rightarrow_c^* Q\).

- **Base case** \( Q = [P]_\xi \). It is enough to take \( P' = P \). Clearly \( P \rightarrow_c P' = P \). Since \( \vdash P : \square_1 \), then \( Q = [P]_\xi \) has no coordinators.

- **Inductive step** \([P]_\xi \rightarrow_c Q'' \rightarrow_c Q\). By inductive hypothesis on \([P]_\xi \rightarrow_c Q''\)
  
  - \( Q'' \equiv c \text{ def } [D'']_{\xi''} \text{ in } [M_1'']_{\xi''} \mid \Pi_{i \in 1..N} R_i'' \mid \Pi_{k \in 1..F} T_k'', \) where \( R_i'' \) are not finished standard Join negotiations, while \( T_k'' \) are finished negotiations. In addition, \( \text{norm}(\Pi_{k \in 1..F} T_k'') \equiv [M_2'']_{\xi''} \mid R'' \), where \( R'' \) stands for the code of coordinators, and
  
  - \( P \rightarrow_c P'' \equiv c \text{ def } D'' \text{ in } M_1'' \mid M_2'' \mid \Pi_{i \in 1..N} N_i'' \text{ where } N_i'' \text{ are not finished standard cJoin negotiations (see Definition 8.10) corresponding to the not finished standard Join negotiations } R_i'' \).

Then the proof proceeds by analysing the definition used for reducing \( Q'' \rightarrow_c Q \).

- for some \( h \in 1..N'' \), \( R_h'' \rightarrow_c R_h' \). There are two different cases:
  1. The applied definition is in \( D_h,c \) or in one auxiliary definition introduced by the encoding. Since the protocol is confluent, then \( \text{norm}(R_h') = \text{norm}(R_h'') \). Then, it is enough to take \( P' = P'' \), which satisfies all conditions.
  2. The reduction occurs by using a definition in \([D'']_{\xi''}^1\). Clearly, such definition is one generated by \([x\langle \bar{a} \rangle \triangleright P_3]_{\xi''}\), for \( \text{count}(P_3) = 1 \) and \( \text{count}(P_3) = 2 \), or \([x\langle \bar{a} \rangle | x_1\langle \bar{a}_1 \rangle \triangleright P_3]_{\xi''}\), where \( \text{count}(P_3) = 1 \). We show here the last case, which is the most interesting one. Hence, there exists a definition with the following shape

\[
x_1^{t_1}\langle \bar{a}_1,p_1,a_1,j_1,\ell_1 \rangle|x_2^{t_2}\langle \bar{a}_2,p_2,a_2,j_2,\ell_2 \rangle \triangleright
\text{def } f_1() \mid f_2() \triangleright 0 \quad \land \quad f_1() \mid t_2(\text{ok},\text{no}) \triangleright \text{no}\rangle
\land t_1(\text{ok},\text{no}) \mid f_2() \triangleright \text{no}\rangle
\land t_1(\text{ok},\text{no}) \mid t_2(\text{ok2},\text{no2}) \triangleright [P_3]_{\xi''}^1_{a_1 j_1 t_1 \bar{a}_1 t_2}{\phi}_h(p_1 a_1 j_1 t_1 \bar{a}_1 t_2) \triangleright [P_3]_{\xi''}^1_{a_2 j_2 t_2}{\phi}_h(p_2 a_2 j_2 t_2)
\mid ok_1\{a_1\}] \mid ok_2\{a_2\}\rangle
\mid j_1(f_1,t_1)\mid j_2(f_2,t_2)
\]

Moreover, \( R_h'' \) contains the messages for activating the rule, i.e.

\[
\Pi_{k \in C_{\ell_h}} [x_{L_h,k}\langle \bar{a}_{L_h,k} \rangle]_{\xi_h}^{\text{put}_{h,k},\text{abi}_{h,k},\text{may}_{h,k},\text{join}_{h,k},\ell_h,k} =
\text{x}_1^{t_1}\langle \bar{p}_1,p_1,a_1,j_1,\ell_1 \rangle|x_2^{t_2}\langle \bar{p}_2,p_2,a_2,j_2,\ell_2 \rangle| M_{L_h}
\]
Hence \( R'_h \) is obtained by removing the messages \( x^1_1 \{ \bar{a}_1, p_1, a_1, j_1, \ell_1 \} \) and \( x^2_2 \{ \bar{a}_2, p_2, a_2, j_2, \ell_2 \} \), and by activating the guarded process of the rule. Then, when normalising, since \( R'_h \) is not finished, then coordinators \( j_1 \) and \( j_2 \) can be joined. Hence, coordinators answer to the join request with \( t_i \). Therefore, it is possible to activate

\[
[e P_3 e^{\bar{a}_1, a_1, j_1, \ell_1} | p_g(\ell_1 \cup \ell_2, \emptyset) | ok_1(\{a_2\}) | ok_2(\{a_1\})]
\]

It is easy to notice that the normalisation with \( ok_1(\{a_2\}) \) and \( ok_2(\{a_1\}) \) produces a symmetric \( abt \) covering, and similarly \( \ell_1 \cup \ell_2 \) will give a symmetric lock covering. Finally, there are several cases for \( P_3 \):

(a) \( P_3 = y(\bar{v}) \), s.t. \( y \) is a message to a local port, then \( R'_h \) contains

\[
[e y(\bar{v}) e^{\bar{a}_1, a_1, j_1, \ell_1} | x^1_1 \{ \bar{a}_1, x_1, \bar{u}_1, u_1, \bar{u}_2, u_2 \} \] Clearly, this reduction corresponds to

\[
N'_h = e \textbf{def} D''_h \text{ in } (\Pi_{j \in G''_h} x^G_{G''_h} \langle \bar{a}_{G''_h} \rangle | \Pi_{j \in \bar{M}''_h} x^M_{M''_h} \langle \bar{a}_{M''_h} \rangle | \Pi_{j \in \bar{L}''_h} x^L_{L''_h} \langle \bar{a}_{L''_h} \rangle) ; Q''_h] \sim_{c,J} [N''_h = e \textbf{def} D''_h \text{ in } (\Pi_{j \in G''_h} x^G_{G''_h} \langle \bar{a}_{G''_h} \rangle | \Pi_{j \in \bar{M}''_h} x^M_{M''_h} \langle \bar{a}_{M''_h} \rangle | \Pi_{j \in \bar{L}''_h} x^L_{L''_h} \langle \bar{a}_{L''_h} \rangle) | \mathcal{L} : Q''_h]
\]

where \( \mathcal{L} \) is obtained from \( \Pi_{j \in \bar{L}''_h} x^L_{L''_h} \langle \bar{a}_{L''_h} \rangle \) by removing \( x_1(\bar{u}_1) \) and \( x_2(\bar{u}_2) \), and by adding \( y(\bar{v}) \). All conditions hold for \( P'' \rightarrow_{c,J} P' \), where \( P' \) is obtained from \( P'' \) by changing \( N''_h \) by \( N'_h \).

(b) If \( P_3 \) consists of a message to a merge port, then the proof is analogous to the previous case.

(c) \( e P_3 e^{\bar{a}_1, a_1, j_1, \ell_1} | x^1_1 \{ \bar{a}_1, x_1, \bar{u}_1, u_1, \bar{u}_2, u_2 \} \) produces a commit vote, there are two cases: (i) if the vote is the last one, then, by normalising, \( R_i \) commits. It is easy to notice that this case corresponds to the case in which all local names have been consumed, then there exist \( P' \) s.t. \( P'' \rightarrow_{c,J} P' \) by using commit; (ii) if some coordinators still wait the vote, then it is enough to take \( P' = P'' \).

(d) \( P_3 = \text{abort} \), then encoding of \( e P_3 e^{\bar{a}_1, a_1, j_1, \ell_1} | x^1_1 \{ \bar{a}_1, x_1, \bar{u}_1, u_1, \bar{u}_2, u_2 \} \) produces the message \( a_1(\{a_1\}, a_1) \). The normalisation makes all coordinators in \( R'_h \) to abort and to release the compensations. It is easy to notice that this corresponds to \( P'' \rightarrow_{c,J} P' \) by producing first the abort in the negotiation \( h \) and then applying rule (ABORT), which releases all compensations.

(e) The encoding of \( P_3 \) produces \( 0 \), in such case it is enough to take \( P' = P'' \).

3. If the reduction is \( \Pi_{k \in 1..k} E T''_k \rightarrow_{J} R' \). Since all \( E T''_k \) are finished negotiations and that normalisation procedure is confluent, then \( \text{norm}(\Pi_{k \in 1..k} E T''_k) = \text{norm}(R) \). Therefore, it is enough to take \( P' = P'' \).

4. The applied rule is a definition in \( \llbracket D''_h \rrbracket_{\mathcal{L}} \):

(a) The rule is part of the encoding of an ordinary definition:
* messages are in $[M_1^\mu]_\nu$, then immediate by reducing $P''$ consuming messages in $M_1^\mu$.

* if a message is in some $T_k''$. This is possible only if some coordinator has finished and released the continuation or the compensation, by Theorem 8.28, the message is in $[M_2^\mu]_\nu$, hence it is possible to fire the corresponding rule in $P''$.

Note that messages cannot be part of some $R_k$ because they have not reached a decision, so global messages are kept by coordinators (b) the applied rule is part of the encoding of a merge definition. This case is similar to the reduction internal to a negotiation, which depends on the pattern of the encoded rule.

There are three different cases:

* The applied rule is $x_{1}^1(\vec{u}, p, a, j, \ell) \Rightarrow p(\ell, \emptyset)$, this is one rule fired by the normalisation process. Since the normalisation process is confluent it is enough to define $P' = P''$.

* The applied rule is $x_{2}^1(\vec{u}, p, a, j, \ell) \Rightarrow x_{2}^1(\vec{u}, p, a, j, \ell)$. It is enough to take $P' = P''$.

* The applied rule is a step of cJoin. As for ordinary definitions, the proof of this case depends on the shape of the definition. We analyse here the shape $[x_1^1(\vec{u}_1) \ldots | x_n^1(\vec{u}_n) \Rightarrow P]_\nu^2$, because it is the most interesting one. There are two main cases:

  - All $x_i$ are consumed from the $R_i$, i.e. from running negotiations. In this case all coordinators are able to be joined, then this case corresponds to the merge of the associated standard cJoin negotiations in $P''$.

  - Some message belongs to some $T_k''$. If the coordinator $D_k, j$ answers $f_j$ to the join request, then the join does not take place. Hence, $P' = P''$. If all coordinators answer $t_j$, note that the join take place. Nevertheless, the only possibility for a coordinator in $T_k''$ to answer yes to a join request is when its negotiation has been aborted, but the abort decision has not reached every coordinator. Clearly, such coordinators have not released the compensation, then the other messages that are going to be joined are part of a concurrent negotiation. At top level, this corresponds to a reduction that first merge the negotiations and then aborts. $P'$ is obtained from $P''$ by removing all negotiations corresponding to the consumed messages, and by activating the corresponding compensations.

\[\Box\]

**Theorem 8.32** (Completeness). Let $P \equiv \text{def } D \in M$ be a canonical flat process. If $[P]_\xi \rightarrow^*_J Q$, $P \rightarrow^*_c P'$ and $\forall x : \text{norm}(Q) \downarrow_x \Rightarrow P' \downarrow_x$.  

8.5 Extension of JoCaml

In this section we use the results presented in the previous section to extend a programming language with Join primitives. In order to straightforwardly use the encoding in Section 8.4.3, we select JoCaml, one of the available implementations of Join. JoCaml adds Join primitives to Objective Caml (Ocaml), which is a functional language with support of object oriented and imperative paradigms. JoCaml provides three main abstractions: process, channels and join-patterns. Processes represent communication and synchronisation tasks. The basic processes are asynchronous messages. Complex processes are obtained by combining expressions with the parallel composition of other processes. Channels are JoCaml abstractions corresponding to Join names. There are two different kind of channels: synchronous and asynchronous. The syntax for defining channels is the following

\[
\text{let def } \text{name}!!(\text{args}) = P(\text{args})
\]

This definition creates a channel (named name) and a receiver for it, which will execute the guarded process \(P\) every time it receives a message. The channel is asynchronous when its name is suffixed with the symbol !, otherwise it is synchronous. Synchronous names must return a value, i.e., \(P\) must explicitly define the return value. Finally, join-patterns are used to describe synchronisation among different channels. A join-pattern definition creates several channels at the same time and states a synchronisation between them: the corresponding guarded process may be executed only when messages on all channels are present. A sample syntax for a process defining a join pattern is below

\[
\begin{align*}
\text{let def } & \ a!(x) \mid b!(y) = P(x, y) \\
\text{or } & \ a!(x) \mid c!(z) = Q(x, z) \\
& ;;
\end{align*}
\]

The above process introduces three new asynchronous ports, namely \(a\), \(b\) and \(c\). Like join processes, the guarded process \(P\) (depending on variables \(x\) and \(y\)) can be activated if both \(a\) and \(b\) have pending messages. Similarly for \(Q(x, z)\), which can be activated when both \(a\) and \(c\) has pending messages. Moreover, the language does not specify which pattern is selected when both are enabled.

Processes can also create fresh ports dynamically. Consider the following program
let def new\_process() =
   let def a!() | b!() = P
   or a!() | c!() = Q
   in reply a, b, c

which defines a synchronous port \textit{new\_process} (i.e., an ordinary function). Any time this function is called, it creates a new process that defines three fresh ports \textit{a}, \textit{b} and \textit{c}. The caller is given back the names of the created ports (by clause \texttt{in reply} \textit{a, b, c}).

At the syntactic level, we extend JoCaml in the following way. A sample transactional process is defined as follow

\texttt{trans let def} b!() | c!() = P_1 
   or b!() | d!() | e!() = P_2 
   in \{b!()|b!()|c!()|d!()\}
\texttt{cmp Q}

\texttt{;;}

The above process is a negotiation that defines the local ports \textit{b}, \textit{c}, \textit{d} and \textit{e}, and initiates by sending the messages \textit{b}!()\textit{b}!()\textit{c}!()\textit{d}!(). Such definition is equivalent to the following \textbf{Join} process

\[ [\textbf{def} \ b\langle c\rangle \triangleright P_1 \land b\langle d\rangle \triangleright P_2 \textbf{ in } \ b\langle b\rangle\langle c\rangle\langle d\rangle : Q] \]

Such transaction will commit if the local computation terminates. Differently it will abort if some rule activates the process \textit{abort}. In such case the compensation \textit{Q} is activated.

Note that in our prototype implementation we do no require processes to be canonical, since we allow join patterns of any length (e.g., \textit{b}!() | \textit{d}!() | \textit{e}!()) and processes with several concurrent agents (e.g. \{\textit{b}!()|\textit{b}!()|\textit{c}!()|\textit{d}!()\}).

Merge patterns are defined by writing the keyword \texttt{board} in front of the corresponding join patterns, like as follows

\texttt{let def board m!(x) | n!(y) = P}

The above definition introduces two merge ports \textit{m} and \textit{n}, with the guarded process \textit{P}, which is required to be an ordinary join process (i.e., without transaction primitives), in order to assure flatness.

\textbf{About the implementation} Firstly, we remark that transactions are not necessarily canonical processes. Nevertheless, we do not transform programs into their
canonical form. In our implementation, we extend the encoding presented in Section 8.4.3 by allowing several threads to be merged or activated simultaneously. This is achieved by: (i) handling the joint of several coordinators as for the case of merge names, and (ii) by creating several coordinators (as many as the number of agents that are in the guarded process) when a rule is fired.

As far as compiler implementation is concerned, we do not translate syntactically JoCaml programs with transactions into ordinary JoCaml code. Differently, since JoCaml is distributed with a license that allows for the freely modification of the software, we changes the original initial phases of the compiler. In particular, the lexer and the parser are slightly changed in order to recognise processes built with the primitives trans_comp, abort and board (as described above). After the parse tree has been generated, and before the typing phase takes place, we translate transactional primitives into JoCaml code. This is done, by directly modifying the parse-tree in order to add all the coordination code described in Section 8.4.3.

The main limitation of our prototype version is that it cannot handle the compilation of separate units, since the translation of processes depends on whether names are ordinary or merge. In particular the environment is initially empty when compiling a new file. As names are being defined, they are added to the environment. Every name not in the environment is assumed to be an ordinary stable name. We believe that this drawback could be overcome by relying on a typing discipline on received names.

8.6 Related works

Protocols for consensus, agreements and commitment in distributed systems have been largely study in the area of Distributed Algorithms [101, 10, 49, 83]. Differently, there have been a few efforts to formalise agreement protocols in process calculi. Notably, the proposal in [9] provides a π-calculus formalisation of the well-known two phase commit protocol, which is aimed at capturing several aspects of real distributed systems, such as message loss, process failures, and time. These aspects are modelled by enriching the asynchronous π-calculus with (i) localities and unreliable communication between sites, (ii) process persistence that provides a recovery mechanism from failures, and (iii) timers. Then, usual process calculi techniques (i.e., bisimulations) are used to prove protocol correctness.

Differently from the d2PC, the 2PC is a centralised protocol. In fact, a manager coordinates the whole execution of the agreement. Clearly, the standard impossibility results for atomic commitment in the presence of failure [50] prevents one to obtain a fully distributed implementation, e.g., in asynchronous π, of agreements when processes may arbitrarily crash. The area of Distributed Algorithms have developed several refinement of the commit problem by assuming particular models of failures, or by extending the computational model. A few works in the arena of process calculi follow also these lines. In particular, [108] proposes a simplified version
of the three phase commit protocol to implement rendezvous mechanisms in systems with asynchronous communications that fails stochastically. A particular case of an agreement protocol, i.e. the distributed implementation of the mixed choice of the $\pi$ calculus, has been shown implementable in the probabilistic $\pi$ [65]. Although several proposals for probabilistic process calculi can be found in literature [92, 64, 18], to the best of our knowledge no randomised commit protocols has been studied through the use of process calculi techniques.

Finally, [54] introduces a process calculus that accounts for unreliable failure detectors [39, 38]. In this model, each processes has an associated failure detector that is able to answer whether another process is locally suspected to have crashed. Failures detectors may be wrong, i.e., they may suspect that one correct process has crashed and vice versa; and the suspicions of two failure detectors may differ. Then, the proposed calculus is used to formalise and prove that the original protocol proposed in [39] terminates, and eventually reach an agreement on a valid value.

We have no consider failures site failures (like crashes) in our presentation because our goal was to show that the commit primitives of $\text{cJoin}$ can be implemented by using a commit protocol. Clearly, a suitable implementation of $\text{cJoin}$ in a computational model in which, unlike join, process or communications links may fail should consider alternative commits protocols (possibly in an enriched $\text{Join}$ calculus along the lines in [9, 54, 65]).
Part III

Compensations and Flow Composition
Roadmap to PART III

A key aspect when aggregating business processes and web services is to assure transactional properties of process executions. Transactional features in web service composition are reminiscent of transactional workflow systems, where the execution of several independent components should be coordinated in order to assure all of them to complete. Since transactions in this context are long running, they rely on compensations. Consequently, several recent language proposals for service composition usually include compensation primitives. Nevertheless, the semantics of such primitives has not been precisely defined. In particular, the relation among compensations and usual programming language primitives is far from being clear. In this part of this thesis we address the problem of giving semantics to compensations in flow composition languages.

Content of PART III. This part of the dissertation consists of Chapter 9, which presents a hierarchy of transactional calculi with increasing expressiveness. We start from a very small language in which activities can only be composed sequentially (Section 9.2). Then, we progressively introduce parallel composition (Section 9.3), nesting (Section 9.4), programmable compensations and exception handling (Section 9.5). A running example illustrates the main features of each calculus in the hierarchy.
Chapter 9

Compensations and Flow Composition

9.1 Motivations

The ultimate goal of web services technologies is to allow the distribution, delivery and interoperability of heterogeneous components over the Internet. Applications achieve interoperability by adhering to standard protocols that provide uniform ways to describe services (namely WSDL), to look for particular services (i.e., UDDI), and to access services (i.e., SOAP). In this way standards facilitate the interaction of different services, not only within an organization but also across organization boundaries. Nevertheless, these standards do not provide yet any support to describe complex interactions between several applications. Recently, many proposals have addressed the problem of aggregating services, giving birth to a family of XML-based composition languages (also known as choreography or orchestration languages), such as BPML [17], XLANG [104], WSFL [81], BPEL4WS [16] and WSCI [109].

Most proposals for orchestration languages contain a large amount of primitives for composing services. Since the official specifications of composition languages for web services mainly consist in an informal textual description of their constructors, many recent efforts have attempted to formalize different subsets of such proposals (see for instance [19, 7, 105]). In this chapter, we study primitives for long running transactions in flow composition languages, and in particular in structured control flows, i.e. flows defined in terms of a fixed set of primitives, like sequencing and branching. We provide a formal semantics for a hierarchy of transactional languages with increasing expressiveness and we prove that the semantics is adequate to the modelled features.

Transactional aspects in composed web services have been mainly inherited from workflow languages. The key idea is that valid executions of a transactional business process (or of a part of it) are those that “complete” all involved activities. Nevertheless, since the execution of a business process may require a very long period of
time in order to complete (perhaps some hours or days), traditional mechanisms for assuring atomicity, such as locking of resources, are regarded as not suitable. Since the seminal work of Sagas [55], the key mechanism for dealing with long running transactions is that of “compensating activities”. Instead of relying on locking and roll-back mechanisms to perfectly undo incomplete executions and avoid interference among transactions, a more relaxed form of atomicity is granted by associating processes with activities that can recover partial executions.

Although compensations can be regarded as an exception handling mechanism [85], the distinctive feature is that compensation handlers are dynamically built during the execution of processes. Consider the saga given in Figure 9.1, where the transactional process $P$ consists in the sequential execution of the activities $A_1$, $A_2$ and $A_3$, that can be compensated respectively by $B_1$, $B_2$ and $B_3$. Suppose now that activity $A_1$ completes successfully while activity $A_2$ fails. In this case, after $A_2$ fails, the compensation $B_1$ (corresponding to the successfully completed activities) is run to undo as much as possible the effects of $A_1$, because the transaction failed as a whole. Note that $B_2$ is not executed, because $A_2$ has not completed. Instead, if both $A_1$ and $A_2$ succeed while $A_3$ fails, then the compensations will be executed in the reverse order, i.e. first $B_2$ and then $B_1$.

After Sagas, several workflow models have been proposed in literature for equipping processes with different (compensation-based) transactional capabilities, such as nesting and forward recovery (for a general overview see [100]). Contrastingly, the study of PDTs with compensations have been less numerous. An extension of the asynchronous $\pi$-calculus, called $\pi$t-calculus, with transactional contexts has been introduced in [14]. The $\pi$t-calculus formalizes the close relation between exception handling and compensations. Nevertheless, this approach is not aimed at capturing the order in which compensations should be activated, i.e. there is not a strong relation between compensations and the control flow of the original processes. For instance, if activity $A_3$ fails during the execution of flow depicted in Figure 9.1, compensations $B_1$ and $B_2$ are activated concurrently.

A different approach is taken by StAC [35], where compensations are installed to be executed in the reverse order w.r.t. that of completion of original activities. StAC is a language in the spirit of process algebras like CSP or CCS with exception handling mechanisms and compensations inspired by BPBeans, a framework for modelling business processes integrated to WebSphere [106]. Although being (to the best of our knowledge) the first process calculus where compensations are closely
related to the control flow of the executed process\footnote{We are aware of previous formal approaches to define compensations, such as ACTA \cite{acta} in the context of database transactions and the work done by C.A.R Hoare \cite{hoare}, but they are not process calculi}, there are several aspects in \texttt{StAC} that deserve further investigation. For instance, compensations in \texttt{StAC} should be explicitly activated through special primitives, i.e. they are not related to the failure or success of the activities of processes, as usually expected in workflows and composition languages, e.g. BPEL4WS. Moreover, to reason about \texttt{StAC} processes, it is necessary to know the low level description of activities. In fact, there is an interplay between data structures used by activities and the control flow of processes. Finally, \texttt{StAC} provides a large number of operators including the imperative fragment of a programming language, whose operational semantics has been given in terms of an even richer intermediate language, called \texttt{StAC$_i$} \cite{staci}. In this way, operators in \texttt{StAC} can only be understood by analyzing their encodings into \texttt{StAC$_i$} operators. Due to the complex definition of the operational semantics of \texttt{StAC$_i$}, it is difficult to reason about the interplay among exception handling, compensations, nesting and parallel composition in \texttt{StAC}. Moreover, some usual behaviors of compensations (for instance, the failure of a branch in a parallel composition requiring the compensation of both branches) are only achieved by combining several operators, making the semantics in \cite{staci} not entirely satisfactory. (Recent ongoing work by Butler, Ferreira and Hoare aims to define a clean trace semantics for a subset of \texttt{StAC}.)

In this chapter we intend to give a more compact description of \texttt{StAC}-like languages: in the spirit of PDLs, we are aimed at providing a minimal set of operators with orthogonal meaning and, in particular, we are interested on marking the distinction between compensations and exception handling mechanisms. Moreover we attempt to provide our operators with the meaning most frequently used in composition languages. Additionally, we relate the behavior of whole processes with the success or failure of atomic activities.

In order to achieve these goals, we start from a very small language formalizing \texttt{Sagas}. First, we show a language corresponding to its sequential version (i.e., allowing only the sequential composition of activities inside a saga). Then we introduce the parallel composition and discuss different alternatives in defining the semantics for the compensation of parallel activities. After that, we add the possibility of defining nested transactions. Finally, we present some extensions, such as programmable compensations, exception handling and forward recovery. Each language in the resulting hierarchy comes with a clean big-step semantics and an adequacy result for such semantics.

As a running example, we select a business process for ordering goods. It is simple enough to require a process that fits in one line, yet it is expressive enough to show how the primitives can enhance business process design.
9.2 Sequential Sagas

As mentioned before, Sagas \[55\] is one of the first proposals for dealing with long running transactions in database applications. A sequential saga (i.e., a long lived transaction) is a sequence of atomic activities (called subtransactions, activities or steps) that should be executed completely. The parallel execution of several sagas can interleave steps in any way, but any single step is guaranteed to be atomic. Subtransactions are atomic in the sense that either they are successfully executed (committed) or no effect is observed when the execution fails (aborted). In addition, no intermediate states computed by an activity are visible to other activities. Activities are transactions with short duration, and therefore they can rely on traditional mechanisms to assure the usual \textit{ACID} properties (i.e., \textit{Atomicity, Consistency, Isolation} and \textit{Durability}). Additionally, any activity \(A_i\) in a saga has a compensating activity \(B_i\) that can be activated to “undo” the effects of a successful execution of \(A_i\) upon a later failure. (We remind that, in this context, the term “undo” does not mean to exactly reverse the effects by restoring the original state, but just to perform an ad hoc activity that moves the system to a sound state).

Any partial execution of a saga is undesirable, and if it occurs, it must be compensated for. A saga involving \(A_1, \ldots, A_n\) (where each \(A_i\) has a compensation \(B_i\)) is guaranteed to execute either the entire series \(A_1; \ldots; A_n\) or the compensated sequence \(A_1; \ldots; A_j; B_j; \ldots; B_1\) for some \(j < n\). The first case stands for the successful execution of the whole saga, i.e., when all activities in the sequence complete. In the second case, the activity \(A_{j+1}\) fails, and all activities already completed \((A_1; \ldots; A_j)\) are recovered by executing the corresponding compensations, in reverse order \((B_j; \ldots; B_1)\).

In this section we introduce a compensation language for sequential sagas. The semantics is intended to describe the behavior of top-level processes but not the low-level computations performed by atomic activities. The only assumption made on subtransactions is that their executions end either successfully or with a failure.

We rely on an infinite set \(\mathcal{A}\) of names for atomic activities, ranged over by \(A, B, \ldots\). Moreover, we will consider a special nil activity \(0 \notin \mathcal{A}\) that always completes and has no effect.

**Definition 9.1** (Sequential Sagas). The set of all sequential sagas is given by the following grammar:

\[
\begin{align*}
\text{(STEP)} & \quad X & := & \ 0 \mid A \mid A \div B \\
\text{(PROCESS)} & \quad P & := & X \mid P; P \\
\text{(SAGA)} & \quad S & := & \{[P]\}
\end{align*}
\]

A sequential saga \(S\) consists in a sequential process \(P\). Each step in \(P\) corresponds either to an activity \(A\) or a compensated activity \(A \div B\), where \(A\) is the activity of the normal flow and \(B\) its compensation. The term \(0\) represents the inert process, and \(P; P\) stands for the sequential composition of processes.

\[
\Gamma \vdash \langle 0, \beta \rangle \xrightarrow{0} \langle \square, \beta \rangle
\]
We define the semantics of sagas up-to structural congruence over processes and steps given by the following axioms:

\[
A \stackrel{0}{\to} A \quad \text{(NULL COMPENSATION)} \\
0; P \equiv P; 0 \equiv P \quad \text{(NULL PROCESS)} \\
(P; Q); R \equiv P; (Q; R) \quad \text{(ASSOC. OF SEQ. COMP.)}
\]

For simplicity we will consider all (instances of) activities in a saga named differently. This does not mean we do not allow the same activity to be executed more than once in a saga, but we consider any execution as a different instance of it and, hence distinguishable from all other instances.

**Definition 9.2** (Activities of a saga). The set of activities of a saga \( S \) is defined as \( \mathcal{A}(S) = \{ A \mid A \text{ occurs in } S \} \).

### 9.2.1 Big Step Semantics

As described above, the execution of a saga \( S \) either commits — i.e., every activity executes successfully — or it aborts and all completed steps are compensated for. This model implicitly assumes that compensations always succeed. In order to relax this assumption, we allow also compensations to fail. In this case, a saga \( S \) has an abnormal termination. Abnormal termination could be managed by suitable exception handling mechanisms. (We informally discuss exception handling in Section 9.5.2). Thus, the set of possible results for the execution of a saga is \( \mathcal{R} = \{ \Box, \Box, \Box \} \), where \( \Box \) stands for commit, \( \Box \) for (compensated) abort, and \( \Box \) for abnormal termination. We let \( \Box \) range over \( \mathcal{R} \).

The execution of a sequential saga is described in terms of the results obtained by performing their constituent activities. As we are not interested on the low-level behavior of individual tasks, we rely on the abstract description of their executions, stating whether they complete successfully or abort. This information is given by a context \( \Gamma \). Formally, \( \Gamma \) is a partial function over \( \mathcal{A} \) that maps any activity to the result obtained with its execution, i.e., \( \Gamma : \mathcal{A} \to \{ \Box, \Box \} \). Note that activities can only commit or abort (they do not terminate abnormally). We denote a particular function \( \Gamma \) as \( A_1 \mapsto \Box_1, \ldots, A_n \mapsto \Box_n \), where \( A_i \neq A_j \) for all \( i \neq j \) (i.e., \( ', ' \) stands for the disjoint union of partial functions).

The semantics of a sequential saga \( S \) is given by the relation \( \Gamma \vdash S \stackrel{\alpha}{\to} \Box \) defined by the inference rules in Figure 9.2. The notation \( \Gamma \vdash S \stackrel{\alpha}{\to} \Box \) denotes that the execution of \( S \) produces \( \Box \) when the atomic activities behave like \( \Gamma \). The observation \( \alpha \) describes the actual flow of control occurring when executing \( S \) under the context \( \Gamma \). The flow \( \alpha \) is a process whose activities have no compensations.

The auxiliary relation \( \Gamma \vdash \langle P, \beta \rangle \stackrel{\alpha}{\to} \langle \Box, \beta' \rangle \) describes the behavior of a process \( P \) within a saga that already installed the compensation \( \beta \) (\( \beta \) stands for a process without compensations). \( \Gamma \) and \( \alpha \) are analogous to the previous case. When \( P \) is executed inside a saga, it can either commit, abort, or fail, but additionally, it
can change the compensations, for instance by installing new activities, like in rule (S-ACT).

Rule (ZERO) states that 0 always commits without changing the installed compensation. Rule (S-ACT) stands for the successful execution of the compensated activity $A \div B$ when $A$ commits. In this case, the observation is $A$ (i.e., the only executed activity), the obtained result is $\Box$, while the compensation is updated by installing $B$ in front of $\beta$. Note that $A$ is the last executed activity, hence the first to be compensated for if the next activity in the saga fails.

Rules (S-CMP) and (F-CMP) describe the execution of $A \div B$ when $A$ fails in a saga that has already installed $\beta$. Both rules activate the compensation procedure by executing $\beta$ (premises of the rules). Note that neither $A$ nor $B$ are really executed. In fact, since $A$ is an atomic activity that aborts, $A$ has no effects and hence, it is not compensated for. For this reason the observation $\alpha$ is just the flow observed by executing $\beta$. In particular, (S-CMP) describes the case in which the compensation procedure completes successfully. Rule (F-CMP) stands for the case in which the compensation procedure fails. In this case, the process finishes abnormally (the corresponding result is $\square$). Since all steps in $\beta$ have trivial nil compensations, the execution of $\beta$ cannot produce $\Box$. For the same reason, the execution of $\beta$ installs no significant compensations, and hence any execution that ends with $\Box$ or $\square$ must have 0 as compensation.

Rule (S-STEP) describes the behavior of a process $P; Q$ when the step $P$ commits. In such case the remaining process $Q$ is executed by taking into account the compensation produced after the execution of $P$. The observation for the whole process $P; Q$ corresponds to the sequential composition of $\alpha$, i.e. the observation of executing $P$, and $\alpha'$, i.e. the flow corresponding to the execution of $Q$. The final result is that obtained when executing $Q$.

Rule (A-STEP) handles the case in which $P; Q$ is stopped because $P$ ends with
abort or abnormal termination. Note that the compensation is activated when \( P \) reaches the abort.

Last rule (SAGA) states that the execution of a saga \( \{P\} \) is the activation of \( P \) with no installed compensations.

Although a more concise set of rules could be used to describe the semantics, we choose this presentation for convenience when extending the language in the next sections.

**Example 9.1 (Sequential Sagas).** Figure 9.3 shows a sequential saga for dealing with purchase orders. It consists on three activities composed sequentially. The first activity (Accept order) handles a request from a client and it is compensated by Refuse order, which will contact the client to notify her/him that the order was canceled. The second step (Update Credit) charge the amount of the order to the balance of the client. This activity could fail, for instance when the client has not enough credit to proceed, activating the compensation, i.e., executing Refuse order. Instead, if it succeeds, then the compensation Refund order is also installed. Refund order is responsible for updating the balance with the amount detracted previously. Last activity (Prepare Order) handles the packaging of the order and update the stock. Its compensation (Update Stock) will increment the stock with the proper values.

The following result states that the execution of a saga corresponds to the intuitive notion we gave initially.

**Theorem 9.1 (Adequacy).** Let \( S \equiv \{A_1 \div B_1; \ldots; A_n \div B_n\} \) be a saga. Then:

- **(Completion)** \( \Gamma \vdash S \xrightarrow{\alpha} \Box \quad \text{iff} \quad \forall i \leq n : A_i \rightarrow \Box \in \Gamma \text{ and } \alpha \equiv A_1; \ldots; A_n; \)

- **(Successful Compensation)** \( \Gamma \vdash S \xrightarrow{\alpha} \Box \quad \text{iff} \quad \exists k, 1 \leq k \leq n \wedge A_k \rightarrow \Box \in \Gamma \wedge \forall i < k : (A_i \rightarrow \Box, B_i \rightarrow \Box \subseteq \Gamma) \text{ and } \alpha \equiv A_1; \ldots; A_{k-1}; B_{k-1}; \ldots; B_1. \)

- **(Failed Compensation)** \( \Gamma \vdash S \xrightarrow{\alpha} \Box \quad \text{iff} \quad \exists j, k, 1 \leq k \leq n \wedge 1 \leq j < k \text{ s.t. } A_k \rightarrow \Box \in \Gamma \wedge B_j \rightarrow \Box \in \Gamma \wedge (\forall i < k : A_i \rightarrow \Box \in \Gamma) \wedge (\forall h \in [j + 1, k - 1] : B_h \rightarrow \Box \in \Gamma) \wedge \alpha \equiv A_1; \ldots; A_{k-1}; B_{k-1}; \ldots; B_{j+1}. \)

It is clear from the above theorem that the last compensation \( B_n \) is never activated. Nevertheless we allow such kind of definitions because they can be useful when specifying more complex sagas in the following sections.
9.2.2 Proof of Adequacy Theorem

In order to proof Theorem 9.1, we first state some auxiliary results. In particular, the following lemma handles successful execution of a sequential process.

**Lemma 9.2.** Let \( P \equiv A_1 \div B_1 ; \ldots ; A_n \div B_n \). Then \( \Gamma \vdash \langle P, \beta \rangle \xrightarrow{A_1;\ldots ;A_n} \langle \square, B_n; \ldots ; B_1; \beta \rangle \) iff \( \forall 1 \leq i \leq n : A_i \rightarrow \Box \in \Gamma \).

**Proof.**

\( \Rightarrow \) It follows by induction on the structure of the proof.

- **Case (zero).** Immediate.
- **Case (s-act).** Immediate.
- **Case (s-step).** The proof have the following shape.

\[
\begin{align*}
A_1 \rightarrow \Box \in \Gamma \\
\Gamma \vdash \langle A_1 \div B_1; \beta \rangle & \xrightarrow{A_1} \langle \Box, B_1; \beta \rangle \\
& \xrightarrow{\alpha'} \langle \Box, \beta' \rangle \\
& \xrightarrow{(s-step)} \langle \Box, \beta' \rangle
\end{align*}
\]

By inductive hypothesis on \( \Gamma \vdash \langle A_2 \div B_2; \ldots ; A_n \div B_n, B_1; \beta \rangle \xrightarrow{\alpha'} \langle \Box, \beta' \rangle \), the following conditions hold \( \alpha' \equiv A_2; \ldots ; A_n \) and \( \beta' \equiv B_n; \ldots ; B_2; B_1; \beta \).

\( \Leftarrow \) The proof follows by induction on the length of the sequence.

- **Base Case** \((n = 0)\). Immediate by rule (zero).
- **Inductive Step** \((n = k)\). The only proof for \( \Gamma \vdash \langle A_1 \div B_1; \ldots ; A_k \div B_{k+1}, \beta \rangle \xrightarrow{\alpha} \langle \Box, \beta' \rangle \) is the following.

\[
\begin{align*}
A_1 \rightarrow \Box \in \Gamma \\
\Gamma \vdash \langle A_1 \div B_1; \beta \rangle & \xrightarrow{A_1} \langle \Box, B_1; \beta \rangle \\
\Gamma \vdash \langle A_2 \div B_2; \ldots ; A_k \div B_{k+1}, \beta \rangle & \xrightarrow{\alpha'} \langle \Box, \beta' \rangle \\
& \xrightarrow{(s-step)} \langle \Box, \beta' \rangle
\end{align*}
\]

By inductive hypothesis we have that \( \Gamma \vdash \langle A_2 \div B_2; \ldots ; A_k \div B_{k+1}, B_1; \beta \rangle \xrightarrow{\alpha'} \langle \Box, \beta' \rangle \) for \( \alpha' \equiv A_2; \ldots ; A_k, \beta' \equiv B_k; \ldots ; B_2; B_1; \beta \), and \( \Box \equiv \Box \).

\( \square \)

Next lemma handles the case in which the saga aborts, but all compensations execute successfully.

**Lemma 9.3.** Let \( P \equiv A_1 \div B_1 ; \ldots ; A_n \div B_n \), and \( \beta \equiv C_1 ; \ldots ; C_m \). Then \( \Gamma \vdash \langle P, \beta \rangle \xrightarrow{\alpha} \langle \Box, 0 \rangle \) where \( \alpha \equiv A_1; \ldots ; A_{k-1}; B_{k-1}; \ldots ; B_1; \beta \) iff there exists \( 1 \leq k \leq n \) such that the following conditions hold:
\begin{itemize}
  
  \item $A_k \mapsto \square \in \Gamma \land \forall i < k : (A_i \mapsto \square \in \Gamma \land B_i \mapsto \square \in \Gamma)$.
  
  \item $\forall l \in 1..m : C_l \mapsto \square \in \Gamma$.
  
\end{itemize}

**Proof.**

$\Rightarrow$) It follows by induction on the structure of the proof.

- **Case (s-cmp).** Then $n = k = 1$ and $A_1 \mapsto \square$. By Lemma 9.2 it holds that $\Gamma \vdash \langle C_1 \div 0; \ldots; C_m \div 0, 0 \rangle \xrightarrow{\alpha} \langle \Box, 0 \rangle$ if $\forall l \in 1..m : C_l \mapsto \square \in \Gamma$ and $\alpha \equiv C_1; \ldots; C_m$.

- **Case (s-step).** Then the proof has the following shape:

$$
\frac{A_1 \mapsto \square \in \Gamma}{\Gamma \vdash \langle A_1 \div B_1, \beta \rangle \xrightarrow{A_1} \langle \Box, B_1 ; \beta \rangle} \quad (s-ACT) \quad \frac{\Gamma \vdash \langle A_2 \div B_2; \ldots; A_n \div B_n, B_1 ; \beta \rangle \xrightarrow{A_1} \langle \Box, 0 \rangle}{\Gamma \vdash \langle A_1 \div B_1; \ldots; A_n \div B_n, B_1 ; \beta \rangle \xrightarrow{A_1} \langle \Box, 0 \rangle} \quad (s-step)
$$

The proof is completed by using inductive hypothesis on $\Gamma \vdash \langle A_2 \div B_2; \ldots; A_n \div B_n, B_1 ; \beta \rangle \xrightarrow{A_1} \langle \Box, 0 \rangle$.

- **Case (a-step).** Then

$$
\frac{\Gamma \vdash \langle A_1 \div B_1, \beta \rangle \xrightarrow{\alpha} \langle \Box, 0 \rangle}{\Gamma \vdash \langle A_1 \div B_1; \ldots; A_n \div B_n, \beta \rangle \xrightarrow{\alpha} \langle \Box, 0 \rangle} \quad (a-step)
$$

The proof follows by using inductive hypothesis on $\Gamma \vdash \langle A_1 \div B_1, \beta \rangle \xrightarrow{\alpha} \langle \Box, 0 \rangle$.

$\Leftarrow$) The proof follows by induction on $n$.

- **Base Case** ($n = 1$). Then $k = 1$. The proof follows by using rule (s-cmp) and Lemma 9.2 for the premise.

- **Inductive Step.** If $k = 1$, the proof is completed by using rule (a-step) and inductive hypothesis on the premise. Otherwise, the proof is built by using rule (s-step), Lemma 9.2 on the first premise and inductive hypothesis on the second one.

$\square$

Last lemma stands for the case in which the normal flow and the compensation procedure abort.

**Lemma 9.4.** Let $P \equiv A_1 \div B_1; \ldots; A_n \div B_n, \beta \equiv C_1; \ldots; C_m$, and $D_m; \ldots; D_1 \equiv B_n; \ldots; B_1; \beta$. Then $\Gamma \vdash \langle P, \beta \rangle \xrightarrow{\alpha} \langle \Box, 0 \rangle$ where $\alpha \equiv A_1; \ldots; A_k; D_l; \ldots; D_j$ with $D_l \equiv B_{k-1}$ iff there exist $1 \leq k \leq n$ and $1 \leq j \leq l$ s.t. the following conditions hold:

- $A_k \mapsto \Box \in \Gamma \land \forall i < k : A_i \mapsto \Box \in \Gamma$
- $D_j \mapsto \Box \in \Gamma \land \forall h \in j + 1..l : D_h \mapsto \Box \in \Gamma$. 


Proof.

\[ \Rightarrow \] It follows by induction on the derivation.

- **Case (f-CMP).** Then \( n = k = 1 \) and \( A_1 \vdash \square \). By Lemma 9.3 it holds that 
  \[ \Delta \vdash \langle C_1 \div 0; \ldots; C_m \div 0, 0 \rangle \overset{\alpha}{\rightarrow} \langle \square, 0 \rangle \]  
  for \( \alpha \equiv C_1; \ldots; C_{k-1} \) if there exists \( 1 \leq j \leq m \) 
  s.t. \( C_k \vdash \square \in \Delta \land \forall i : (C_i \vdash \square \in \Delta) \).

- **Case (s-STEP).** Then the proof has the following
  \[
  \frac{
  A_1 \vdash \square \in \Delta 
  }{
  \Delta \vdash \langle A_1 \div B_1, \beta \rangle \overset{A_1}{\rightarrow} \langle \square, B_1, \beta \rangle
  } \quad \text{(s-ACT)}
  \]

  Then the proof is completed by using inductive hypothesis on \( \Delta \vdash \langle A_2 \div B_2; \ldots; A_n \div B_n, B_1; \beta \rangle \overset{A_1}{\rightarrow} \langle \square, 0 \rangle \).

- **Case (a-STEP).** Then 
  \[
  \frac{
  \Delta \vdash \langle A_1 \div B_1, \beta \rangle \overset{a}{\rightarrow} \langle \square, 0 \rangle
  }{
  \Delta \vdash \langle A_1 \div B_1; \ldots; A_n \div B_n, \beta \rangle \overset{A_1}{\rightarrow} \langle \square, 0 \rangle \quad \text{(a-STEP)}
  }
  \]

  The proof follows by using inductive hypothesis on \( \Delta \vdash \langle A_1 \div B_1, \beta \rangle \overset{a}{\rightarrow} \langle \square, 0 \rangle \).

\[ \Leftarrow \] The proof follows by induction on \( n \).

- **Base Case (n = 1).** Then \( k = 1 \). The proof follows by using rule (f-CMP) and 
  Lemma 9.3 for the premise.

- **Inductive Step.** If \( k = 1 \), the proof is completed by using rule (a-STEP) and 
  inductive hypothesis on the premise. Otherwise, the proof is built by using rule 
  (s-STEP), Lemma 9.2 on the first premise and inductive hypothesis on the second one.

\[ \square \]

**Proof of Theorem 9.1.** The proof is immediate by Lemmata 9.2, 9.3 and 9.4.

\[ \square \]

### 9.3 Parallel Sagas

In order to allow several activities to be executed concurrently, the language of 
sequential sagas is extended with the operator \( \mid \), denoting the parallel composition 
of processes.

**Definition 9.3 (Parallel Sagas).** The set of all parallel sagas is defined by the grammar:

...
(STEP) \[ X ::= 0 \mid A \mid A \div B \]

(PROCESS) \[ P ::= X \mid P ; P \mid P | P \]

(SAGA) \[ S ::= \{P\} \]

In addition to the structural axioms for sequential sagas, we require \(\mid\) to be associative and commutative with unit 0. We let sequential composition have higher priority than parallel, i.e. \(P; Q|R; S\) stands for \((P; Q)|(R; S)\).

Note that the dependencies among activities are described by a structured flow, and in particular synchronizations between processes take place only when composing sequentially. We will not consider descriptions based on links dependencies like those allowed in WSFL until Section 9.5.5.

As for sequential sagas, a computation of a parallel saga is successful only when all its activities commit, while the whole saga should be compensated for when an activity fails. Also we like compensations to be performed in the reverse order of the normal flow. Composition languages usually express this requirement by stating that all compensation handlers for completed activities run in the reverse order of completion. In our approach the compensations of concurrent activities are concurrent, because we want a semantics where compensations do not depend on the particular interleaving of executed concurrent activities.

We first give a semantics where parallel branches are completely independent (Section 9.3.1). This semantics is simple but not entirely satisfactory when modelling real problems, because it does not allow to force the failure in one branch as soon as a failure is detected in the other branch. A more complex semantics is then given in Section 9.3.2, which can deal properly with this kind of optimization.

### 9.3.1 Naïve definition for the semantics of \(\mid\)

In a first attempt at defining the semantics of the parallel composition we add the rules in Figure 9.4 to those of sequential sagas. Note that transition labels \(\alpha\) can now take the form \(\gamma|\gamma'\), where \(\gamma\) and \(\gamma'\) are uncompensated processes. In this way we quotient out all possible interleaving executions of \(\gamma|\gamma'\). We recall that activities are atomic steps, and therefore there is no interaction among them. For this reason any interleaving of \(\gamma|\gamma'\) is a valid execution of the process. Hence, the parallel branches of \(P|Q\) are executed independently, i.e. each branch performs until completion in its own thread, which has no initial compensation.

The first rule (S-PAR) handles the successful execution of \(P|Q\), i.e. every activity commits and both \(P\) and \(Q\) produce \(\equiv\) as result. The compensation for the whole process is updated by installing the parallel composition of \(\beta'\) and \(\beta''\) at the top, i.e. the compensation of parallel processes corresponds to the parallel compensation of its branches. The observed flow is the parallel composition of the flows for \(P\) and \(Q\).

The remaining four rules handle the cases where at least one activity aborts. If both branches fail during the normal flow but their compensation completes, then
(S-PAR)

\[ \Gamma \vdash \langle P, 0 \rangle \xrightarrow{\alpha} \langle \Box, \beta' \rangle \quad \Gamma \vdash \langle Q, 0 \rangle \xrightarrow{\alpha'} \langle \Box, \beta'' \rangle \]

\[ \Gamma \vdash \langle P | Q, \beta \rangle \xrightarrow{\alpha[\alpha']} \langle \Box, \beta'; \beta'' \rangle \]

(F-PAR-NAÏVE-1)

\[ \square_2 = \begin{cases} \Box & \text{if } \Box_1 = \Box \\ \text{otherwise} & \end{cases} \]

(F-PAR-NAÏVE-2)

\[ \Gamma \vdash \langle P, 0 \rangle \xrightarrow{\alpha} \langle \Box, 0 \rangle \quad \Gamma \vdash \langle Q, 0 \rangle \xrightarrow{\alpha'} \langle \Box, \beta' \rangle \quad \Gamma \vdash \langle \beta', 0 \rangle \xrightarrow{\alpha''} \langle \Box_1, 0 \rangle \]

\[ \Gamma \vdash \langle P | Q, \beta \rangle \xrightarrow{(\alpha[\alpha'])\alpha''} \langle \Box, 0 \rangle \]

(F-PAR-NAÏVE-3)

\[ \Gamma \vdash \langle P, 0 \rangle \xrightarrow{\alpha} \langle \Box, 0 \rangle \quad \Gamma \vdash \langle Q, 0 \rangle \xrightarrow{\alpha'} \langle \Box, \sigma_0 \rangle \quad \text{with } \sigma \in \{\Box, \Box\} \]

(F-PAR-NAÏVE-4)

\[ \Gamma \vdash \langle P, 0 \rangle \xrightarrow{\alpha} \langle \Box, 0 \rangle \quad \Gamma \vdash \langle Q, 0 \rangle \xrightarrow{\alpha'} \langle \Box, \beta' \rangle \quad \Gamma \vdash \langle \beta'; \beta, 0 \rangle \xrightarrow{\alpha''} \langle \Box_1, 0 \rangle \]

\[ \Gamma \vdash \langle P | Q, \beta \rangle \xrightarrow{(\alpha[\alpha'])\alpha''} \langle \Box_2, 0 \rangle \]

\[ \square_2 = \begin{cases} \Box & \text{if } \Box_1 = \Box \\ \text{otherwise} & \end{cases} \]

Figure 9.4: Naïve semantics of parallel composition.
the execution ends by activating the original compensation $\beta$ (rule F-PAR-NAÏVE-1). The result is $\lhd$ when $\beta$ finishes without problems. On the contrary, if $\beta$ aborts, then the whole process terminates abnormally (producing $\lhd$).

Rules (F-PAR-NAÏVE-2) and (F-PAR-NAÏVE-3) stand for the cases in which $P$ terminates abnormally, i.e. some activity in the normal flow of $P$ aborts activating its compensation procedure, which also fails. In such cases, the original compensation $\beta$ is not executed, because it should follow the compensation of $P$ that has failed. The behavior is similar to the sequential case, where the compensation procedure stops when some activity aborts. In particular, rule (F-PAR-NAÏVE-2) handles the situation in which the remaining process $Q$ has completed successfully and it is compensated for. For this reason the compensation $\beta'$ installed by $Q$ is activated. The final result is in any case $\lhd$, and the observed flow corresponds to the parallel execution of both branches followed by the execution of the compensation $\beta'$. Instead (F-PAR-NAÏVE-3) describes the cases in which $Q$ has already been compensated for (i.e., the result is $\lhd$ or $\lhd$), and therefore no further compensation is activated. Also, in this case the process $P|Q$ terminates abnormally.

Last rule (F-PAR-NAÏVE-4) describes the behavior when one of the branches finishes successfully and the other has been aborted and properly compensated for. Hence, the execution done by the successful branch needs to be reversed (by running $\beta'$) before activating the original compensation $\beta$. If the whole compensation (i.e., the execution of $\beta'; \beta$) finishes with success, then the final result is $\lhd$, otherwise the whole process terminates abnormally.

**Example 9.2** (Parallel Sagas). The second and third activities in Example 9.1 could be performed in parallel as shown in Figure 9.5. Nevertheless, in case some activity aborts, we would like all completed steps to be compensated for.

Although given rules allow a failed branch to start its compensation as soon as it aborts, the successful branch is forced to execute until completion, even when it will be compensated for. Consider the following parallel saga:

$$S \equiv \{ [A_1 \div B_1; A_2 \div B_2 | C_1 \div D_1] \}$$

and the context in which all activities but $C_1$ commit, i.e. $\Gamma = A_1 \mapsto \Box, A_2 \mapsto \Box, C_1 \mapsto \Box, B_1 \mapsto \Box, B_2 \mapsto \Box$, the only possible computation for $S$ produces as result $\Box$ with observation $(A_1; A_2|0); B_2; B_1$. 

![Figure 9.5: A parallel saga for handling orders.](image)
In a real execution of $S$ where $C_1$ fails while $A_1$ is still executing, it would be desirable to avoid the execution of both $A_2$ and its compensation $B_2$ by starting the compensation procedure as soon as $A_1$ finishes. (We remind that activities are atomic and hence their execution cannot be stopped once they have started). In general, when several processes execute concurrently and some activity aborts, then the whole saga aborts and every completed activity should be compensated for. Hence, it would be desirable to stop all processes before completion and to start the compensation procedure of partially executed branches as soon as a concurrent activity aborts. The following section presents a semantics for parallel sagas that allows such kind of behaviors.

### 9.3.2 Parallel Sagas Revised

To handle partial executions of successful branches during an aborted execution of a saga, we introduce two new kinds of results for a process running in a saga: (i) $\square$ denoting that the process has been forced to compensate and then it has been compensated for successfully, and (ii) $\Box$ denoting that the process has been forced to compensate but the compensation procedure has failed. Let $\emptyset, \sigma_1, \sigma_2$ range over $\mathcal{R} = \mathcal{R} \cup \{\square, \Box\}$. Moreover, we use the binary operator $\wedge$ over $\mathcal{R}$ to express the result obtained by combining the execution of two parallel branches. The associative and commutative operator $\wedge$ is defined in the following table (because of commutativity we omit half of the table).

<table>
<thead>
<tr>
<th>$\wedge$</th>
<th>$\square$</th>
<th>$\Box$</th>
<th>$\emptyset$</th>
<th>$\Box$</th>
<th>$\square$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\square$</td>
<td>$\square$</td>
<td>$\Box$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\square$</td>
</tr>
<tr>
<td>$\Box$</td>
<td>$\Box$</td>
<td>$\Box$</td>
<td>$\Box$</td>
<td>$\Box$</td>
<td>$\Box$</td>
</tr>
<tr>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$\Box$</td>
<td>$\Box$</td>
<td>$\Box$</td>
<td>$\Box$</td>
<td>$\Box$</td>
<td>$\Box$</td>
</tr>
<tr>
<td>$\square$</td>
<td>$\square$</td>
<td>$\Box$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\square$</td>
</tr>
</tbody>
</table>

Note that $\wedge$ is not defined when one operand is $\square$ and the other not. In fact, it is not possible for a branch to commit when the other aborts or fails: in the process $P|Q$, when $P$ commits but $Q$ does not, $P$ is forced to compensate. The other interesting cases are the two last rows on the table, in which one of the branches is forced to compensate (producing either $\Box$ or $\square$). If the remaining branch really fails (i.e., it reduces to a configuration with result $\Box$ or $\square$) then the parallel composition actually fails. Otherwise—if it is also forced to compensate—then the whole process has been forced to compensate.

The semantics for parallel sagas is given in Figure 9.6. All rules for sequential sagas remain unchanged but A-STEP, whose side condition considers also the new kinds of results for $\sigma$, and four new rules are added.

Rules (S-PAR), (F-PAR) and (C-PAR) specify the behavior of parallel composition. As for the naive semantics, parallel branches are run in parallel without initial
Figure 9.6: Semantics of parallel sagas.
compensations. If both branches commit (rule (S-PAR)), then the original compensation \( \beta \) is updated with the compensations \( \beta' \) and \( \beta'' \) installed by both branches. In particular, if the whole process \( P \mid Q \) has to be compensated, then \( \beta' \) and \( \beta'' \) are activated in parallel and \( \beta \) is started only when they finish.

If some branch has activated its compensation procedure, then also the other branch is required to be compensated for. If one of the branches fails during the compensation procedure (rule (F-PAR)), then the final result for \( P \mid Q \) will be a (possibly forced) abnormal termination (i.e., \( \Box \) or \( \Diamond \)). In this case the compensation \( \beta \) installed before the execution of \( P \mid Q \) is not even activated.

Finally, rule (C-PAR) handles the case in which both \( P \) and \( Q \) are successfully compensated for. In such case, also the previously installed compensation \( \beta \) is run.

The new rule (FORCED-ABT) handles the forced compensation of a process \( P \), i.e. \( P \) can activate the compensation procedure before starting its execution that will produce a forced termination \( \Box \) or \( \Diamond \). Nevertheless, by rule (SAGA), the execution of a saga ends only when \( P \) produces \( \Box \), \( \Box \) or \( \Diamond \). Hence, a valid execution forces a process to compensate only when it is a branch of a parallel composition. Moreover, in order to remove the tag of forced termination, the other branch is required to actually abort or finish abnormally. This is achieved by rules (F-PAR) and (C-PAR) that use the operator \( \land \) to combine the results of concurrent executions.

As done for sequential sagas, we state the correspondence between the proposed semantics and the intended meaning of parallel sagas. We start by defining some notions that are needed to formalize the correspondence.

**Definition 9.4** (Forward flow). The forward flow \(|S|\) of a parallel saga \( S \) is obtained by removing all compensations from \( S \), i.e. terms \( A \div B \) are replaced by \( A \).

In defining the order of a saga, we assume all activities to have a compensation \( (A \equiv A \div 0) \). Since activities in a saga are named differently, we univocally identify the compensation of an activity \( A \div B \) with \( A^{-1} \equiv B \).

**Definition 9.5** (Order of a saga). Let \( S \) be a parallel saga, the order of a saga \( S \) is the least transitive relation \( \prec_S \) on \( \mathcal{A}(S) \) s.t.:

1. if \( A \div B \) occurs in \( S \) then \( A \prec_S B \);
2. if \( P \mid Q \) then \( A \prec_S B \quad \forall A \in \mathcal{A}(|P|) \) and \( \forall B \in \mathcal{A}(|Q|) \);
3. if \( A, B \in \mathcal{A}(S) \) and \( A \prec_S B \) then \( B^{-1} \prec_S A^{-1} \).

Given \( A \subseteq \mathcal{A}(S) \), we write \( \prec_{S|A} \) for the order \( \prec_S \) restricted to the elements of \( A \). We will use \( A \prec_S \{A_1, \ldots, A_n\} \) (and \( \{A_1, \ldots, A_n\} \prec_S A \) if \( \exists i \) s.t. \( A \prec_S A_i \) (resp. \( A_i \prec_S A \).

The adequacy is now expressed by three theorems.

**Theorem 9.5** (Completion). Given a parallel saga \( S \), \( \Gamma \vdash S \overset{\alpha}{\rightarrow} \Box \) iff \( \forall A \in \mathcal{A}(|S|) : A \rightarrow \Box \in \Gamma \) and \( \alpha \equiv |S| \).
Theorem 9.6 (Successful compensation). Let \( S \) be a parallel saga. \( \Gamma \vdash S \xrightarrow{\alpha} \emptyset \) iff there exists a non empty set \( \mathcal{F}_A \subseteq \mathcal{A}(|S|) \) of failed activities (i.e., \( \forall A \in \mathcal{F}_A, A \rightarrow \emptyset \in \Gamma \)) s.t. \( \forall A, B \in \mathcal{F}_A: A \neq_S B \) and the following conditions hold:

1. \( \prec_A = \prec_{\mathcal{A}(\alpha)} \), i.e. the observed flow respects the flow given by \( S \);
2. if \( A \in \mathcal{A}(\alpha) \) then \( A \rightarrow \emptyset \in \Gamma \), i.e. all observed activities commit;
3. if \( A \in \mathcal{F}_A \) then \( A \notin \mathcal{A}(\alpha) \), i.e. failed activities are not observed;
4. if \( A \in \mathcal{A}(|S|) \) and \( A \prec_S \mathcal{F}_A \) then \( A \in \mathcal{A}(\alpha) \), i.e. all activities that precede \( \mathcal{F}_A \) are executed successfully;
5. if \( A \in \mathcal{A}(|S|) \) and \( \mathcal{F}_A \prec_S A \) then \( A \notin \mathcal{A}(\alpha) \), i.e. all activities after \( \mathcal{F}_A \) in the forward flow are not run;
6. if \( A \in \mathcal{A}(|S|) \) and \( A \in \mathcal{A}(\alpha) \) then \( A^{-1} \in \mathcal{A}(\alpha) \), i.e. all executed activities are compensated successfully.

Theorem 9.7 (Abnormal termination). Given a parallel saga \( S \). Then \( \Gamma \vdash S \xrightarrow{\alpha} \emptyset \) iff there exist a non empty set of failed activities \( \mathcal{F}_A \subseteq \mathcal{A}(|S|) \) s.t. \( \forall A_1, A_2 \in \mathcal{F}_A: A_1 \neq_S A_2 \), and a non empty set of failed compensations \( \mathcal{F}_C \subseteq \mathcal{A}(\mathcal{S}) \) s.t. \( \mathcal{F}_C \cap \mathcal{A}(|S|) = \emptyset \) and \( \forall B_1, B_2 \in \mathcal{F}_C: B_1 \neq_S B_2 \), and the following conditions hold: 1 – 5 as in Theorem 9.6 and

6'. if \( A \in \mathcal{A}(|S|) \) and \( A \in \mathcal{A}(\alpha) \) and \( A^{-1} \prec_S \mathcal{F}_C \) then \( A^{-1} \in \mathcal{A}(\alpha) \), i.e. activities whose compensations precede \( \mathcal{F}_C \) are compensated successfully;
7. if \( A^{-1} \in \mathcal{F}_C \) then \( A^{-1} \notin \mathcal{A}(\alpha) \), i.e. failed compensations are not executed;
8. if \( A \in \mathcal{A}(|S|) \) and \( A \in \mathcal{A}(\alpha) \) and \( \mathcal{F}_C \prec_S A^{-1} \) then \( A^{-1} \notin \mathcal{A}(\alpha) \), i.e. activities whose compensations follows \( \mathcal{F}_C \) are not compensated.

Above results are a generalization of Theorem 9.1. In fact, the order of a sequential saga is a total order, and constraints in Theorems 9.5–9.7 reduce to conditions in Theorem 9.1.

### 9.3.3 Proof of Adequacy Theorems

As in the previous case we start by proving several auxiliary results. The first one handles the case in which a flow executes successfully.

**Definition 9.6.** Let \( P \) be a process. The compensation for the completed execution of \( P \), written \( P^{-1} \), is defined as follow.

\[
0^{-1} = 0 \quad (A \div B)^{-1} = B \quad (P; Q)^{-1} = P^{-1}; Q^{-1} \quad (P \mid Q)^{-1} = P^{-1} \mid Q^{-1}
\]
Lemma 9.8. Let $P$ be a process and $\beta$ a process without compensations. Then
\[ \Gamma \vdash \langle P, \beta \rangle \xrightarrow{\alpha} \langle \Box, \beta' \rangle \text{ iff} \]
1. $\alpha \equiv |P|$ 
2. $\forall A \in \mathcal{A}(|P|) : A \rightarrow \Box \in \Gamma$ 
3. $\beta' \equiv P^{-1} \cdot \beta$

Proof.
\(\Rightarrow\) It follows by induction on the structure of the proof.

- **Case** (ZERO). Immediate.
- **Case** (s-ACT). Immediate.
- **Case** (s-STEP). Then the proof has the following shape
  \[
  \frac{
  \Gamma \vdash \langle Q, \beta \rangle \xrightarrow{\gamma} \langle \Box, \beta'' \rangle 
  \quad
  \Gamma \vdash \langle R, \beta'' \rangle \xrightarrow{\gamma'} \langle \Box, \beta' \rangle
  }{
  \Gamma \vdash \langle Q; R, \beta \rangle \xrightarrow{\gamma \gamma'} \langle \Box, \beta' \rangle
  }
  \]
  By inductive hypothesis on the first premise we have $\gamma \equiv |Q|$ and $\beta'' \equiv Q^{-1} \cdot \beta$. By inductive hypothesis on the second premise $\gamma' \equiv |R|$ and $\beta'' \equiv R^{-1} \cdot \beta''$ hold. Hence $\alpha \equiv \gamma; \gamma' \equiv |Q|; |R| \equiv |Q; R|$ and $\beta' \equiv R^{-1}; \beta'' \equiv R^{-1}; Q^{-1}; \beta \equiv (Q; R)^{-1}$.

- **Case** (s-PAR). Immediate by inductive hypothesis applied to both premises

\(\Leftarrow\) The proof follows by induction on the structure of $P$.

- **Case** $P \equiv 0$. Immediate by constructing the proof with rule (ZERO).
- **Case** $P \equiv A \div B$. Immediate by using rule (s-ACT).
- **Case** $P \equiv Q; R$. By applying inductive hypothesis on both $\langle R, \beta \rangle$ and $\langle Q, R^{-1}; \beta \rangle$ and then building the proof with the rule (s-STEP).
- **Case** $P \equiv Q|R$. By applying inductive hypothesis on both $\langle Q, 0 \rangle$ and $\langle R, 0 \rangle$ and then building the proof with the rule (s-PAR).

\(\Box\)

The following case handles the case in which a successful flow is forced to abort and the compensation procedure succeeds.

Lemma 9.9. Let $P$ be a process and $\beta$ a process without compensations. Then
\[ \Gamma \vdash \langle P, \beta \rangle \xrightarrow{\alpha; \beta} \langle \Box, 0 \rangle \text{ iff} \]
1. $\langle \alpha \rangle = s \cdot A(\alpha)$, i.e. the observed flow respects the flow given by $S$;
2. if \( A \in \mathcal{A}(\alpha) \) then \( A \rightarrow \Box \in \Gamma \), i.e. all observed activities commit;

3. if \( A \in \mathcal{A}(|S|) \) and \( A \in \mathcal{A}(\alpha) \) then \( A^{-1} \in \mathcal{A}(\alpha) \), i.e. all executed activities are compensated successfully.

4. \( \Gamma \vdash \langle \beta, 0 \rangle \xrightarrow{\beta} \langle \Box, 0 \rangle \)

Proof.

\( \Rightarrow \) It follows by induction on the structure of the proof.

- **Case (s-step).** Then \( P \equiv Q; R \), such that \( \Gamma \vdash \langle Q, \beta \rangle \xrightarrow{\alpha_1} \langle \Box, \beta_1 \rangle \) and \( \Gamma \vdash \langle R, \beta_1 \rangle \xrightarrow{\alpha_2} \langle \Box, 0 \rangle \). By Lemma 9.8, \( \alpha_1 \equiv |Q|, \beta_1 \equiv |Q|^{-1}; \beta \) and \( \forall A \in \mathcal{A}(Q) : A \rightarrow \Box \in \Gamma \). By inductive hypothesis, \( \alpha_2 \equiv \alpha'_2; |Q|^{-1}; \beta \) where \( \alpha'_2 \) and \( R \) satisfy the conditions (1)–(4). Note that the whole observation is \( \alpha = |Q|; \alpha'_2; |Q|^{-1}; \beta \), which makes conditions (1) and (3) to hold. Condition (2) is assured either by Lemma 9.8 or inductive hypothesis. While condition (4) is guarantied by inductive hypothesis.

- **Case (a-step).** By inductive hypothesis.

- **Case (c-par).** By inductive hypothesis on the two first premises and Lemma 9.8 for the third premise.

**Case (forced-abt).** By Lemma 9.8 applied to the premise.

\( \Leftarrow \) The proof follows by induction on the structure of \( P \).

- **Case** \( P \equiv 0 \). By constructing the proof with rule (forced-abt).

- **Case** \( P \equiv A \div B \). There are two cases: if \( A \in \mathcal{A}(\alpha) \), then by using rule (s-step). Otherwise by using rule (forced-abt).

- **Case** \( P \equiv Q; R \). There are two cases: if \( \mathcal{A}(Q) \subseteq \mathcal{A}(\alpha) \), then the proof is build by using rule (s-step), the premise \( \Gamma \vdash \langle Q, \beta \rangle \xrightarrow{\alpha_1} \langle \Box, \beta_1 \rangle \) follows by Lemma 9.8, while \( \Gamma \vdash \langle R, \beta_1 \rangle \xrightarrow{\alpha_2} \langle \Box, 0 \rangle \) is obtained by applying inductive hypothesis. Otherwise, by using rule (a-step), where the premise follow by applying inductive hypothesis.

- **Case** \( P \equiv Q|R \). The proof is built by using rule (c-par), where the two first premises follow by inductive hypothesis and the third one by Lemma 9.8.

Next lemma stands for flows that abort and are compensated successfully.

**Lemma 9.10.** Let \( P \) be a process and \( \beta \) a process without compensations. Then \( \Gamma \vdash \langle P, \beta \rangle \xrightarrow{\alpha_1} \langle \Box, 0 \rangle \) iff there exists a non empty set \( \mathcal{F}_A \subseteq \mathcal{A}(|P|) \) of failed activities (i.e., \( \forall A \in \mathcal{F}_A, A \rightarrow \Box \in \Gamma \)) s.t. \( \forall A, B \in \mathcal{F}_A : A \not\rightarrow B \) and the following conditions hold:

1. \( \prec_a = \prec_{P|\mathcal{A}(\alpha)} \), i.e. the observed flow respects the flow given by \( P \);
2. if \( A \in \mathcal{A}(\alpha) \) then \( A \mapsto \square \in \Gamma \), i.e. all observed activities commit;

3. if \( A \in \mathcal{F}_A \) then \( A \notin \mathcal{A}(\alpha) \), i.e. failed activities are not observed;

4. if \( A \in \mathcal{A}(|P|) \) and \( A \prec_P \mathcal{F}_A \) then \( A \in \mathcal{A}(\alpha) \), i.e. all activities that precede \( \mathcal{F}_A \) are executed successfully;

5. if \( A \in \mathcal{A}(|P|) \) and \( \mathcal{F}_A \prec_P A \) then \( A \notin \mathcal{A}(\alpha) \), i.e. all activities after \( \mathcal{F}_A \) in the forward flow are not run;

6. if \( A \in \mathcal{A}(|P|) \) and \( A \in \mathcal{A}(\alpha) \) then \( A^{-1} \in \mathcal{A}(\alpha) \), i.e. all executed activities are compensated successfully.

7. \( \Gamma \vdash \langle \beta, 0 \rangle \xrightarrow{\beta} \langle \square, 0 \rangle \)

Proof.

\( \Rightarrow \) It follows by induction on the structure of the proof.

- **Case** \((s\text{-CMP})\). Then \( A \in \mathcal{F}_A \), \( \alpha \equiv 0 \) and \( \prec_\alpha \equiv \emptyset \), which satisfy all conditions.

- **Case** \((s\text{-STEP})\). Then \( P \equiv Q; R \), such that \( \Gamma \vdash \langle Q, \beta \rangle \xrightarrow{\alpha_1} \langle \square, \beta_1 \rangle \) and \( \Gamma \vdash \langle R, \beta_1 \rangle \xrightarrow{\alpha_2} \langle \square, 0 \rangle \). By Lemma 9.8, \( \alpha_1 \equiv |Q|, \beta_1 \equiv |Q|^{-1}; \beta \) and \( \forall A \in \mathcal{A}(Q) : A \mapsto \square \in \Gamma \). By inductive hypothesis, \( \alpha_2 \equiv \alpha_2'; |Q|^{-1}; \beta \) where \( \alpha_2 \) and \( R \) satisfy the conditions \((1)-(7)\). Note that the whole observation is \( \alpha = |Q|; \alpha_2'; |Q|^{-1}; \beta \).

**Case** \((a\text{-STEP})\). By inductive hypothesis.

- **Case** \((c\text{-PAR})\). There are two cases. If both branches abort, the the proof follow by inductive hypothesis on the two first premises and Lemma 9.8 for the third premise. Otherwise, one branch aborts and the other is forced to abort. Then the proof is built by applying inductive hypothesis on the branch that aborts, Lemma 9.9 on the branch that is forced to abort and Lemma 9.8 for the execution of the compensation.

\( \Leftarrow \) The proof follows by induction on the structure of \( P \).

- **Case** \( P \equiv 0 \). There not exists \( \mathcal{F}_A \).

- **Case** \( P \equiv A \div B \). Then, the only possibility is \( \mathcal{F}_A = \{ A \} \). Hence the proof follows by \((s\text{-CMP})\).

- **Case** \( P \equiv Q; R \). There are two cases: if \( \forall A \in \mathcal{A}(|P|) \Rightarrow A \in \mathcal{A}(\alpha) \), then the proof is built by using \((s\text{-STEP})\). By Lemma 9.8, we have \( \Gamma \vdash \langle Q, \beta \rangle \xrightarrow{\alpha_1} \langle \square, \beta_1 \rangle \), and then by inductive hypothesis, while \( \Gamma \vdash \langle R, \beta_1 \rangle \xrightarrow{\alpha_2} \langle \square, 0 \rangle \).

- **Case** \( P \equiv Q|R \). There are two cases, if both branches abort, i.e., \( \mathcal{F}_A \cap \mathcal{A}(|Q|) \neq \emptyset \) and \( \mathcal{F}_A \cap \mathcal{A}(|R|) \neq \emptyset \), then the proof is obtained by rule \((c\text{-PAR})\). The two first premises hold by inductive hypothesis, while the third one is assured by condition \((7)\). In the other case, i.e., one branch aborts and the other is forced to compensate, e.g. \( \mathcal{F}_A \cap \mathcal{A}(|Q|) \neq \emptyset \) and \( \mathcal{F}_A \cap \mathcal{A}(|R|) = \emptyset \), then the proof is obtained also by using rule \((c\text{-PAR})\). By inductive hypothesis we have \( \Gamma \vdash \langle Q, \beta \rangle \xrightarrow{\alpha_1} \langle \square, 0 \rangle \), while \( \Gamma \vdash \langle R, \beta \rangle \xrightarrow{\alpha_2} \langle \square, 0 \rangle \) follows by Lemma 9.9.
9.3. PARALLEL SAGAS

The following lemma handles the case in which a flow is forced to compensate, but the compensation procedure fails.

**Lemma 9.11.** Let \( P \) be a process and \( \beta \) a process without compensations. Then \( \Gamma \vdash \langle P, \beta \rangle \xrightarrow{\alpha} \langle 
\bar{\alpha}, 0 \rangle \) if and only if

1. \( \prec_{\alpha} \prec_{P,} A(\alpha) \), i.e. the observed flow respects the flow given by \( P \);
2. if \( A \in A(\alpha) \) then \( A \mapsto \Box \in \Gamma \), i.e. all observed activities commit;
3. if \( A \in \mathcal{F}_A \) then \( A \not\in A(\alpha) \), i.e. failed activities are not observed;
4. if \( A \in A([P]) \) and \( A \prec_P \mathcal{F}_A \) then \( A \in A(\alpha) \), i.e. all activities that precede \( \mathcal{F}_A \) are executed successfully;
5. if \( A \in A([P]) \) and \( \mathcal{F}_A \prec_P A \) then \( A \not\in A(\alpha) \), i.e. all activities after \( \mathcal{F}_A \) in the forward flow are not run;
6. if \( \mathcal{F}_C \not\subseteq A(\beta) \) and \( A \in A([P]) \) and \( A \in A(\alpha) \) and \( A^{-1} \prec_s \mathcal{F}_C \) then \( A^{-1} \in A(\alpha) \), i.e. when the failed compensations are in \( P \), then only those compensations preceding \( \mathcal{F}_C \) are compensated successfully;
7. if \( \mathcal{F}_C \not\subseteq A(\beta) \) then:
   - \( \forall A \in A([P]) : A \in A(\alpha) \) implies \( A^{-1} \in A(\alpha) \); and
   - if \( A \in A(\beta) \) and \( A \prec_P \mathcal{F}_C \) then \( A \in A(\alpha) \), i.e. all executed activities in \( P \) are compensated, and all activities in \( \beta \) preceding \( \mathcal{F}_C \) are successful;
8. if \( A^{-1} \in \mathcal{F}_C \) then \( A^{-1} \not\in A(\alpha) \), i.e. failed compensations are not executed;
9. if \( A \in A(S; \beta) \) and \( \mathcal{F}_C \prec_{S; \beta} A \) then \( A \not\in A(\alpha) \), i.e. all compensations after \( \mathcal{F}_C \) are not executed.

**Proof.**

\( \Rightarrow \) It follows by induction on the structure of the proof.

- **Case** (f-CMP). The proof follow by Lemma 9.10 for obtaining the premise. Note that in this case all observed activities belong to \( \beta \), hence conditions are trivially satisfied.

- **Case** (s-STEP). By Lemma 9.8 for the first premise and inductive hypothesis on the second premise.

- **Case** (A-STEP). By inductive hypothesis.
• **Case** (f-PAR). In this case there are two main cases. If both branches have been forcer to compensate and both fail, then the proof follow by inductive hypothesis. Otherwise, if just one of them fails the proof follow by Lemma 9.9 for the branch that compensate successfully and inductive hypothesis for the other.

• **Case** (c-PAR). The proof follow by using Lemma 9.9 for the first two premises, while Lemma 9.10 for the third one.

• **Case** (f-ABT). By Lemma 9.10.

\[
\implies \text{The proof follows by induction on the structure of } P.
\]

*Case* \( P \equiv 0 \). By Lemma 9.10 we have \( \Gamma \vdash \langle \beta, 0 \rangle \xrightarrow{\alpha} \langle \emptyset, 0 \rangle \), hence the proof is construct with rule (FORCED-ABT).

*Case* \( P \equiv A \vdash B \). By using rule (f-CMP), where the premise follows by Lemma 9.10.

*Case* \( P \equiv Q; R \). There are two cases: if \( \mathcal{A}(\{Q\}) \subseteq \mathcal{A}(\alpha) \), then the proof is build by using rule (S-STEP), the premise \( \Gamma \vdash \langle Q, \beta \rangle \xrightarrow{\alpha_1} \langle \emptyset, \beta_1 \rangle \) follows by Lemma 9.8, while \( \Gamma \vdash \langle R, \beta_1 \rangle \xrightarrow{\alpha_2} \langle \emptyset, 0 \rangle \) is obtained by applying inductive hypothesis. Otherwise, by using rule (A-STEP), where the premise follow by applying inductive hypothesis.

*Case* \( P \equiv Q\mid R \). There are two cases: (i) \( \mathcal{F}_C \cap \mathcal{A}(Q\mid R) = \emptyset \), then the proof by using rule (c-PAR), where third first premises follow by Lemma 9.9. (ii) if \( \mathcal{F}_C \cap \mathcal{A}(Q\mid R) \neq \emptyset \), then the proof follows by rule (f-PAR)

\[\square\]

**Lemma 9.12.** Let \( P \) be a process and \( \beta \) a process without compensations. Then \( \Gamma \vdash \langle P, \beta \rangle \xrightarrow{\alpha} \langle \emptyset, 0 \rangle \) iff

1. \( \ll \alpha = \ll_{P\mid \mathcal{A}(\alpha)} \), i.e. the observed flow respects the flow given by \( P \);
2. if \( A \in \mathcal{A}(\alpha) \) then \( A \rightarrow \Box \in \Gamma \), i.e. all observed activities commit;
3. if \( A \in \mathcal{F}_A \) then \( A \not\in \mathcal{A}(\alpha) \), i.e. failed activities are not observed;
4. if \( A \in \mathcal{A}(\{|P|\}) \) and \( A \ll_{P\mid \mathcal{F}_A} A \) then \( A \in \mathcal{A}(\alpha) \), i.e. all activities that precede \( \mathcal{F}_A \) are executed successfully;
5. if \( A \in \mathcal{A}(\{|P|\}) \) and \( \mathcal{F}_A \ll_{P} A \) then \( A \not\in \mathcal{A}(\alpha) \), i.e. all activities after \( \mathcal{F}_A \) in the forward flow are not run;
6. if \( \mathcal{F}_C \not\subseteq \mathcal{A}(\beta) \) and \( A \in \mathcal{A}(\{|P|\}) \) and \( A \in \mathcal{A}(\alpha) \) and \( A^{-1} \ll_{S} \mathcal{F}_C \) then \( A^{-1} \in \mathcal{A}(\alpha) \), i.e. when the failed compensations are in \( P \), then only those compensations preceding \( \mathcal{F}_C \) are compensated successfully;
7. if \( \mathcal{F}_C \not\subseteq \mathcal{A}(\beta) \) then:
9.4 Adding Nesting to Sagas

Nesting has been introduced in database transactions to localize failures within a transaction and to allow partial roll-backs [90]. Basically, a nested transaction is decomposed into a hierarchy of activities called subtransactions. The root of the hierarchy is usually referred to as the top-level transaction. In this scheme, any subtransaction executes independently and concurrently with respect to its parent and siblings, deciding autonomously to commit or abort. When a transaction aborts all its subtransactions should abort and consequently all committed subtransactions must be rolled back. Nevertheless, a top-level transaction can commit even though some subtransactions have aborted.

**Definition 9.7** (Nested Sagas). Nested sagas are defined by the following grammar:

\[
\begin{align*}
\text{(STEP)} \quad X &::= 0 \mid A \mid A \div B \\
\text{(PROCESS)} \quad P &::= X \mid P;P \mid P|P \mid S \\
\text{(SAGA)} \quad S &::= \{[P]\}
\end{align*}
\]

The additional rules for nested sagas are in Figure 9.7. The main idea is that a subtransaction \{[P]\} executes \(P\) in an independent thread without initial compensation. The successful completion of \{[P]\} (rule SUB-CMT) is analogous to the case of a successful activity (rule S-ACT). When the subtransaction commits, the compensation \(\beta'\) computed by \(P\) is installed on top of the compensations.

Rule (SUB-ABT) describes the silent abortion of a subtransaction. As aforementioned, nesting is intended to allow the commit of a transaction even when some activities fail. That is, if an activity in \(P\) fails while running \{[P]\}, and the executed activities of \(P\) are successfully compensated for, then the abort is hidden to the parent. For this reason the result associated with \{[P]\} is \(\Box\) even though \(P\) aborts. (We discuss another possibility for handling this situation in Section 9.5.3). The observed flow corresponds to the execution of \(P\) and the original compensation \(\beta\) is not modified.
\[(\text{SUB-CMT})\]
\[
\Gamma \vdash \langle P, 0 \rangle \xrightarrow{\alpha} \langle \Box, \beta' \rangle \\
\Gamma \vdash \langle [[P]], \beta \rangle \xrightarrow{\alpha} \langle \Box, \beta' ; \beta \rangle
\]

\[(\text{SUB-ABT})\]
\[
\Gamma \vdash \langle P, 0 \rangle \xrightarrow{\alpha} \langle \Box, 0 \rangle \\
\Gamma \vdash \langle [[P]], \beta \rangle \xrightarrow{\alpha} \langle \Box, \beta \rangle
\]

\[(\text{SUB-FAIL})\]
\[
\Gamma \vdash \langle P, 0 \rangle \xrightarrow{\alpha} \langle \Box, 0 \rangle \\
\Gamma \vdash \langle [[P]], \beta \rangle \xrightarrow{\alpha} \langle \Box, 0 \rangle
\]

\[(\text{SUB-FORCED-1})\]
\[
\Gamma \vdash \langle P, 0 \rangle \xrightarrow{\alpha} \langle \Box, 0 \rangle \\
\Gamma \vdash \langle [[P]], \beta \rangle \xrightarrow{\alpha} \langle \Box, 0 \rangle
\]

\[(\text{SUB-FORCED-2})\]
\[
\Gamma \vdash \langle P, 0 \rangle \xrightarrow{\alpha} \langle \Box, 0 \rangle \\
\Gamma \vdash \langle [P], \beta \rangle \xrightarrow{\gamma} \langle \Box_1, 0 \rangle \\
\square_2 = \left\{ \begin{array}{l}
\Box \quad \text{if } \Box_1 = \Box \\
\Box \quad \text{if } \Box_1 \in \{ \Box, \Box \}
\end{array} \right.
\]

Figure 9.7: Semantics of nested Sagas

Instead, \{\{P\}\} ends abnormally when \(P\) has an abnormal termination (SUB-FAIL). This result is propagated until the top-level transaction, which will finish abnormally. (Section 9.5.2 introduces local handlers for abnormal termination).

The three rules described above do not allow a subtransaction to be stopped and compensated for when a concurrent activity aborts.

Consider the saga \(S \equiv \{ \{P\} \mid A \div B \}\) and a context \(\Gamma = A \mapsto \Box, \Gamma'\) such that \(\Gamma \vdash \langle P, 0 \rangle \xrightarrow{\alpha} \langle \Box, \beta \rangle\) and \(\Gamma \vdash \langle \beta, 0 \rangle \xrightarrow{\alpha'} \langle \Box, 0 \rangle\), the whole saga \(S\) should abort because \(A\) aborts. With the rules seen until now we can build only the two derivations shown in Figure 9.8. The branch \{\{P\}\} is forced to abort either before (Figure 9.8(a)) or after (Figure 9.8(b)) the whole execution of \(P\). Nevertheless, if \(P\) is a composed process, for instance a sequence, it is not possible to stop the execution of \(P\) once it starts. To allow the activation of the compensation procedure in subtransactions as soon as possible, we add the last two rules in Figure 9.7, which handle the interruption and compensation of a subtransaction. Rule (SUB-FORCED-1) handles the failure of the forced compensation. In this case the compensation \(\beta\) previously installed is not activated since the compensation procedure fails. On the contrary, rule (SUB-FORCED-2) activates \(\beta\) when \(P\) is compensated successfully.

\textbf{Example 9.3 (Nested sagas).} The organization has a reward program in which users accumulate points when they purchase. The activity \textbf{Add points} updates the reward balance of a user. This activity aborts when the buyer is not part of the reward program. Clearly, we do not like the whole process to abort when the user is not registered, for this reason we model this activity as a nested transaction (see Figure 9.9). The compensation \textbf{Subtract points} undoes the step \textbf{Add points} when some activity in the top level flow fails.

To formalize the adequacy theorems we need some preliminary definitions.
9.4. ADDING NESTING TO SAGAS  

\[
\begin{align*}
\text{[F-A]} & \quad \Gamma \vdash \langle 0, 0 \rangle \xrightarrow{0} \langle \boxempty, 0 \rangle \\
\text{[C-P]} & \quad \Gamma \vdash \langle 0, 0 \rangle \xrightarrow{0} \langle \boxempty, 0 \rangle \\
\text{[Z]} & \quad \Gamma \vdash \langle 0, 0 \rangle \xrightarrow{0} \langle \boxempty, 0 \rangle \\
\text{[S-C]} & \quad A \mapsto \text{false}, \Gamma' \vdash \langle A \div B, 0 \rangle \xrightarrow{0} \langle \boxempty, 0 \rangle \\
\text{[S-A]} & \quad \Gamma \vdash \langle \{P\} \mid A \div B, 0 \rangle \xrightarrow{0} \langle \boxempty, 0 \rangle \\
\text{[S-F]} & \quad \Gamma \vdash \langle \{P\} \mid A \div B \rangle \xrightarrow{\alpha \gamma} \langle \boxempty, 0 \rangle
\end{align*}
\]

(a) \{P\} is not activated

(b) \( P \) is completely executed

**Figure 9.8:** Possible executions of \( S \equiv \{ \{P\} \mid A \div B \} \) when \( \Gamma = A \mapsto \text{false}, \Gamma' \).

**Definition 9.8** (Subtransactions). The set of all subtransactions of \( S \) are \( \mathcal{S}(S) = \{ \{P\} \mid \{P\} \text{ is a proper subterm of } S \} \) while the top-level subtransactions are \( \mathcal{S}_{top}(S) = \{ S' \mid S' \in \mathcal{S}(S) \land \forall S'' \in \mathcal{S}(S) : S' \not\in \mathcal{S}(S'') \} \). The set of all top-level activities of \( S \) is \( \mathcal{A}_{top}(S) = \{ A \mid A \in \mathcal{A}(\{P\}) \text{ and } A \text{ does not occur in a subterm } \{P\} \} \).

The definition of order of a saga considers only top activities and subtransactions (i.e., \( A \) and \( B \) range over top activities and subtransactions, and \( A^{-1} \) denotes also compensations of subtransactions seen as symbolic atoms). Again, we break the adequacy results in three theorems.

**Theorem 9.13** (Completion). Let \( S \) be a nested saga. Then \( \Gamma \vdash S \xrightarrow{\alpha} \boxempty \) iff

1. if \( A \in \mathcal{A}_{top}(\{P\}) \) then \( A \mapsto \boxempty \in \Gamma \);

2. if \( S' \in \mathcal{S}_{top}(S) \) then \( \Gamma \vdash \langle S', 0 \rangle \xrightarrow{\alpha'} \langle \boxempty, \beta \rangle \) for some \( \alpha', \beta \).

**Theorem 9.14** (Successful compensation). Let \( S \) be a nested saga. Then \( \Gamma \vdash S \xrightarrow{\alpha} \boxempty \) iff there exists a non empty set of failed activities \( \mathcal{F}_A \subseteq \mathcal{A}_{top}(\{P\}) \) s.t. \( \forall A, B \in \mathcal{F}_A : A \not\in S B \) and the following conditions hold:

1. if \( A \in \mathcal{A}(\alpha) \) then \( A \mapsto \boxempty \in \Gamma \);

2. if \( A \in \mathcal{F}_A \) then \( A \not\in \mathcal{A}(\alpha) \);

3. if \( A \in \mathcal{A}_{top}(\{P\}) \) and \( A \not\in \mathcal{F}_A \) then \( A \in \mathcal{A}(\alpha) \), i.e. all top activities that precede \( \mathcal{F}_A \) are executed successfully;

\[
\begin{align*}
\text{[F-A]} & \quad \Gamma \vdash \langle 0, 0 \rangle \xrightarrow{0} \langle \boxempty, 0 \rangle \\
\text{[C-P]} & \quad \Gamma \vdash \langle 0, 0 \rangle \xrightarrow{0} \langle \boxempty, 0 \rangle \\
\text{[Z]} & \quad \Gamma \vdash \langle 0, 0 \rangle \xrightarrow{0} \langle \boxempty, 0 \rangle \\
\text{[S-C]} & \quad A \mapsto \text{false}, \Gamma' \vdash \langle A \div B, 0 \rangle \xrightarrow{0} \langle \boxempty, 0 \rangle \\
\text{[S-A]} & \quad \Gamma \vdash \langle \{P\} \mid A \div B, 0 \rangle \xrightarrow{0} \langle \boxempty, 0 \rangle \\
\text{[S-F]} & \quad \Gamma \vdash \langle \{P\} \mid A \div B \rangle \xrightarrow{\alpha \gamma} \langle \boxempty, 0 \rangle
\end{align*}
\]
4. if $S' \in S_{\text{top}}(S)$ and $S' \prec_S F_A$ then $\Gamma \vdash \langle S', 0 \rangle \xrightarrow{\alpha'} \langle \Box, \beta \rangle$, and $\Gamma \vdash \langle \beta, 0 \rangle \xrightarrow{\gamma} \langle \Box, 0 \rangle$ for some $\alpha', \beta$ and $\gamma$, i.e. all top subtransactions before $F_A$ (and their compensations) are successful;

5. if $A \in A_{\text{top}}(|S|)$ and $F_A \prec_S A$ then $A \not\in A(\alpha)$, i.e. all activities in the forward flow after $F_A$ are not executed;

6. if $S' \in S_{\text{top}}(S)$ and $F_A \prec_S S'$ then $\forall A \in A(S') : A \not\in A(\alpha)$, i.e. activities of subtransactions following $F_A$ are not executed;

7. if $A \in A(|S|)$ and $A \in A(\alpha)$ then $A^{-1} \in A(\alpha)$, i.e. all executed activities are compensated successfully.

**Theorem 9.15** (Abnormal termination). Let $S$ be a nested saga. Then $\Gamma \vdash S \xrightarrow{\sigma} \Box$ if and only if there exist: (i) a set of failed activities $F_A \subseteq A(|S|)$ and (ii) a set of abnormal terminated subtransactions $F_S \subseteq S_{\text{top}}(S)$ s.t.: $F_A = F_A \cup F_S$ is not empty and $\forall A_1, A_2 \in F_A$: $A_1 \not\prec_S A_2$, (iii) a set of failed compensations $F_C \subseteq A_{\text{top}}(S)$ s.t. $F_C \cap A(|S|) = \emptyset$, (iv) a set of precommitted subtransactions with failed compensations $F_P \subseteq S_{\text{top}}(S)$ s.t. $F_C = F_C \cup F_S^{-1} \cup F_P^{-1}$ is not empty and $\forall A_1, A_2 \in F_C$: $A_1 \not\prec_S A_2$, and the following conditions hold: conditions 1, 3, 5, 6 as in Theorem 9.14 and

2'. if $A \in F_A$ then $A \not\in A(\alpha)$

4'. if $S' \in S_{\text{top}}(S)$, $S' \prec_S F_A$, and $S'^{-1} \prec_S F_C$ then $\Gamma \vdash \langle S', 0 \rangle \xrightarrow{\alpha'} \langle \Box, \beta \rangle$ and $\Gamma \vdash \langle \beta, 0 \rangle \xrightarrow{\gamma} \langle \Box, 0 \rangle$

7'. if $A \in A_{\text{top}}(|S|)$ and $A \in A(\alpha)$ and $A^{-1} \prec_S F_C$ then $A^{-1} \in A(\alpha)$, i.e. activities whose compensations precede $F_C$ are compensated successfully.

8. if $S' \in F_S$ then $\Gamma \vdash \langle S', 0 \rangle \xrightarrow{\sigma'} \langle \sigma, 0 \rangle$ with $\sigma \in \{\Box, \Box\}$, i.e. failed subtransactions terminate abnormally;

9. if $S' \in F_P$ then $S' \prec_S F_A$, $\Gamma \vdash \langle S', 0 \rangle \xrightarrow{\alpha'} \langle \Box, \beta \rangle$ and $\Gamma \vdash \langle \beta, 0 \rangle \xrightarrow{\gamma} \langle \Box, 0 \rangle$, i.e. failed precommitted subtransactions complete successfully but their installed compensations fail.

10. if $(S' \in S_{\text{top}}(S)$ and $F_C \prec_S S'^{-1}$ and $A \in A(S') \) or $(A \in A_{\text{top}}(S)$ and $F_C \prec_S A^{-1})$ then $A^{-1} \not\in A(\alpha)$, i.e. compensations after $F_C$ are skipped.

The proofs of Theorems 9.13–9.15 proceed as in Section 9.3.3.

Previous results do not characterize precisely the order of the observation $\alpha$ (as in previous sections). Instead they state the set of executed steps and the result they produce.
9.5 Additional features

This section presents further extensions of nested sagas.

9.5.1 Programmable compensations

The compensation mechanism described until now is usually referred to as implicit or default compensation. In addition, some composition languages (such as BPEL4WS) allow the programmer to explicitly define the compensation procedure associated with a completed subtransaction. In our case, the syntax of steps should include terms with the following form: $S \div P$. Consider the long running transaction $S \equiv \{ [A_1 \div B_1 ; A_2 \div B_2 ] \div P ; A_3 \}$, which should behave as follows: when $A_1$ commits and $A_2$ aborts, the default mechanism should compensate $A_1$ by activating $B_1$. Instead, if $A_1$ and $A_2$ commit while $A_3$ aborts, the programmed compensation $P$ (and not the default $B_2 ; B_1$) should be run.

The difference between default and programmable compensations is that the former are always flat processes without compensations whose executions always produce results like $\langle \Box, 0 \rangle$, while the latter can produce $\langle \Box, \beta \rangle$. Let $P \equiv C_1 \div D_1 ; \{ Q \} \div Q' ; C_2 \div D_2$ in $S$ above. Clearly, if $A_3$ fails, then $P$ is activated and it can commit, abort or terminate abnormally, but it may also generate compensations. In particular, if $P$ commits, then it produces $\langle \Box, D_2 ; Q' ; D_1 \rangle$ but the rules used in previous sections, which assume compensations not to generate new compensations (i.e. S-CMP, F-CMP, C-PAR, FORCED-ABT and SUB-FORCED-2), will not handle this expected behavior.

One alternative for dealing with programmable compensations is to restrict their syntax to allow only basic activities or processes without compensations. Similarly, without imposing a syntactical restriction, to make compensations to behave as their forward flow, as follow.

\[
\begin{align*}
\Gamma \vdash \langle P, 0 \rangle & \xrightarrow{\alpha} \langle \Box, \beta \rangle \\
\Gamma \vdash \langle \{ P \} \div Q, \beta \rangle & \xrightarrow{\alpha} \langle \Box, \{ Q \}; \beta \rangle
\end{align*}
\]
This rule is similar to (SUB-CMT) in Figure 9.7 that installs the default compensation $\beta'$. Differently, rule (PGM-CMP) discards $\beta'$ and replaces it by the forward flow $|Q|$ of $Q$. We recall that the forward flow of a process is obtained by replacing in $P$ each term $Q \div Q^{-1}$ by $Q$. This will assure that the execution of a compensation never generates new compensations neither terminates abnormally.

On the contrary, it could be possible to take into account compensations produced by compensations (as done in StAC) and to install and use them to repeatedly compensate a process. For instance by adding the following rule

\[ \frac{\Gamma \vdash \langle P, \beta \rangle \xrightarrow{\alpha} \langle \mathbb{K}, \beta'' \rangle \quad \Gamma \vdash \langle \beta'', 0 \rangle \xrightarrow{\alpha'} \langle \boxslash, \beta' \rangle \quad \beta'' \neq 0}{\Gamma \vdash \langle P, \beta \rangle \xrightarrow{\alpha; \alpha'} \langle \boxslash, \beta' \rangle} \]

Consider the process $\llbracket \{ [P] \} \div A_0 ; A_1 \div (B_1 \div C_1) ; R \rrbracket$ and an execution in which $\llbracket P \rrbracket$ commits and installs $A_0$ as a compensation. Then, $A_1$ commits and installs $B_1 \div C_1$ on top. Suppose now that $R$ fails and starts the compensation procedure by executing $B_1 \div C_1$. If $B_1$ commits, $C_1$ will be installed and could be activated later on, for instance when $A_0$ aborts. Note that this kind of definitions generates an upward flow of control when a compensation fails, i.e. the failure of $A_0$ activates $C_1$. In our approach, where compensations are used to undo committed steps, the meaning of such a construction is quite obscure. Basically, it would mean that a successful execution of $A_1$ can be undone by running $B_1$, which can be in turn compensated with $C_1$. In particular if they are perfect compensations, i.e. they remove all the effects, the term $A_1 \div (B_1 \div C_1)$ leaves all the effects of $A_1$ when the compensation procedure fails. Moreover it is difficult to figure out real cases in which repeated compensation is really necessary. In our opinion the failure of a compensation can be modelled more naturally by exploiting an exception handling mechanism like the one presented in Section 9.5.2. For this reason, we prefer rule (PGM-CMP) instead of (REPEATED-CMP) for handling programmable compensations.

### 9.5.2 Exception handling

A basic exception handling mechanism can be added to the presented languages by interpreting the result $\langle \mathbb{E}, \beta \rangle$ as a process that raises an exception. At the syntactic level, we can consider exception handlers introduced by steps like `try S with P`, where $S$ is a saga and $P$ a generic process. The behavior for such processes is defined in Figure 9.10.

The first rule handles the case in which $S_1$ finishes without raising an exception. As usual, the exception handler $P_2$ is discarded, and the compensation created by $S_1$ is installed on top of the stack. The second rule describes the activation of the handler $P_2$ when $S_1$ raises an exception. Note that $P_2$ starts with the original compensation $\beta$. The last rule handles the activation of the compensation handler when the abnormal termination is reached during a forced compensation. The handler is
also run in this case because it is intended to finish the compensation procedure. Note that the final result is always a forced termination.

Although the described mechanism is naïve, it illustrates the interplay between both concepts: compensations undo partial executions of transactions, while exception handling deals with incomplete compensations.

### 9.5.3 Alternatives to aborted subtransactions

In the nested model we presented in Section 9.4, the behavior of a parent transaction does not depend on the completion/abortion of its subtransactions. In fact, rule SUB-ABT hides the parent transaction the fact that one (or more) of its subtransactions have not been executed (i.e., compensated). Nevertheless, workflow systems usually allow the possibility of specifying forward recovery strategies for a process that fails to commit, such as the retry of the activity or the execution of an alternative process.

These aspects can be modelled by extending the language with new primitive steps `try S or P` whose behavior can be described with the following rules

\[
\frac{ \Gamma \vdash \langle S, 0 \rangle \xrightarrow{\alpha} \langle \square, \beta' \rangle \quad \text{with} \quad \square \in \{\square, \mathbb{E}, \mathbb{M}\} }{ \Gamma \vdash \langle \text{try } S \text{ with } P, \beta \rangle \xrightarrow{\alpha} \langle \square, \beta' ; \beta \rangle }
\]

\[
\frac{ \Gamma \vdash \langle S, 0 \rangle \xrightarrow{\alpha} \langle \mathbb{E}, 0 \rangle \quad \Gamma \vdash \langle P, \beta \rangle \xrightarrow{\alpha'} \langle \square, \beta' \rangle }{ \Gamma \vdash \langle \text{try } S \text{ with } P, \beta \rangle \xrightarrow{\alpha;\alpha'} \langle \square, \beta' \rangle }
\]

\[
\frac{ \Gamma \vdash \langle S, 0 \rangle \xrightarrow{\alpha} \langle \mathbb{M}, 0 \rangle \quad \Gamma \vdash \langle P, \beta \rangle \xrightarrow{\alpha'} \langle \square_1, \beta' \rangle }{ \Gamma \vdash \langle \text{try } S \text{ with } P, \beta \rangle \xrightarrow{\alpha;\alpha'} \langle \square_2, \beta' \rangle }
\]

\[
\text{with} \quad \square_2 = \begin{cases} \mathbb{M} & \text{if } \square_1 = \mathbb{M} \\ \mathbb{M} & \text{otherwise} \end{cases}
\]
(5-choice)
\[ \Gamma \vdash \langle P, 0 \rangle \xrightarrow{\alpha} \langle \Box, 0 \rangle \quad \Gamma \vdash \langle Q, 0 \rangle \xrightarrow{\alpha'} \langle \Box, 0 \rangle \quad \text{with } \Box \in \{\mathfrak{F}, \mathfrak{F}^\perp\} \]

(F-choice)
\[ \Gamma \vdash \langle P, 0 \rangle \xrightarrow{\alpha} \langle \sigma_1, 0 \rangle \quad \Gamma \vdash \langle Q, 0 \rangle \xrightarrow{\alpha} \langle \sigma_2, 0 \rangle \quad \{ \begin{array}{l} \sigma_1 \in \{\mathfrak{F}, \mathfrak{F}^\perp\} \\ \sigma_2 \in \{\mathfrak{F}, \mathfrak{F}^\perp, \mathfrak{F}^\perp, \mathfrak{F}\} \end{array} \]

(c-choice)
\[ \Gamma \vdash \langle P, 0 \rangle \xrightarrow{\alpha} \langle \sigma_1, 0 \rangle \quad \Gamma \vdash \langle Q, 0 \rangle \xrightarrow{\alpha'} \langle \sigma_2, 0 \rangle \quad \Gamma \vdash \langle \beta, 0 \rangle \xrightarrow{\gamma} \langle \Box_1, 0 \rangle \]
\[ \Gamma \vdash \langle P \oplus Q, \beta \rangle \xrightarrow{(\alpha|\sigma_1') \cdot \gamma} \langle \sigma_1 \land \sigma_2 \land \Box_2, 0 \rangle \]
\[ \sigma_1, \sigma_2 \in \{\mathfrak{F}, \mathfrak{F}^\perp\} \quad \text{and} \quad \Box_2 = \begin{cases} \mathfrak{F} & \text{if } \Box_1 = \Box \\ \mathfrak{F}^\perp & \text{otherwise} \end{cases} \]

Figure 9.11: Semantics for the choice of a successful branch: \( P \oplus Q \)


(5-alt)
\[ \Gamma \vdash \langle S, 0 \rangle \xrightarrow{\alpha} \langle \Box, 0 \rangle \]

(F-alt)
\[ \Gamma \vdash \langle S, 0 \rangle \xrightarrow{\alpha} \langle \Box, 0 \rangle \quad \text{with } \Box \in \{\mathfrak{F}, \mathfrak{F}^\perp, \mathfrak{F}^\perp, \mathfrak{F}\} \]

(t-alt)
\[ \Gamma \vdash \langle \text{try } S \text{ or } P, \beta \rangle \xrightarrow{\alpha} \langle \Box, 0 \rangle \quad \Gamma \vdash \langle P, \beta \rangle \xrightarrow{\alpha'} \langle \Box, 0 \rangle \]
\[ \Gamma \vdash \langle \text{try } S \text{ or } P, \beta \rangle \xrightarrow{\alpha | \alpha'} \langle \Box, 0 \rangle \]

This mechanism is similar to the exception handling described above. Nevertheless, while exception handling is used during backward computation (for failed compensations) alternative procedures are used as a forward recovery mechanism. For this reason, an alternative is activated only when the subtransaction aborts. Moreover, by rule (F-alt), which shift the forced abortion \( \mathfrak{F} \) to the parent level, alternatives are not executed during a forced termination. In fact, alternatives are intended to be used while executing towards a completion not during the compensation procedure.

9.5.4 Choices
The recovery capability introduced above allows the sequential search of one process that executes successfully. Some composition languages (like BPML [17]) allow alternatives to be explored in parallel (this kind of choice is known as discriminator).
Once one branch finishes successfully all the remaining alternatives are stopped and compensated for. We will write these processes as $P \parallel Q$, and we assume $\parallel$ associative and commutative. Inference rules are in Figure 9.11. The last two rules (i.e., abnormal termination ($\tau$-CHOICE) and abort (($C$-CHOICE)) are analogous to those for parallel composition. Differently, a choice $P \parallel Q$ succeeds only when one branch commits and the other has been successfully (possible forced) compensated for (rule $S$-CHOICE). A computation cannot go forward when one branch terminates abnormally because the state of the system is inconsistent. Hence, the successful branch is forced to compensate and the whole process $P \parallel Q$ ends abnormally ($\tau$-CHOICE).

These kind of choices, where the selection takes place once one of the branches has completed successfully, are quite different to usual internal or external choices. Internal choices $P \sqcap Q$ can be defined straightforwardly by defining a unique rule and requiring $\sqcap$ to be associative and commutative.

\[
\frac{\gamma \vdash \langle P, \beta \rangle}{\Gamma \vdash \langle \square, \beta' \rangle} \quad (\text{INT-CHOICE})
\]

\[
\frac{\gamma \vdash \langle P \sqcap Q, \beta \rangle}{\Gamma \vdash \langle \square, \beta' \rangle}
\]

External choices are related to the notion of synchronization of events that make the description of flows not structured. We analyze more in detailed the synchronization between flows in the following section.

### 9.5.5 Link dependencies

This section discusses the synchronization of concurrent flows. Consider the flow depicted in Figure 9.12, and let

\[P \equiv A_1 \parallel B_1; (A_2 \parallel B_2; A_4 \parallel B_4; A_3 \parallel B_3; A_5 \parallel B_5); A_6 \parallel B_6\]

with the additional constraint stating that $A_4$ must be executed after $A_3$, written $\text{link}(A_3, A_4)$. Hence, any valid execution $\alpha$ of $P$ must hold both the order given by $P$ and the additional constraints $A_3 \prec_\alpha A_4$. Although all languages agree on this meaning for links (or synchronization) while computing forward, it is less clear which is the desired behavior when compensating. For instance, $\text{StAC}$ (which provides an operator for parallel composition with synchronization over a name set) ignores all synchronizations when computing backward. For instance, if $A_6$ fails during the execution of $P$, then, according to the compensation policy of $\text{StAC}$, the compensation procedure could activate $B_3$ before the termination of $B_4$. In our opinion this semantics has a main drawback in that the encoding of sequential composition as a synchronization between parallel flows has a different meaning when compensating. Consider the sequential process $P \equiv A_1 \parallel B_1; A_2 \parallel B_2; Q$, and $P' \equiv A_1 \parallel B_1 \parallel A_2 \parallel B_2; Q$ with $\text{link}(A_1, A_2)$. It is clear that requiring $A_1 \prec_\alpha A_2$ does not make any execution $\alpha$ of $P$ a valid execution of $P'$: we also need $B_2 \prec_\alpha B_1$.

The following definitions formalize the notion of valid execution for a structured flow process with links.
**Definition 9.9** (Order of a saga with links). Let $S$ be a parallel saga and $L = \{\text{link}(A_i, A_j)| A_i, A_j \in A(S)\}$ be a set of links. The order $\prec_{S, L}$ is the least transitive and antisymmetric relation (if defined) satisfying: (i) $\prec_S \subseteq \prec_{S, L}$, and (ii) $\forall \text{link}(A_i, A_j) \in L, A_i \prec_{S, L} A_j$ and $A_j^{-1} \prec_{S, L} A_i^{-1}$.

Clearly, when $L$ introduces cycles in the control flow the order is not defined. The following definition singles out those executions that satisfy a set of well-defined links.

**Definition 9.10** (Valid execution with links). Let $S$ be a parallel saga and $L$ a set of links s.t. $\prec_{S, L}$ is defined. An order $\alpha$ is a valid execution of $S$ with links $L$ iff $\, \Gamma \vdash S \xrightarrow{\alpha'} \Box$ and $\alpha = \prec_{\alpha', L}$.

The definition above simply states that a valid execution of a process with links is an execution of the process that also satisfies the dependency constraints.

### 9.6 Implementation issues

The hierarchy of compensable flow composition languages presented in this chapter has been used in [102] as a starting point for the implementation of the API for *Java Transactional Web Services* (JTWS). JTWS is a Java library that allows for the definition of composed web services as compensable flows. The main feature of JTWS is that, instead of executing the workflow in a centralised way, it generates standard service wrappers that are responsible for coordinating the execution of the component services. The design of the wrappers for sequential and parallel composition was guided by the semantics presented here.
Chapter 10

Conclusions

In this thesis, we have investigated the fundamentals of models and languages for handling transactional aspects in global computing applications. Our research contributes to the three most representative approaches for dealing with transactions, namely:

- **Atomic transactions** (PART I): starting from the basic atomic multiway transactions of $zs$ nets, we progressively add value passing, reconfigurability, and dynamic features to define a framework for atomic multiway transactions with name mobility, respectively by introducing coloured $zs$ nets (Chapter 4), reconfigurable $zs$ nets (Chapter 5), and dynamic $zs$ nets (Chapter 6). We show that the $zs$ approach is orthogonal to all of these aspects, in the sense that it can be smoothly applied to the hierarchy proposed in [31]. Moreover, we prove that for coloured and reconfigurable $zs$ nets is still possible to construct the abstract view of the system (Sections 4.5 and 5.5). Additionally, these constructions are generalisations of the basic one, since, e.g., given a basic $zs$ net, it is possible to build the abstract view either as an ordinary place/transition net, as coloured net, or as reconfigurable net, and all views coincide (Sections 4.6 and 5.6). Differently, we show that for dynamic $zs$ nets, it is not always possible to build the corresponding abstract net (Section 6.3).

As an additional contribution, we give a novel characterization of deterministic processes for coloured and reconfigurable nets (Sections 4.2 and 5.2).

- **Conversion-based compensable transactions** (PART II): we define a language for a compensable, multiway, and nested transactions with mobility. The obtained language, named $cJOIN$, allows for the description of transactional services that are composed by following the interaction-based approach, i.e., each participant exhibits the behaviour that allows it to collaborate with other participants (Chapter 7). We give a suitable notion of serializability for interacting transactions, and we prove that the subcalculus of shallow processes (characterised by a syntactical restriction) is serializable (Section 7.4). We also prove that $zs$ nets are a special case of $cJOIN$ processes (Section 7.5).
Finally (Chapter 8), we implement the subcalculus of flat processes, defined in terms of a type system (Section 8.1), into Join (Section 8.4). The encoding is given only for processes that are written with few patterns, called canonical (Section 8.2). Nevertheless, canonical forms are enough to encode any flat process (Section 8.3). The implementation of flat cJoin in Join is then used to develope a prototype extension of the JoCAML language with transactions (Section 8.5).

- **Transactional long-running flows (PART III):** We provide a hierarchy of compensable languages for designing transactional long-running processes (Chapter 9). These languages were aimed at providing formal semantics to several technological standards for business processes. The main features of the approach are: (i) the fact that only the interaction among components is explained, while low-level computations are hidden, (ii) the definition of the languages is modular, in the sense that several extensions can be added incrementally. We start from a language with sequential composition (Section 9.2), then we add parallelism (Section 9.3) and nesting (Section 9.4). For each language, we prove that the defined semantics is adequate to the intended behaviour. Finally, we define several further extensions (Section 9.5).

The approaches presented in this thesis can be used, clearly, to describe systems requiring different flavours of transactions, but also to provide complementary views of the same system. The following are just a few possibilities that illustrate the way in which these models can be combined:

- At the very high-level of abstraction of a system (e.g., when gathering system requirements) it would be enough to describe the interaction of several components just as atomic interactions by using (coloured, reconfigurable or dynamic) zS nets. Differently, the actual implementations of such interactions may rely on compensations, and hence a more detailed representation (written in cJoin or as a compensable flow) could make explicit the user-programmed roll-backs.

- Usually business processes are represented as transactional flows that abstract away from the low-level computations performed by the component steps or activities. Since activities are assumed to be atomic, they are in fact real transactions. Therefore, any atomic step can be described either as a compensable flow, as a zS transaction, or as a cJoin program.

- Usually, transactional flows do not describe the coordination mechanisms involved in their execution. For instance, consider the case in which the parallel branches of a process that fails have to be aborted. In this case, a coordination mechanism informs all branches to stop and to compensate for. When such coordination mechanisms have to be made explicit, they can be described as cJoin programs.
Open problems and research lines. The work done in this thesis bears several lines for future work. In particular, and although not being strictly related with the main contributions of this thesis, the extension of the classical notion of deterministic processes to dynamic nets remains an interesting open problem. Such a construction may allow us to characterise processes equivalences for Join programs that take into account the truly concurrent nature of the CHAM model. Moreover, the impossibility of constructing the abstract view for dynamic ZS nets due to the input capability restriction, invites us to explore the extension of the hierarchy of nets proposed in [31] and of their associated process calculi.

As far as cJoin is concerned (PART II), probably the most appealing line of research is the study of more expressive variants in the lines of [9, 54, 65] that may take into account hostile environments, like those in which processes may crash and messages get lost. In particular, these extensions will require to change the implementation of transaction primitives by using a suitable variant of the distributed commit protocol.

On the other hand, since the hierarchy of compensable flow languages/calculi is proposed in a modular way (PART II), it would be interesting to consider advanced features appearing in some commercial proposals, such as selective compensations, imperative features like state or variables, control structures like branching and iteration, and data communication between activities. A step forward would be to explore how mobility can contribute to compensable flows. Another interesting question arises from the fact that flow composition languages are implemented in a centralized way, where a manager decides which activity executes next. Something that naturally comes up is whether compensable flows can be implemented as a distributed coordination process among involved parties.

As far as implementation issues is concerned, the encoding of flat cJoin in Join and the corresponding extension of JoCaml presented in this thesis appear as a promising approach to provide real programming languages with well-disciplined transaction primitives. In this respect, some lines of work could be the extension of our implementation to consider aspects like distributed nested negotiations, hostile environments, quality of service.
Bibliography


