Università degli Studi di Pisa
Dipartimento di Informatica

Dottorato di Ricerca in Informatica
Università di Pisa

Ph.D. Thesis: TD-1/99

Tile Logic for Synchronized Rewriting of Concurrent Systems

Roberto Bruni

March 1999

Addr: Corso Italia 40, 56125 Pisa, Italy
Tel: +39-050-887268 — Fax: +39-050-887226 — E-mail: bruni@di.unipi.it
http://www.di.unipi.it/~bruni

Thesis Supervisor:
Prof. Ugo Montanari

Abstract

Tile logic is a framework to reason about the dynamic evolution of concurrent systems in a modular way. It extends *rewriting logic* (in the unconditional case) by adding rewriting synchronization and side effects. This dissertation concerns both theoretical and implementation issues of tile models of computation where the mathematical structures representing *configurations* (i.e., system states) and *observations* (i.e., observable actions) rely on the same common auxiliary structure (e.g., for tupling, projecting, etc.).

We proceed by considering the simplest tile model (whose configurations and observations are just commutative monoids, and their sequential and parallel compositions coincide) and showing that it yields an original net model, called *zero-safe net*, which comes equipped with a primitive notion of transition synchronization, which is missing in ordinary Petri nets. The operational and abstract semantics of the model are expressed in the language of Net Theory and also as universal constructions in the language of Category Theory.

Then, two original versions of tile logic (called *process* and *term tile logic* respectively) are fully discussed, where *net-process-like* and usual *term* structures are employed for modelling configurations and observations. The categorical models for the two logics are introduced in terms of suitable classes of double categories. Then, the new model theory of *2EVH-categories* is proposed to relate the categorical models of tile logic and rewriting logic. This is particularly important, because rewriting logic is the semantic basis of several language implementation efforts (Cafe, ELAN, Maude), and therefore a conservative mapping of tile logic back into rewriting logic can be useful to get executable specifications of tile systems.

The theory required to relate the two logics is presented in *partial membership equational logic*, a recently developed framework, which is particularly suitable to deal with categorical structures. The comparison yields a correct rewriting implementation of tile logic that uses a meta-layer to make sure that only tile computations are performed. Exploiting the *reflective* capabilities of the Maude language, such meta-layer is specified in terms of *internal strategies*. Several detailed examples illustrating the implementation of tile systems for *concurrent process calculi* conclude the presentation.

To Daniela, my parents, and my friends. In the memory of my grandfather Pier Luigi.

A soul in tension that's learning to fly
Condition grounded but determined to try
Can't keep my eyes from the circling skies
Tongue-tied and twisted just an earth-bound misfit. I

— MOORE/EZRIN/GILMOUR/CARIN, Learning to Fly

Acknowledgements

The first person I wish to thank is, of course, my advisor Ugo Montanari, who influenced my research with many original ideas and contributions. It is not an overstatement to say that without his patience and guidance this thesis would not exist. Ugo, it has been an honour to work with you.

The other person to whom I am greatly indebted is José Meseguer. The months I spent in California visiting him at SRI have been very pleasant and have represented a nice professional opportunity. José taught me a lot, and I appreciate very much his friendship. Thank you Pepe, I hope you will find this thesis in a "good shape."

I would also invite the reader to join me in thanking the reviewers of this thesis, Narciso Martí-Oliet and Andrzej Tarlecki, for all the errors, misspellings, and technical mistakes that you won't find in the following more than three hundred pages (but for any error that you will discover the blame is just on me). Their accurate and valuable comments have improved a lot the presentation. Indeed, I have to thank Narciso twice, because of his helpful suggestions on an earlier version of this thesis. For the same reason, I am especially grateful to Gianluigi Ferrari and Paolo Baldan, with whom I have enjoyed many interesting discussions.

Many thanks to Fabio Gadducci for introducing me to the "world of tiles," and to Vladimiro Sassone for his help in some preliminary investigations on the notion of symmetric double categories at the end of 1996 and for his hospitality in Aarhus. I want to thank also my colleagues and all the people at the Computer Science Department in Pisa, and in particular (here the order is random) Chiara Bodei, Alessandra Raffaetà, Marco Pistore, and Stefano Bistarelli. I still have to thank the co-ordinator of the PhD courses in Pisa, Andrea Maggiolo-Schettini, and also Simone Martini, who followed my research progresses in the last two years.

I would like to thank Manuel Clavel for his help in the area of "internal strategies," and all the people at SRI, especially Pierangela Samarati, Sabrina De Capitani di Vimercati, Nina Mascardo, Alicia Siciliano, Grit Denker, Patrick Lincoln, Francisco Duran, and Livio Ricciulli. I am also grateful to Mogens Nielsen, to Glynn Winskel, and to Hélène and Claude Kirchner, who gave me the opportunity to present preliminary results of this thesis in Aarhus and in Pont-à-Mousson.

Before switching to Italian to thank the people closest to me, I want to thank Susan and Len (hi mom, hi pop), Letizia (nice car, Letty!), Rita, Lilia, Steve (if you read this, say hi to Pinky from me), Elizabeth, Betty and Phil, Les and Günther: They have helped me and Daniela in many ways when we were in California.

Ringrazio i miei genitori, Didi e Giampiero per avermi sempre aiutato e supportato: vi voglio bene e spero di riuscire sempre a darvi le soddisfazioni che meritate. Tengo a ringraziare anche i miei suoceri Renza e Pier Luigi, che mi hanno praticamente adottato, la mia "sorellina" acquisita Lucia ed il gatto Quasimodo che vive con i miei genitori e da quando mi sono sposato spadroneggia nella mia stanza. Grazie anche a tutti i miei parenti, tra cui tengo a commemorare mio nonno Pier Luigi a cui devo il mio interesse per le materie scientifiche, e la matematica in particolare.

Voglio ringraziare anche tutti i miei amici di Livorno, anche quelli con cui ultimamente ho perso un po' i contatti, per avermi aiutato a trascorrere il tempo libero. In particolare, tengo a salutare Roberto, che è una persona davvero speciale.

Per ultima ho volutamente lasciato Daniela, la mia compagna di vita. Sicuramente è la persona (assieme a me) a cui questa tesi ha procurato lo stress maggiore. Potrei ringraziarla per molte cose, come avermi aiutato più volte a correggere il mio inglese labronico, ma per fortuna posso farlo di persona.

Water, water, every where,
and all the boards did shrink;
water, water, every where,
nor any drop to drink.

— SAMUEL COLERIDGE, The Rime of the Ancient Mariner

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The huge growth of network-computing applications has characterized the latest years and will probably be a leading field of interest through the beginning of the new millennium. Faster and faster developments of tools and services supplied to public and private users call for new programming paradigms and formal methods that can help the analysis of *heterogeneous* and *reactive* systems.

In the literature, many mathematical formalisms have been proposed for the representation of collections of *agents* (also *processes*) that can evolve asynchronously, or interact, either competing for the use of some shared resources, or cooperating by exchanging some information, or taking some agreement on the next action to perform. In particular, such formalisms allow to focus exactly on some aspects of interest (e.g., causal dependencies, mobility, observable effects, logical properties, safeness, liveness, algebraic structure), abstracting from the real nature of the system under analysis (e.g., software architectures, network structures, industrial automation, real time and distributed systems). Each formalism has to provide both the *syntactic* and *semantic* description of systems, i.e., has to describe *what a distributed system is* and *how it behaves*.

We like to distinguish between two opposite styles of system specification, namely *monolithic* and *compositional*. In the first case, the system is viewed as an entity which cannot be decomposed, whose behaviour is specified by means of global rules. The second approach aims at describing a complex system as a *structured* composition of simpler entities, such that the global behaviour can be specified as a coordination of local evolutions of the subcomponents.

The compositional viewpoint looks more appealing to us because it provides a natural treatment of heterogeneous systems and does not require the control to be centralized. Moreover, compositional specifications promote the use of algebraic tools and can be extended by adding new constructors and by adopting more efficient implementations of components without having to rebuild the theory from scratch.

This dissertation is about the theoretical and implementation issues of a general, compositional formalism, called *tile logic* [69], for the modular description of asynchronous and synchronized reactive systems.

The more practical part concerns the implementation of a layer for the execution of tile specifications. The theoretical part provides an original categorical semantic setting for tiles, which can be used to prove the correctness of the implementation. Indeed, such categorical setting yields a semantic basis for the analysis of several useful extensions of the classical tile model (i.e., the one presented in [69]).

As a bright result, we define a methodology for the development (specification and execution) of reactive systems that is illustrated by several examples, mostly related to the field of *process algebras*. The great generality of such methodology is due to the following facts (that will be discussed throughout the thesis and that we propose here as a sort of *manifesto* for tile logic).

- Tiles are a natural model for reactive systems that are *compositional in space*: the behaviour of structured states can be defined as a coordination of the activities of the subcomponents.

- Tiles deal in the proper way with reactive systems that are *compositional in time*: they offer a powerful framework for modelling the composition of computations, where the observable actions are not necessarily uninterpreted actions.

- The structure of tiles is designed to take into account most of the distinctive features of distributed systems, as e.g., *concurrency*, *parallelism*, *distribution*, *mobility*, *synchronization*, and *coordination*.

- Tile logic allows us to reason about *open* systems (i.e., partially specified), rather than just *ground* systems, as it is the case for most *structural operational semantics* (SOS) approaches.

- Tile logic deals with algebraic structures on configurations that are different from the ordinary, tree-like presentation of terms (e.g., term graphs, partitions, preorders, relations, net processes). Analogously, it deals in a uniform way with more structured observations than basic uninterpreted actions.

- Tile logic has been already employed to model several different calculi and architectures, and there are many ongoing works in this direction. This permits to recast all such structures in a basic common framework, which is very useful for their extensive comparison, and for understanding differences and analogies.

- Tile logic yields an algebraic framework for computation proofs.

- Tile logic comes equipped with a natural notion of behavioural equivalence for open systems based on (*tile*) *bisimulation*.

- Tiles have nice logical (tiles as sequents and deduction as computation) and categorical (based on monoidal double categories) characterizations. This duality between logic and category has always played a foundational rôle for formal frameworks.
- Tiles are equipped with an appropriate *type system* (on interfaces) that can be used to characterize relevant properties of modelled systems (see e.g., [116]).
- Tiles admit a suggestive graphical presentation, that allows us to carry on proofs in tile logic by pasting rectangular diagrams.
- Several tile models can be implemented using existing languages based on rewriting logic, as e.g., Maude [34].
- The tile paradigm gives many new insights also in well-established formalisms, as illustrated in the first part of this thesis.
- *Different tile formats could allow the characterization of a general meta-theory for several interesting properties (e.g., "tile bisimulation is a congruence"), as it has been done for several SOS formats. Some preliminary result can be found in [69], but this aspect deserves further research.*
- *The introduction of higher-order features in the tile model could lead to the definition of a general type system for reactive calculi.*

1.1 Background: Formal Tools and Models

Petri Nets

Petri nets, introduced by Petri in [120] (see also [123]), are one of the most used and representative *models for concurrency*, because of the simple formal description of the net model, and because it allows the natural characterization of *concurrent* and *distributed systems*.

The distributed nature of Petri nets is fully exploited in the operational semantics. The states of a Petri net are multisets of *places*, called *markings*. Each place represents a different idealized class of resources, which is made available by the system. Each element of a marking is called *token* and can be interpreted as an instance of the corresponding resource (e.g., `printer` could be a class of resources, and each active printing device could be an instance in the current state). An elementary evolution is a *transition* t that rewrites a multiset of resources u_t to a multiset v_t. A transition can *fire* (i.e., be executed) in every state u with $u_t \preceq u$, leading to the state $v = u \ominus u_t \oplus v_t$, where $\_ \preceq \_$, $\_ \ominus \_$, and $\_ \oplus \_$ respectively denote multiset inclusion, difference and union. In this way, evolutions of a multiset are defined in terms of its subsets, and disjoint subsets of resources may *concurrently* evolve. This gives a compositional structure to the framework.

Structural Operational Semantics

The tile model is the result of a progressive exploration of mathematical structures allowing for finitary descriptions of complex, context-dependent *transition systems*. In Computer Science, (labelled) transition systems are one of the most used formalisms, for the operational understanding of computational systems. First, an abstract description of the system is defined, whose set of configurations (i.e., the feasible assignments to memory cells, registers, data structures, etc.) gives the set of *states* S of the transition system. Then, a *transition relation* $T \subseteq S \times S$ is defined, which represents the possible moves of the system. A set of *actions* (or *labels*) A is sometimes introduced to deal with suitable observational aspects: T becomes a ternary relation $T \subseteq S \times A \times S$, and so an external observer can discriminate different evolutions between the same pair of states.

In many cases, exploiting the structure of states, the relation T is inductively defined according to that structure. The well-known *structural operational semantics* approach (SOS) [121] is one of the most successful methodologies where the state structure is fully exploited. We are especially interested in SOS specifications for *process algebras* [5, 78, 111], where states are terms over the signature of processes — whose operators reflect the basic composition aspects of the system — and a set of *inference rules* (driven by the structure of the states) inductively defines the transition relation. Moreover, the resulting transition system provides a more abstract semantics based on bisimilarity of observable actions.

The expressiveness and properties of a variety of SOS rule formats have been investigated in several works [127, 11, 77, 7], with the common aim of providing syntactic constraints able to ensure certain useful properties (e.g., bisimulation is a congruence). Hence, these formats define some sort of "general correctness guidelines" to good operational semantics, in the sense that knowing that rules of a certain kind may have bad properties, we will try to replace them with equivalent ones in a good format. However, since SOS formats only deal with closed terms, such meta-theory cannot be applied to calculi with binding mechanisms (e.g., mobile and higher order process calculi). Moreover, the meta-theory is designed for transition systems labelled with uninterpreted actions, whereas it can be useful to have some mechanism for dealing with more informative actions (e.g., concurrent abstract semantics is often defined by decorating actions with causality links or with abstract locations and by introducing specialized versions of bisimulation [52, 46, 15, 85, 40, 117, 114]). The tile model offers a framework where the specification of calculi with name passing, causality and locality becomes uniform (see [61]).

Context systems [90], and *structured transition systems* (STS) [45, 58] also extend the SOS approach. In a context system, the transition relation is defined over *contexts* (that is, terms where free variables may occur) instead of just over *closed terms*, thus characterizing the behaviour of partially specified components of a system. In STS's, also transitions are equipped with an algebraic structure. Usually this is achieved by lifting the state structure to transitions and to computations.

Indeed, "programs" of many computational formalisms (including, among others, P/T Petri nets as described above, term rewriting systems, term graph rewriting [55, 41]) can be encoded as *heterogeneous graphs* having as collection of nodes algebras with respect to a suitable algebraic specification, and a poorer structure on arcs (e.g., just a set). Structured transition systems are defined instead as graphs having algebraic structure both on nodes and on arcs. A free construction associates with each program its induced structured transition system, from which a second free construction is used to generate the free model of computation, i.e., a structured category which lifts the algebraic structure to the transition sequences. This induces an equivalence relation on the computations of a system, which is shown to capture some basic properties of true concurrency. Moreover, since the construction of the free model is a left adjoint functor, it is compositional with respect to operations on programs expressible as colimits.

This approach was first applied by Meseguer and Montanari in the seminal work on algebraic models for concurrent semantics of Petri nets [106] (see also [49]). They showed that for each Petri net N, there exists a freely generated strictly symmetric, strict monoidal category whose arrows exactly represent the concurrent computations of N. Indeed, the functoriality axiom (of tensor product $\_\otimes\_$) expresses a basic fact about the *true concurrency* of the model, namely that any two concurrent actions $t_1 : u_1 \longrightarrow v_1$ and $t_2 : u_2 \longrightarrow v_2$ can occur in any order:[1]

$$(t_1 \otimes u_2);(v_1 \otimes t_2) = t_1 \otimes t_2 = (u_1 \otimes t_2);(t_1 \otimes v_2).$$

This is due to the fact that the multiset structure of states is naturally lifted to firings and to firing sequences, thus recovering also the notion of step and of step sequence. We will refer to this approach using the terminology "Petri nets are monoids."

Rewriting Logic

The introduction of *term rewriting* as an autonomous research field can be dated back to the early Seventies, when the *Knuth-Bendix completion procedure* [88] was presented. Such algorithm takes as input a signature Σ together with a set of axioms E on the terms of the signature, and returns a *convergent* rewriting system R. The idea is to transform the equations in E into one-way rewrite rules in such a way that R gives a unique possible result for any term, that is, its *normal form*, and that two terms are equated in the input specification if and only if they have the same normal form. Since then, term rewriting has been often considered as a technique to model an efficient form of equational deduction.

The seminal work on *rewriting logic* by Meseguer [99], shifted the attention from this *reduction* view (from terms to their unique normal form) to a more intriguing perspective, where term rewriting is fully exploited as a basic computational paradigm: terms are states of an abstract machine whose state-transforming function is given by the rewriting rules of the system.

[1] The operator $\_;\_$ represents sequential composition, and u_1, u_2, v_1, v_2 represents idle resources.

Moreover, a logic theory is associated to a rewriting system in such a way that each computation represents a *sequent* entailed by the theory (the *entailment* relation is specified by means of simple inference rules, accordingly to the term algebra under consideration). For example, rewriting rules (possibly containing free variables) can be freely contextualized and instantiated, and also sequentially composed, yielding new sequents in the logic. Derivations are then naturally provided with an algebraic structure by decorating the sequents with proof terms (simple inference rules define the schema for decorating the sequents). As an important result, decorated sequents can be axiomatized in such a way that deduction in the logic becomes equivalent to concurrent computing. In this sense, rewriting logic can be considered as a *logic of concurrent systems with state changes.*

Given this correspondence, a rewrite rule $t \Rightarrow t'$ has two readings: *computationally*, it means that when the system is in a state s, any instance of the pattern t in s can evolve to the corresponding instance of t', possibly in parallel with other changes; *logically*, it just means that we can derive the *formula* t' from t.

Notice that the notion of system state is entirely user-definable as an algebraic data type satisfying certain equational properties. Therefore, rewriting logic can be parametrized w.r.t. the underlying equational theory on state (e.g., one sorted, many sorted, order sorted, partial membership) and plays the rôle of a *semantic framework* where many different languages, systems and models of computation (e.g., labelled transition systems, grammars, Petri nets and algebraic nets, chemical abstract machine, concurrent objects, actors, graph rewriting, data flow, neural networks, real time systems, and many others) can be nicely expressed by natural encodings. For example, specifications of programming languages given in terms of rewrite theories become *de facto* interpreters for the languages in question. On the other hand, rewriting logic is a *logical framework*, where many other logics can be naturally represented (several examples can be found in [94, 95]).

Rewriting logic has also been used as a semantic framework for software architectures, providing a formal semantics for *architecture description languages* and their interoperation [110]. Other examples regard formal methods tools based on *inference systems* (e.g., *theorem provers*) that can be specified and prototyped in rewriting logic, and also the verification of *communication protocols*, including *secure* ones [53].

Notably several programming languages based on rewriting logic have been developed in different countries (e.g., Maude [35], ELAN [13], CafeObj [64]). This growing community has recently organized two workshops to discuss a great miscellany of different aspects of rewriting logic [103, 86].

Conditional Rewriting Logic

The semantics of process algebras finds natural presentations in the SOS format. Such representation yields a *conditional* rewriting system [94], whose rules can have the more general form:

$$t \Rightarrow t' \text{ if } s_1 \Rightarrow s_1' \wedge \cdots \wedge s_n \Rightarrow s_n' \ .$$

Unfortunately, the implementation of conditional rules increases the expressive power of rewrite theories as much as the complexity of the underlying rewrite machine. Indeed, conditional rules are not supported by languages based on rewriting logic because of efficiency reasons. Therefore, conditional specification must be partially adapted before becoming executable. Such modification can be pursued in an *ad hoc* fashion for each model, but a better approach obviously consists of finding a methodology able to automatically perform the translation for an entire class of specifications.

Tile logic extends rewriting logic along this direction. As already noticed, a basic assumption in rewriting logic is that rewriting rules can be applied in any context with their parameters (i.e., the free variables) being substituted by suitable terms. The basic assumption of tile logic is that *rewritings can be synchronized*. In particular, the free variables could be substituted only by terms verifying certain properties, and some rewriting could be forbidden in certain contexts. The side-effects of tile logic allow one to model such constraints in a natural way. In this sense, tile logic is a *logic of concurrent systems with coordination*.

Algebraic Structures

Since models of computation based on the notion of free and bound *names* are widespread, the notion of *name sharing* is essential for several applications ranging from logic programming, λ-calculus and functional programming to mobile processes (where local names may be communicated to the external world, thus becoming global names). We can think of names as links to communication channels, or to objects, or to locations, or to remote shared resources, or also to some *cause* in the event history of the system. Often, private names can be freely α-converted, because the only important information they offer is *sharing*.

An informal "wire and box notation" can give an intuitive understanding of the name sharing mechanism, and more generally of the rôle played by the *auxiliary* structure in the ordinary representation of terms. In the wire and box notation, variables are represented by wires and the operators of the signature are denoted by boxes labelled with the corresponding operation symbols. For instance, the term $f(x_1, g(x_2), h(x_1, a))$ over the signature

$$\Sigma = \{a : 0 \longrightarrow 1, b : 0 \longrightarrow 1, g : 1 \longrightarrow 1, h : 2 \longrightarrow 1, f : 3 \longrightarrow 1\}$$

and variables x_1, x_2 admits the graphical representation in Figure 1.1. Notice that wire duplications (e.g., of x_1) and wire swappings (e.g., of x_2 and a copy of x_1) are auxiliary, in the sense that they belong to *any* wire and box model, independently from the underlying signature.

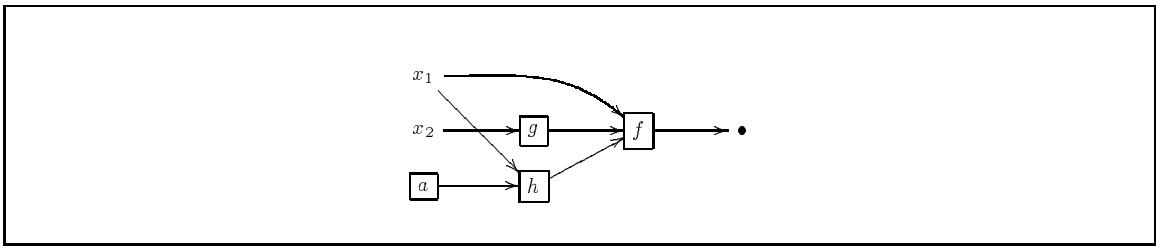


Figure 1.1: Example of wire and box notation.

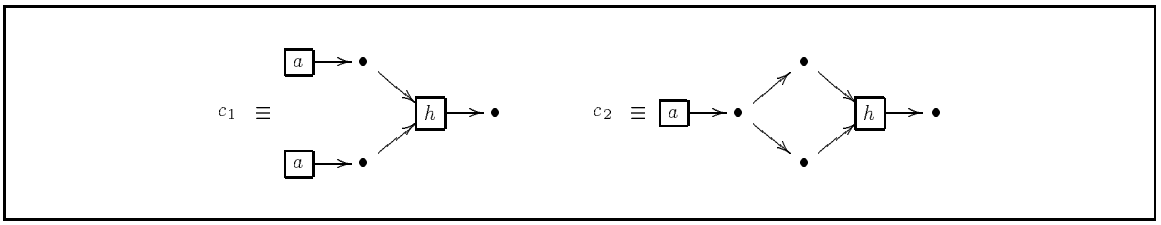


Figure 1.2: Example of explicit sharing.

The properties of the auxiliary structure are far from trivial and could lead to somehow misleading system representations if their interpretation is not well formalized. For example, let us consider the wire and box diagrams c_1 and c_2 in Figure 1.2. In a *value-oriented* interpretation, both c_1 and c_2 yield the same term $h(a, a)$. Instead, in a *reference-oriented* interpretation, c_1 and c_2 define different situations: in the former case the two arguments of h are uncorrelated, while in the latter case they point to the same *shared* location.

The difference between c_1 and c_2 becomes more evident if we add the rewriting rule $a \Rightarrow b$ for modelling the dynamics of our diagrams. Indeed, while we need two (concurrent) rewritings to transform c_1 into the representation of $h(b, b)$, only one rewriting is needed to apply a similar transformation to c_2. Eventually, suppose that we have two rewriting rules with side-effects for the constant a and one rule for the context $h(x_1, x_2)$ that coordinates the evolution of its arguments:

- $a \Rightarrow b$ with side-effect u,
- $a \Rightarrow b$ with side-effect v, and
- $h(x_1, x_2) \Rightarrow b$ provided that x_1 and x_2 are rewritten yielding side-effects u and v respectively.

It follows that c_1 can be rewritten to b, whereas c_2 cannot (at most one rule can be used to rewrite the shared subterm a, and therefore either the effect u or the effect v can be propagated, but not both).

Term graphs [55] are a reference-oriented generalization of the ordinary (value-oriented) notion of term, where the sharing of subterms can be specified also for closed (i.e., without variables) terms.[2]

[2] Terms can share variables, but shared subterms of a closed term can be freely copied, always yielding an equivalent term.

The distinction between terms and term graphs is made very precise by the axiomatization of *algebraic theories*: terms and term graphs differ for two axioms, representing, in a categorical setting, the *naturality* of transformations for copying and discharging arguments [42].

Many other mathematical structures have been proposed in the literature to express formalisms different from the ordinary tree-like presentation of terms. They range from the *flownomial calculus* of Stefanescu [32, 128], to the *bicategories of processes* of Walters et alii [80, 81], to the *pre-monoidal categories* of Power and Robinson [122], to the *action structures* of Milner [112], to the *interaction categories* of Abramsky [1], and to the *gs-monoidal categories* of Corradini and Gadducci [42, 41], to mention just a few (see also [31, 125, 79, 2, 43, 69, 59]). An element common to all these structures is the fact that they can be thought of as suitable enrichments of symmetric strict monoidal categories, which give the basis for the description of a distributed environment in terms of a wire and box diagram (the enrichment usually relies on the addition of categorical *transformations* lacking the naturality requirement).

In a recent joint work with Gadducci and Montanari [20] we proposed a schema for describing normal forms for this kind of structures, obtaining a framework where each structure finds its unique standard representation. The central notion is that of *distributed space*, which is a set of *assignments* over sets of variables. We distinguish between four different kinds of assignments, each representing a basic functionality of a generic distributed space, namely *input* and *output interfaces*, *basic modules*, and *connections*. Changing the constraints on the admissible connections (also called *links*) is the key to move from a formalism to another. We can then establish a sort of triangular correspondence between various formalisms presented in the literature (usually given in a set-theoretical way), different classes of distributed spaces and suitable enriched symmetric monoidal categories.

Very interestingly, tile logic can deal with different combinations of these structures for the description of states and observations. Moreover, the methodology that we will define for the implementation of tile specifications can be parametrized w.r.t. the choice of the underlying structures.

Tile Logic

The *tile model* [69] is a formalism for modular descriptions of the dynamic evolution of concurrent systems. The idea is that a set of rules defines the behaviour of certain *open configurations* (e.g., partially specified configurations) of the system, which may interact through their *interfaces*. Then, the behaviour of a whole system is defined as a coordinated evolution of its components. A tile has the form

$$\alpha : s \xrightarrow[b]{a} s'$$

and states that the *initial configuration* s of the system evolves to the *final configuration* s' producing an *effect* b, which can be observed by the rest of the system.

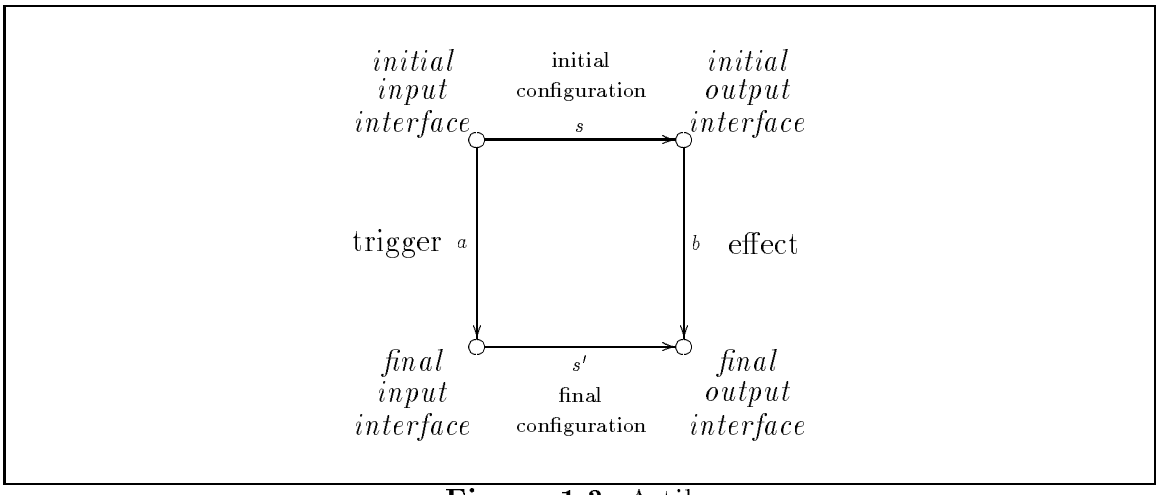


Figure 1.3: A tile.

However, such a step is allowed only if the subcomponents of s (which is in general an open configuration) also evolve (to the subcomponents of s') producing the *trigger* a. Both trigger and effect are called *observations* and model the interaction with the environment during a computation. Configurations are equipped with an *input* and an *output* interface.

For example, we can think of configurations as term contexts whose *typed holes* are represented by the input interface, and whose *type* is represented by the output interface. Analogously, observations are equipped with an *initial* and a *final* interface. Notice that the tile structure forces some interfaces to be identified (e.g., the initial interface of a coincides with the input interface of s). To better understand the interplay between configurations and observations, it is useful to visualize a tile as a two-dimensional structure (see Figure 1.3), from which the name "tile." In Figure 1.3 the horizontal dimension corresponds to the extension of the system, while the vertical dimension corresponds to the extension of the computation. Since the tile model is proposed as a model of computation for concurrent systems, we should also imagine a third dimension (the thickness of the tile), which models parallelism: configurations, observations, interfaces and tile themselves are all assumed to consist of several components in parallel. The initial configuration s of a tile α can also be denoted by $\mathtt{n}(\alpha)$, where $\mathtt{n}$ stands for "north," and likewise $\mathtt{s}(\alpha) = s'$, $\mathtt{w}(\alpha) = a$, $\mathtt{e}(\alpha) = b$ (standing for "south," "west," and "east" respectively).

The notions of *configuration* and *observation* are very general here, the only requirement is that they come equipped with operations of parallel and sequential composition[3] (represented by the infix operators $\_ \otimes \_$ and $\_;\_$ respectively). Tiles themselves possess three operations of composition (satisfying certain *exchange* axioms): parallel ($\_ \otimes \_$), horizontal ($\_ * \_$), and vertical composition ($\_ \cdot \_$). These operations are illustrated in Figure 1.4.

[3]Parallel composition is total, whilst sequential composition is defined only if the target of the first argument matches the source of the second argument.

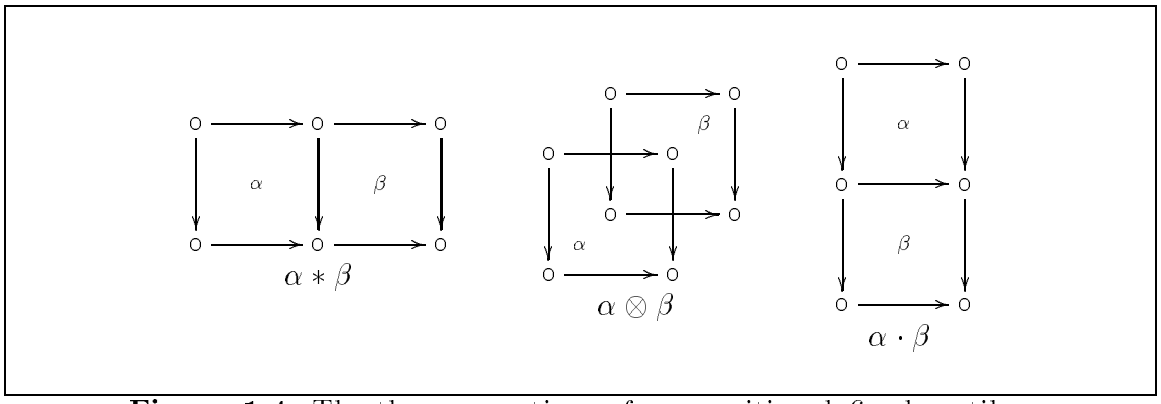


Figure 1.4: The three operations of composition defined on tiles.

The operation of parallel composition corresponds to building concurrent steps, where two (or more) disjoint configurations can concurrently evolve. Of course, the trigger (effect) of a concurrent step is the parallel composition of those triggers (effects) required for each component of the step.

Horizontal composition yields rewriting synchronization: the effect of the first tile provides the trigger for the second tile, and the resulting tile expresses the synchronized behaviour of both. Notice however that, if the trigger is an identity, then the synchronization is ephemeral, in the sense that the two tiles can be executed in any order yielding the same result (because of the exchange axioms).

Vertical composition models the execution of a sequence of steps starting from an initial configuration. It corresponds to sequential composition of computations.

It is evident that the tile model extends rewriting logic [99] (in the unconditional case), taking into account rewriting with side effects and rewriting synchronization. In fact, in unconditional rewriting systems, both triggers and effects are just identities; therefore rewriting steps may be freely applied, i.e., without interacting with the rest of the system and unconditional rewriting logic is embedded in the tile formalism as a special case.

By analogy with rewriting logic, the tile model also comes equipped with a purely logical presentation, where tiles are just considered as special (decorated) sequents subject to certain inference rules (see e.g., [69, 21]). Given a tile system, the associated *tile logic* is obtained by adding some *auxiliary* tiles (that can be necessary for allowing suitable rearrangements of the interfaces) and then freely composing in all possible ways (i.e., horizontally, vertically and in parallel) both auxiliary and basic tiles.

Moreover, tile models can be naturally equipped with observational equivalences (and congruences) based on bisimilarity: we can view a tile system as a labelled transition system whose states are the configurations, whose labels are pairs of the form "(trigger,effect)" and whose transition relation is given by the freely generated tiles. Hence, we obtain a notion of tile bisimulation that is defined also for open configurations.

Different mathematical structures have been employed to model configurations and observations, depending on the nature of the systems one wants to model. Basically, we can distinguish two approaches.

The first approach, proposed in [69], considers only models arising from internal constructions in suitable categories with structure. But in this case the auxiliary structure can be exploited only in one dimension. For example, within this class we find *zero-safe nets*, discussed in the first part of this thesis. Zero-safe nets define a tile model relying on the simplest possible interpretation of structured configurations and observations, namely Petri net markings. As an interesting result, horizontal composition yields a notion of *transition synchronization*, an important feature for compositionality, which is missing in ordinary nets (where only *token synchronization* is provided), and usually achieved through complex constructions.

Other examples consist of the monoidal tile system for finite CCS presented in [107], where discharged choices in the nondeterministic operator are managed with explicit garbaging, and the algebraic tile system for finite CCS [69], where configurations have the cartesian structure associated to the term algebra of processes, and free discharging of choices is allowed. Finally, we mention the simple coordination model based on graph rewriting and synchronization of [115], whose configurations, called *open graphs*, have an algebraic characterization as suitable gs-graphs (a structure with explicit subterm sharing and garbaging, offering a partial algebraic semantics for modelling graphs with subsets of sharable nodes), and allow recasting the hard computational problem of tile synchronization into a distributed version of constraint solving.

The second approach is presented in this thesis and takes into account a richer class of models, where both configurations and observations can have similar auxiliary structures. Rather than based on internal constructions, such models rely on the notion of *hypertransformation*, which is able to characterize the analogies between the mathematical structures employed in the two dimensions.

Process Algebras

Process algebras [5, 78, 111] offer a natural way to describe concurrent systems considered as structured entities (i.e., *agents*) interacting by message passing on common channels.

The basic idea is that each system is represented by a term of an algebra over a set of process constructors. New systems can then be constructed from existing ones, assuming that algebraic operators model basic features of concurrent systems.

Usually, the dynamic behaviour of agents is then specified by an operational semantics given in the SOS style, exploiting the structured nature of the framework. As already said, in general the translation of the operational semantics into a rewrite system yields conditional rules that are difficult to manage. Instead, tile logic is a very reasonable setting for modelling the behaviour of most process algebras, because it can naturally express the interaction between agents and "the rest of the world."

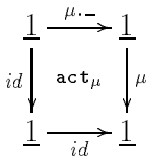


Figure 1.5: The tile for action prefix.

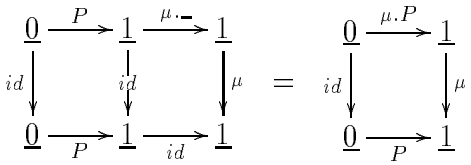


Figure 1.6: Tile proof of transition $\mu.P \xrightarrow{\mu} P$.

The advantage of modelling process algebras in tile logic (using the trigger/effect synchronization mechanism of rewritings) is evident just considering the usual *action prefix* operation, denoted by $\mu.\_$. When applied to a certain process P it returns a process $\mu.P$ that can perform the action μ and then behave like P. The corresponding tile $\mathtt{act}_\mu$ is represented in Figure 1.5 (horizontal arrows are process contexts, vertical arrows denote computations, and interfaces are underlined natural numbers denoting the number of components in the interface).

Notice that the horizontal operator $\mu.\_$ and the vertical operator μ are very different: the former represents the prefix context, which is a syntactic operator, and the latter denotes the execution of the observable action μ. The label id denotes trivial configurations and observations (i.e., identities). The tile $\mathtt{act}_\mu$ can be composed horizontally with the identity tile of any process P (representing an idle component) to model the step associated to the action prefix (see Figure 1.6).

Now, let nil be the inactive process, and consider the process $Q = \mu_1.\mu_2.nil$. If Q tries to execute the action μ_2 before executing μ_1 it gets stuck, because there is no tile having μ_2 as trigger and $\mu_1.\_$ as initial configuration (see Figure 1.7). Using ordinary rewrite rules, as $\mathtt{act}_\mu(p) : \mu.p \Rightarrow p$, where p is a process variable, wrong computations cannot be discarded automatically, because rewritings can be freely instantiated and contextualized. In our example, we can instantiate the variable p to nil and contextualize the rewriting in $\mu_1.\_$, obtaining $\mu_1.\mathtt{act}_{\mu_2}(nil) : \mu_1.\mu_2.nil \Rightarrow \mu_1.nil$, which is forbidden in the ordinary operational semantics. The correctness of the implementation must then rely either on some *type system* or on some *rewriting strategy* that avoids reduction to occur inside a prefix context.

This problem is well-known, and *ad hoc* solutions have been already proposed in [94, 133]. Our methodology offers a unifying approach for many related problems.

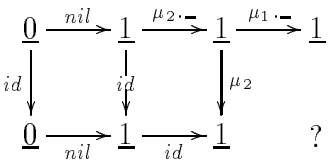


Figure 1.7: An incomplete tile proof.

Category Theory

The advantages of using *category theory* in computer science are well summarized in [73]. We believe that category theory with its abstraction, purity and elegance can provide the right guidance in the analysis and development of many concepts, properties and results.

In particular, we want to remark here the following aspects: (a) suitable classes of (structure-preserving) *functors* between categories (representing transition systems) offer an immediate definition of the *simulation morphism* between the underlying systems; (b) considering categories "in the small" (i.e., objects are states and arrows are computations), a commuting diagram may identify "computationally equivalent behaviours" also from a concurrent viewpoint; (c) considering categories "in the large" (i.e., objects are categorical models and arrows are simulation functors), isomorphisms may be used to characterize equivalent models; (d) *universal constructions* (i.e., *adjunctions*, *reflections*, *coreflections*, etc.) may be used to define a notion of *optimal* model; (e) *limits* and *colimits* often summarize useful compositions also from a model theoretic viewpoint (e.g., *products*, *sums*, *pushouts*, *pullbacks*, *initial and terminal objects*, just to cite a few).

Moreover, categories generalize transition systems in an obvious way: states are *objects* and transitions are *arrows* equipped with a partial composition operator $\_;\_$ (associative and with *identities*), corresponding to the intuitive sequential composition of transitions for expressing computations (identities represent idle components of the system).

The most archetypical example of categorical semantics is given by *Lawvere theories*. An *algebraic theory* [91, 92, 87] is just a cartesian category having underlined natural numbers as objects. The free algebraic theory associated to a (one-sorted) signature Σ, called the Lawvere theory for Σ and denoted by $\mathbf{Th}[\Sigma]$ (also $\mathcal{L}_\Sigma$), is defined as follows: the arrows from $\underline{m}$ to $\underline{n}$ are in one-to-one correspondence with n-tuples of terms of the free Σ-algebra with (at most) m variables, and composition is term substitution. In a certain sense, a Lawvere theory is just an alternative presentation of a signature, because the additional structure (for tupling, projecting and permuting the elements of a tuple) is generated in a completely free way: only the operators of the signature contain *information*, whereas the other constructors add nothing but auxiliary structure.

Lawvere theories introduce a very general notion of *model* (i.e., chosen functor from $\mathbf{Th}[\Sigma]$ to a cartesian category with chosen products $\mathcal{C}$) and *model morphism* (i.e., natural transformation between two models). This fact has been well-exploited in the categorical semantics of rewriting systems. In fact, it has been shown in [98] that a rewriting theory $\mathcal{R}$ yields a cartesian *2-category* $\mathcal{L}_{\mathcal{R}}$, which does for $\mathcal{R}$ what a Lawvere theory does for a signature (i.e., models can be defined as 2-product-preserving 2-functors).

Gadducci and Montanari pointed out in [67], that if also observations are to be taken into consideration during the rewriting process, then *double categories* [57, 82] should be considered as a natural model. A double category can be informally described as the superposition of a horizontal and a vertical category of *cells*, the former defining side-effect propagation, and the latter describing dynamic evolution. Then, in the same way as the term algebra is freely generated by a signature, and the initial model of rewriting logic is freely generated from the rules of the rewriting system, the basic tiles of a tile system freely generate a (monoidal) double category which constitutes the natural operational characterization[4] in the spirit of initial model semantics.

1.2 Main Contributions

The main aim of this dissertation consists in giving evidence of the foundational expressiveness of computational models based on tile logic.

In the FIRST PART, this is achieved by illustrating how the tile paradigm can suggest new insights in well-established research fields. We proceed by considering maybe the simplest tile model (whose configurations and observations are just commutative monoids, and their sequential and parallel compositions coincide) and showing that it yields a synchronization mechanism between *transitions* of *P/T Petri nets* [123]. This mechanism is very useful for compositionality issues, but it is missing in the classical net theory that provides only token synchronization. The new models, called *zero-safe nets*, offer a refined view of systems whose abstract counterparts consist of ordinary P/T nets.

Our approach is analyzed and compared under both the *collective* and the *individual token philosophies*, which correspond to the two different schools of thought for defining concurrent semantics of nets. According to the collective token philosophy, net semantics should not distinguish among different instances of the idealized resources (the so-called *tokens*) that rule the basics of net behaviour. The rationale for this is that any such instance is *operationally* equivalent to all the others. Obviously this approach disregards that operationally equivalent resources may have different origins and histories, and may, thus, carry different *causality* information.

[4] The tiles are cells, the configurations are arrows of the 1-horizontal category, the observations are the arrows of the vertical 1-category, and the interfaces are objects and model connections between the somehow syntactic horizontal category and the dynamic vertical evolution.

Selecting one instance of a resource rather than another, may be as different as being or not being causally dependent on some previous event. And this may well be information one is not ready to discard, which is the point of view of the individual token philosophy. The distinction between collective and individual token philosophy is discussed and clarified on the basis of the abstract representations of a simple zero-safe net that models a multicasting system.

Transition synchronization is shown to be very useful when using nets as a semantic foundation to interpret concurrent languages. We give evidence of this fact by illustrating the translation of simple process algebras. Moreover, our results can be expressed, using the *category theory* language, by means of *adjunctions* and *coreflections*, which ensure that suitable constructions for composing the refined models are preserved in the operational and abstract models.

Our feeling is that, by keeping the tile paradigm in mind, analogous investigations could be pursued for many other frameworks as, e.g., *control structures* [112], functional and logic programming, software architectures, and communication protocols.

The work presented in the SECOND and in the THIRD PART of this thesis aims at developing a general methodology for the implementation of several tile formats using languages based on rewriting logic. In the SECOND PART we consider two interesting cases of auxiliary structures shared between configurations and observations, introducing the notions of *process tile logic* and of *term tile logic*:

- The auxiliary tiles of process tile logic express consistent permutations of interfaces along the horizontal and vertical structures (configurations and observations define symmetric strict monoidal categories). *Flat* versions (e.g., any two sequents having the same "border" are identified, thus no emphasis is given upon the axiomatization of logic proofs) of process tile logic have been used for defining compositional models of computation of mobile calculi, and causal and located concurrent systems [59, 60].

- The auxiliary tiles of term tile logic allow consistent permutations of interfaces along the horizontal and vertical structures (as for process tile logic), consistent free copying, and consistent projections on subcomponents. Term tile logic represents the obvious extension of term rewriting logic (configurations and observations define suitable cartesian categories). Connections between the two logics are particularly interesting because in both logics the underlying cartesian structure manifests itself at the level of syntax, allowing the use of the standard term notation and the use of ordinary term substitution as arrow composition.

The two paradigms are illustrated by several examples on many different levels of complexity. For term tile logic we briefly discuss the expressiveness of the tile format in comparison with other widely used rule formats.

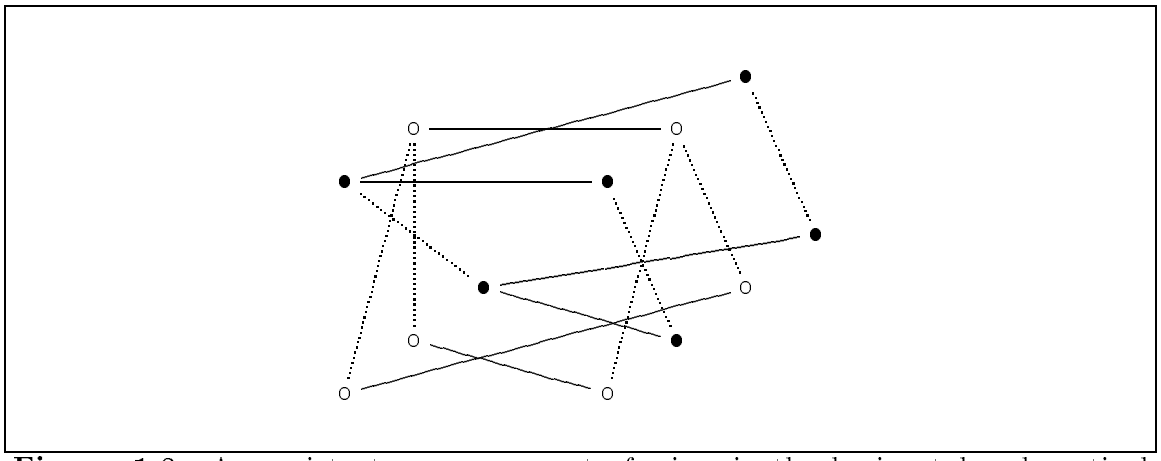


Figure 1.8: A consistent rearrangement of wires in the horizontal and vertical dimensions (vertical lines are dotted to improve the readability).

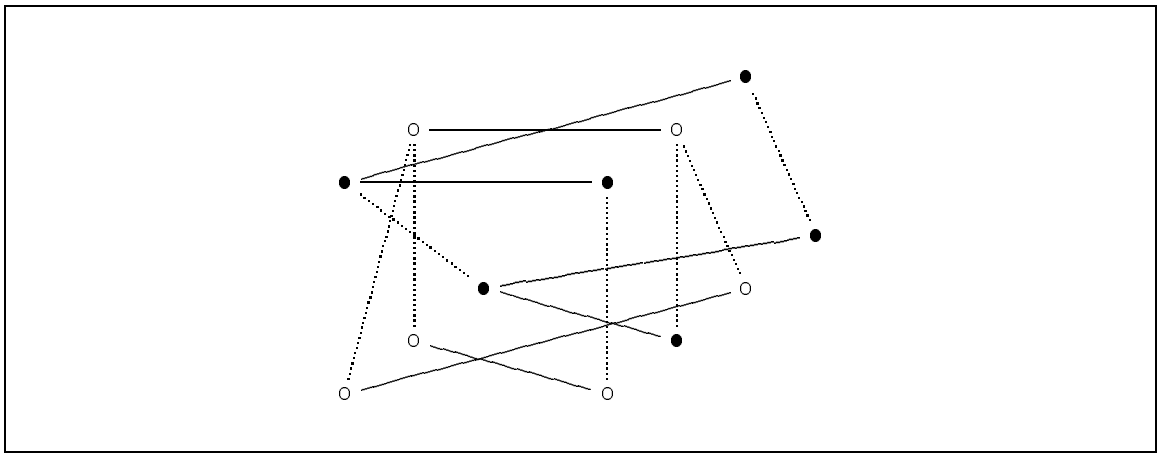


Figure 1.9: A non consistent rearrangement of wires.

A naïve illustration of the use of auxiliary tiles can be given in terms of the wire and box notation. We have seen that wire duplications and crossing of wires are auxiliary, in the sense that they belong to any wire and box model, independently from the underlying signature. It follows that, if we adopt the wire and box notation for both configurations and observations, then this kind of operations (e.g., rearrangements of wires) belongs to both dimensions (i.e., they are *shared*). Moreover, *consistent* rearrangements of wires on both dimensions do not change the meaning of a rule, but only its interface. We have the following naïve characterization of auxiliary tiles:

> *Auxiliary tiles are consistent rearrangements of interfaces in both dimensions, where consistency means that the rearrangement due to the initial configuration followed by the effect of the tile is equivalent to the rearrangement due to the trigger followed by the final configuration.*

In Figure 1.8 it is illustrated a consistent rearrangement, and in Figure 1.9 it is pictured a slightly different rearrangement that is not consistent. Indeed tile consistency can be easily verified as follows: given a tile let us say that two elements, one of the initial input interface, and the other of the final output interface, are *hv-connected* if they are connected by two wires attached to the same element of the initial output interface; similarly, let us say that they are *vh-connected* if they are connected by two wires attached to the same element of the final input interface; then the tile is consistent if it has the property that *two elements are hv-connected if and only if they are vh-connected.* It can be easily checked that this condition holds for the tile in Figure 1.8, but it does not hold for the tile in Figure 1.9 (e.g., notice that the first component $\bullet$ of the initial input interface is hv-connected to the first component $\circ$ of the final output interface, but they are not vh-connected).

Algebraic theories provide a clear mathematical representation of auxiliary constructors as suitable natural transformations, whose components are called *symmetries*, *duplicators*, and *dischargers*. This result will be very useful to relate our naïve definition with a more formal definition.

The semantics of process and term tile logic are given in terms of suitable classes of double categories whose equational axioms identify intuitively equivalent tile computations. For this purpose, we introduce the notions of *symmetric strict monoidal double category* and of *cartesian double category* ("with consistently chosen products"). As far as we know these definitions are new, because all the previous attempts (based on internal constructions) for analogous notions have led to asymmetric models, where the auxiliary structure (i.e., symmetries, duplicators, and dischargers) is fully exploited in one dimension only. Therefore we develop an alternative approach, following the idea of *hypertransformations* [57] for many-fold categories, and exploiting the results for double categories. In particular, we define the notion of *generalized transformation*, which acts in both dimensions, and asserts the coherence of the two ways of transforming the structure. Then, we instantiate the definition to the special cases of symmetries, duplicators, and dischargers, in a similar way as it happens for the 1-dimensional case. This approach motivates the following formal characterization of auxiliary tiles:

> *Auxiliary tiles for process and term tile logic are suitable generalized transformations respecting some coherence equations (where coherence means that they are uniquely defined).*

The comparison between tile logic and rewriting logic is carried out by embedding their corresponding categorical models in the recently developed framework of *partial membership equational logic* [104]. In doing so, we extend the result of [107], to deal with richer structures for configurations and observations. This task is accomplished by defining an extended version of 2-categories, called 2EVH-categories, which provides a systematic connection between models of tile logic and of rewriting logic.

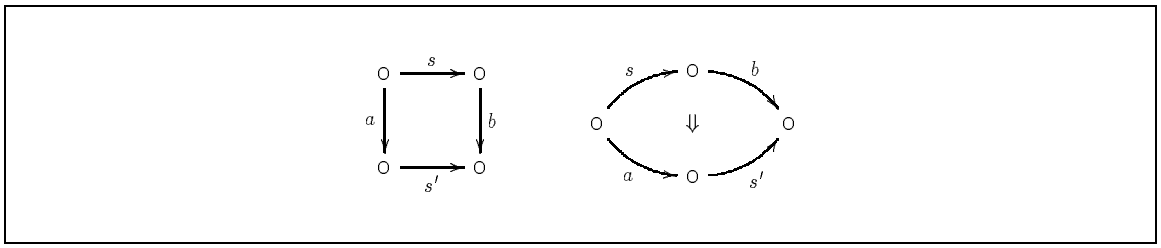


Figure 1.10: A tile (on the left) and the associated *stretched* rewrite rule.

The idea is to "stretch" double cells into ordinary 2-cells (see Figure 1.10), maintaining the capability to distinguish between configurations and effects, whereas the auxiliary structure becomes shared between the two classes.

2EVH-categories are able to simulate — in the sense that the algebraic structure of double categories is recoverable in terms of operations on 2-cells — the structure of double categories, where both the horizontal and vertical 1-categories *share some non-trivial structure* other than objects. In this translation we must be careful about two issues, namely, the possible identification of distinct double cells, and the possible existence of 2-cells having correct horizontal-vertical partition of the source and vertical-horizontal partition of the target, but which do not represent any double cell. Given that: (1) each arrow of a 2-category can be viewed as an identity 2-cell, (2) each auxiliary operator is a shared arrow, and (3) auxiliary tiles are consistent, it follows that 2EVH-categories allow for a third characterization of auxiliary tiles:

> *Auxiliary tiles coincide with the possible square-shaped decompositions of the identity 2-cells associated to auxiliary constructors.*

We will show that the three definitions of auxiliary tiles sketched in this introduction coincide.

In the THIRD PART we apply the theoretical results of the second part to define an executable framework for tile specifications using existing languages based on rewriting logic. To discard *wrong* computations, we need suitable meta-strategies that take control over the possible nondeterminism contained in a tile specification (and in its translation). To overcome this difficulty, we make use of the *reflective* capabilities [37, 38] of the rewriting logic language Maude [35] to define suitable *internal strategies* [39], which help the user to control the computation and to collect (some of) the possible (correct) results.

The key point is that the internal strategies defined here for simulating tile systems can also be thought of as general meta-strategies for nondeterministic rewriting systems equipped with a user-definable notion of success. We have experimented with Maude paradigmatic tile specifications of several CCS-like process calculi (namely finite CCS, full CCS and located CCS), and we have successfully developed and applied general internal strategies to analyze tile computations.

Moreover, Maude allows users to define specifications in a friendly algebraic framework without requiring any additional expertize in programming languages. Users can then improve the performance of their specifications simply by declaring their own *ad hoc* strategies.

In many other systems such strategies are often *external* to the language they control (e.g., they can be defined using a separate programming language external to the logic). One advantage of using Maude is that strategies can be made *internal* to the logic, i.e., they can be defined using rewrite rules. The logical basis of the system allows users to reason about their specifications and their strategies within the same logic.

The outcome of our theoretical and experimental results is a general methodology for providing an executable semantic framework for tile specifications of reactive systems. The general methodology for modelling reactive systems is summarized in the Conclusions.

1.3 Outline of the Thesis

In Chapter 2 we introduce some technical background that is essential to understand the notation and the constructions presented in the rest of the thesis. The presentation of more specific mathematical tools however is postponed to the "background" section of the chapter where they are required (e.g., this is the case of Petri nets, double categories and partial membership equational logic). More precisely:

- Section 2.1 introduces some elementary category theory (e.g., adjunctions, 2-categories, symmetric monoidal categories, cartesian categories), while we remark that the definition of double categories is postponed to Section 5.2.
- In Section 2.2 we recall the characterization of the auxiliary structure of terms via a progressive enrichment of suitable monoidal graphs generated by (hyper)signatures.
- In Section 2.3 we describe the logical and categorical semantics of rewriting systems, and in Section 2.4 we present the algebraic version of tile logic, proposed by Gadducci and Montanari [69].

Each of these sections makes use of concepts introduced in the previous ones.

The original contributions of the thesis are organized in three different parts. At the beginning of each part, a short abstract describes the contents and the main results. Although this introduces some redundancy in the presentation, the expected outcome is that each part can be read (almost) separately.

In the FIRST PART (ZERO-SAFE NETS), we introduce a net model equipped with a primitive notion of transition synchronization, and we discuss its operational and abstract semantics under both the collective and the individual token philosophy.

The FIRST PART is divided in two chapters.

- Chapter 3 introduces zero-safe nets and their operational and abstract semantics using the language of Net Theory. This chapter is almost self-contained; in particular, Section 3.2 provides an elementary introduction to the theory of Petri nets. Section 3.3 introduces zero-safe nets and their operational and abstract semantics.

  The simpler collective token approach is analyzed first, then the individual token approach is presented and compared with the collective one. The operational semantics for zero-safe nets in the collective token philosophy is defined in Section 3.4.1, while Section 3.4.2 concerns the related abstract semantics. In this context, the basic evolutions of zero-safe nets are equivalence classes of ordinary firing sequences induced by the diamond transformation, and are called *abstract stable transactions*.

  To illustrate the individual token version, Section 3.5.1 introduces the notion of *causal firing sequence*, which allows for a concise representation of concatenable processes, and has a suggestive implementation on a machine whose states are collections of token stacks. Then, the basic evolutions of a zero-safe net are the equivalence classes of causal firing sequences having isomorphic underlying processes.

  The translation of a simple process algebra into the zero-safe net model concludes the chapter.

- In Chapter 4, the language of category theory is used to give evidence that the definitions and the constructions presented in Chapter 3 are the best possible choices. In particular, many notions given in Section 2.1 are employed in this chapter (e.g., monoidal categories, adjunctions).

  After a brief survey of the *Petri nets are monoids* approach (given in Sections 4.2.1 and 4.2.2, for the collective and for the individual token philosophy respectively), the categorical characterization of the operational and abstract semantics of zero-safe nets under the collective (individual) token philosophy are presented in Sections 4.3.1 and 4.3.2 (Sections 4.4.1 and 4.4.2).

  The relationship between zero-safe nets and the tile model is discussed in Section 4.5. Related approaches to the refinement and the abstraction of nets are discussed at the end of the chapter.

In the SECOND PART (MAPPING TILE LOGIC INTO REWRITING LOGIC), we introduce two original versions of tile logic, called *process* and *term tile logic*, and define their categorical models by developing suitable notions of *symmetric monoidal* and *cartesian double category*. Then, we define a theory in partial membership equational logic presenting an extended version of the theory of 2-categories that is useful to relate the categorical models of tile logic and those of rewriting logic.

The SECOND PART is divided in two chapters.

- Chapter 5 introduces process and term tile logic, illustrates their use by several examples, and defines their categorical models. In this chapter, we make extensive use of concepts introduced in Chapter 2.

  In Section 5.2 we give the basics of double categories and we introduce some technical tools for the definition of symmetric monoidal and cartesian double categories: *generalized transformations* and *diagonal categories* (indeed in Sections 5.4.1 and 5.4.2 we incrementally enrich the basic monoidal structure of cells, first with *generalized symmetries* and then with *generalized dischargers* and *generalized duplicators*).

  In Sections 5.3.1 and 5.3.2 we introduce process and term tile logic. For each model we first present the flat version (Sections 5.3.1.1 and 5.3.2.1), then we define a suitable algebra of proofs (Sections 5.3.1.2 and 5.3.2.2), and eventually axiomatize the proof terms (Sections 5.3.1.3 and 5.3.2.3).

  In Section 5.4, we present the categorical models for process and term tile logic, formalizing the notion of *symmetric strict monoidal double categories* and of *cartesian double categories with chosen products*.

- Chapter 6 focuses on the relationships between categorical models of tile logic and of rewriting logic. To build a bridge from tiles to ordinary 2-cells we make use of a recent specification methodology, called *(one-kinded) partial membership equational logic* [104], which is summarized in Section 6.2.1.

  In Section 6.2.2 we recall the theory of 2VH-categories to give evidence that it is not suitable to deal with auxiliary structures shared between the horizontal and the vertical dimension.

  The extended model theory of 2EVH-categories is illustrated in Section 6.3, where also the monoidal, symmetric and cartesian extension are presented.

  Section 6.4 formalizes the notion of computad as model of rewrite theories that can be used to freely generate the associated categorical model.

  In Section 6.5 we define the logics associated to a rewrite theory by taking its interpretation either in double categories or in 2EVH-categories, and then we show the adequacy of the *extended* logic w.r.t. *tile* logic. Moreover it is shown that for the class of *uniform* systems, the rewriting strategies required in rewriting logic to perform controlled deductions in the extended logic become very simple.

In the THIRD PART (IMPLEMENTING TILE LOGIC), we present an executable framework for tile logic specifications, which is based on the logical language Maude, and discuss in detail the implementation of several CCS-like process algebras.

In particular, in Section 7.2.1 we present the reflective capabilities of rewriting logic and in Section 7.2.2 explain how they are supported by the language Maude. Then, Section 7.3 addresses the definition of internal strategies for non Church-Rosser systems, and Section 7.4 explains how they can be used to control the simulation of tile proofs. In Sections 7.5 and 7.6 we formalize the application of the meta-layer defined in Section 7.3 to the implementation of uniform tile systems for finite, full and located CCS.

In the CONCLUSIONS we summarize the main results and techniques used in the thesis, and sketch possible future lines of research.

At the end of the thesis four technical APPENDICES are included:

- Appendices A and B present the axiomatization of proof sequents for process and term tile logic.
- Appendix C explains Ehresmann's *hypertransformations* that motivated our definition of generalized transformations.
- Appendix D summarizes the syntax of the language Maude, which is extensively used in Chapters 6 and 7, respectively for the presentation of theories in partial membership equational logic (defined as *functional modules* in Maude), and for the implementation of tile specifications (defined as *system modules*).

1.4 Origins of the Chapters

Since many results of this thesis have been already presented at some conferences and some of them have been already published, we give here a list of pointers to their original sources, organized by topics:

- ZERO-SAFE NETS have been introduced in [28], where their semantics is discussed under the collective token philosophy. The individual token approach to zero-safe nets is presented in [29]. A detailed comparison between the two approaches, together with some interesting applications to concurrent language translations is given in [30]. Although not discussed in this thesis, we would like to mention also [26, 27], where the relationships between the *behavioural*, *categorical* and *logical* semantics of ordinary Petri nets are investigated and clarified under both the collective and the individual token philosophy.
- PROCESS AND TERM TILE LOGIC have been introduced in the Technical Report [21]. Their application to the specification of process calculi has been discussed in [22, 23, 24]. Specifically, the process tile system for CCS with locations and the term tile system for finite CCS are presented in [23], and the term tile system for full CCS is introduced in [24] (the term tile system for finite CCS is also sketched in [22] to illustrate the application of internal strategies for tile logic).

- Symmetric Monoidal and Cartesian Double Categories have been introduced in the Technical Report [21], and a journal version on this topic has been accepted for publication [25].

- Internal Strategies for Tile Logic have been introduced in the Technical Report [21], and published in [22]. They have been summarized and used in [23, 24].

- A Common Framework for Different Mathematical Structures has been illustrated in [20]. This work has been only sketched in this Introduction (see page 9) and it is not discussed in the rest of the thesis, but it could offer a basis for several expressive tile formats.

A presentation concerning most of the topics investigated in this thesis has been given at the 2nd Workshop on Rewriting Logic and its Applications (WRLA'98), Pont-à-Mousson, France [19].

Chapter 2

Background

Contents

In this chapter, we recall some basic notions that will be useful in the rest of the thesis. In Section 2.1 we introduce some elementary extracts of category theory (leaving the part related to double categories for Section 5.2). In Section 2.2 we recall the characterization of the auxiliary structure of terms via a progressive enrichment of suitable monoidal graphs. In Section 2.3 we describe the logical and categorical semantics of rewriting systems, and in Section 2.4 we present the algebraic version of tile logic, proposed by Gadducci and Montanari for dealing with SOS specifications that are more general than those in the DeSimone format [127], but less general than those in the positive GSOS format [11].

Some other preliminaries, e.g., on Petri nets, double categories, partial membership equational logic, are presented *by need* inside the background section of the proper chapter.

2.1 Category Theory

In this section we recall some basics of category theory that will be used in the rest of the thesis. For a comprehensive introduction to the subject we refer to [93, 3]. The reader already acquainted with the basic concepts of category theory can skip this section.

We start with a quotation from Goguen [73] that we found very appropriate.

> "To each species of mathematical structure, there corresponds a *category* whose objects have that structure, and whose arrows preserve it."

DEFINITION 2.1.1 (CATEGORY)
A category $\mathcal{C}$ *consists of a class* $O_{\mathcal{C}}$ *of* objects *(ranged over by* a*,* b*,...) and a class* $A_{\mathcal{C}}$ *of* arrows *(ranged over by* f*,* g*,...) with the following structure:*

- *Each arrow has a* domain $dom(f)$ *and a* codomain $cod(f)$ *(also called* source *and* target*) that are objects. We write*

$$f : a \longrightarrow b \quad \text{or also} \quad a \xrightarrow{f} b$$

  if a *is the domain of* f *and* b *is the codomain of* f*.*

- *Given two arrows* f *and* g *such that* $cod(f) = dom(g)$*, the* composition *of* f *and* g*, written* $f;g$*, is an arrow with domain* $dom(f)$ *and codomain* $cod(g)$*:*

$$a \xrightarrow{f} b \xrightarrow{g} c = a \xrightarrow{f;g} c.$$

- *The composition operator is associative, i.e.,*

$$f;(g;h) = (f;g);h$$

  (whenever f*,* g *and* h *can be composed).*

- *For every object a there is an* identity *arrow $id_a : a \longrightarrow a$, s.t. for any arrow f*

$$id_{dom(f)}; f = f = f; id_{cod(f)}.$$

To shorten the notation, identities are often denoted by the associated object. As usual, we write either $f \in A_{\mathcal{C}}$ or just $f \in \mathcal{C}$ to say that f is an arrow in $\mathcal{C}$, and similarly for objects. In Section 6.2.1 we will see an alternative formulation of the theory of categories making use of partial membership equational logic (Example 6.2.2).

EXAMPLE 2.1.1 (CATEGORY **Set**) *The* category of sets *(denoted by* **Set***) is the category whose objects are sets and whose arrows are functions, with the ordinary composition of functions as arrow composition and the identity function id_X as identity for each set X.*

EXAMPLE 2.1.2 (CATEGORY **Mon**) *The* category of monoids *(denoted by* **Mon***) is the category whose objects are monoids*[1] *and whose arrows are monoid homomorphisms.*

EXAMPLE 2.1.3 (PRODUCT CATEGORY) *Given a category $\mathcal{C}$, the product category $\mathcal{C} \times \mathcal{C}$ has as objects the pairs (a, b) of objects in $\mathcal{C}$, and as arrows the pairs $(f, g) : (a, b) \longrightarrow (c, d)$ for arrows $f : a \longrightarrow c$ and $g : c \longrightarrow d$ in $\mathcal{C}$ (composition is defined pairwise).*

Other examples are the *empty* category **0**, the category **1** with one object $\bullet$ and one arrow $id_\bullet$, and the category of *relations*. A category is called *discrete* if every arrow is the identity of some object. For example, **1** is a discrete category.

DEFINITION 2.1.2 (HOMSET)
The collection of arrows from an object a to an object b in $\mathcal{C}$ is called homset *and is denoted by $\mathcal{C}[a, b]$.*

DEFINITION 2.1.3 (SUBCATEGORY)
A category $\mathcal{A}$ is a subcategory *of a category $\mathcal{C}$, if*

- *$O_{\mathcal{A}} \subseteq O_{\mathcal{C}}$,*
- *for all $a, b \in O_{\mathcal{A}}$, then $\mathcal{A}[a, b] \subseteq \mathcal{C}[a, b]$,*
- *composition and identities in $\mathcal{A}$ coincide with those of $\mathcal{C}$.*

A subcategory is full *if $\mathcal{A}[a, b] = \mathcal{C}[a, b]$, for all $a, b \in O_{\mathcal{A}}$.*
A subcategory is lluf *if $O_{\mathcal{A}} = O_{\mathcal{C}}$*

[1] A *monoid* is a set X together with a binary operation which is associative and has a unit element e.

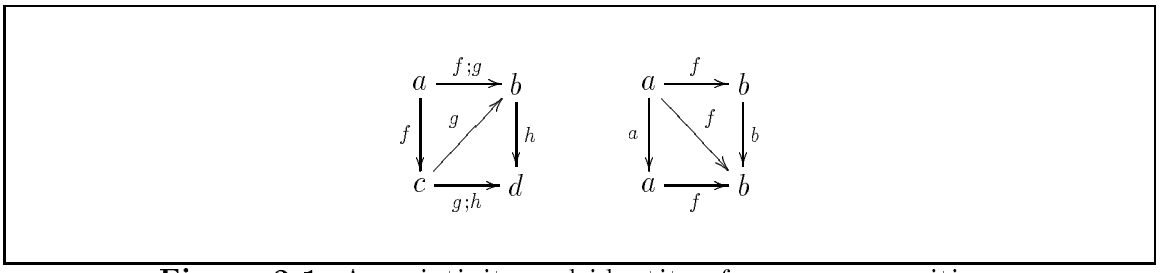


Figure 2.1: Associativity and identity of arrow composition.

In category theory, it is common practice to use *diagrams* made up of arrows and objects to express suitable properties of a category by saying that such diagrams *commute*. This is equivalent to saying that picking any two objects in the diagram and tracing any path of arrows from one object to the other, the composition of the arrows yields always the same result.

EXAMPLE 2.1.4 *The associativity of composition can equivalently be expressed by saying that the diagram on the left in Figure 2.1 commutes. Likewise, the identities are characterized by saying that the diagram on the right in Figure 2.1 commutes.*

DEFINITION 2.1.4 (INVERSE)
Suppose that $\mathcal{C}$ is a category and that a and b are two objects of $\mathcal{C}$. An arrow $f : a \longrightarrow b$ is said to be an isomorphism *if there is an arrow $g : b \longrightarrow a$ such that $f;g = a$ and $g;f = b$. In that case, we say that g is the* inverse *of f (and vice versa) and denote g by f^{-1} (it is immediate to prove that the inverse, if it exists, is unique).*

Given any category $\mathcal{C}$, we can always construct another category $\mathcal{C}^{op}$ by reversing all the arrows of $\mathcal{C}$.

DEFINITION 2.1.5 (DUAL OF A CATEGORY)
The dual *(or* opposite*) $\mathcal{C}^{op}$ of a category $\mathcal{C}$ is defined as follows:*

- *the objects and arrows of $\mathcal{C}^{op}$ are the objects and arrows of $\mathcal{C}$,*
- *if $f : a \longrightarrow b \in \mathcal{C}$, then $f : b \longrightarrow a \in \mathcal{C}^{op}$,*
- *if $f;g = h$ in $\mathcal{C}$, then $g;f = h$ in $\mathcal{C}^{op}$.*

To avoid confusion, if f is an arrow in $\mathcal{C}$, then we denote by f^{op} the corresponding arrow taken in $\mathcal{C}^{op}$.

2.1.1 Functors

Since categories themselves are mathematical entities, the quotation at the beginning of this section suggests that suitable morphisms between such entities can play an important rôle.

DEFINITION 2.1.6 (FUNCTOR)
Given two categories $\mathcal{A}$ and $\mathcal{B}$, a functor $F : \mathcal{A} \longrightarrow \mathcal{B}$ *consists of a pair of operations* (F_O, F_A) *such that:*

- $F_O : O_{\mathcal{A}} \longrightarrow O_{\mathcal{B}}$,
- $F_A : A_{\mathcal{A}} \longrightarrow A_{\mathcal{B}}$,
- *for each* $f : a \longrightarrow b \in \mathcal{A}$, *then* $F_A(f) : F_O(a) \longrightarrow F_O(b)$ *(but usually we omit subscripts and write* $F(f) : F(a) \longrightarrow F(b)$*),*
- *for each* $f : a \longrightarrow b, g : b \longrightarrow c \in \mathcal{A}$, *then* $F(f;g) = F(f);F(g)$,
- *for each object* $a \in \mathcal{A}$, *then* $F(id_a) = id_{F(a)}$.

EXAMPLE 2.1.5 (CATEGORY **Cat**) *Let $F : \mathcal{A} \longrightarrow \mathcal{B}$ and $G : \mathcal{B} \longrightarrow \mathcal{C}$ be two functors. We can define their composition $F;G : \mathcal{A} \longrightarrow \mathcal{C}$ as $F;G(\_) = G(F(\_))$, which is associative and has identities (the identity functors). This yields the category* **Cat** *whose objects are categories and whose arrows are functors.*

Given a functor $F : \mathcal{C}_1 \times \cdots \times \mathcal{C}_n \longrightarrow \mathcal{C}$ and a permutation

$$\phi : [1, ..., n] \longrightarrow [1, ..., n],$$

we will denote by $F(\_\phi(1), ..., \_\phi(n))$ the functor $F_\phi : \mathcal{C}_{\phi(1)} \times \cdots \times \mathcal{C}_{\phi(n)} \longrightarrow \mathcal{C}$, which is defined as $F_\phi(f_1, ..., f_n) = F(f_{\phi^{-1}(1)}, ..., f_{\phi^{-1}(n)})$, for any $f_i \in \mathcal{C}_{\phi(i)}$ for $i = 1, ..., n$. If ϕ is the identity we will omit the subscripts. For instance, let $F : \mathcal{C} \times \mathcal{C} \longrightarrow \mathcal{C}$ be a functor and let $X : \mathcal{C} \times \mathcal{C} \longrightarrow \mathcal{C} \times \mathcal{C}$ be the functor that swaps its arguments, i.e., such that for all $f, g \in \mathcal{C}$, $X(f, g) = (g, f)$, then $F(\_1, \_2) = F(\_, \_) = F$ and $F(\_2, \_1) = X; F$.

EXAMPLE 2.1.6 (HOM FUNCTOR) *Let $\mathcal{C}$ be locally small (in the sense that the collection of arrows between any two objects is a set). The* hom functor

$$\mathcal{C}[\_, \_] : \mathcal{C}^{op} \times \mathcal{C} \longrightarrow \mathbf{Set}$$

assigns to each pair of objects (a, b) the set $\mathcal{C}[a, b]$, and to each pair of arrows $(f : a' \longrightarrow a, g : b \longrightarrow b')$ (with $f, g \in \mathcal{C}$) the function $\mathcal{C}[f, g] : \mathcal{C}[a, b] \longrightarrow \mathcal{C}[a', b']$ sending each arrow $h : a \longrightarrow b$ into the arrow $f; h; g : a' \longrightarrow b'$.

DEFINITION 2.1.7 (PROPERTIES OF A FUNCTOR)
Any functor $F : \mathcal{A} \longrightarrow \mathcal{B}$ induces a set mapping $\mathcal{A}[a, b] \rightarrow \mathcal{B}[F(a), F(b)]$.

- *We say that F is* full *if the induced mapping is surjective for every homset.*
- *We say that F is* faithful *if the induced mapping is injective on every homset.*

Moreover, we say that F preserves *a property P of arrows if whenever f has property P, so does $F(f)$. We say that F* reflects *a property P of arrows if whenever $F(f)$ has property P, so does f (for any arrow that F maps to $F(f)$).*

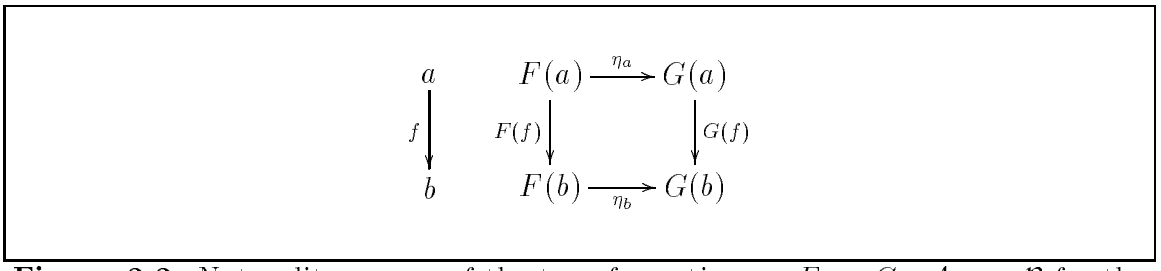


Figure 2.2: Naturality square of the transformation $\eta : F \Rightarrow G : \mathcal{A} \longrightarrow \mathcal{B}$ for the arrow $f \in \mathcal{A}$.

2.1.2 Natural Transformations

Given two categories $\mathcal{A}$ and $\mathcal{C}$, also their homset in **Cat** comes equipped with a category structure, i.e., $\mathbf{Cat}[\mathcal{A},\mathcal{C}]$ (also denoted by $\mathcal{C}^{\mathcal{A}}$) is a category whose objects are functors and whose arrows are called *natural transformations.*

Definition 2.1.8 (Natural transformation)
A natural transformation *between two functors $F, G : \mathcal{A} \longrightarrow \mathcal{B}$ consists of a family*

$$\eta = \{\eta_a : F(a) \longrightarrow G(a)\}_{a \in O_{\mathcal{A}}}$$

of arrows in $\mathcal{B}$ indexed by the objects of $\mathcal{A}$, such that the diagram in Figure 2.2 commutes for every arrow $f : a \longrightarrow b \in \mathcal{A}$, expressing the naturality *of the transformation η. We call η_a the component at a of the natural transformation η.*

We use either the notation $\eta : F \Rightarrow G : \mathcal{A} \longrightarrow \mathcal{B}$, or the notation $\eta : F \Rightarrow G$ to say that η is a transformation from F to G.

Two natural transformations $\eta : F \Rightarrow G$ and $\nu : G \Rightarrow H$ can be composed by elementwise composing their components, obtaining the natural transformation $\eta \circ \nu : F \Rightarrow H$, with $(\eta \circ \nu)_a = \eta_a ; \nu_a$. Moreover, for each functor $F : \mathcal{A} \longrightarrow \mathcal{B}$ there exists the obvious identity transformation $1_F : F \Rightarrow F$ given by $1_F = \{id_{F(a)}\}_{a \in O_{\mathcal{A}}}$.

2.1.3 Symmetric and Monoidal Categories

Definition 2.1.9 (Symmetric, strict monoidal category [93])
A strict monoidal category, *is a triple $(\mathcal{C}, \_ \otimes \_, e)$ where:*

- *$\mathcal{C}$ is the underlying category (with composition $\_ ; \_$ and with identity id_a for each object a);*
- *$\_ \otimes \_ : \mathcal{C} \times \mathcal{C} \longrightarrow \mathcal{C}$ is a functor called the* tensor product*;*
- *e is an object of $\mathcal{C}$ called the* unit object*;*
- *the tensor product $\_ \otimes \_$ is associative on both objects and arrows and e is the unit for $\_ \otimes \_$.*

A symmetric, strict monoidal category *(ssmc) is a 4-tuple* $(\mathcal{C}, \otimes, e, \gamma)$ *where:*

- $(\mathcal{C}, \otimes, e)$ *is a strict monoidal category,*
- $\gamma : \_1 \otimes \_2 \Rightarrow \_2 \otimes \_1$ *is a natural isomorphism called* symmetry *satisfying the Kelly-MacLane* coherence axioms *expressed by the following equations:*

$$\gamma_{x\otimes y,z} = (id_x \otimes \gamma_{y,z}); (\gamma_{x,z} \otimes id_y) \quad (2.1)$$

$$\gamma_{x,y}; \gamma_{y,x} = id_{x\otimes y} \quad (2.2)$$

REMARK 2.1.7 *Translating the more general notion of symmetric monoidal category into the special case of symmetric* strict *monoidal category it could seem that also the axiom* $\gamma_{e,x} = id_x$ *should be stated. However it is immediate to show that the others are sufficient to guarantee this constraint. In fact, since* $e \otimes e = e$*, it follows that* $\gamma_{e,x} = \gamma_{e\otimes e,x} = (id_e \otimes \gamma_{e,x}); (\gamma_{e,x} \otimes id_e) = \gamma_{e,x}; \gamma_{e,x}$*. Thus, by composing the leftmost and the rightmost expressions with* $\gamma_{x,e}$*, we obtain* $id_{e\otimes x} = \gamma_{e,x}$*. Finally, recalling that* $e \otimes x = x$ *we can conclude that* $id_x = \gamma_{e,x}$*.*

A *symmetry* in a ssmc is any arrow obtained as composition and tensor of *identities* and *components of* γ. We denote by $Sym_{\mathcal{C}}$ the subcategory of the symmetries of $\mathcal{C}$. If γ is the identity natural transformation (i.e., the tensor product is commutative) then the category is called *strictly symmetric, strict monoidal.*

DEFINITION 2.1.10 (CATEGORIES **SSMC** AND **SSMC**$^{\oplus}$)
Let $\mathcal{C}$ *and* $\mathcal{C}'$ *be two symmetric, strict monoidal categories. A* symmetric strict monoidal functor *from* $\mathcal{C}$ *to* $\mathcal{C}'$ *is a functor* $F : \mathcal{C} \longrightarrow \mathcal{C}'$ *such that:*

$$F(e) = e' \quad (2.3)$$

$$F(\_1 \otimes \_2) = F(\_1) \otimes' F(\_2) \quad (2.4)$$

$$F(\gamma_{x,y}) = \gamma'_{Fx,Fy} \quad (2.5)$$

Let **SSMC** *be the category of symmetric strict monoidal categories (as objects) and symmetric strict monoidal functors (as arrows). We denote by* **SSMC**$^{\oplus}$ *the full subcategory consisting of the monoidal categories whose objects form* free *commutative monoids.*

2.1.4 Cartesian Categories

In Category Theory, several notions are often stated by means of *universal properties*, that is, by stating the existence of a *unique* arrow that verifies certain properties.

DEFINITION 2.1.11 (TERMINAL OBJECT)
Let $\mathcal{C}$ *be a category. We say that an object* t *of* $\mathcal{C}$ *is* terminal *if for any object* c *of* $\mathcal{C}$ *there is exactly one arrow* $!_c$ *from* c *to* t*.*

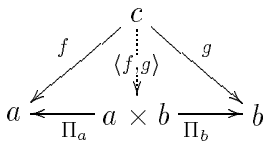


Figure 2.3: The diagram for products.

Definition 2.1.12 (Product)
We say that $\mathcal{C}$ has binary products *if for any pair of objects $a, b \in \mathcal{C}$ there exists an object u together with two* projections *$\Pi_a : u \longrightarrow a$ and $\Pi_b : u \longrightarrow b$ satisfying the following condition:*

> *for each object c and arrows $f : c \longrightarrow a$ and $g : c \longrightarrow b$ in $\mathcal{C}$ there exists a unique arrow $q : c \longrightarrow u$ such that $f = q;\Pi_a$ and $g = q;\Pi_b$, i.e., there exists a unique arrow $q : c \longrightarrow u$ (also denoted by $\langle f, g\rangle$) such that the diagram in Figure 2.3 commutes (the dotted arrow associated to q expresses the universal property of q, i.e., that it exists and is unique).*

The product (u, Π_a, Π_b) of two objects a and b is often denoted by $a \times b$ (but notice that it is unique only up to isomorphism, and that it is uniquely determined only specifying also its projections).

The category $\mathcal{C}$ has *canonical binary products* (also called *chosen binary products*) if a specific product diagram is given for each pair of objects.

Definition 2.1.13 (Cartesian Category)
The category $\mathcal{C}$ is cartesian *if it has a terminal object and all (binary) products.*

2.1.5 Limits and Colimits

The dual notions of terminal object and products are called *initial object* and *coproducts* (also *sums*). These and other useful definitions (e.g., pullbacks, pushouts, equalizers, coequalizers, ...) that often represent useful constructions (e.g., in model categories), are all instances of the more general concept of *limit* (and of the dual notion of *colimit*).

Definition 2.1.14 (Cone)
Let $\mathcal{G}$ be a graph and $\mathcal{C}$ be a category. Let $D : \mathcal{G} \longrightarrow \mathcal{C}$ be a diagram in $\mathcal{C}$ with shape $\mathcal{G}$. A cone *with base D is an object c of $\mathcal{C}$ together with a family $\{p_a\}_{a \in \mathcal{G}}$ of arrows of $\mathcal{C}$ indexed by the nodes of $\mathcal{G}$, such that $p_a : c \longrightarrow D(a)$ for each node a of $\mathcal{G}$. The arrow p_a is the* component *of the cone at a. We denote the cone by writing $p : c \longrightarrow D$.*

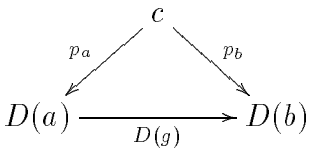


Figure 2.4: Commutative cone.

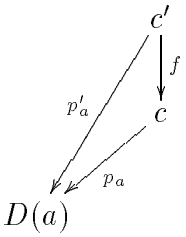


Figure 2.5: An arrow f from the cone p' to p.

A cone is called *commutative* if for any arrow $g : a \longrightarrow b \in \mathcal{G}$, the diagram in Figure 2.4 commutes.[2] If $p' : c' \longrightarrow D$ and $p : c \longrightarrow D$ are cones, an *arrow* from the first to the second is an arrow $f : c' \longrightarrow c$ such that for each node $a \in \mathcal{G}$, the diagram in Figure 2.5 commutes.

DEFINITION 2.1.15 (LIMIT)
A commutative cone over the diagram D is called universal *if every commutative cone over the same diagram has a unique arrow to it. A universal cone, if such exists, is called a* limit *of the diagram D.*

EXAMPLE 2.1.8 *A limit for the empty diagram is then an object t such that for any other object c there is exactly one arrow from c to t, i.e., it is a terminal object. Dually, the* initial *object is the colimit of the empty diagram.*

EXAMPLE 2.1.9 *A limit of the diagram (c_a, c_b) (i.e., a pair of objects), which is associated to the* discrete *graph[3] with only two nodes a and b, is an object u, together with two arrows $p_1 : u \longrightarrow c_a$ and $p_2 : u \longrightarrow c_b$, such that for any other cone $(u', p'_1 : u' \longrightarrow c_a, p'_2 : u' \longrightarrow c_b)$ there exists a unique arrow $h : u' \longrightarrow u$ with $p'_1 = h; p_1$ and $p'_2 = h; p_2$. But this is just the definition of* product *of c_a and c_b. Dually, the colimit of the diagram (c_a, c_b) defines the* coproduct *of c_a and c_b (if it exists).*

[2] We remark that the diagram D is not assumed to commute.
[3] A graph is called *discrete* if it has no arrows.

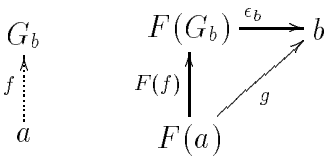


Figure 2.6: The left adjoint F.

DEFINITION 2.1.16 (PULLBACK AND PUSHOUT)
Let $\mathcal{G}$ be the graph $\bullet \longrightarrow \bullet \longleftarrow \bullet$. A diagram of this type in a category $\mathcal{C}$ is given by three objects a, b and c, and two morphisms $f : a \longrightarrow c$ and $g : b \longrightarrow c$. A cone for such diagram is an object d together with three arrows $h_1 : d \longrightarrow a$, $h_2 : d \longrightarrow b$ and $h_3 : d \longrightarrow c$, such that $h_3 = h_1; f$ and $h_3 = h_2; g$. Hence, $h_1; f = h_2; g$ and the cone is equivalently expressed by (d, h_1, h_2), because h_3 is uniquely determined by h_1 and by h_2. A limit for this diagram (if it exists) is called pullback *of f and g.*

The dual notion is called pushout. *It can be defined as the colimit for the diagram $(a, b, c, f : c \longrightarrow a, f : c \longrightarrow b)$, which is associated to the graph $\bullet \longleftarrow \bullet \longrightarrow \bullet$.*

2.1.6 Adjunctions

Adjunctions are one of the categorical tools that will be used more extensively in the rest of the thesis (to characterize our constructions by means of *universal properties*). There are several equivalent definitions of adjunction. We think that the more "constructive" one relies on the scenario consisting of two categories $\mathcal{A}$ and $\mathcal{B}$ and a functor $F : \mathcal{A} \longrightarrow \mathcal{B}$, where given an object b in $\mathcal{B}$ we want to find the object in the image of $\mathcal{A}$ through F that better approximates b, where *approximation* means that there exists an arrow from $F(a)$ to b, and "better" refers to the other approximations in $F(\mathcal{A})$, not to all the objects in $\mathcal{B}$. In this sense, *better than the others* means that any other arrow $f : F(c) \longrightarrow b$ factorizes through the better approximation of b via the image of a morphism in $\mathcal{A}$ in a unique way.

DEFINITION 2.1.17 (ADJUNCTION)
Given two categories $\mathcal{A}$ and $\mathcal{B}$ and a functor $F : \mathcal{A} \longrightarrow \mathcal{B}$, we say that F is a left adjoint *if for each object b in $\mathcal{B}$ there exists an object G_b in $\mathcal{A}$ and an arrow $\epsilon_b : F(G_b) \longrightarrow b$ in $\mathcal{B}$ such that for any object $a \in \mathcal{A}$ and for any arrow $g : F(a) \longrightarrow b$ there exists a* unique *arrow $f : a \longrightarrow G_b \in \mathcal{A}$ such that $g = F(f); \epsilon_b$ (see Figure 2.6).*

An immediate consequence of this fact is the existence of a functor from $\mathcal{B}$ to $\mathcal{A}$ that maps each object b into its approximation G_b, i.e., G can be extended to a functor. To prove this point, one can notice that given an arrow $f : b \longrightarrow b'$ then the arrow $\epsilon_b; f : F(G_b) \longrightarrow b'$ factorizes through $\epsilon_{b'}$ and the image of a unique arrow h from G_b to $G_{b'}$. Therefore the functor G can be defined by assuming $G(f) = h$ (see Figure 2.7). The functor G is called the *right adjoint* of F, and we write $F \dashv G$.

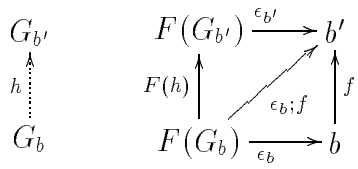


Figure 2.7: The definition of the right adjoint of F.

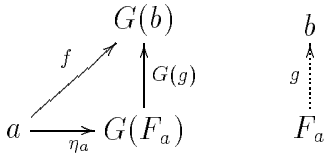


Figure 2.8: The right adjoint G.

Adjoints are unique up to isomorphism. The collection $\epsilon = \{\epsilon_b\}_{b \in \mathcal{B}}$ is called the *counit* of the adjunction and defines a natural transformation from $G;F$ to $1_{\mathcal{B}}$.

REMARK 2.1.10 *We could have employed an equivalent* dual *approach to the definition of adjoints, by starting with G and defining the "least upper" approximation F_a for each object a. This construction yields the* unit $\eta = \{\eta_a : a \longrightarrow G(F_a)\}_{a \in \mathcal{A}}$ *of the adjunction (see Figure 2.8).*

Another equivalent definition of adjunction can be given by saying that given two functors $F : \mathcal{A} \longrightarrow \mathcal{B}$ and $G : \mathcal{B} \longrightarrow \mathcal{A}$, F is left adjoint to G (and G is right adjoint to F) if there is a natural bijection

$$\mathcal{B}[F(a), b] \xrightarrow{\varphi_{a,b}} \mathcal{A}[a, G(b)]$$

for each pair of objects $a \in \mathcal{A}$ and $b \in \mathcal{B}$.

An important property of adjunctions is the preservation of universal constructions: left adjoints preserve colimits, and right adjoints preserve limits.

A typical example of adjunction consists of a *forgetful functor* $U : \mathcal{S} \longrightarrow \mathcal{C}$, from a category $\mathcal{S}$ of certain *structures* and *structure-preserving* functions (e.g., **Mon**) to a category $\mathcal{C}$ with *less* structure (e.g., **Set**). Forgetful functors have often a left adjoint that adds the structure missing in $\mathcal{C}$ by means of a *free construction*.

REMARK 2.1.11 *We will not give a formal definition of forgetful functor. It is reasonable to expect any forgetful functor U to be faithful and that if f is an isomorphism and $U(f)$ is an identity arrow, then also f is an identity arrow. We refer to [63] (pages 9–19) for a formal definition of forgetful functor.*

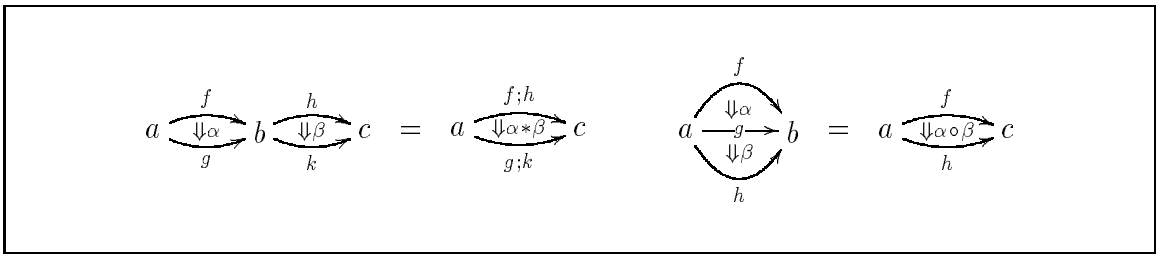


Figure 2.9: Horizontal and vertical composition of 2-cells.

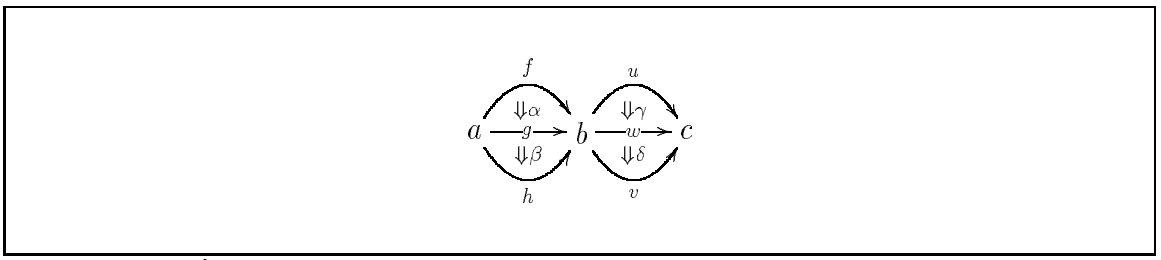


Figure 2.10: Composition of 2-cells for the exchange law.

Reflection and *coreflection* are two particularly important kinds of adjunction, where respectively the counit and the unit define natural isomorphisms, yielding optimal approximations.

2.1.7 2-categories

2-categories are a special case of *enriched categories*, namely the categories enriched over **Cat**. Basically, the idea is that given a monoidal category $\mathbf{V}$, a category enriched over $\mathbf{V}$ is just a category such that, for any two objects a and b, the homset $\mathcal{C}[a,b]$ is an object of $\mathbf{V}$; moreover, this hom-objects satisfy suitable coherence axioms. Therefore, a **Cat**-enriched category is a category such that each homset is a category. We adapt the following from the classical reference for 2-categories [82].

DEFINITION 2.1.18 (2-CATEGORY)
A 2-category!definition $\mathcal{C}$ *consists of a collection* $a, b, c, \ldots$ *of* objects, *or 0-cells, a collection* $f, g, h, \ldots$ *of* morphisms, *or 1-cells, and a collection* $\alpha, \beta, \gamma, \ldots$ *of* transformations, *or 2-cells. 1-cells are assigned a source and a target 0-cell, written as* $f : a \longrightarrow b$, *and 2-cells are assigned a source and a target 1-cell, say* f *and* g, *in such a way that* $f, g : a \longrightarrow b$, *and this is indicated as* $\alpha : f \Rightarrow g : a \longrightarrow b$, *or simply* $\alpha : f \Rightarrow g$. *Moreover, the following operations are given:*

- a *partial operation* $\_;\_$ *of* horizontal composition *of 1-cells, which assigns to each pair of arrows* $f : a \longrightarrow b$, $g : b \longrightarrow c$ *a 1-cell* $f;g : a \longrightarrow c$,
- a *partial operation* $\_*\_$ *of* horizontal composition *of 2-cells, which assigns to each pair* $\alpha : f \Rightarrow g : a \longrightarrow b$, $\beta : h \Rightarrow k : b \longrightarrow c$ *a 2-cell (see Figure 2.9)*

$$\alpha * \beta : f;h \Rightarrow g;k : a \longrightarrow c,$$

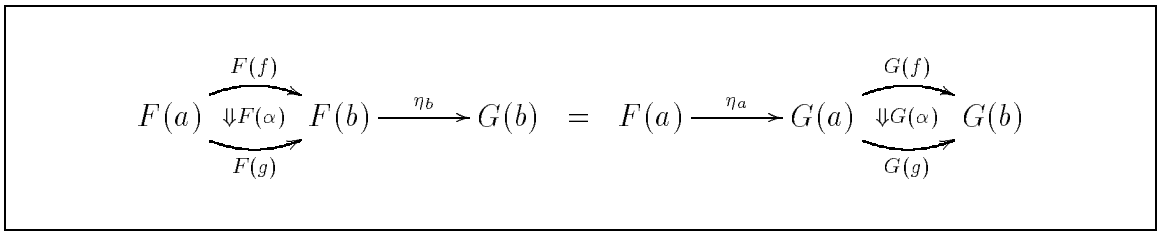


Figure 2.11: Naturality axiom of the 2-transformation η for the 2-cell α.

- *a partial operation* $\_\circ\_$ *of* vertical composition *of 2-cells, which assigns to each pair* $\alpha : f \Rightarrow g : a \longrightarrow b$, $\beta : g \Rightarrow h : a \longrightarrow b$ *a 2-cell (see Figure 2.9)*

$$\alpha \circ \beta : f \Rightarrow h : a \longrightarrow c.$$

Moreover, to each object a *there is associated an identity 1-cell* id_a*, and to each morphism* f *there is associated a 2-cell identity* 1_f*, satisfying the following axioms:*

- *the objects and the morphisms with the horizontal composition of 1-cells and the identities* id_a *form a category* $\mathcal{C}^!$*, called the* underlying category *of* $\mathcal{C}$*,*
- *for any pair of objects* a *and* b*, the morphisms of the kind* $f : a \longrightarrow b$ *and their 2-cells form a category under vertical composition of 2-cells with identities* 1_f*,*
- *the objects and the 2-cells form a category under the operation of horizontal composition of 2-cells with identities* 1_{id_a}*,*
- *for all* $f : a \longrightarrow b$ *and* $g : b \longrightarrow c$*, it is* $1_f * 1_g = 1_{f;g}$*,*
- *for all the situations as the one in Figure 2.10, it holds the* exchange law*:*

$$(\alpha * \gamma) \circ (\beta * \delta) = (\alpha \circ \beta) * (\gamma \circ \delta).$$

Since the identity 2-cell 1_f is essentially f, we usually denote the horizontal composition of 2-cells with the same symbol $\_;\_$ of horizontal composition of 1-cells (in place of $\_ * \_$).

EXAMPLE 2.1.12 *The category* **Cat** *of categories and functors is a 2-category (indeed, we remind that* $\mathbf{Cat}[\mathcal{C}, \mathcal{C}']$ *is the category having the functors from* $\mathcal{C}$ *to* $\mathcal{C}'$ *as objects, and the natural transformations between such functors as arrows).*

DEFINITION 2.1.19 (2-FUNCTOR)
Given the 2-categories $\mathcal{C}$ *and* $\mathcal{C}'$*, a* 2-functor $F : \mathcal{C} \longrightarrow \mathcal{C}'$ *is a function which maps objects to objects, morphisms to morphisms and 2-cells to 2-cells, preserving identities and composition of all kinds.*

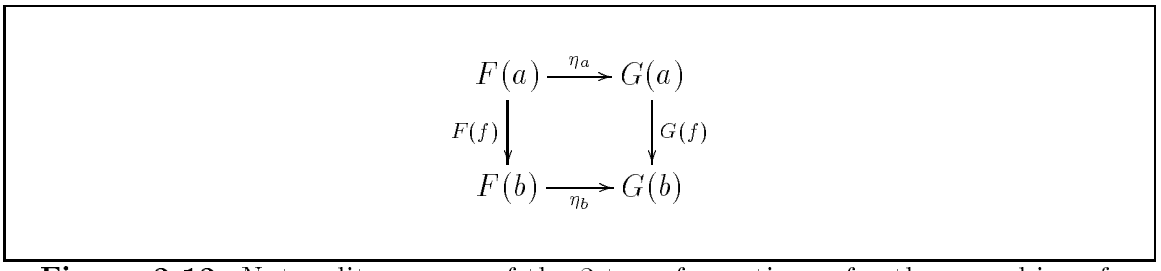


Figure 2.12: Naturality square of the 2-transformation η for the morphism f.

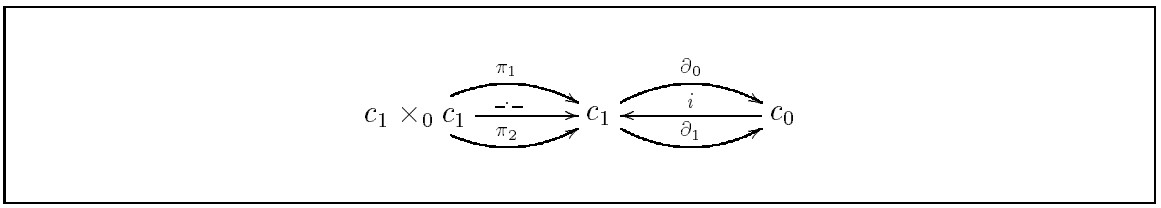


Figure 2.13: Internal category of $\mathcal{C}$.

DEFINITION 2.1.20 (2-NATURAL TRANSFORMATION)
Given the 2-functors $F, G : \mathcal{C} \longrightarrow \mathcal{C}'$, a 2-natural transformation *η from F to G is a $\mathcal{C}$-indexed family of morphisms $\eta_a : F(a) \longrightarrow G(a)$ in $\mathcal{C}'$ such that for any $\alpha : f \Rightarrow g : a \longrightarrow b$ the equation $F(\alpha); \eta_b = \eta_a; G(\alpha)$ holds (Figure 2.11).*

Note that, for any $f : c \longrightarrow d$ in $\mathcal{C}$, we can take $\alpha = 1_f$, thus obtaining the usual commutative square for the naturality of η w.r.t. f (see the diagram in Figure 2.12).

The notions of monoidal 2-category and symmetric 2-category, monoidal 2-functor and symmetric 2-functor are the obvious ones (but we remark that the naturality of symmetries also holds on 2-cells). Also the definition of product and of terminal object are the natural extension of the ordinary ones (e.g., a 2-category has binary 2-products if its underlying category $\mathcal{C}_!$ has binary products and for every pair $\alpha : f \Rightarrow g : c \longrightarrow a$ and $\beta : h \Rightarrow k : c \longrightarrow b$ of 2-cells, there exists a unique cell $\gamma = \langle \alpha, \beta \rangle : \langle f, h \rangle \Rightarrow \langle g, k \rangle : c \longrightarrow a \times b$ satisfying $\gamma; \Pi_a = \alpha$ and $\gamma; \Pi_b = \beta$).

2.1.8 Internal Constructions

Double categories can be defined as internal categories in **Cat**. The idea is to take a category whose arrows and objects have some structure, and the operations of the category preserve such structure. In the case of double categories the additional structure is again that of categories.

REMARK 2.1.13 (PULLBACK NOTATION) *We write $x \times_0 y$ (instead of the heavier notation $x \times_{f,g} y$) for the pullback of x and y along morphisms f and g with common target z. The subscript should be considered essentially as a warning that we are dealing with a pullback, and not just a product. The pullback projections will be usually denoted by π indexed by a number or some other symbol.*

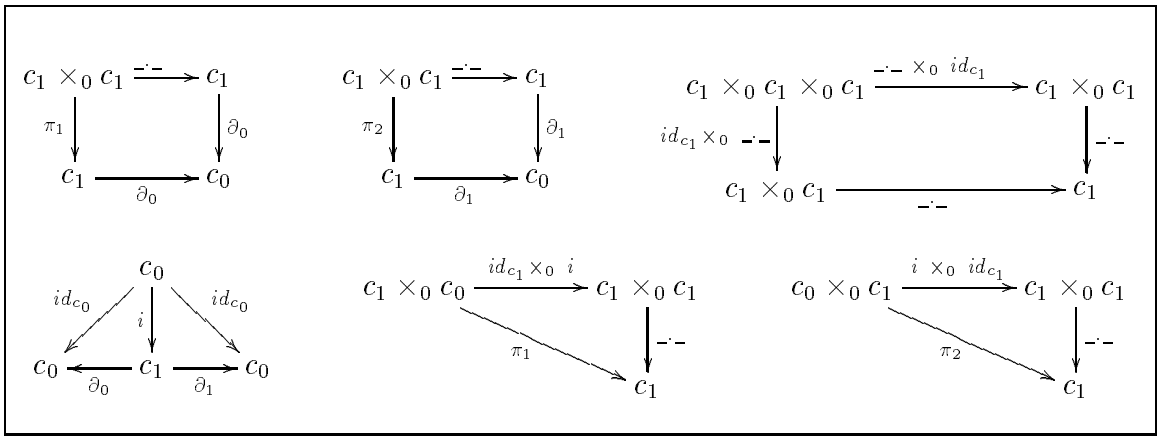


Figure 2.14: Commutative squares of internal categories.

The notation $\langle \_, \_ \rangle_0$ will be used as a "pullback pairing" map, that is, given suitable arrows $h : w \longrightarrow x$ and $k : w \longrightarrow y$, we have $\langle h, k \rangle_0 : w \longrightarrow x \times_0 y$.

DEFINITION 2.1.21 (INTERNAL CATEGORY)
Let $\mathcal{C}$ be a category[4] *with all finite limits. An* internal category *of $\mathcal{C}$ is a tuple $\mathbf{c} = (c_0, c_1, \partial_0, \partial_1, \_\cdot\_, i)$ such that:*

- *c_0 and c_1 are objects of $\mathcal{C}$;*
- *$\partial_0 : c_1 \longrightarrow c_0$, $\partial_1 : c_1 \longrightarrow c_0$, $\_\cdot\_ : c_1 \times_0 c_1 \longrightarrow c_1$ and $i : c_0 \longrightarrow c_1$ (where $c_1 \times_0 c_1$, together with two arrows $\pi_1, \pi_2 : c_1 \times_0 c_1 \longrightarrow c_1$, is a pullback object in $\mathcal{C}$ of ∂_0, ∂_1, representing the pair of composable arrows accordingly to the source and target operations ∂_0 and ∂_1) are arrows of $\mathcal{C}$ (see Figure 2.13);*
- *the following equations hold in $\mathcal{C}$ (see the commutative squares in Figure 2.14):*
  - *$(\_\cdot\_) * \partial_0 = \pi_1 * \partial_0$ and $(\_\cdot\_) * \partial_1 = \pi_2 * \partial_1$ (source and target of the internal composition $\_\cdot\_$),*
  - *$(c_1 \times_0 (\_\cdot\_)) * (\_\cdot\_) = ((\_\cdot\_) \times_0 c_1) * (\_\cdot\_)$ and $i * \partial_0 = c_0 = i * \partial_1$ (associativity of $\_\cdot\_$, and source and target of identities),*
  - *$(c_1 \times_0 i) * (\_\cdot\_) = \pi_1$ and $(i \times_0 c_1) * (\_\cdot\_) = \pi_2$ (identity as neutral element for the internal composition $\_\cdot\_$).*

We write $\mathbf{c} \in Cat(\mathcal{C})$ to state that $\mathbf{c}$ is an internal category of $\mathcal{C}$.

REMARK 2.1.14 *Notice that there is an implicit isomorphism between $c_1 \times_0 (c_1 \times_0 c_1)$ and $(c_1 \times_0 c_1) \times_0 c_1$. In the following, this isomorphism will be always skipped in order to maintain the notation at a simpler level.*

EXAMPLE 2.1.15 *An internal category in* **Mon** *is a monoidal category.*

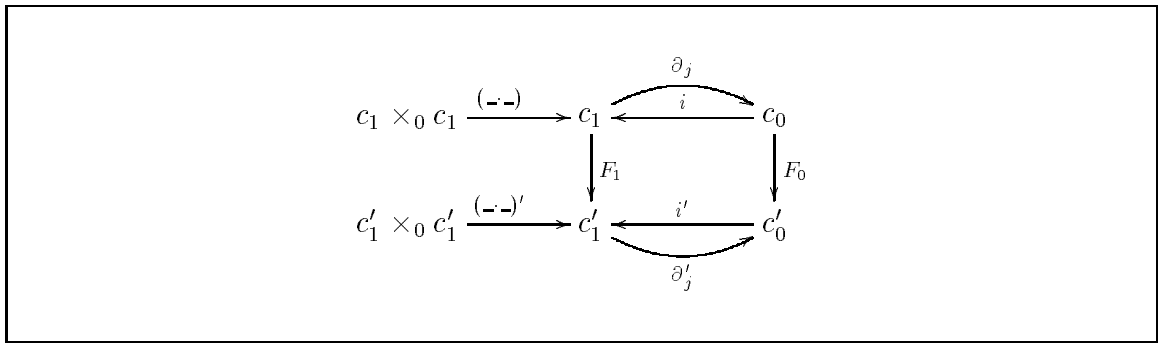


Figure 2.15: Internal functor.

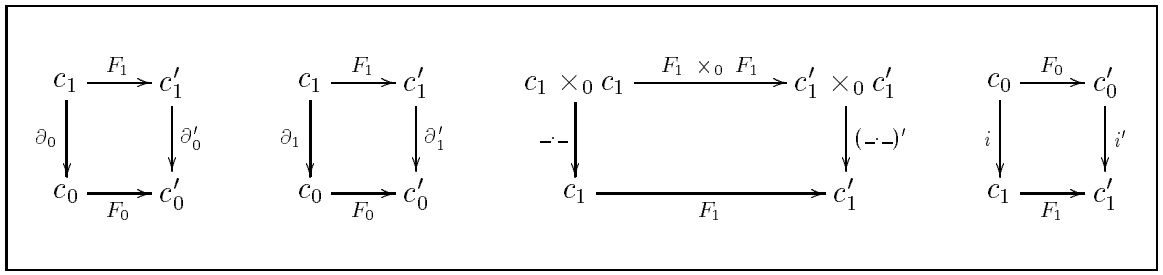


Figure 2.16: Commutative squares of internal functors.

DEFINITION 2.1.22 (INTERNAL FUNCTORS)
Let $\mathbf{c}, \mathbf{c}' \in Cat(\mathcal{C})$. *An* internal functor *from* $\mathbf{c}$ *to* $\mathbf{c}'$ *(written* $F : \mathbf{c} \longrightarrow \mathbf{c}'$*) is a couple* $F = (F_0, F_1)$ *of arrows in* $\mathcal{C}$, $F_0 : c_0 \longrightarrow c'_0$ *and* $F_1 : c_1 \longrightarrow c'_1$ *(see Figure 2.15) which satisfy (see the commutative squares in Figure 2.16):*

- $F_1 * \partial'_0 = \partial_0 * F_0$ *and* $F_1 * \partial'_1 = \partial_1 * F_0$ *(source and target are preserved),*
- $(F_1 \times_0 F_1) * (\_\cdot\_)' = (\_\cdot\_) * F_1$ *and* $F_0 * i' = i * F_1$ *(composition and identities are preserved).*

DEFINITION 2.1.23 (CATEGORY $Cat(\mathcal{C})$)
The category $Cat(\mathcal{C})$ *has as objects the internal categories of* $\mathcal{C}$ *and as morphisms the internal functors (composition of functors is defined in the obvious way).*

EXAMPLE 2.1.16 *The category* $Cat(\mathbf{Set})$ *is the category* **Cat**.

DEFINITION 2.1.24 (INTERNAL NATURAL TRANSFORMATIONS)
Let F, G *be two internal functors from* $\mathbf{c}$ *to* $\mathbf{c}'$. *An* internal natural transformation *from* F *to* G *is an arrow* $\Phi : c_0 \longrightarrow c'_1$ *(see Figure 2.17) which satisfies (see the commutative squares in Figure 2.18):*

- $\Phi * \partial'_0 = F_0$ *and* $\Phi * \partial'_1 = G_0$ *(source and target),*
- $\langle \partial_0 * \Phi, G_1 \rangle_0 * (\_\cdot\_)' = \langle F_1, \partial_1 * \Phi \rangle_0 * (\_\cdot\_)'$ *(naturality).*

[4] We denote composition in $\mathcal{C}$ by $\_*\_$ and identities either by id_a or by just their object name a.

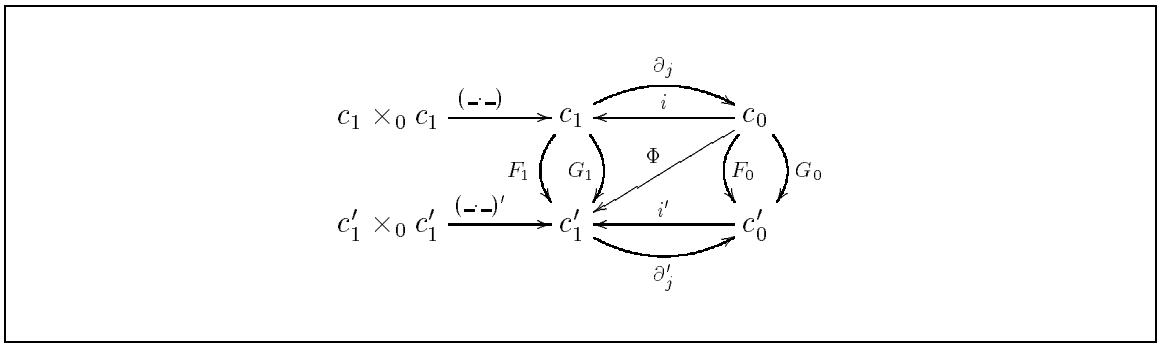


Figure 2.17: Internal natural transformation.

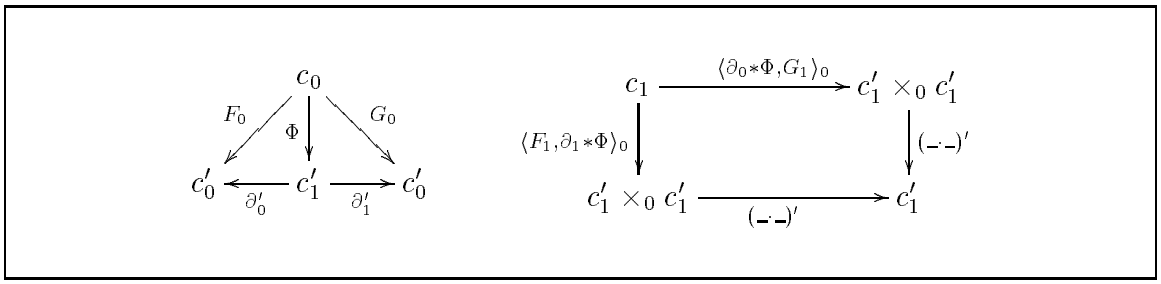


Figure 2.18: Commutative squares of internal natural transformations.

Taking $Cat(\mathbf{Cat})$, we can notice that the naturality of an internal transformation is required only for the (internal) composition $\_\cdot\_$, whereas Φ is functorial w.r.t. the composition in **Cat**. Hence the internal view is not suitable for dealing with auxiliary structures common to the two dimensions, because it yields an asymmetric notion of transformation that is exploited in one dimension but not in the other.

2.2 Algebraic Theories

We recall here some basic definitions from graph theory, used to recast the usual notion of term over a signature in a more general setting, where suitable equivalence classes of monoidal (hyper)graphs equipped with auxiliary arrows are considered.

DEFINITION 2.2.1 (HYPERSIGNATURE)
A many-sorted hypersignature Σ *over a set* S *of sorts is a family* $\{\Sigma_{w,w'}\}_{w,w' \in S^*}$ *of sets of operators, where each operator* $f \in \Sigma_{w,w'}$ *takes* $|w|$ *arguments typed accordingly to* w*, and returns a tuple of* $|w'|$ *values, typed accordingly to* w'*.*

If S *is a singleton, the hypersignature* Σ *is called* one-sorted *and is simply denoted by the family* $\{\Sigma_{n,m}\}_{n,m \in \mathbb{N}}$*.*

DEFINITION 2.2.2 (SIGNATURE)
A many-sorted signature *is a signature* Σ *such that* $\Sigma_{w,w'} \neq \emptyset \Rightarrow w' \in S$*, i.e., a family* $\{\Sigma_{w,s}\}_{w \in S^*, s \in S}$*. Moreover, if* S *is a singleton, the signature* Σ *is called* one-sorted *and is simply denoted by the family* $\{\Sigma_n\}_{n \in \mathbb{N}}$ *(only the arities are important).*

DEFINITION 2.2.3 (GRAPHS)
A graph G *is a 4-tuple* $(O_G, A_G; \partial_0, \partial_1)$*, where* O_G *is the set of* objects*,* A_G *is the set of* arrows*, and* $\partial_0, \partial_1 : A_G \longrightarrow O_G$ *are functions, called respectively* source *and* target*.*

A graph G *is* reflexive *if there exists an* identity *function* $id : O_G \longrightarrow A_G$ *such that* $\partial_0(id(a)) = a = \partial_1(id(a))$*, for all* $a \in O_G$*; it is* with pairing *if* O_G *is a monoid; it is* monoidal *if it is reflexive, both* O_G *and* A_G *are monoids, and the functions* ∂_0*,*∂_1*, and* id *are monoid homomorphisms (i.e., preserve the monoidal operator and the neutral element).*

We use the standard notation $f : a \longrightarrow b$ to denote an arrow f with source a and target b.

It is immediate that a many-sorted hypersignature Σ over S can be seen as a graph with pairing G_Σ such that its objects are strings on S (i.e., $O_{G_\Sigma} = S^*$, string concatenation is the monoidal operator, and the empty string λ is the neutral element), and its arcs are labelled with operators in Σ (i.e., $f : w \longrightarrow w' \in A_{G_\Sigma}$ if and only if $f \in \Sigma_{w,w'}$).

For simplicity, we will often present our constructions considering one-sorted hypersignatures only, then sketching how the results can be extended to deal with the many-sorted case (most extensions are trivial, but the notation for the many-sorted case is more complex).

DEFINITION 2.2.4 (GRAPH THEORIES)
Given a one-sorted (hyper)signature Σ*, the associated* graph theory $\mathbf{G}(\Sigma)$ *is the monoidal graph with objects the elements of the additive monoid of underlined natural numbers (i.e.,* $\underline{0}$ *is the neutral element, and the monoidal operation* $\_ \otimes \_$ *is defined as* $\underline{n} \otimes \underline{m} = \underline{n+m}$*), and arrows those generated by the following inference rules:*

$$(\text{generators})\ \frac{f \in \Sigma_{n,m}}{f : \underline{n} \longrightarrow \underline{m} \in \mathbf{G}(\Sigma)} \qquad (\text{identities})\ \frac{n \in \mathbb{N}}{id_n : \underline{n} \longrightarrow \underline{n} \in \mathbf{G}(\Sigma)}$$

$$(\text{pairing})\ \frac{t : \underline{n} \longrightarrow \underline{m},\ t' : \underline{n}' \longrightarrow \underline{m}' \in \mathbf{G}(\Sigma)}{t \otimes t' : \underline{n} \otimes \underline{n}' \longrightarrow \underline{m} \otimes \underline{m}' \in \mathbf{G}(\Sigma)}$$

Monoidality implies that $\_ \otimes \_$ *is associative on arrows,* id_0 *is the neutral element of the monoid of arrows, and that the* monoidality *axiom* $id_{n \otimes m} = id_n \otimes id_m$ *holds for all* $n, m \in \mathbb{N}$*.*

This view is very useful to define a chain of further structural enrichments on graphs, finally leading to the usual algebraic notion of terms over a signature. We are particularly interested in this final level, and also in the intermediate level corresponding to symmetric theories.

REMARK 2.2.1 *The name "monoidal-sequential theories" might seem more appropriate than "monoidal theories," because graph theories already possess the monoidal structure. However, to keep the notation consistent with the literature (see, e.g., [69]) we prefer to use the terminology "monoidal theories."*

DEFINITION 2.2.5 (MONOIDAL THEORIES)
Given a (hyper)signature Σ, the associated monoidal theory $\mathbf{M}(\Sigma)$ *is the monoidal graph generated by the same inference rules and axioms given for $\mathbf{G}(\Sigma)$, together with the following inference rules:*

$$(\text{composition})\ \frac{t : \underline{n} \longrightarrow \underline{m},\ t' : \underline{m} \longrightarrow \underline{k} \in \mathbf{M}(\Sigma)}{t;t' : \underline{n} \longrightarrow \underline{k} \in \mathbf{M}(\Sigma)}$$

Moreover, the composition operator $\_;\_$ is associative, and the arrows of $\mathbf{M}(\Sigma)$ satisfy the identity axiom *(for all $t : \underline{n} \longrightarrow \underline{m}$), $id_n;t = t = t;id_m$, and the* functoriality axiom *$(s \otimes t);(s' \otimes t') = (s;s') \otimes (t;t')$ (whenever compositions $s;s'$ and $t;t'$ are defined).*

DEFINITION 2.2.6 (SYMMETRIC THEORIES)
The symmetric theory $\mathbf{S}(\Sigma)$ *associated to the (hyper)signature Σ is the monoidal graph generated by the same inference rules and axioms given for $\mathbf{M}(\Sigma)$, together with the following inference rule:*

$$(\text{symmetries})\ \frac{n, m \in \mathbb{N}}{\gamma_{n,m} : \underline{n} \otimes \underline{m} \longrightarrow \underline{m} \otimes \underline{n} \in \mathbf{S}(\Sigma)}$$

Moreover, the symmetries satisfy the naturality axiom *(for all $t : \underline{n} \longrightarrow \underline{m}$, and $t' : \underline{n}' \longrightarrow \underline{m}'$ in $\mathbf{S}(\Sigma)$),*

$$(t \otimes t');\gamma_{m,m'} = \gamma_{n,n'};(t' \otimes t),$$

and the coherence axioms *(for all $n, m, k \in \mathbb{N}$),*

$$\gamma_{n\otimes m,k} = (id_n \otimes \gamma_{m,k});(\gamma_{n,k} \otimes id_m),\quad \text{and}\quad \gamma_{n,m};\gamma_{m,n} = id_{n\otimes m}.$$

Actually, a (symmetric) monoidal theory is just a particular (symmetric) strict monoidal category [93], i.e., the free such category generated by the signature Σ.

DEFINITION 2.2.7 (ALGEBRAIC THEORIES)
Given a signature Σ, the associated algebraic theory $\mathbf{A}(\Sigma)$ *is the monoidal graph generated by the same inference rules and axioms given for $\mathbf{S}(\Sigma)$ together with the following inference rules:*

$$(\text{duplicators})\ \frac{n \in \mathbb{N}}{\nabla_n : \underline{n} \longrightarrow \underline{n} \otimes \underline{n} \in \mathbf{A}(\Sigma)} \qquad (\text{dischargers})\ \frac{n \in \mathbb{N}}{!_n : \underline{n} \longrightarrow \underline{0} \in \mathbf{A}(\Sigma)}$$

Moreover, the arrows of $\mathbf{A}(\Sigma)$ *verify the* naturality axioms *(for all* $t : \underline{n} \longrightarrow \underline{m}$*),*

$$t;\nabla_m = \nabla_n;(t \otimes t), \quad \textit{and} \quad t;!_m =!_n,$$

and the coherence axioms *(for all* $n, m \in \mathbb{N}$*),*

$$\nabla_{n\otimes m} = (\nabla_n \otimes \nabla_m);(id_n \otimes \gamma_{n,m} \otimes id_m), \quad \nabla_0 = id_0 =!_0, \quad !_{n\otimes m} =!_n \otimes !_m,$$

$$\nabla_n;(1_n \otimes \nabla_n) = \nabla_n;(\nabla_n \otimes 1_n), \quad \nabla_n;\gamma_{n,n} = \nabla_n, \quad \textit{and} \quad \nabla_n;(1_n \otimes !_n) = id_n.$$

Coherence axioms tell how $\nabla_{n\otimes m}$ and $!_{n\otimes m}$ are defined in terms of the basic components $\nabla_{1,1}$ and $!_1$.

It can be considered categorical folklore that a cartesian category can actually be decomposed into a symmetric monoidal category, together with a family of suitable natural transformations, usually denoted as *diagonals* and *projections*. Then, Definition 2.2.7 can be proved equivalent to the Lawvere theory construction $\mathbf{Th}[\Sigma]$, dating back to the early work of Lawvere [91]. A classical result states the equivalence of these theories with the usual term algebra.

Definition 2.2.8 (Σ-Algebra)
Given a signature $\Sigma = \{\Sigma_n\}_{n\in\mathbb{N}}$*, a* Σ-algebra *is a set* A*, together with an assignment of a function* $A_f : A^n \longrightarrow A$ *for each* $f \in \Sigma_n$*.*

As usual, we write T_Σ to denote the Σ-algebra of *ground* Σ*-terms*, and $T_\Sigma(X)$ to denote the Σ-algebra of Σ-terms with variables in a set X.

Proposition 2.2.1
Let Σ *be a signature. Then, for all* $n, m \in \mathbb{N}$*, there exists a one-to-one correspondence between the set* $\mathbf{A}(\Sigma)[\underline{n}, \underline{m}]$ *of arrows from* $\underline{n}$ *to* $\underline{m}$ *in* $\mathbf{A}(\Sigma)$ *and the* m*-tuples of elements of the term algebra* $T_\Sigma(X)$ *over a set* X *of* n *variables.*

We believe that this presentation of algebraic theories separates very nicely the auxiliary structure from the Σ-structure (better than the ordinary description involving the meta-operation of substitution).

In particular, the naturality axioms of ∇ and ! allow a controlled form of *duplication* and *discharging* of information. For example *term graphs* [55, 42] differ from terms exactly for the two naturality axioms of ∇ and !, and *gs-graphs* [41, 42] have the same axiomatization of term graphs but can deal with hypersignatures.

2.3 Rewriting Logic

Rewriting logic [98, 99, 101] is an elegant and expressive semantic framework for the specification of languages and systems, and it is a good candidate as a logical framework in which many other logics can be represented [95, 96].

Reflexivity.

$$\frac{[t] \in T_{\Sigma,E}(X)}{[t] \Rightarrow [t]}$$

Congruence.

$$\frac{[t_1] \Rightarrow [t'_1],\ \ldots,\ [t_n] \Rightarrow [t'_n],\ f \in \Sigma_n}{[f(t_1,\ldots,t_n)] \Rightarrow [f(t'_1,\ldots,t'_n)]}$$

Replacement.

$$\frac{[w_1] \Rightarrow [w'_1],\ \ldots,\ [w_n] \Rightarrow [w'_n],\ r : [t(x_1,\ldots,x_n)] \Rightarrow [t'(x_1,\ldots,x_n)] \in R}{[t(\vec{w}/\vec{x})] \Rightarrow [t'(\vec{w}/\vec{x})]}$$

Transitivity.

$$\frac{[t_1] \Rightarrow [t_2],\ [t_2] \Rightarrow [t_3]}{[t_1] \Rightarrow [t_3]}$$

Table 2.1: Inference rules for flat rewriting sequents.

Here we just sketch an introductory description of the subject and the original 2-algebraic semantics as proposed by Meseguer in [98]. A short summary of the reflective capabilities of rewriting logic will be given in Section 7.2.1.

Let Σ be a signature and let X be a set of objects. Given a set E of Σ-equations (i.e., sentences of the form $t = t'$ with $t, t' \in T_\Sigma(X)$), $T_{\Sigma,E}$ (respectively $T_{\Sigma,E}(X)$) denotes the Σ-algebra of equivalence classes of ground Σ-terms modulo the equations in E (respectively the Σ-algebra of equivalence classes of Σ-terms with variables in X modulo the equations in E).

We denote the E-equivalence class of a Σ-term t either by $[t]_E$, or just by $[t]$.

Definition 2.3.1 (Rewrite Theories)
A labelled rewrite theory $\mathcal{R}$ *is a 4-tuple* (Σ, E, L, R) *where* Σ *is a signature,* E *is a set of* Σ*-equations,* L *is the set of* labels, *and* $R \subseteq L \times T_{\Sigma,E}(X) \times T_{\Sigma,E}(X)$ *is the set of* labelled rewrite rules. *For* $(r, [t], [t']) \in R$ *we use the notation* $r : [t] \Rightarrow [t']$.

Rewrite rules in $\mathcal{R}$ may be understood as basic sequents *entailed* by $\mathcal{R}$. More complex deductions in the logic of $\mathcal{R}$ can be obtained by a finite application of four simple rules.

Definition 2.3.2 (Rewriting Sequents)
Let $\mathcal{R} = (\Sigma, E, L, R)$ *be a rewrite theory. We say that* $\mathcal{R}$ entails a flat sequent $[t] \Rightarrow [t']$, *written* $\mathcal{R} \vdash [t] \Rightarrow [t']$ *iff* $[t] \Rightarrow [t']$ *can be obtained by a finite number of applications of the* rules of deduction *in Table 2.1, where* $t(\vec{w}/\vec{x})$ *denotes the* simultaneous *substitution of* w_i *for* x_i *in* t.

Identities.

$$\frac{[t] \in T_{\Sigma,E}(X)}{[t] : [t] \Rightarrow [t]}$$

Σ-structure.

$$\frac{\alpha_1 : [t_1] \Rightarrow [t'_1],\ \ldots,\ \alpha_n : [t_n] \Rightarrow [t'_n],\ f \in \Sigma_n}{f(\alpha_1, \ldots, \alpha_n) : [f(t_1, \ldots, t_n)] \Rightarrow [f(t'_1, \ldots, t'_n)]}$$

Replacement.

$$\frac{\alpha_1 : [w_1] \Rightarrow [w'_1],\ \ldots,\ \alpha_n : [w_n] \Rightarrow [w'_n],\ r : [t(x_1, \ldots, x_n)] \Rightarrow [t'(x_1, \ldots, x_n)] \in R}{r(\alpha_1, \ldots, \alpha_n) : [t(\vec{w}/\vec{x})] \Rightarrow [t'(\vec{w}/\vec{x})]}$$

Composition.

$$\frac{\alpha : [t_1] \Rightarrow [t_2],\ \beta : [t_2] \Rightarrow [t_3]}{\alpha \circ \beta : [t_1] \Rightarrow [t_3]}$$

Table 2.2: Inference rules for decorated rewriting sequents.

A rewrite theory is just a *static description* of "what a system can do." The *meaning* of the theory should be given by *computational models* of its actual behaviour. Taking advantage of the correspondence between *deductions* in rewriting logic and *(concurrent) computations*, it is natural, in the spirit of initial model semantics, to define the *initial model* $\mathcal{T}_{\mathcal{R}}$ of $\mathcal{R}$ as a system whose states are E-equivalence classes of Σ-terms, and whose transitions are equivalence classes of terms representing *proofs* in rewriting deduction, i.e., concurrent rewritings using the rules in R. The rules for generating such proof terms are obtained from the rules of deduction of Definition 2.3.2 by decorating the sequents with appropriate proof terms.

DEFINITION 2.3.3 (PROOF TERMS OF REWRITING LOGIC)
Let $\mathcal{R} = (\Sigma, E, L, R)$ be a rewrite theory such that each rewrite rule has a different label. We say that $\mathcal{R}$ entails the proof term $\alpha : [t] \Rightarrow [t']$, *written $\mathcal{R} \vdash \alpha : [t] \Rightarrow [t']$ (or just $\mathcal{R} \vdash \alpha$), iff the proof term α is generated by a finite number of applications of the* decorated *rules of deduction in Table 2.2.*

Each of the rules presented in Table 2.2 defines a different operation, taking certain proof terms as arguments and returning a resulting proof term. In other words, proof terms form an algebraic structure $\mathcal{P}_{\mathcal{R}}(X)$ consisting of a graph with nodes $T_{\Sigma,E}(X)$, with identity arrows, and with operations f (for each $f \in \Sigma$), r (for each rewrite rule), and $\_ \circ \_$ (for composing arrows).

REMARK 2.3.1 *Notice that we use diagrammatic order for the sequential composition of proofs, and that the composition operator is denoted by the same symbol of* vertical *composition of natural transformations to enhance the relations with the categorical semantics described at the end of this section.*

DEFINITION 2.3.4 (MODEL $\mathcal{T}_{\mathcal{R}}(X)$)
Given a rewrite theory $\mathcal{R}$, the model $\mathcal{T}_{\mathcal{R}}(X)$ of $\mathcal{R}$ is the quotient of the algebra of proof terms $\mathcal{P}_{\mathcal{R}}(X)$ modulo the following equations (when composition of arrows is involved, we always implicitly assume that the corresponding source and target match):

Category

Associativity *(for all composable $\alpha, \beta, \gamma \in \mathcal{P}_{\mathcal{R}}(X)$):* $\alpha \circ (\beta \circ \gamma) = (\alpha \circ \beta) \circ \gamma$

Identities *(for any $\alpha : [t] \Rightarrow [t'] \in \mathcal{P}_{\mathcal{R}}(X)$):* $\alpha \circ [t'] = \alpha = [t] \circ \alpha$

Functoriality of the Σ-algebraic structure *(for any $f \in \Sigma_n$)*

Preservation of composition *(for all $\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_n \in \mathcal{P}_{\mathcal{R}}(X)$:*

$$f(\alpha_1 \circ \beta_1, \ldots, \alpha_n \circ \beta_n) = f(\alpha_1, \ldots, \alpha_n) \circ f(\beta_1, \ldots, \beta_n)$$

Preservation of identities *(for all $[t_1], \ldots, [t_n]$):*

$$f([t_1], \ldots, [t_n]) = [f(t_1, \ldots, t_n)]$$

E-axioms *(for any $t(x_1, \ldots, x_n) = t'(x_1, \ldots, x_n) \in E$)*

$$\forall \alpha_1, \ldots, \alpha_n, \quad t(\alpha_1, \ldots, \alpha_n) = t'(\alpha_1, \ldots, \alpha_n)$$

Exchange *(for any $r : [t(x_1, \ldots, x_n)] \Rightarrow [t'(x_1, \ldots, x_n)] \in R$)*

$$\frac{\alpha_1 : [w_1] \Rightarrow [w'_1], \ \ldots, \ \alpha_n : [w_n] \Rightarrow [w'_n]}{r(\overrightarrow{[w]}) \circ t'(\vec{\alpha}) = r(\vec{\alpha}) = t(\vec{\alpha}) \circ r(\overrightarrow{[w']})}$$

Note that the set X of variables is actually a parameter of these constructions. In particular, for $X = \emptyset$ we adopt the notation $\mathcal{T}_{\mathcal{R}}$.

The **Category** equations make $\mathcal{T}_{\mathcal{R}}(X)$ a category. The **Functoriality** equations make each operator f of Σ a functor. The **E-axioms** equations extend axioms in E also to proof terms. The **Exchange** law is particularly relevant, because it states that the *simultaneous* rewriting of a "context" t via r and of its "subcomponents" $w_1, \ldots, w_n$ via $\alpha_1, \ldots, \alpha_n$ is equivalent to the sequential composition $r(\overrightarrow{[w]}) \circ t'(\vec{\alpha})$ (first rewriting on top and then on subcomponents) and also to the sequential composition $t(\vec{\alpha}) \circ r(\overrightarrow{[w']})$ (first rewriting the subcomponents and then the top of the term). It follows that each proof term in $\mathcal{T}_{\mathcal{R}}(X)$ is a description of a concurrent computation, according to an equational theory of *true concurrency*. Moreover, since $[t(x_1, \ldots, x_n)]$ and $[t'(x_1, \ldots, x_n)]$ can be regarded as functors from $\mathcal{T}_{\mathcal{R}}(X)^n$ to $\mathcal{T}_{\mathcal{R}}(X)$, the exchange law asserts that r is a *natural transformation*. This situation is illustrated in Figure 2.19.

LEMMA 2.3.1 (CF. [99])
For each rewrite rule $r : [t(x_1, \ldots, x_n)] \Rightarrow [t'(x_1, \ldots, x_n)]$ in R, the family of morphisms $\{r(\overrightarrow{[w]}) : [t(\vec{w}/\vec{x})] \Rightarrow [t'(\vec{w}/\vec{x})] \mid \overrightarrow{[w]} \in T_{\Sigma,E}(X)^n\}$ defines a natural transformation from the functor $[t(x_1, \ldots, x_n)] : \mathcal{T}_{\mathcal{R}}(X)^n \longrightarrow \mathcal{T}_{\mathcal{R}}(X)$ to the functor $[t'(x_1, \ldots, x_n)] : \mathcal{T}_{\mathcal{R}}(X)^n \longrightarrow \mathcal{T}_{\mathcal{R}}(X)$.

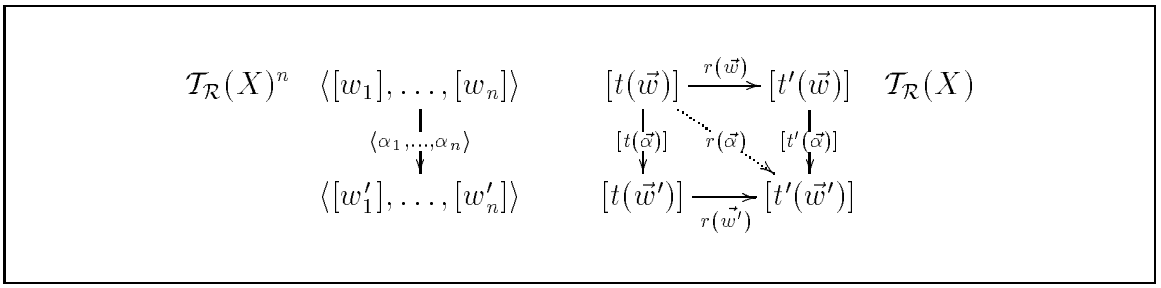


Figure 2.19: Graphical representation of the **Exchange** law as a natural transformation.

The category $\mathcal{T}_\mathcal{R}(X)$ is a very particular model of the rewrite theory $\mathcal{R}$, in that its objects are the elements of a very particular Σ-algebra, namely $T_{\Sigma,E}(X)$. The general notion of model, called $\mathcal{R}$-*system*, is defined as follows.

DEFINITION 2.3.5 ($\mathcal{R}$-SYSTEM)
Given a rewrite theory $\mathcal{R} = (\Sigma, E, L, R)$, an $\mathcal{R}$-system $\mathcal{S}$ *is a category $\mathcal{S}$ together with:*

1. *a family of functors $\{\mathcal{S}_f : \mathcal{S}^n \longrightarrow \mathcal{S} \mid f \in \Sigma_n\}$ satisfying the equations in E (i.e., for any $t(x_1, \ldots, x_n)$ the functor $\mathcal{S}_t$ is inductively defined in the obvious way from the functors $\mathcal{S}_f$, and for each E-equation $t(x_1, \ldots, x_n) = t'(x_1, \ldots, x_n)$ the identity of functors $\mathcal{S}_t = \mathcal{S}_{t'}$ holds);*

2. *for each rewrite rule $r : [t(\vec{x})] \Rightarrow [t'(\vec{x})]$ in R, a natural transformation $\mathcal{S}_r$ from $\mathcal{S}_t$ to $\mathcal{S}_{t'}$.*

An $\mathcal{R}$-homomorphism *$F : \mathcal{S} \longrightarrow \mathcal{S}'$ between two $\mathcal{R}$-systems is then a functor from $\mathcal{S}$ to $\mathcal{S}'$ such that it is a Σ-algebra homomorphism (i.e., for each $f \in \Sigma_n$, $\mathcal{S}_f; F = F^n; \mathcal{S}'_f$), and such that F "preserves" R (i.e., for each rewrite rule $r : [t(x_1, \ldots, x_n)] \Rightarrow [t'(x_1, \ldots, x_n)]$ in R we have that $\mathcal{S}_r; F = F^n; \mathcal{S}'_r$). This defines a category* $\mathcal{R}$-**Sys** *of* models *for the rewrite theory $\mathcal{R}$.*

The following theorem (cf. [99]) characterizes the relevance of $\mathcal{T}_\mathcal{R}$ and $\mathcal{T}_\mathcal{R}(X)$.

THEOREM 2.3.2 (INITIAL AND FREE MODEL OF $\mathcal{R}$)
$\mathcal{T}_\mathcal{R}$ is an initial object in the category $\mathcal{R}$-**Sys**. *More generally, $\mathcal{T}_\mathcal{R}(X)$ has the following universal property: Given an $\mathcal{R}$-system $\mathcal{S}$, each function $F : X \longrightarrow |\mathcal{S}|$ extends uniquely to an $\mathcal{R}$-homomorphism $\hat{F} : \mathcal{T}_\mathcal{R}(X) \longrightarrow \mathcal{S}$.*

Given an equational theory $T = (\Sigma, E)$ let us denote by $\mathbf{Alg}_{\Sigma,E}$ the category of T-algebras, and by $\mathcal{L}_T$ the Lawvere theory of T, having underlined natural numbers as objects, and where an equivalence class $[t(x_1, \ldots, x_n)]$ is viewed as an arrow

$$[t(x_1, \ldots, x_n)] : \underline{n} \longrightarrow \underline{1}$$

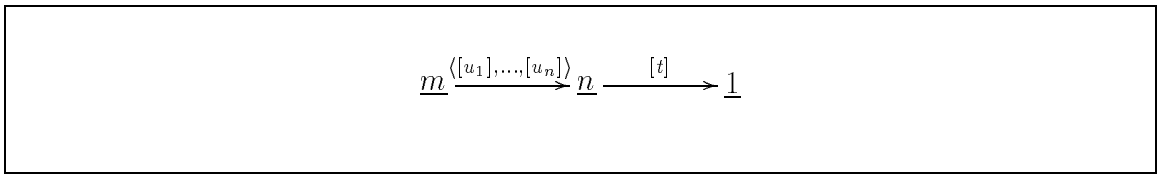


Figure 2.20: Composition as substitution.

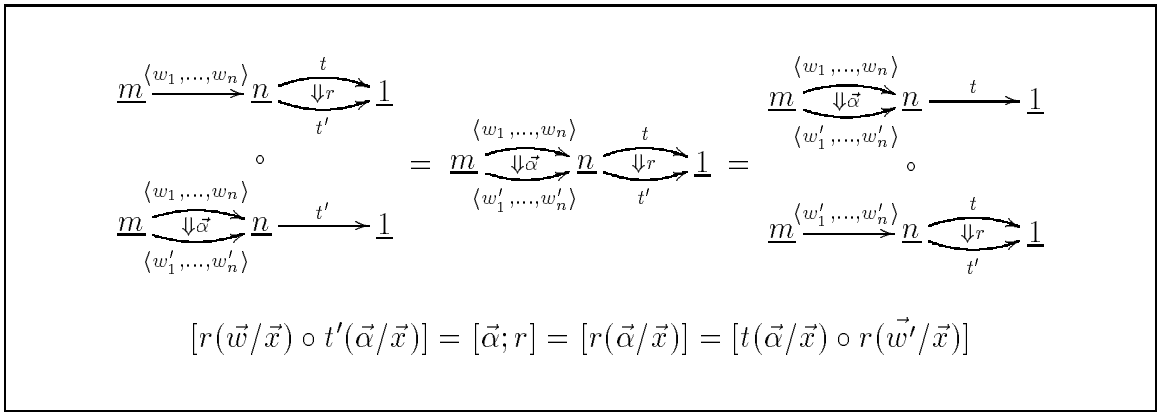


Figure 2.21: Graphical representation of the **Exchange** law in $\mathcal{L}_{\mathcal{R}}$.

(from n placeholders for the n ordered variables of t to the placeholder for the result), arrow composition being substitution (i.e., given n arrows $[u_i(y_1,\ldots,y_m)] : \underline{m} \longrightarrow \underline{1}$, for $i = 1,\ldots,n$, and an arrow $[t(x_1,\ldots,x_n)] : \underline{n} \longrightarrow \underline{1}$, the composition between $\langle [u_1],\ldots,[u_n]\rangle : \underline{m} \longrightarrow \underline{n}$ and $[t(x_1,\ldots,x_n)]$ (see Figure 2.20) yields the arrow $[t(\vec{u}/\vec{x})] : \underline{m} \longrightarrow \underline{1}$ as a result). In particular, for $T = (\Sigma, \emptyset)$, we have $\mathcal{L}_T \simeq \mathbf{A}(\Sigma)$, thanks to Proposition 2.2.1.

Lawvere made the seminal discovery that, given a Σ-algebra A satisfying E, the function mapping each E-equivalence class $[t(x_1,\ldots,x_n)]$ to its functional interpretation $A_{[t]} : A^n \longrightarrow A$ in the Σ-algebra A defines exactly a product-preserving functor $\hat{A} : \mathcal{L}_T \longrightarrow \mathbf{Set}$. Moreover, if we choose canonical set-theoretic products in the targets of such functors, and denote by $\mathbf{Mod}(\mathcal{L}_T, \mathbf{Set})$ the category with objects those functors and morphisms natural transformations between them, then the assignment $A \longmapsto \hat{A}$ corresponds to an isomorphism of categories $\mathbf{Alg}_{\Sigma,E} \simeq \mathbf{Mod}(\mathcal{L}_T, \mathbf{Set})$.

This situation generalizes very naturally to the case of rewriting logic: it suffices to change the "ground" on which models exist from the category **Set** to the 2-category **Cat**. Hence, models for rewriting logic are algebraic structures on categories (i.e., sets with additional structure) rather than on sets.

Indeed, given a rewrite theory $\mathcal{R}$, the 2-category with 2-products $\mathcal{L}_{\mathcal{R}}$ has underlined natural numbers as objects, E-equivalence classes of terms $[t(x_1,\ldots,x_n)]$ as arrows from $\underline{n}$ to $\underline{1}$, and equivalence classes of proof terms

$$[\alpha] : [t(x_1,\ldots,x_n)] \Rightarrow [t'(x_1,\ldots,x_n)]$$

as cells, where vertical composition is given by $[\alpha] \circ [\gamma] = [\alpha \circ \gamma]$, and horizontal composition $\alpha;\beta$ is given by $[t(\vec{\alpha}/\vec{x}) \circ \beta(\vec{v}/\vec{x})] : [t(\vec{u}/\vec{x})] \Rightarrow [t'(\vec{v}/\vec{x})]$.

As illustrated in Figure 2.21, the exchange law states the coherence between $\_;\_$ and $\_ \circ \_$.

As a matter of fact, given an $\mathcal{R}$-system $\mathcal{S}$, the assignment to each rule $r : [t] \Rightarrow [t']$ in R of a natural transformation $\mathcal{S}_r$ from the functor $\mathcal{S}_t : \mathcal{S}^n \longrightarrow \mathcal{S}$ to $\mathcal{S}_{t'} : \mathcal{S}^n \longrightarrow \mathcal{S}$ extends naturally to a 2-product preserving 2-functor $\hat{\mathcal{S}} : \mathcal{L}_{\mathcal{R}} \longrightarrow \mathbf{Cat}$, and the assignment $\mathcal{S} \longmapsto \hat{\mathcal{S}}$ yields an isomorphism of 2-categories $\mathcal{R}-\mathbf{Sys} \simeq \mathbf{Mod}(\mathcal{L}_{\mathcal{R}}, \mathbf{Cat})$, where $\mathbf{Mod}(\mathcal{L}_{\mathcal{R}}, \mathbf{Cat})$ is the category of canonical 2-product preserving 2-functors from $\mathcal{L}_{\mathcal{R}}$ to $\mathbf{Cat}$. This result can be summarized by saying that $\mathcal{L}_{\mathcal{R}}$ does for $\mathcal{R}$-systems what in the $\mathbf{Set}$ case $\mathcal{L}_T$ does for T-algebras, i.e., $\mathcal{L}_{\mathcal{R}}$ extends the result of Lawvere to systems with *state changes*.

More recently, alternative semantics have been proposed for rewriting logic. In [44] it is noticed that when the rules of $\mathcal{R}$ are not right linear — that is, there is a repeated occurrence of a variable in the righthand side of a rule — then $\mathcal{L}_{\mathcal{R}}$ is in a sense too abstract. This is made clear by associating a poset of partial orders of events to $\mathcal{L}_{\mathcal{R}}$ and observing that it is not a prime algebraic domain. A uniform construction for a *sesqui-category* model (similar to $\mathcal{L}_{\mathcal{R}}$ but satisfying fewer equations, and in particular such that the exchange axiom is not imposed) is then provided, and it is shown that its associated poset is a prime algebraic domain. In this way, the relationship between rewriting logic models and event structures is clarified, and useful connections with other concurrency models are provided.

In [118] the treatment of conditional rules in the functorial model for conditional rewriting logic of [99] is generalized and reformulated in terms of *weighted limits* rather than 2-limits, i.e., using *inserters* instead of subequalizers (which however coincide in $\mathbf{Cat}$).

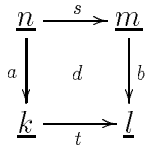
In what follows we restrict ourselves to the semantics proposed by Meseguer [98], and to rewrite theories with the empty set of equations (i.e., $E = \emptyset$).

2.4 Algebraic Tile Logic

The rule format of *algebraic tile systems* [69] extends the one of rewriting systems to deal with observations viewed as basic (unary) actions, in the style of SOS semantics for several process algebras. In this sense, each rule should be considered as a description of a possible behaviour of a *module* depending on the behaviours of its sub-components (i.e., the system evolves if and only if all of its active modules synchronize their actions).

Definition 2.4.1 (Algebraic Tile System)
An algebraic tile system *(*ats*)* $\mathcal{R}$ *is a 4-tuple* $\langle \Sigma_H, \Sigma_V, N, R \rangle$, *where* Σ_H *and* Σ_V *are (one-sorted) signatures,* N *is a set of rule names, and* R *is a function* $R : N \longrightarrow \mathbf{A}(\Sigma_H) \times \mathbf{G}(\Sigma_V) \times \mathbf{G}(\Sigma_V) \times \mathbf{A}(\Sigma_H)$ *such that for all* $d \in N$, *if* $R(d) = \langle s, a, b, t \rangle$, *then we have* $s : \underline{n} \longrightarrow \underline{m}$, $t : \underline{k} \longrightarrow \underline{l}$, $a : \underline{n} \longrightarrow \underline{k}$, *and* $b : \underline{m} \longrightarrow \underline{l}$ *for suitable* $n, m, k, l \in \mathbb{N}$.

We will write such a rule d either as a sequent $d : s \xrightarrow[b]{a} t$, or as the tile



thus making explicit the source and target of each operator. It follows that a term rewriting system is just an ATS with $\Sigma_V = \emptyset$, and a context system [90] is an ATS where Σ_V contains unary operators only and $R : N \longrightarrow \Sigma_H \times \mathbf{G}(\Sigma_V) \times \mathbf{G}(\Sigma_V) \times \Sigma_H$. The actual behaviour of a large system can be recovered from the behaviour of its modules (as specified by the rules of the given ATS) by regarding the rewriting system as a logical theory, and its rules as basic sequents entailed by that theory. Then, some simple inference rules allow us to obtain many other "structured" sequents. A *proof* of a sequent is given by the sequence of inference rules applied to prove it. It is possible to decorate the sequents with *proof terms* to obtain a more concrete framework, in which it is possible to distinguish between different proofs for the same *flat* sequent. To be more concise, we show the decorated version only (the rules for flat sequents are just the same, but without proof terms).

DEFINITION 2.4.2 (ALGEBRAIC TILE LOGIC)
Let $\mathcal{R} = \langle \Sigma_H, \Sigma_V, N, R \rangle$ be an ATS. *We say that $\mathcal{R}$* entails *the class $P_a(\mathcal{R})$ of* decorated algebraic sequents *$\alpha : s \xrightarrow[b]{a} t$, written $\mathcal{R} \vdash_a \alpha : s \xrightarrow[b]{a} t$, obtained by a finite number of applications of the* deduction rules[5] *in Table 2.3. The decoration α is called the* proof term *for its associated sequent $s \xrightarrow[b]{a} t$. Moreover, we say that $\mathcal{R}$* entails *the class $S_a(\mathcal{R})$ of* flat algebraic sequents *$s \xrightarrow[b]{a} t$ (written $\mathcal{R} \vdash_{fa} s \xrightarrow[b]{a} t$) if there exists a decorated algebraic sequent $\alpha \in P_a(\mathcal{R})$ such that $\alpha : s \xrightarrow[b]{a} t$.*

While $P_a(\mathcal{R})$ gives a very precise but too concrete description of $\mathcal{R}$, flat sequents are sometimes too abstract, identifying too much. However, a natural equivalence over proof terms can be expressed by means of a simple set of axioms, in such a way that computationally equivalent derivations are identified.

DEFINITION 2.4.3 (ABSTRACT ALGEBRAIC TILE LOGIC)
Let $\mathcal{R} = \langle \Sigma_H, \Sigma_V, N, R \rangle$ be an ATS. *We say that $\mathcal{R}$* entails *the class $A_a(\mathcal{R})$ of* abstract algebraic sequents, *whose elements are equivalence classes of decorated algebraic sequents in $P_a(\mathcal{R})$ modulo the following set of axioms:*
Associativity Axioms for $\_ \otimes \_$, $\_ * \_$, and $\_ \cdot \_$ *(whenever the sequents α, β and γ can be correctly composed):*

$$\alpha \otimes (\beta \otimes \gamma) = (\alpha \otimes \beta) \otimes \gamma \qquad \alpha * (\beta * \gamma) = (\alpha * \beta) * \gamma \qquad \alpha \cdot (\beta \cdot \gamma) = (\alpha \cdot \beta) \cdot \gamma$$

[5] Notice that the effects of basic tiles are in $\mathbf{G}(\Sigma_V)$, but those of generated tiles are in $\mathbf{M}(\Sigma_V)$.

Basic Proof Sequents. *Generators and Identities:*

$$(gen)\ \frac{d : s \xrightarrow[b]{a} t \in R(N)}{d : s \xrightarrow[b]{a} t \in P_a(\mathcal{R})}$$

$$(v\text{-}ref)\ \frac{a : \underline{n} \longrightarrow \underline{k} \in \mathbf{M}(\Sigma_V)}{1_a : id_n \xrightarrow[a]{a} id_k \in P_a(\mathcal{R})} \qquad (h\text{-}ref)\ \frac{t : \underline{n} \longrightarrow \underline{m} \in \mathbf{A}(\Sigma_H)}{1^t : t \xrightarrow[id_m]{id_n} t \in P_a(\mathcal{R})}$$

Auxiliary Sequents. *Horizontal symmetries, duplicators and dischargers.*

$$(v\text{-}swap)\ \frac{a : \underline{n} \longrightarrow \underline{k},\ b : \underline{m} \longrightarrow \underline{l} \in \mathbf{M}(\Sigma_V)}{\gamma_{a,b} : \gamma_{n,m} \xrightarrow[b \otimes a]{a \otimes b} \gamma_{k,l} \in P_a(\mathcal{R})}$$

$$(v\text{-}dup)\ \frac{a : \underline{n} \longrightarrow \underline{k} \in \mathbf{M}(\Sigma_V)}{\nabla_a : \nabla_n \xrightarrow[a \otimes a]{a} \nabla_k \in P_a(\mathcal{R})} \qquad (v\text{-}dis)\ \frac{a : \underline{n} \longrightarrow \underline{k} \in \mathbf{M}(\Sigma_V)}{!_a : !_n \xrightarrow[id_0]{a} !_k \in P_a(\mathcal{R})}$$

Composition Rules *Parallel, Horizontal and Vertical compositions:*

$$(par)\ \frac{\alpha : s \xrightarrow[b]{a} t \in P_a(\mathcal{R}),\ \beta : s' \xrightarrow[b']{a'} t' \in P_a(\mathcal{R})}{\alpha \otimes \beta : s \otimes s' \xrightarrow[b \otimes b']{a \otimes a'} t \otimes t' \in P_a(\mathcal{R})}$$

$$(hor)\ \frac{\alpha : s \xrightarrow[b]{a} t \in P_a(\mathcal{R}),\ \beta : s' \xrightarrow[c]{b} t' \in P_a(\mathcal{R})}{\alpha * \beta : s ; s' \xrightarrow[c]{a} t ; t' \in P_a(\mathcal{R})}$$

$$(vert)\ \frac{\alpha : s \xrightarrow[b]{a} t \in P_a(\mathcal{R}),\ \beta : t \xrightarrow[b']{a'} t' \in P_a(\mathcal{R})}{\alpha \cdot \beta : s \xrightarrow[b ; b']{a ; a'} t' \in P_a(\mathcal{R})}$$

Table 2.3: Inference rules for decorated algebraic sequents.

Identity Axioms *(for each $\alpha : s \xrightarrow[b]{a} t \in P_a(\mathcal{R})$):*

$$1_a * \alpha = \alpha = \alpha * 1_b \qquad 1^s \cdot \alpha = \alpha = \alpha \cdot 1^t$$

Monoidality Axioms *(for each $s, t \in \mathbf{A}(\Sigma_H)$, $\alpha \in P_a(\mathcal{R})$, and $a, b \in \mathbf{M}(\Sigma_V)$):*

$$1^{s \otimes t} = 1^s \otimes 1^t \qquad 1_{id_0} \otimes \alpha = \alpha = \alpha \otimes 1_{id_0} \qquad 1_{a \otimes b} = 1^a \otimes 1^b$$

Functoriality Axioms:
Identities *(for each $n \in \mathbb{N}$, and composable arrows $s, t \in \mathbf{A}(\Sigma_H)$ and $a, b \in \mathbf{M}(\Sigma_V)$):*

$$1^{s;t} = 1^s * 1^t \qquad 1_{id_n} = 1^{id_n} \qquad 1_{a;b} = 1_a \cdot 1_b$$

Compositions *(whenever both sides are defined):*

$$(\alpha \otimes \beta) \cdot (\gamma \otimes \delta) = (\alpha \cdot \gamma) \otimes (\beta \cdot \delta) \qquad (\alpha \otimes \beta) * (\gamma \otimes \delta) = (\alpha * \gamma) \otimes (\beta * \delta)$$

$$(\alpha * \beta) \cdot (\gamma * \delta) = (\alpha \cdot \gamma) * (\beta \cdot \delta)$$

Auxiliary operators *(for each $n \in \mathbb{N}$, and composable arrows $a, b \in \mathbf{M}(\Sigma_V)$ and $c, d \in \mathbf{M}(\Sigma_V)$):*

$$\gamma_{(a;b),(c;d)} = \gamma_{a,c} \cdot \gamma_{b,d} \qquad \gamma_{id_n, id_m} = 1^{\gamma_{n,m}}$$
$$\nabla_{a;b} = \nabla_a \cdot \nabla_b \qquad \nabla_{id_n} = 1^{\nabla_n}$$
$$!_{a;b} = !_a \cdot !_b \qquad !_{id_n} = 1^{!_n}$$

Naturality Axioms *(for each $\alpha : s \xrightarrow[b]{a} t$, $\alpha' : s' \xrightarrow[b']{a'} t' \in P_a(\mathcal{R})$):*

$$(\alpha \otimes \alpha') * \gamma_{b,b'} = \gamma_{a,a'} * (\alpha' \otimes \alpha) \qquad \alpha * \nabla_b = \nabla_a * (\alpha \otimes \alpha) \qquad \alpha * !_b = !_a$$

Coherence Axioms *(for each $a, b, c \in \mathbf{M}(\Sigma_V)$):*

$$\gamma_{a \otimes b, c} = (1_a \otimes \gamma_{b,c}) * (\gamma_{a,c} \otimes 1_b) \qquad \nabla_a * \gamma_{a,a} = \nabla_a$$
$$\nabla_{a \otimes b} = (\nabla_a \otimes \nabla_b) * (1_a \otimes \gamma_{a,b} \otimes 1_b) \qquad \nabla_a * (1_a \otimes \nabla_a) = \nabla_a * (\nabla_a \otimes 1_a)$$
$$!_{a \otimes b} = !_a \otimes !_b \qquad \nabla_a * (1_a \otimes !_a) = 1_a$$
$$\gamma_{id_0, id_0} = 1_{id_0} = \nabla_{id_0} = !_{id_0} \qquad \gamma_{a,b} * \gamma_{b,a} = 1_{a \otimes b}$$

2.4.1 Tile Bisimulation

Algebraic tile logic allows defining a suitable notion of behavioural equivalence which is reminiscent of the well-known technique of *bisimulation*.

DEFINITION 2.4.4 (TILE BISIMULATION)
Let $\mathcal{R} = \langle \Sigma_H, \Sigma_V, N, R \rangle$ be an ATS. *An equivalence relation*

$$\_ \equiv \_ \subseteq \mathbf{A}(\Sigma_H) \times \mathbf{A}(\Sigma_H)$$

is a tile bisimulation *for $\mathcal{R}$ if, whenever $s \equiv t$ for generic $s, t \in \mathbf{A}(\Sigma_H)$, then for any sequent $s \xrightarrow[b]{a} s'$ entailed by $\mathcal{R}$, there exists $t' \in \mathbf{A}(\Sigma_H)$ such that also $t \xrightarrow[b]{a} t'$ is entailed by $\mathcal{R}$, with $s' \equiv t'$.*

Tile bisimulations are closed under union. The maximal tile bisimulation is called *strong tile bisimulation* and is denoted by $\_ \equiv_{st} \_$. For some conditions on $\mathcal{R}$ ensuring that $\_ \equiv_{st} \_$ is a congruence we refer the interested reader to [69]. An algebraic theory for CCS recovering the ordinary strong congruence via tile bisimulation has been defined in [66, 69]. In Section 7.5.3 we will adapt such a model to give an executable specification in term tile logic.

2.4.2 A Comparison with SOS Rule Formats

Plotkin's SOS approach is particularly suitable to deal with those systems where the structure of the state determines its behaviour: The operational semantics is expressed by a labelled transition system, where the nodes represent the states of the system and the transitions (i.e., the activities of the system) are defined using a mechanism of structural induction that is based on inference rules having the format:

$$\frac{s_1 \xrightarrow{a_1} s_1' \ \cdots \ s_n \xrightarrow{a_n} s_n'}{s \xrightarrow{a} s'}.$$

Transitions in the upper part of the rule are called *premises* (also *assumptions*), whereas the transition in the lower part is called the *conclusion* of the rule.

Usually, each inference rule can be thought of as a *schema* of rules, i.e. has the form:

$$\frac{x_1 \xrightarrow{a_1} x_1' \ \cdots \ x_n \xrightarrow{a_n} x_n'}{C[X] \xrightarrow{a} D[Y]} \text{ for } Pr(a_1, ..., a_n, a)$$

where the x_i and x_i' are variables, a_i, a range over a given set of actions Act, Pr is an $(n+1)$-ary relation over Act, X and Y are sets of variables, and C and D are contexts, containing any number of times each variable. The idea is that the schema applies to any instantiation of the variables to (ground) states. As a matter of notation, when dealing with process algebras, we will often denote variables by P, P_1, Q, Q',... and call them *process variables*.

2.4.2.1 Algebraic Tile Format, De Simone Format, and GSOS Format

The format of tiles allowed by algebraic tile systems is called *algebraic tile format* and corresponds to SOS rules of the form:

$$\frac{P_i \xrightarrow{a_i} Q_i, \text{ for } i \in I}{C[P_1, ..., P_n] \xrightarrow{a} D[Q_1, ..., Q_n]} \text{ for } Pr(a_1, ..., a_n, a)$$

where $I \subseteq \{1, ..., n\}$, a_i, a range over a given set of actions Act, and Pr is an $(n+1)$-ary relation over Act. C and D are process contexts, containing any number of times each process variable, and all the P_i's, Q_j's are different, except for $P_k = Q_k$ with $k \notin I$.

Therefore, it extends the *De Simone format*, where the rules have the form

$$\frac{P_i \xrightarrow{a_i} Q_i, \text{ for } i \in I}{f(P_1, ..., P_n) \xrightarrow{a} D[Q_1, ..., Q_n]} \text{ for } Pr(a_1, ..., a_n, a)$$

with the same restrictions as before, but where f is a n-ary operator (and not a generic context) and the Q_i's can be used *at most once* in the context D.

However, notice that both formats do not allow the *reuse* of a variable P_i tested in the premises. Also multiple tests of the same variable are forbidden by both formats. Hence they are less general than the *GSOS format* [11].

EXAMPLE 2.4.1 *An axiom such as*

$$\frac{}{\delta P \xrightarrow{\tau} P||\delta P}$$

which represents the spawning with replication *of a process, is in the algebraic tile format, but not in the De Simone format.*

EXAMPLE 2.4.2 *The GSOS rule modelling* guarded *replication*

$$\frac{P \xrightarrow{a} Q}{!P \xrightarrow{a} Q|!P}$$

is neither in the algebraic format, nor in the De Simone format.

Part I

Zero-Safe Nets

Abstract of Part I

The main feature of *zero-safe nets* is a primitive notion of *transition synchronization*. To this aim, besides ordinary places, called *stable* places, zero-safe nets are equipped with *zero* places, which in an observable marking cannot contain any token. This yields the notion of *net transaction*: a basic atomic computation, which may use zero tokens as triggers, but defines an evolution between observable markings only. Transactions satisfy two main requirements: 1) they model interacting activities which cannot be decomposed into disjoint sub-activities, and 2) they do not consume stable tokens previously generated in the same transaction.

The abstract counterpart of a generic zero-safe net B consists of an ordinary P/T net whose places are the stable places of B, and whose transitions represent the transactions of B. The two nets offer both the refined and the abstract model of the same system, where the former can be much smaller than the latter, because of the transition synchronization mechanism.

Depending on the chosen approach — *collective* vs *individual token philosophy* — two notions of transaction may be defined, each one leading to different operational and abstract models. Their comparison is fully discussed on the basis of a *multicasting system* example.

In Chapter 4, following the *Petri nets are monoids* approach of [106], we make use of category theory to analyze and motivate the framework presented in Chapter 3. More precisely, the two operational semantics of zero-safe nets are characterized as adjunctions, and the derivation of abstract P/T nets as coreflections. Since left adjoints preserve colimits, and right adjoints preserve limits, it follows that several constructions in the refined model are preserved by its operational and abstract semantics.

We conclude by remarking the similarities between zero-safe nets and a simple class of tile systems.

The results presented in this part are joint work with Ugo Montanari.

And now for something completely different.

— MONTY PYTHON, Monty Python Flying Circus

Chapter 3

Transition Synchronization

Contents

3.1 Motivations

Petri nets [123] are unanimously regarded as one of the most attractive models of concurrency. As a matter of fact, this model offers a basic concurrent framework that has often been used as a semantic foundation for system analysis and for the interpretation of many concurrent languages [136, 72, 119, 51, 75, 10]. However, though net transitions allow for *token* synchronization, the basic net model does not offer any synchronization mechanism among transitions, while this feature is essential to write modular and expressive programs. In fact, all the above translations involve complex constructions for the net defining the synchronized composition of two programs

Zero-safe nets (also *ZS nets*), extend Petri nets along this direction, coming equipped with a simple but very general notion of transition synchronization as a built-in feature. ZS nets are based on the notion of *zero* places. Tokens which are produced in a zero place force the firing of transitions which are able to consume them. In addition to zero places, a distinguished set of *stable* places is also present. Stable markings (consisting only of *stable tokens*, i.e., tokens in stable places) define the observable states of the system, whilst non-stable markings (those involving *zero tokens*, i.e., tokens in zero places) are the hidden states of the refined model, in the sense that they are not observable at the abstract level. Therefore, any synchronized evolution of a ZS net starts at some observable marking, then evolves through hidden states and eventually terminates into a new observable state.

A *refined* ZS net and an *abstract* P/T net are supposed to model the same given system. The former specifies how every transition of the latter is actually achieved as a different coordinated collection of firings (called *transaction*) and the latter offers the synchronized view, which corresponds to the abstraction from the hidden mechanism that controls the firings through zero tokens. This should favour a uniform approach to concurrent languages translation. For instance, in the case of process algebras in the CCS style [111], the parallel composition of two nets modelling communicating processes involves the combinatorial analysis of all the admissible synchronizations, whereas if zero places were used to model communication channels, then the parallel composition of two nets would just merge the common channels.

We remark that in our setting the term "transaction" denotes a certain activity of the system that might be composed by many, possibly concurrent, coordinated subactivities. Moreover, we require that at the right level of abstraction, where transactions are considered as atomic activities of the system, all the intermediate states (except the initial and the final states) must be unobservable. Since the concurrent semantics of an operational model is usually defined by considering as equivalent all the computations where the same concurrent events are executed in different orders, it follows that we should identify those transactions which are equivalent from a concurrent viewpoint, in such a way that the actual order of execution of concurrent transitions in the refined net is invisible in the abstract net.

Having this in mind, a real dichotomy runs on the distinction between *collective* and *individual token philosophies* (see e.g., Van Glabbeek and Plotkin [71] and also [26, 27] where it is explained how the two approaches can influence the *categorical*, *logical* and *behavioural* semantics of ordinary Petri nets).

The simplest approach relies on the *collective token philosophy*, *CTph* for short, according to which net semantics should not distinguish among different instances of the idealized resources (i.e., the tokens) that rule the basics of net behaviour. This is true, of course, only if any such instance is *operationally* equivalent to all the others. This school of thought identifies all those firing sequences that are obtained by repeatedly permuting pairs of (adjacent) concurrently enabled firings. We call *abstract stable transactions* the resulting equivalence classes of basic evolutions of the refined net. As a major drawback, this approach disregards that operationally equivalent resources may have different origins and histories, and may, therefore, carry different *causality* information (e.g., selecting one instance of a resource rather than another, may be as different as being or not being causally dependent on some previous event). Therefore, causal dependencies on zero tokens are lost. It follows that the class of computations captured by abstract nets may turn out to be too abstract for many applications.

An alternative approach is based on the *individual token philosophy*, *ITph* for short. According to the *ITph*, causal dependencies are a central aspect in the dynamic evolution of a net. As a consequence, only the transactions which refer to isomorphic Goltz-Reisig processes [74] are identified, and we call *connected transactions* the induced equivalence classes. In this case, the actual order of execution of concurrent transitions in the refined net is invisible in the abstract net, but all the causal dependencies are preserved.

In many cases, the distinction between the *CTph* and the *ITph* arises more from an academic debate than from some real issue. On the contrary, in the case of ZS nets, we want to emphasize that the distinction between the two philosophies is motivated also from a pragmatic perspective. In particular, depending on the chosen approach, two different notions of transaction may be defined, each leading to different operational and abstract models.

3.1.1 The Multicasting System Example

We will use the ZS net *MS* pictured in Figure 3.1 as a running example throughout Chapters 3 and 4.

Notice that we extend the standard graphical representation for nets (where boxes stand for transitions, circles for places, dots for tokens, and oriented arcs for the flow relation) by using smaller circles to represent zero places.

The ZS net *MS* will be useful to illustrate some instances of the definitions. In addition, it will serve as a basis to better illustrate and understand the differences between the *CTph* and the *ITph* approaches.

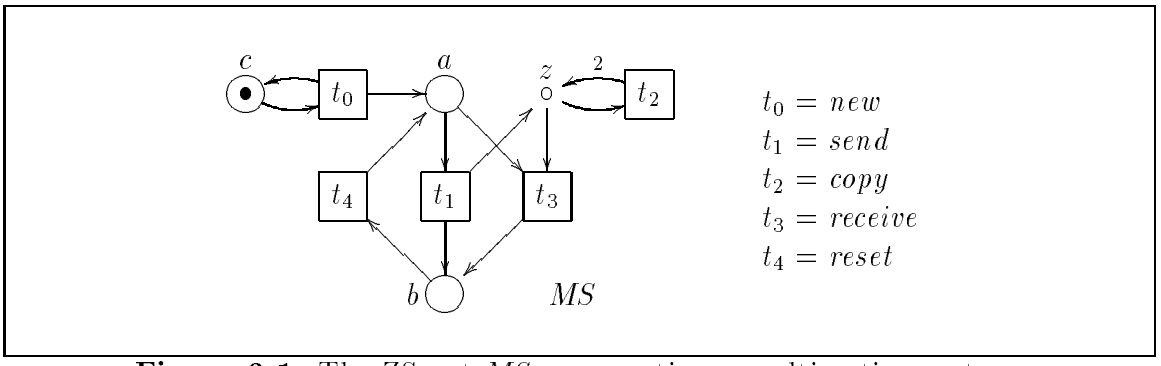


Figure 3.1: The ZS net *MS* representing a multicasting system.

The net *MS* is intended to represent a *multicasting system.*

As in a broadcasting system, an agent can simultaneously send the same message to an unlimited number of receivers, but here the receivers are not necessarily all the remaining agents, and thus, several one-to-many communications can take place concurrently.

Each token in place a represents a different *active* agent (i.e., an agent which is ready to communicate), while the tokens in place b are *inactive* agents. The zero place z models a buffer in which each token is a message.

The transition *new* permits to create an unlimited number of agents. Each firing of *send* opens a one-to-many communication: a message is put in the buffer z and the agent which started the communication is suspended until the end of the current transaction. Each time the transition *copy* fires, a new copy of the same message is created. To complete a transaction, as many firings of transition *receive* are needed as the number of copies created by *copy* plus one. Each firing of the transition *receive* synchronizes an active agent with a copy of the message and then suspends the agent. At the end of a session, all the suspended agents are moved into place b. The transition *reset* activates an inactive agent.

We call *copy policy* any strategy for making copies of the messages in the buffer z. For instance, the *sequential one-to-n copying* policy acts as follows: once the message is produced, a firing of *copy* creates two copies of the message in the buffer, one of them is retrieved by the firing of *receive*, the remaining message is duplicated again by a second firing of *copy*, one of the two messages is retrieved by a second agent via a second firing of *receive*, and so on, until the n-th agent consumes the last copy of the message, i.e., the firing sequence starts with *send*, then repeats $n-1$ times *copy* followed by *receive*, and is concluded by the n-th *receive*.

As another example, the *parallel one-to-n copying* policy begins with *send*, then creates all the needed copies of the message by firing *copy* with the maximal parallelism allowed (e.g., for $n = 8$, first *copy* fires, then two instances of *copy* fire concurrently, producing four copies of the message, then four instances of *copy* fire concurrently), and at last, the n agents retrieve the n copies concurrently (by firing n instances of *receive*).

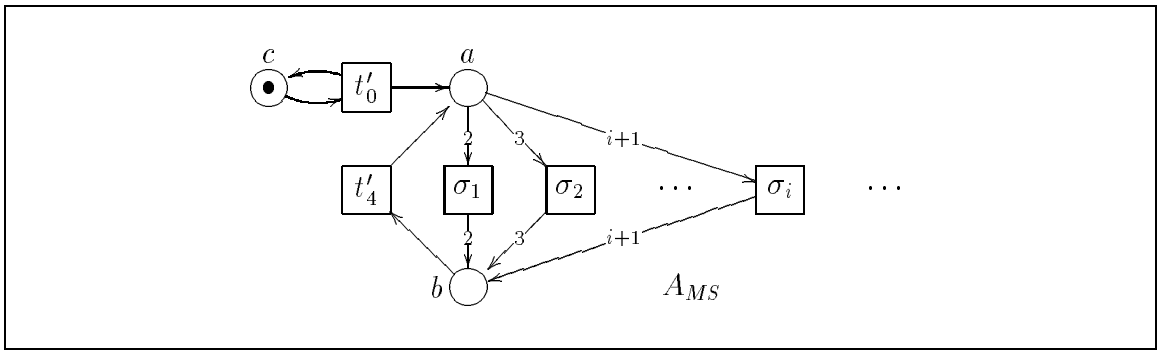


Figure 3.2: The abstract net for the multicasting system under the *CTph*.

As suggested before, the ZS net *MS* may be used as an example to better illustrate the difference between the *CTph* and the *ITph*. In the first case, transactions are distinguished only if they differ for the number of involved agents, whereas in the second case also different copy policies involving the same number of agents can be distinguished.

In Figure 3.2 we see the infinite abstract P/T net A_{MS} for the refined ZS net *MS*, according to the *CTph* (see Definition 3.4.5).

As it will be explained in Section 4.3.2, the *abstract* net A_{MS} comes equipped with a *refinement morphism* ϵ^C_{MS} to the refined net *MS*. The refinement morphism maps each place of A_{MS} into the homonymous stable place of *MS* and defines a bijection between the transitions of A_{MS} and (the equivalence classes under the *CTph* of) the transactions of *MS*. Therefore, the transition σ_n of A_{MS} represents a one-to-n transmission, i.e., it represents any transaction which corresponds to a one-to-n transmission.

For instance, provided that we have reached a marking with at least five active agents, the sequential copying one-to-four transmission sc_4 defined as

$$sc_4 \quad = \quad \textit{send-copy-receive-copy-receive-copy-receive-receive}$$

and the parallel copying one-to-four transmission bc_4 defined as

$$bc_4 \quad = \quad \textit{send-copy-}\{\textit{copy,copy}\}\textit{-}\{\textit{receive,receive,receive,receive}\}$$

(where we have used a self-explanatory informal notation to denote sequences of firings and concurrent firings), are in the same equivalence class. In fact, it is possible to repeatedly apply some diamond transformation to transaction sc_4 to derive the transaction

$$\textit{send-copy-copy-copy-receive-receive-receive-receive},$$

as well as any other linearization of bc_4.

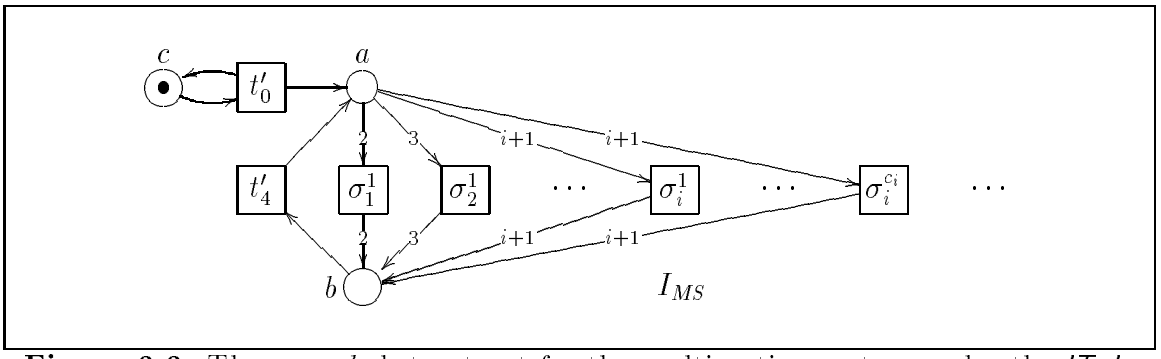


Figure 3.3: The *causal* abstract net for the multicasting system under the *ITph*.

On the contrary, if we adopt the *ITph*, then different copy policies may be distinguished. The infinite *causal abstract* P/T net I_{MS} corresponding to the refined ZS net *MS* under the *ITph* (see Definition 3.5.6) is displayed in Figure 3.3.

As it will be explained in Section 4.4.2, the net I_{MS} comes equipped with a *causal refinement morphism* ϵ^I_{MS} to the refined net *MS*. Such morphism maps each place of I_{MS} into the homonymous stable place of *MS*, and defines a bijection between the transitions of I_{MS} and (the equivalence classes under the *ITph* of) the transactions of *MS*. We assume that the generic transition σ^k_n corresponds to the one-to-n transmission that follows the k-th codified copy policy (for each one-to-n transmission there are c_n different copy policies). As an example transitions σ^1_4 and σ^2_4 denote the equivalence classes of sc_4 and bc_4 that are no longer equivalent.

Zero places can be used to coordinate and synchronize in a single transaction any number of transitions of the refined net. Thus it may well happen that the refined net is finite while the abstract net is infinite. This is the case, for instance, for our running example, which models a multicasting system where a message can be delivered to an unlimited number of receivers.

Notice also that the abstract and the refined net both rely on the same basic token-pushing mechanism which is used to express their behaviours. This similarity is the key of the constructions described in Chapter 4.

3.1.2 Process Algebra Encodings

Though we believe that the multicasting system representation offers a good example about the expressivity of ZS nets, another interesting application presented in this chapter concerns the compositional representation of a simple process algebra equipped with action prefix, parallel composition and restriction operators (both for communications in the CCS-style and in the CSP-style).

In particular, we map each agent in a ZS net (equipped with a sort of *interface*) such that its abstract net is able to (bi)simulate the behaviour of the agent.

Moreover, the hiding of local names to the external world is handled very well, because agents that are equivalent up to α-conversion yield isomorphic nets.

3.1.3 Structure of the Chapter

After briefly recalling in Section 3.2 some basic definitions of net theory, we present ZS nets and their operational and abstract semantics throughout Section 3.3, where the simpler *CTph* approach is analyzed first.

The operational semantics for ZS nets in the *CTph* is defined in Section 3.4.1, while Section 3.4.2 concerns the related abstract semantics. In this context, the basic evolutions of ZS nets are equivalence classes of ordinary firing sequences induced by diamond transformation, and are called *abstract stable transactions*. To illustrate the *ITph* versions, in Section 3.5.1 we introduce the notion of *causal firing sequence*, which allows for a concise representation of concatenable processes, and has a suggestive implementation on a machine whose states are collections of token stacks. Then, we define the evolution of a ZS net in terms of the equivalence classes of causal firing sequences induced by isomorphic underlying processes. As for the *CTph* based approach, in Section 3.5.2, ordinary P/T nets are defined as the abstract counterparts of ZS nets. The translation of a simple process algebra into zero-safe nets concludes the chapter.

In Chapter 4 we will employ category theory to give evidence that the definitions and the constructions presented in this chapter are the best possible choices. Related approaches to the refinement and the abstraction of nets are discussed at the end of Chapter 4.

3.2 Background: Petri Nets

Definition 3.2.1 (Net)
A net N *is a triple* $N = (S_N, T_N; F_N)$ *where*

- S_N *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*
- $F_N \subseteq (S_N \times T_N) \cup (T_N \times S_N)$ *is called the* flow relation. *The elements of the flow relation* F_N *are called* arcs *and we write* xF_Ny *for* $(x, y) \in F_N$.

We will denote $S_N \cup T_N$ by N whenever no confusion arises, and subscripts will be omitted if they are obvious from the context.

For $x \in N$, the set ${}^\bullet x = \{y \in N \mid yFx\}$ is called the *pre-set* of x, and the set $x^\bullet = \{y \in N \mid xFy\}$ is called the *post-set* of x. We only consider nets such that for any transition t, ${}^\bullet t \neq \emptyset$. Moreover, let ${}^\circ N = \{x \in N \mid {}^\bullet x = \emptyset\}$ and $N^\circ = \{x \in N \mid x^\bullet = \emptyset\}$ denote the sets of *initial* and *final elements* of N respectively. A place a is said to be *isolated* if ${}^\bullet a \cup a^\bullet = \emptyset$.

Definition 3.2.2 (P/T net)
A marked place/transition net *is a 5-tuple* $N = (S, T; F, W, u_{\text{in}})$ *such that* $(S, T; F)$ *is a net, the function* $W : F \longrightarrow \mathbb{N}$ *assigns a positive* weight *to each arc in* F*, and the finite multiset* $u_{\text{in}} : S \longrightarrow \mathbb{N}$ *is the* initial marking *of* N.

We find it convenient to interpret the relation F as a function

$$F : ((S \times T) \cup (T \times S)) \longrightarrow \{0, 1\},$$

assuming $xFy \iff F(x, y) \neq 0$. Then, for nets with weighted arrows we may safely replace $\{0, 1\}$ by $\mathbb{N}$. Thus, relation F becomes a *multiset* relation

$$F : (S \times T) \cup (T \times S) \longrightarrow \mathbb{N}$$

and W is discharged.

A *marking* $u : S \longrightarrow \mathbb{N}$ is a finite multiset of places. It can be written either as $u = \{n_1 a_1, ..., n_k a_k\}$ where the natural number $n_i > 0$ (if $n_i = 0$ then the corresponding term $n_i a_i$ is safely omitted) dictates the number of occurrences (*tokens*) of the place a_i in u, i.e., $n_i = u(a_i)$, or as a formal sum $u = \bigoplus_{a_i \in S} n_i a_i$ denoting an element of the commutative monoid $S^{\oplus}$ freely generated from the set of places S (the order of summands is immaterial and the monoidal composition is defined by taking $(\bigoplus_i n_i a_i) \oplus (\bigoplus_i m_i a_i) = (\bigoplus_i (n_i + m_i) a_i)$ and 0 as the neutral element). For any transition $t \in T$, let $pre(t)$ and $post(t)$ be the multisets over S such that $pre(t)(a) = F(a, t)$ and $post(t)(a) = F(t, a)$, for all $a \in S$.

The evolution of a net (i.e., its interleaving behaviour) is usually described in terms of firing sequences.

Definition 3.2.3 (Enabling, firing and firing sequence)
Given a P/T net N, let u and u' be markings of N. Then a transition $t \in T_N$ is enabled *at u if $pre(t)(a) \leq u(a)$, for all $a \in S_N$.*

Moreover, we say that u evolves to u' under the firing *of t, written $u[t\rangle u'$, if and only if t is enabled at u and $u'(a) = u(a) - pre(t)(a) + post(t)(a)$, for all $a \in S$.*

A firing sequence *from u_0 to u_n is a sequence of markings and firings such that $u_0[t_1\rangle u_1 ... u_{n-1}[t_n\rangle u_n$.*

Given a marking u of N the set $[u\rangle$ of its *reachable markings* is the smallest set of markings such that $u \in [u\rangle$, and moreover, for any $u' \in [u\rangle$ such that $u'[t\rangle u''$ for some transition t, then $u'' \in [u\rangle$.

Definition 3.2.4 (Reachable markings of N)
The reachable markings *of the net $N = (S, T; F, u_{\text{in}})$ are the elements of the set $[u_{\text{in}}\rangle$, i.e., the markings that are reachable from the initial marking of N.*

Besides firings and firing sequences, *steps* and *step sequences* are usually introduced. A step allows for the simultaneous execution of several concurrently enabled transitions, i.e., the execution of a multiset of transitions.

A multiset $X : T \longrightarrow \mathbb{N}$ is enabled at u if

$$\sum_{t \in T} X(t) \cdot pre(t)(a) \leq u(a),$$

for all $a \in S_N$.

We say that u evolves to u' under the *step* X, written $u[X\rangle u'$, if and only if X is enabled at u and

$$u'(a) = u(a) + \sum_{t \in T} X(t) \cdot (post(t)(a) - pre(t)(a)),$$

for all $a \in S$.

To deal with a more informative semantic account of the causal relationships between firings (e.g., which transitions produced the tokens consumed by a certain transition) the notion of *process* has been introduced by Goltz and Reisig.

DEFINITION 3.2.5 (CAUSAL NET AND PROCESS)
A net K is a causal net *(also called* deterministic occurrence net*) if*

- $\forall a \in S_K,\ |{}^\bullet a| \leq 1 \wedge |a^\bullet| \leq 1$ *and*
- F_K^* *is acyclic,*[1] *i.e.,* $\forall x, y \in K,\ xF_K^*y \wedge yF_K^*x \Longrightarrow x = y$.

A (Goltz-Reisig) process *for a P/T net N is a mapping $P : K \longrightarrow N$, from a causal net K to N, such that*

- $P(S_K) \subseteq S_N$, $P(T_K) \subseteq T_N$,
- ${}^\circ K \subseteq S_K$, *and*
- $\forall t \in T_K,\ \forall a \in S_N,\ F_N(a, P(t)) = |P^{-1}(a) \cap {}^\bullet t| \wedge F_N(P(t), a) = |P^{-1}(a) \cap t^\bullet|$.

As usual we denote the set of *origins* (i.e., minimal or initial places) and *destinations* (i.e., final or maximal places) by $O(K) = {}^\circ K$ and $D(K) = K^\circ \cap S_K$, respectively.

Two processes P and P' of N are *isomorphic* and thus identified if there exists an isomorphism $\psi : K_P \longrightarrow K_{P'}$ such that $\psi; P' = P$.

Concatenable processes [49, 50, 126] are obtained from processes by imposing a total ordering on the origins that are instances of the same place and, similarly, on the destinations.

DEFINITION 3.2.6 (CONCATENABLE PROCESS)
Given a set S with a labelling function $l : S \longrightarrow S'$, a label-indexed ordering function *for l is a family $\beta = \{\beta_a\}_{a \in S'}$ of bijections, where $\beta_a : l^{-1}(a) \longrightarrow \{1, \ldots, |l^{-1}(a)|\}$.*

A concatenable process *for a P/T net N is a triple $C = (P, \mathscr{o}, \ell^\circ)$ where*

- $P : K \longrightarrow N$ *is a process for N and*
- $\mathscr{o}$, ℓ° *are label-indexed ordering functions for the labelling function P restricted to $O(K)$ and $D(K)$, respectively.*

[1] F^* denotes the reflexive and transitive closure of relation F.

Two concatenable processes C and C' are *isomorphic* if P_C and $P_{C'}$ are isomorphic via a morphism that preserves all the orderings.

A partial binary operation $\_;\_$ (associative up to isomorphism and with identities) of concatenation of concatenable processes (whence their names) can be easily defined: we take as source (target) the image through P of the initial (maximal) places of K_P; then the composition of $C = (P, \ell, \ell^\circ)$ and $C' = (P', \ell', \ell'^\circ)$ is realized by merging, when it is possible, the maximal places of K_P with the initial places of $K_{P'}$ according to their labelling and ordering functions so to match these places one-to-one.

Concatenable processes admit also a monoidal *parallel* composition $\_ \otimes \_$, which can be represented by putting two processes side by side. We refer the interested reader to [50] for the formal definitions of $C; C'$ and $C \otimes C'$. The concatenable processes of a net yield a symmetric strict monoidal category (see Definition 2.1.9).

We conclude this introductory section by recalling the notion of *safety*, which plays an important rôle in Net Theory.

DEFINITION 3.2.7 (n-SAFE NETS)
A place is n-safe *if it contains at most* n *tokens in any reachable marking. A net is* n*-safe if all its places are* n*-safe. A net is* safe *if a bound can be given for the number of tokens in each place (for all reachable markings), i.e., if there exists* $n \in \mathbb{N}$ *such that for all* $u \in [u_{\text{in}}\rangle$*, and for all* $a \in S$*, we have* $u(a) \leq n$.

3.3 Zero-Safe Nets

We augment P/T nets with special places called zero places. Their role is to coordinate the atomic execution of complex collections of transitions. From an abstract viewpoint, all the coordinated transitions will appear as being synchronized. However no new interaction mechanism is needed, and the coordination of the transitions participating in a step is handled by the ordinary token-pushing rules of nets. Notice that, whereas in the standard terminology a n-safe net is a net whose places are all n-safe, in the case of zero-safe nets only a subset of places (the zero places) are required to satisfy a 0-safe condition w.r.t. reachable markings.

DEFINITION 3.3.1 (ZS NET)
A zero-safe net *(also* ZS net *for short) is a 5-tuple* $B = (S_B, T_B; F_B, u_B; Z_B)$ *where* $N_B = (S_B, T_B; F_B, u_B)$ *is the* underlying *P/T net and the set* $Z_B \subseteq S_B$ *is the set of* zero places. *The places in* $S_B \setminus Z_B$ *are called* stable places. *A* stable marking *is a multiset of stable places, and the initial marking of* B *must be stable.*

Stable markings describe *observable* states of the system, whereas the presence of one or more zero tokens in a given marking marks it as internal and hence unobservable. We call *stable tokens* and *zero tokens* the tokens that respectively belong to stable places and to zero places.

3.4 Collective Token Approach

3.4.1 Operational Semantics

A *stable step* of a ZS net B may involve the execution of several transitions of the underlying P/T net N_B (it is actually a firing sequence of N_B). At the beginning, the state must contain enough stable (i.e., nonzero) tokens to enable all these transitions at the same time. As the computation progresses, the firings can only consume the stable tokens that were also available at the beginning of the computation and the zero tokens that have been produced by some fired transition. The stable tokens that are produced by some firings cannot be consumed in the same transaction. However, no token can be left on zero places at the end of the computation (nor can belong to the starting configuration). A stable step whose intermediate markings are all not stable and which consumes all the available stable tokens is called *stable transaction*. *Stable step sequences* are sequences of stable steps.

In a certain sense, each stable step can be thought of as a collection of transactions plus a collection of idle resources; this means that once the basic transactions are known, then all the *correct* behaviours of the system may be derived. We ask the reader to keep in mind this observation because it constitutes the basis for our approach.

DEFINITION 3.4.1 (STABLE STEP, TRANSACTION AND STEP SEQUENCE)
Let B be a ZS net and let $s = u_0[t_1\rangle u_1 \ldots u_{n-1}[t_n\rangle u_n$ be a firing sequence of the underlying net N_B of B. Sequence s is a stable step *of B if:*

- $\forall a \in S_B \setminus Z_B,\ \sum_{i=1}^{n} pre(t_i)(a) \leq u_0(a)$ *(concurrent enabling);*
- *u_0 and u_n are stable markings of B (stable fairness).*

We write $u_0\{\!\![s\rangle u_n$ to denote the stable step s, and $O(s)$ and $D(s)$ to denote the markings u_0 and u_n respectively. A stable step s is a stable transaction *of B if in addition:*

- *markings $u_1, \ldots, u_{n-1}$ are not stable (atomicity);*
- $\forall a \in S_B \setminus Z_B,\ \sum_{i=1}^{n} pre(t_i)(a) = u_0(a)$ *(perfect enabling).*

A stable step sequence *is a sequence $u_0\{\!\![s_1\rangle u_1 \ldots u_{n-1}\{\!\![s_n\rangle u_n$. We also say that u_n is* reachable *from u_0 and we write $u_n \in \{\!\![u_0\rangle$. Sometimes we will denote the set $\{\!\![u_B\rangle$ of* reachable (stable) markings *of B by $\{\!\![B\rangle$.*

In a stable transaction, each transition represents a micro-step carrying on the atomic evolution through invisible states. Stable tokens produced during the transaction become operative in the system only at the end of the transaction (i.e., after the firing of the *commit* transition t_n).

EXAMPLE 3.4.1 *Consider the ZS net MS of Figure 3.1.*

The firing sequence

$$\{2a\}[t_1\rangle\{a,b,z\}[t_4\rangle\{2a,z\}[t_3\rangle\{a,b\}$$

is not a stable step since the concurrent enabling condition is not satisfied (indeed, $pre(t_1)(b) + pre(t_4)(b) + pre(t_3)(b) = 1 \not\leq 0$*).*

The firing sequence

$$\{4a\}[t_1\rangle\{3a,b,z\}[t_2\rangle\{3a,b,2z\}[t_3\rangle\{2a,2b,z\}[t_3\rangle\{a,3b\}$$

is a stable step but not a stable transaction since the perfect enabling condition is not satisfied ($pre(t_1)(a) + pre(t_2)(a) + 2pre(t_3)(a) = 3 < 4$*).*

The firing sequence

$$s' = \{2a,b\}[t_1\rangle\{a,2b,z\}[t_3\rangle\{3b\}[t_4\rangle\{a,2b\}$$

is a stable step but not a stable transaction since the atomicity constraint is not satisfied (the inner marking $\{3b\}$ *is stable).*

The firing sequence

$$s'' = \{2a,b\}[t_1\rangle\{a,2b,z\}[t_4\rangle\{2a,b,z\}[t_3\rangle\{a,2b\}$$

is a stable transaction (as opposed to the first sequence of this example that starts from the marking $2a$ *mimicking* s''*, but it is not a stable transaction).*

The concurrent semantics of an operational model is usually defined by considering as equivalent all the computations where the same concurrent events are executed in different orders. In the case of P/T nets, the simplest approach relies on the collective token philosophy, which forces the identification of all the firing sequences obtained by repeatedly permuting pairs of firings which are concurrently (i.e., independently) enabled [97]. Thus, Definition 3.4.2 allows for a more satisfactory notion of stable step (transaction) in a concurrent setting.

An alternative approach, following the individual token philosophy, is presented in Section 3.5.

DEFINITION 3.4.2 (DIAMOND TRANSFORMATION)
Given a P/T net N*, let*

$$s = u_0[t_1\rangle u_1 \cdots u_{i-1}[t_i\rangle u_i[t_{i+1}\rangle u_{i+1} \cdots u_{n-1}[t_n\rangle u_n$$

be a firing sequence of N*, where* t_i *and* t_{i+1} *are concurrently enabled by* u_{i-1}*, i.e.,* $pre(t_i)(a) + pre(t_{i+1})(a) \leq u_{i-1}(a)$ *for any place* a*. Let* s' *be the firing sequence obtained by permuting the firing ordering of* t_i *and* t_{i+1}*, i.e.:*

$$s' = u_0[t_1\rangle u_1 \cdots u_{i-1}[t_{i+1}\rangle u_i'[t_i\rangle u_{i+1} \cdots u_{n-1}[t_n\rangle u_n.$$

The sequence s' *is* a diamond transformation *of* s.

The reflexive and transitive closure of the relation induced by diamond transformations gives the natural equivalence in the *CTph* interpretation. Notice that all the equivalent sequences have the same first and last markings u_0 and u_n.

DEFINITION 3.4.3 (ABSTRACT SEQUENCE)
Equivalence classes of sequences (w.r.t. diamond transformations) are called abstract sequences *and are denoted by* σ. *The abstract sequence of* s *is written* $[\![s]\!]$. *We also write* $pre([\![s]\!]) = O(s)$ *and* $post([\![s]\!]) = D(s)$ *to denote the origins and the destinations of* $[\![s]\!]$, *respectively.*

DEFINITION 3.4.4 (ABSTRACT STABLE STEP AND TRANSACTION)
Given a ZS net B, *an* abstract stable step *is an abstract sequence* $[\![s]\!]$ *of the underlying net* N_B, *where* s *is a stable step. An* abstract stable transaction *is an abstract sequence of* N_B *which contains only stable transactions of* B. *We denote by* Υ_B *the set of all abstract stable transactions of* B.

As a matter of fact, the equivalence induced by diamond transformations preserves stable steps (because the diamond transformation preserves the properties of concurrent enabling and of stable fairness that are required by Definition 3.4.1) but not stable transactions. Generally speaking, the problem is that two stable transactions that are concurrently enabled could be interleaved in such a way that the resulting sequence also is a stable transaction. Of course, such transaction cannot be considered as a representative of an atomic activity of the system, because it can be expressed in terms of two sub-activities. Thus, it is not enough to require s to be a stable transaction to make sure that $[\![s]\!]$ is an abstract stable transaction, and we need a stronger constraint, namely that all the sequences in the equivalence class are stable transactions.

EXAMPLE 3.4.2 *As a counterexample showing that stable transactions are not preserved by our equivalence, consider the net MS defined in Figure 3.1 and the stable steps* s' *and* s'' *defined in Example 3.4.1. It is easy to verify that* $[\![s']\!] = [\![s'']\!]$, *since* s'' *is obtained from* s' *by a diamond transformation. However* s'' *is a stable transaction whereas* s' *is not. Thus* $[\![s'']\!]$ *is not an abstract stable transaction.*

Conversely, the firing sequence

$$s = \{4a\}[t_1\rangle\{3a,b,z\}[t_2\rangle\{3a,b,2z\}[t_2\rangle\{3a,b,3z\}[t_3\rangle\{2a,2b,2z\}[t_3\rangle\{a,3b,z\}[t_3\rangle\{4b\}$$

defines an abstract stable transaction $[\![s]\!]$. *In fact the stable transaction*

$$\overline{s} = \{4a\}[t_1\rangle\{3a,b,z\}[t_2\rangle\{3a,b,2z\}[t_3\rangle\{2a,2b,z\}[t_2\rangle\{2a,2b,2z\}[t_3\rangle\{a,3b,z\}[t_3\rangle\{4b\}$$

is the unique diamond transformation of s *(and vice versa).*

3.4.2 Abstract Semantics

According to the *CTph*, since the basic execution steps of a system modelled via ZS nets consist of abstract stable transactions, it is natural to define a high-level description of such a model as a net whose transitions are abstract stable transactions.

DEFINITION 3.4.5 (ABSTRACT NET)
Let $B = (S_B, T_B; F_B, u_B; Z_B)$ be a ZS net. The net $A_B = (S_B \setminus Z_B, \Upsilon_B; F, u_B)$, with $F(a,\sigma) = pre(\sigma)(a)$ and $F(\sigma,a) = post(\sigma)(a)$, is the abstract net *of B (we recall that $pre(\sigma)$ and $post(\sigma)$ yield the first and last marking of any stable transaction in the equivalence class σ, and that Υ_B is the set of all the abstract stable transactions of B).*

EXAMPLE 3.4.3 *Let MS be the ZS net of our running example and let $\{c\}$ be its initial marking. Consider the following firing sequences of the underlying net N_{MS} of MS:*

$$
\begin{aligned}
s_{new} &= \{c\}[t_0\rangle\{a,c\},\\
s_{res} &= \{b\}[t_4\rangle\{a\},\\
s_1 &= \{2a\}[t_1\rangle\{a,b,z\}[t_3\rangle\{2b\},\\
&\cdots\\
s_i &= \{(i+1)a\}[t_1\rangle\{ia,b,z\}[t_2\rangle\cdots[t_2\rangle\{ia,b,iz\}[t_3\rangle\cdots[t_3\rangle\{(i+1)b\},\\
&\cdots
\end{aligned}
$$

where s_i has $i-1$ firings of t_2 and i firings of t_3.

Thus $\Upsilon_{MS} = \{t'_0, t'_4, \sigma_1, \ldots, \sigma_i, \ldots\}$ with $t'_0 = [\![s_{new}]\!]$, $t'_4 = [\![s_{res}]\!]$ and $\sigma_i = [\![s_i]\!]$, for $i \geq 1$. The (infinite) abstract net A_{MS} of MS is (partially!) pictured in Figure 3.2. This abstract net consists of three places and infinitely many transitions: one for creating a new active process, one for reactivating a process after a synchronization, and one for each possible multicasting communication involving i receivers.

3.5 Individual Token Approach

In this section, the basic activities of ZS nets are defined accordingly to the *ITph*, instead of the *CTph*. This choice has a great impact on the resulting notion of transaction, yielding dramatically different abstract models of ZS nets. To better understand the difference between the two philosophies, we propose the following example.

EXAMPLE 3.5.1 *Let MS be the net in Figure 3.1, and suppose that the current marking is $\{a,b\}$. If t_4 fires then a new token is produced into place a. A firing of t_1 consumes a token from place a.*

- *In the ITph approach, it makes a difference if t_1 gets the token produced by t_4 or the one already present in a (in the former case the firing of t_1 causally depends on that of t_4 while in the latter case the firings of t_1 and of t_4 are concurrent activities).*
- *In the CTph approach the two firings are always concurrent, since the initial marking enables both t_1 and t_4, i.e., the execution of t_4 does not modify the enabling condition of t_1. Thus t_1 and t_4 may fire in any order always originating equivalent computations.*

3.5.1 Operational Semantics

In the ITph, a marking can be thought of as an indexed (over the places of the net) collection of ordered sequences of tokens. Moreover, the firing of a transition specifies which tokens (of each ordered sequence) are consumed and also the correspondence between each token in the reached marking with either some produced token or an idle token of the original marking. Using multisets instead of ordered sequences would make it impossible to recognize which token was produced by which firing, as it happens for the CTph.

3.5.1.1 The Stacks Based Approach

The approach we propose is very similar to the one adopted in [124]: we choose a canonical interpretation of the tokens that have to be consumed and produced in a firing and we introduce *permutation firings* with the task of rearranging the orderings of the indexed sequences of tokens. A marking is represented as a collection of stacks, one for each place, where transitions can only access through a firing in order to extract and to insert tokens. Therefore, the extraction and the insertion of tokens in each place follow the LIFO policy. However, permutation firings are introduced to modify the token positions in the state of the system (i.e., the collection of stacks). Informally, permutation firings permit to choose which tokens to consume next. In what follows, we will denote the token stack associated to a place a by the term a-stack.

Definition 3.5.1 (Causal firing)
Let N be a P/T net, and $s = u[t\rangle u'$ be a firing of N for some marking u and transition t. We interpret the firing s as a causal firing *by assuming that s consumes the first $pre(t)(a)$ tokens of the a-stack of u and produces the first $post(t)(a)$ tokens of the a-stack of u', for each place a.*

Definition 3.5.2 (Permutation firing)
Let N be a P/T net. Given a marking $u = \{n_a a\}_{a \in S_N}$ of N, a symmetry *p on u is a vector of* permutations *$p = \langle \pi_a \rangle_{a \in S_N}$ with $\pi_a \in \Pi(n_a)$, for all $a \in S_N$, i.e., each π_a is a permutation of n_a elements. We denote by $\Pi(u)$ the set of all symmetries on u. Each symmetry p on u induces a* permutation firing *$s = u[p\rangle u$ on the net.*

DEFINITION 3.5.3 (CAUSAL FIRING SEQUENCE)
Given a P/T net N, a causal firing sequence *is a finite sequence $\omega = s_1 \cdots s_n$ of causal and permutation firings such that $s_i = u_{i-1}[X_i\rangle u_i$ with $X_i \in T_N \cup \Pi(u_{i-1})$ for $i = 1, \ldots, n$. We say that ω* starts at *u_0 (written $O(\omega) = u_0$) and* ends in *u_n (written $D(\omega) = u_n$).*

EXAMPLE 3.5.2 *Let N_{MS} be the underlying net of the ZS net MS defined in Figure 3.1 (i.e., in N_{MS} we do not distinguish between stable and zero places). A causal firing sequence for N_{MS} is*

$$\omega = \{b, c\}[t_0\rangle\{a, b, c\}[t_4\rangle\{2a, c\}[t_1\rangle\{a, b, c, z\}[t_3\rangle\{2b, c\}.$$

At the beginning of the sequence ω, the stacks of places a and z are empty while the stacks of places b and c contain one token each. After the firing of t_0, the token in the c-stack is replaced by a new one and a token is also inserted in the a-stack. The firing of t_4 consumes the token in the b-stack and puts a new token on top of the a-stack. The firing of t_1 consumes the token on top of the a-stack (the one produced by the firing of t_4) and produces a token in the b-stack and a token in the z-stack. The firing of t_3 consumes the token in the z-stack and also the token produced by the firing of t_0 in the a-stack; then, it inserts a token on top of the b-stack. Since the sequence ω does not involve any symmetry, it follows that the latest tokens produced are the first to be consumed next (for any place).

To represent the sequence in which t_1 depends on t_0 and t_3 depends on t_4, we have two possibilities. The first one is to execute t_0 after t_4 (they are concurrently enabled), thus obtaining the sequence

$$\omega' = \{b, c\}[t_4\rangle\{a, c\}[t_0\rangle\{2a, c\}[t_1\rangle\{a, b, c, z\}[t_3\rangle\{2b, c\}.$$

The second possibility is to reorganize the a-stack just before executing t_1. This can be done via the symmetry $p = \langle \pi_a, \pi_b, \pi_c, \pi_z \rangle$, where π_b and π_z are the empty permutations in $\Pi(0)$, $\pi_a = \{1 \to 2, 2 \to 1\} \in \Pi(2)$, and π_c is the unique identity permutation in $\Pi(1)$. Indeed we have the sequence

$$\omega'' = \{b, c\}[t_0\rangle\{a, b, c\}[t_4\rangle\{2a, c\}[p\rangle\{2a, c\}[t_1\rangle\{a, b, c, z\}[t_3\rangle\{2b, c\}.$$

Causal firing sequences establish a correspondence among the tokens produced and consumed via firings. This is due to the implicit orders which are imposed on the markings and is strictly related to a *process* view of computations, and in particular to concatenable processes (see Definition 3.2.6).

3.5.1.2 From Causal Sequences to Concatenable Processes

It may be easily noticed that each causal firing sequence uniquely determines a concatenable process. Informally the construction associates an elementary (concatenable) process to each causal and permutation firing.

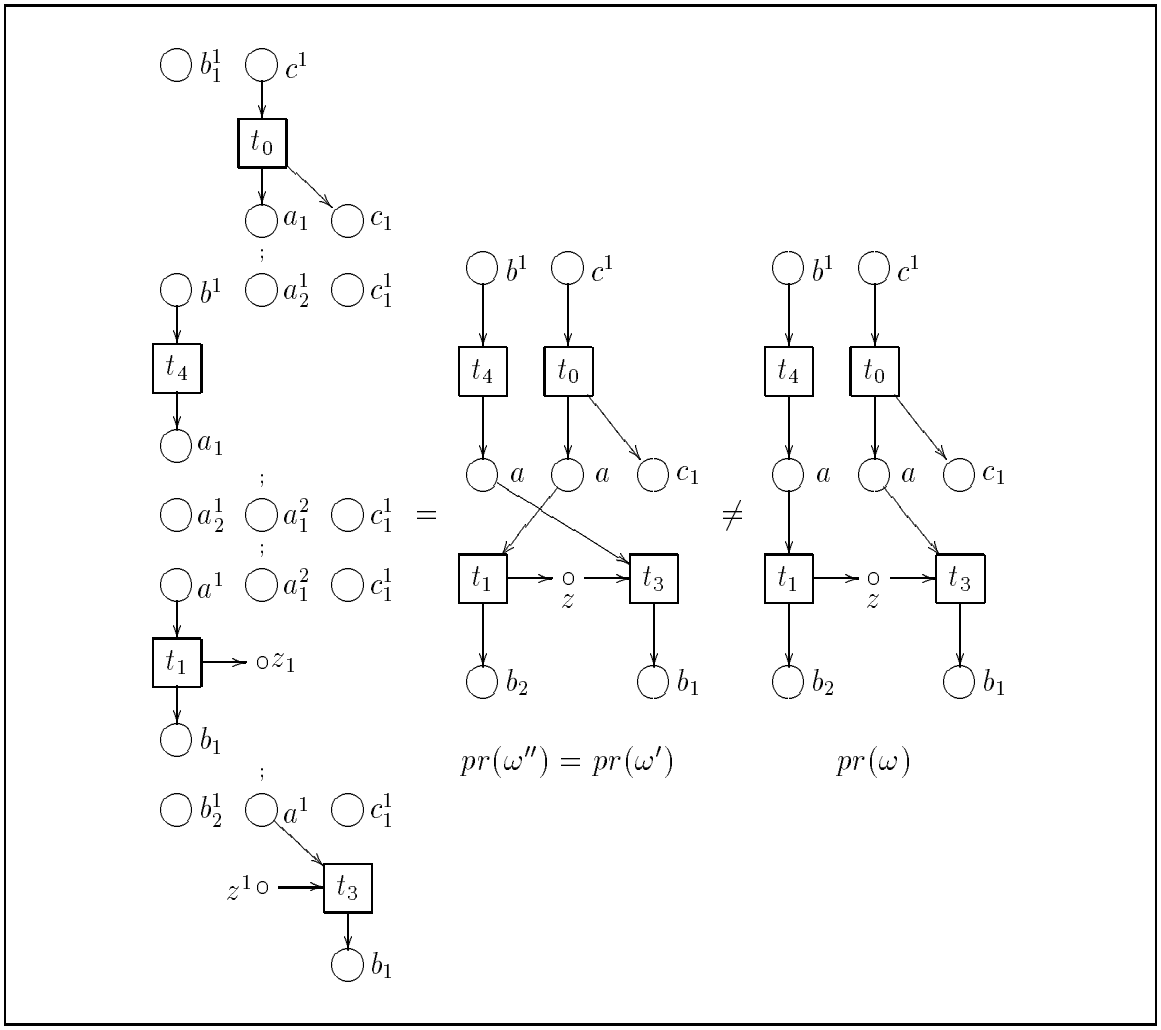


Figure 3.4: The concatenable processes derived from sequences ω'', ω', and ω of Example 3.5.2.

From causal firings to processes. Let N be a P/T net and $s = u[t\rangle u'$ be a causal firing, with $u = \{n_a a\}_{a \in S_N}$, $pre(t) = \{h_a a\}_{a \in S_N}$, $post(t) = \{k_a a\}_{a \in S_N}$. The associated concatenable process $pr(s) = (P : K \longrightarrow N, {}^\circ\ell, \ell^\circ)$ is defined as follows:

- $T_K = \{\tilde{t}\},\ P(\tilde{t}) = t$;
- $S_K = \{\tilde{a}_i \mid a \in S_N,\ 1 \leq i \leq n_a + k_a\},\ P(\tilde{a}_i) = a$;
- ${}^\bullet\tilde{t} = \{\tilde{a}_i \mid a \in S_N,\ 1 \leq i \leq h_a\}$, $\tilde{t}^\bullet = \{\tilde{a}_i \mid a \in S_N,\ h_a + 1 \leq i \leq h_a + k_a\}$. Therefore
  - $O(K) = \{\tilde{a}_i \in S_K \mid i \leq h_a \vee i \geq h_a + k_a + 1\}$ and
  - $D(K) = \{\tilde{a}_i \in S_K \mid i \geq h_a + 1\}$;
- $\forall \tilde{a}_i \in O(K),\ {}^\circ\ell_a(\tilde{a}_i) = \begin{cases} i & \text{if } 1 \leq i \leq h_a \\ i - k_a & \text{if } h_a + 1 + k_a \leq i \leq n_a + k_a; \end{cases}$
- $\forall \tilde{a}_i \in D(K),\ \ell^\circ_a(\tilde{a}_i) = i - h_a$.

A brief explanation is necessary: the causal net K contains a unique transition $\tilde{t}$ (which is mapped in the transition t of N), and a place for each consumed, produced, and idle token of s.

For each place $a \in S_N$ we need exactly $n_a + k_a$ different places in S_K. We denote the generic i-th place associated to place a by $\tilde{a}_i$.

The set $\{\tilde{a}_i \mid a \in S_N,\ 1 \leq i \leq h_a\}$ represents the tokens which are consumed by the causal firing of t, i.e., the first h_a tokens of each a-stack in the starting state.

The set $\{\tilde{a}_i \mid a \in S_N,\ h_a + 1 \leq i \leq h_a + k_a\}$ represents the tokens which are produced by the causal firing of t, i.e., the first k_a tokens of each a-stack in the ending state.

The set $\{\tilde{a}_i \mid a \in S_N,\ h_a + k_a + 1 \leq i \leq n_a + k_a\}$ contains the remaining idle tokens, i.e., the last $n_a - h_a$ tokens of each a-stack of both u and u'.

Functions ${}^\circ\ell$ and ℓ° are defined accordingly with these assumptions.

From permutation firings to processes. Let N be a P/T net and $s = u[p\rangle u'$ be a permutation firing, with $u = \{n_a a\}_{a \in S_N}$ and $p = \langle \pi_a \rangle_{a \in S_N}$. The associated concatenable process $pr(s) = (P : K \longrightarrow N, {}^\circ\ell, \ell^\circ)$ is defined as follows:

- $T_K = \emptyset$ (it follows that $O(K) = D(K) = S_K$);
- $S_K = \{\tilde{a}_i \mid a \in S_N,\ 1 \leq i \leq n_a\},\ P(\tilde{a}_i) = a$;
- $\forall \tilde{a}_i \in O(K),\ {}^\circ\ell_a(\tilde{a}_i) = i$;
- $\forall \tilde{a}_i \in D(K),\ \ell^\circ_a(\tilde{a}_i) = \pi_a(i)$.

In this case the set of transitions is empty and all the tokens stay idle. The generic place $\tilde{a}_i$ of k denotes the instance of place a which corresponds to the i-th token (from the top) of the a-stack of the starting state. The re-organization induced by the permutation firing is realized by differentiating the functions $\mathscr{C}$ and ℓ°.

The concatenable process associated to a (finite) causal firing sequence is given by the sequential composition of the concatenable processes associated to each step of the given sequence. In what follows we denote by $pr(\omega)$ the concatenable process associated to the causal firing sequence ω (up to isomorphism).

EXAMPLE 3.5.3 *The concatenable processes derived from the sequences ω, ω' and ω'' of Example 3.5.2 are presented in Figure 3.4.*

We use the standard notation that labels the places and transitions of the causal net K with their images in N. A superscript for any initial place and a subscript for any final place denotes the value of $\mathscr{C}$ and ℓ°, respectively.

The construction of $pr(\omega'')$ is explained in details by making explicit the concatenable processes associated to each causal and permutation firing of ω''.

3.5.1.3 Terminology

Let us introduce the suitable terminology for several properties of processes that will be used extensively. A process is *active* if it includes at least one transition, *inactive* otherwise.

An active process is *decomposable into parallel activities* if it is the parallel composition of two (or more) active processes. If such a decomposition does not exist, then the process is called *connected*.

A connected process may involve idle places, but it does not admit disjoint activities. The resources which are first produced and then consumed (i.e., the inner places) are called *evolution places*.

More formally, a concatenable process $C = (P : K \longrightarrow N, \mathscr{C}, \ell^\circ)$ is *connected* if and only if the set of transitions of K is non-empty and moreover, for each pair (t, t') of transitions of K there exists an undirected path (through the arcs of the flow relation) connecting t and t'. Process C is *full* if it does not contain idle (i.e., isolated) places (i.e., $\forall a \in S_K,\ |{}^\bullet a| + |a^\bullet| \geq 1$). Finally, the set of *evolution places* of process C is the set $E_C = \{P(a) \mid a \in K,\ |{}^\bullet a| = |a^\bullet| = 1\}$.

3.5.1.4 Connected Transactions

A causal firing sequence is essentially a "linearization" of a (concatenable) process and more than one sequence may correspond to the same (up to isomorphism) concatenable process. In fact, sequences differing in the order in which concurrent firings are executed or for the way equivalent symmetries are performed are equivalent.

For example, repeatedly swapping two tokens (of the same place) for an odd number of times is equivalent to apply the swap just once.

We will consider the equivalence over sequences induced by isomorphic processes. Moreover, we will notice that, for the kind of sequences under consideration, the label-indexed ordering functions of origins and destinations are no longer important, so we define the equivalence on the underlying Goltz-Reisig processes, rather than on the concatenable processes.

DEFINITION 3.5.4 (EQUIVALENT CAUSAL FIRINGS)
Let N be a net and ω, ω' be two causal firing sequences. We say that ω and ω' are causally equivalent, *written $\omega \approx \omega'$ iff $pr(\omega) = (P : K \longrightarrow N, \mathscr{C}, \ell^\circ)$ and $pr(\omega') = (P' : K' \longrightarrow N, \mathscr{C}', \ell'^\circ)$ with process P isomorphic to P'. The equivalence class of ω is denoted by $[\![\omega]\!]_\approx$. We use ξ to range over equivalence classes. Since relation $\approx$ respects the initial and final marking, we extend the notation letting $O(\xi) = O(\omega)$ and $D(\xi) = D(\omega)$, for $\xi = [\![\omega]\!]_\approx$.*

In the *ITph* based approach, state changes are given in terms of *connected steps*, which may involve the concurrent execution and synchronization of several transitions.

DEFINITION 3.5.5 (CONNECTED STEP AND TRANSACTION)
Given a ZS net B, let $\omega = s_1 \cdots s_n$ be a causal firing sequence of the underlying P/T net N_B. The equivalence class $\xi = [\![\omega]\!]_\approx$ is a connected step *of B, written $O(\xi)[\![\xi\rangle D(\xi)$, if:*

- *$O(\omega)$ and $D(\omega)$ are stable markings (stable fairness);*
- *$E_{pr(\omega)} \subseteq Z_B$ (atomicity).*

A connected step sequence *is a sequence $u_0[\![\xi_1\rangle u_1 \ldots u_{n-1}[\![\xi_n\rangle u_n$ of connected steps, and we say that u_n is reachable from u_0.*

Furthermore, the connected step ξ is a connected transaction *of B if:*

- *$pr(\omega)$ is connected;*
- *$pr(\omega)$ is full.*

We denote by Ξ_B (ranged by δ) the set of connected transactions of B.

As for the *CTph* approach, a connected step may be applied if the starting state contains enough stable tokens to enable all the transitions independently, and no token may be left on zero places at the end of the step (nor may be found on them at the beginning of the step). This means that all the zero tokens which are produced are also consumed in the same step, and defines the synchronization mechanism. A connected transaction is a connected step such that no intermediate marking is stable, and which consumes all the available stable tokens of the starting state. Connected steps differ from stable steps in that they allow for a finer causal relationship among firings. Let us point out that all conditions in Definition 3.5.5 impose constraints only over the Goltz-Reisig process associated with $pr(\omega)$, thus the proof of the following proposition is straightforward.

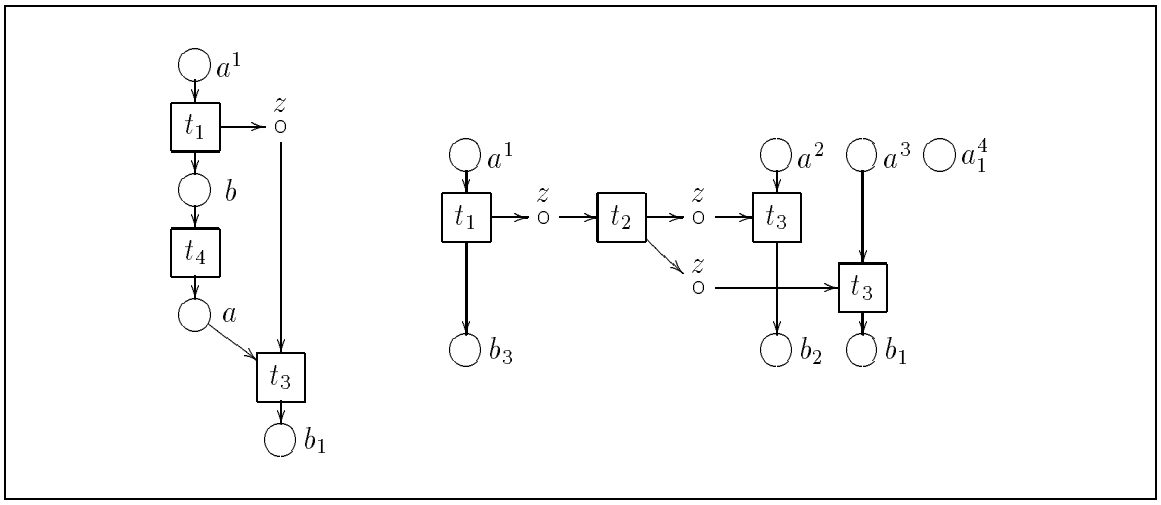


Figure 3.5: The concatenable processes $pr(\omega_1)$ (left) and $pr(\omega_2)$ (right).

PROPOSITION 3.5.1
Conditions "stable fairness," "atomicity," "connectedness," and "fullness" (see Definition 3.5.5) are preserved by the equivalence $\approx$.

We claim that, in the *ITph* context, connected transactions are a good definition for the basic behaviours of the systems. Our assertion is supported by the fact that connected transactions denote *basic computations that cannot be extended further.* The fact of being basic follows immediately from the connectedness of the associated processes (e.g., a transaction cannot be decomposed in two disjoint activities, because they must interact by definition of connectedness). The second argument deserves a more precise explanation. A basic behaviour can be extended if there exists a broader basic behaviour of which the former is a sub-part. From our viewpoint, the only interaction allowed in a transaction of a ZS net is given by the flow of tokens through zero places. Since connected steps and connected transactions start and also end in stable markings, it is impossible to hook them in a wider basic computation by means of zero tokens. This is very clear for connected transactions, because they consume all the needed resources. This is not the case of connected steps, since it could be possible for some resource to stay idle during the whole sequence of moves. However this kind of resources are stable and not connected to the activities performed in the rest of the step; thus, any other activity involving them is intrinsically concurrent w.r.t. the step under consideration. It follows that any wider behaviour extending a connected step is not basic (e.g., it can be expressed in terms of concurrent components).

EXAMPLE 3.5.4 *Let us consider the ZS net MS of Figure 3.1.*
The equivalence class of the causal firing sequence

$$\omega_1 = \{a\}[t_1\rangle\{b,z\}[t_4\rangle\{a,z\}[t_3\rangle\{b\}$$

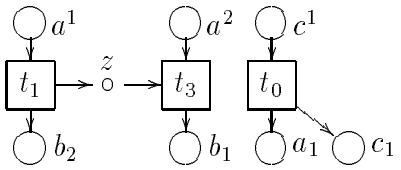


Figure 3.6: The concatenable process $pr(\omega_3)$.

is not a connected step since the "atomicity" requirement is not fullfilled. More precisely, this sequence defines a communication between an agent and itself, which is forbidden (see Figure 3.5).

The equivalence class of the causal firing sequence

$$\omega_2 = \{4a\}[t_1\rangle\{3a, b, z\}[t_2\rangle\{3a, b, 2z\}[t_3\rangle\{2a, 2b, z\}[t_3\rangle\{a, 3b\}$$

is a connected step but not a connected transaction since the associated process is not full (see Figure 3.5).

The equivalence class of the causal firing sequence

$$\omega_3 = \{2a, c\}[t_1\rangle\{a, b, c, z\}[t_3\rangle\{2b, c\}[t_0\rangle\{a, 2b, c\}$$

is a connected step but not a connected transaction since the associated process is not connected (see Figure 3.6).

The equivalence class of the causal firing sequence

$$\{5a\}[t_1\rangle\{\cdots\}[t_2\rangle\{\cdots\}[t_2\rangle\{\cdots\}[t_2\rangle\{\cdots\}[t_3\rangle\{\cdots\}[t_3\rangle\{\cdots\}[t_3\rangle\{\cdots\}[t_3\rangle\{5b\}$$

(where the obvious intermediate markings have been omitted), is a connected transaction.

3.5.2 Abstract Semantics

Also in the *ITph* based approach it is possible to define an abstract view of the systems modelled via zero-safe nets. As for the *CTph*, since transactions rewrite multisets of stable tokens, it is natural to choose a net as a candidate for the abstraction. Furthermore, since the ordering of tokens in the pre-set (post-set) of a transition is useless we should abstract from it. This is already done because we consider equivalence classes of causal firing sequences which denote different Goltz-Reisig processes.

When restricted to connected steps, this equivalence intuitively corresponds to limiting the symmetries of permutation firings to be vectors of permutations over the zero places only, with the assumption that the stable tokens which are produced in a transaction are not reused during the same transaction. The last statement was also the basis for the *CTph* approach.

EXAMPLE 3.5.5 *Consider the ZS net MS defined in Section 3.1.1, and let s denote the permutation firing $\{2a\}[p\rangle\{2a\}$, where $p \in \Pi(2a)$ is the symmetry which swaps the two tokens in the place a, let $s' = \{2b\}[p'\rangle\{2b\}$, where $p' \in \Pi(2b)$ is the symmetry which swaps the two tokens in the place b, and let $\omega = \{2a\}[t_1\rangle\{a, b, z\}[t_3\rangle\{2b\}$. Clearly the causal sequences ω, $s\omega$, and $\omega s'$ define the same connected transaction $\xi = [\![\omega]\!]_{\approx}$, but $pr(s\omega) \neq pr(\omega) \neq pr(\omega s')$. If we represent the connected transaction ξ as a transition t of a net, then its pre-set (as well as its post-set) is an unordered multiset. This means that when t fires it is impossible to distinguish among the tokens in a and in b that it produces and consumes. It follows that it makes no sense to have many different transitions to represent behaviours that we cannot reproduce at the abstract level. Thus we are forced to identify $pr(s\omega)$, $pr(\omega)$, $pr(\omega s')$, and also $pr(s\omega s')$.*

DEFINITION 3.5.6 (CAUSAL ABSTRACT NET)
Let $B = (S_B, T_B; F_B, u_B; Z_B)$ be a ZS net. The net $I_B = (S_B \setminus Z_B, \Xi_B; F, u_B)$, with $F(a, \delta) = pre(\delta)(a)$ and $F(\delta, a) = post(\delta)(a)$, is the causal abstract net *of B (we recall that $pre(\delta)$ and $post(\delta)$ denote the multisets $O(\delta)$ and $D(\delta)$, respectively, and that Ξ_B is the set of all the connected transactions of B).*

We conclude this section by examining the causal abstract net of our multicasting system.

EXAMPLE 3.5.6 *Let MS be the zero-safe net defined in Figure 3.1. Its causal abstract net I_{MS} is (partially!) depicted in Figure 3.3. We now comment on the possible connected transactions. Transition t'_0 is the basic activity which creates a new communicating process and it corresponds to $[\![\omega_0]\!]_{\approx}$ with $\omega_0 = \{c\}[t_0\rangle\{a, c\}$. Similarly t'_4 is the equivalence class of the firing $\{b\}[t_4\rangle\{a\}$. Each σ_i^k describes a different one-to-i communication. The index k identifies the copy policy (see Section 3.1.1) under consideration. For each i, we denote by c_i the number of different copy policy for the communication one-to-i.*

Essentially, a generic one-to-i communication can be described as follows: a firing of t_1 initiates the communication, then the system executes as many firings of t_2 as the number of copies of the message that are needed (i.e., $i-1$ since a message is already present in the buffer), and finally i firings of t_3 synchronize the remaining i messages with i active processes. In the ITph *we distinguish among tokens in a marking, which were created by different firings. In this way we have a one-to-one correspondence among copy policies and the* complete *binary trees*[2] *with exactly i leaves. For any $i \in \mathbb{N}$ ($i \neq 0$), the total number of copy policies c_i can be defined recursively.*

- *For $i = 1$ there is only one tree whose root is the unique leaf, and therefore $c_1 = 1$.*

[2] We recall that a binary tree is *complete* if any internal node has exactly two children (we do not distinguish between *left* and *right* children).

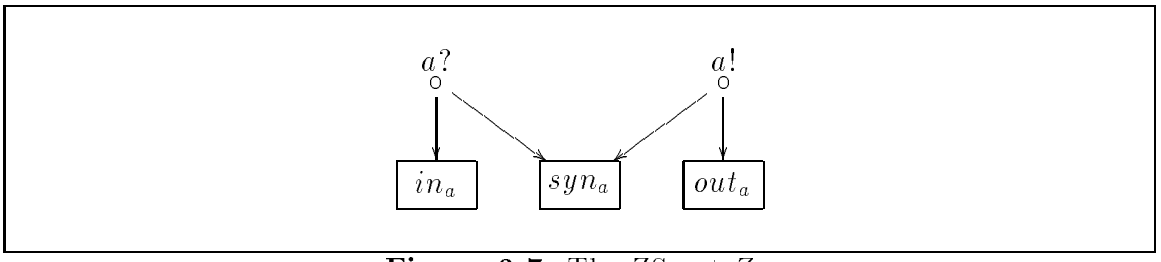


Figure 3.7: The ZS net Z_a.

- *If $i = 2h+1$ for some positive integer h, it follows that one of the two children of the root originates a complete binary tree with $j \leq h$ leaves, while the other child originates a complete binary tree with $i - j$ leaves; for any j, we know that there are $c_j \cdot c_{i-j}$ possible trees made in this way, thus $c_i = \sum_{j=1}^{h} c_j \cdot c_{i-j}$.*

- *If $i = 2h$ for some positive integer h, we adopt an analogous reasoning to deduce that for any $j < h$ there are $c_j \cdot c_{i-j}$ possible complete binary trees such that exactly j leaves originates from one of the two children of the root. However we have to pay attention to the case $j = h$. In fact, if the two subtrees have the same number h of leaves then there are $\frac{c_h \cdot (c_h+1)}{2}$ possible combinations for choosing them. It follows that $c_i = \frac{c_h \cdot (c_h+1)}{2} + \sum_{j=1}^{h-1} c_j \cdot c_{n-j}$.*

Since there are no transitions in MS requiring two or more zero tokens, there are no other transactions (it is impossible to synchronize separate communications).

3.6 Concurrent Language Translation

Before introducing the categorical constructions (Chapter 4), we conclude the more descriptive part of our presentation of ZS nets by giving some general hints for the translation of concurrent languages equipped with a CCS-like communication mechanism, into ZS nets.

The idea is to represent each channel by a pair of zero places, one for input and one for output, and to model each input (output) action on a channel with a transition, which produces a token on the input (output) zero place associated to the channel. A synchronization transition, also associated to the channel, is fired by a token in the input and a token in the output zero place. If the channel is restricted, this setting forces synchronization of input/output actions. If the channel is not restricted, two additional transitions are associated to the channel, simulating the effect of the environment.

More precisely, for every channel name a we define a ZS net Z_a consisting of two zero places $a!$ and $a?$, and three transitions in_a, syn_a, and out_a as depicted in Figure 3.7. We denote by $Z(\{a_1, ..., a_n\})$ the ZS net obtained as the disjoint union of $Z_{a_1}, \ldots, Z_{a_n}$, where $\{a_1, ..., a_n\}$ is any set of channel names.

$$\frac{}{a.p \xrightarrow{a} p} \qquad \frac{}{\bar{a}.p \xrightarrow{\bar{a}} p} \qquad \frac{p \xrightarrow{\mu} q}{p\backslash a \xrightarrow{\mu} q\backslash a}\ \mu \notin \{a, \bar{a}\}$$

$$\frac{p \xrightarrow{\mu} q}{p|r \xrightarrow{\mu} q|r} \qquad \frac{p \xrightarrow{\lambda} q,\ p' \xrightarrow{\bar{\lambda}} q'}{p|p' \xrightarrow{\tau} q|q'} \qquad \frac{p \xrightarrow{\mu} q}{r|p \xrightarrow{\mu} r|q}$$

Table 3.1: SOS rules for the simple process algebra SPA, where λ ranges over input/output actions, and μ ranges over input (a), output ($\bar{a}$) and silent (τ) actions.

DEFINITION 3.6.1 (INTERFACED NET)
A $\{a_1, ..., a_n\}$-interfaced net is a triple $\langle B, \{a_1, ..., a_n\}, P\rangle$, where B is a (marked) ZS net — in our translation the initial marking will always be a set — and P is an injective mapping from $Z(\{a_1, ..., a_n\})$ to B, which preserves the ZS net structure. The set of names $\{a_1, ..., a_n\}$ is also called the interface *of the net.*

Then, each agent p is modelled by an $fv(p)$-interfaced net $[\![p]\!]$, where the set $fv(p)$ is the set of the free (i.e., non-restricted) channel names in p. We consider a *simple process algebra* (SPA) equipped with the operations of inaction *nil*, input and output action prefix $a._-$ and $\bar{a}._-$, parallel composition $\_|\_$, and restriction $\_\backslash a$, i.e., the agents of the process algebra SPA are defined by the grammar

$$p ::= nil \ \mid\ a.p \ \mid\ \bar{a}.p \ \mid\ p\backslash a \ \mid\ p|p.$$

The operational semantics is given in Table 3.1 using the SOS style. Other operations, such as nondeterministic sum and recursion, could also be added, with limitations similar to those described in the literature for the existing approaches. The definition of $[\![p]\!]$ is given by initiality (i.e., it is the unique SPA-algebra homomorphism from the term algebra), and thus it is enough to define the corresponding operations on interfaced nets.

Inaction. The inactive net *nil* is a $\emptyset$-interfaced net $\langle B, \emptyset, \emptyset\rangle$, where B consists of a single place, and has one token in it as the initial marking.

Action prefix. The interfaced net $a.\langle B, A \cup \{a\}, P\rangle$ is given by adding a new stable place b and a new transition t to B. The initial marking consists of a token in b. Transition t takes a token in b, and produces the initial marking of B plus a token in the zero place $P(a?)$. If the name a is not contained in the interface of the given net, then also a copy of Z_a has to be added, and the injective mapping P is extended in the obvious way. A similar construction is defined for an output action prefix $\bar{a}.p$ (we substitute $a!$ for $a?$ in the post-set of the new transition t).

Parallel composition. We have $\langle B_1, A_1, P_1\rangle|\langle B_2, A_2, P_2\rangle = \langle B, A_1 \cup A_2, P\rangle$, with B given by the union of B_1 and B_2 where only $P_1(Z(A_1 \cap A_2))$ and $P_2(Z(A_1 \cap A_2))$ are identified, and with the mapping P given by the union of P_1 and P_2. The initial marking of B is the union of the initial markings of B_1 and B_2.

Restriction. We have $\langle B, A, P\rangle\backslash a = \langle B', A', P'\rangle$, with $B' = B\backslash\{P(in_a), P(out_a)\}$, $A' = A \setminus \{a\}$, and P' is P restricted to $Z(A \setminus \{a\})$.

Now it should be clear that the image of $Z(fv(p))$ plays the role of the interface, since it is the only part of the net $[\![p]\!]$ that is modified by the construction defined above: it can be increased (as in the case of action prefix), it can be merged with another interface (as in the case of parallel composition) and it can also be restricted (as in the case of the restriction operator).

Since each transition of the interfaced nets associated to the agents consumes and produces at most one token in each zero place and there is no transitions from zero places to zero places, it follows that the corresponding abstract nets obtained under the *CTph* and the *ITph* coincide.

PROPOSITION 3.6.1
For each agent p, with $[\![p]\!] = \langle B, A, P\rangle$, we have $A_B = I_B$.

Proof. The only transitions that can consume zero tokens are those in $P(Z(A))$, and alla have an empty postset. Hence, for each channel $a \in A$ there are three kinds of basic activities:

- $u_0[t_0\rangle v_0[in_a\rangle w_0$,
- $u_0 \oplus u_1[t_0, t_1\rangle v_0 \oplus v_1[syn_a\rangle w_0 \oplus w_1$,
- $u_1[t_1\rangle v_1[out_a\rangle w_1$, and

for any $t_0 \in B$ that produces a token in $a?$, for any $t_1 \in B$ that produces a token in $a!$ and for suitable markings u_i, v_i and w_i (in the case of restricted channel, then only the second activity is allowed). Each activity obviously identify a different abstract stable transaction and also a different abstract causal transaction, thus establishing a bijective correspondence between the two abstractions. □

To formalize a behavioural relation between SPA agents and their associated interfaced nets, we add a labelling function ϕ from the transitions of abstract nets to the set of actions. We first need the following proposition.

PROPOSITION 3.6.2
Let p be an agent, then each (connected) transaction ξ of $[\![p]\!]$ contains at most one firing of transitions in $P(Z(fv(p)))$.

Proof. By construction, only the transitions in $P(Z(fv(p)))$ can consume zero tokens, and all the remaining transitions have (exactly) a zero place in the postset. □

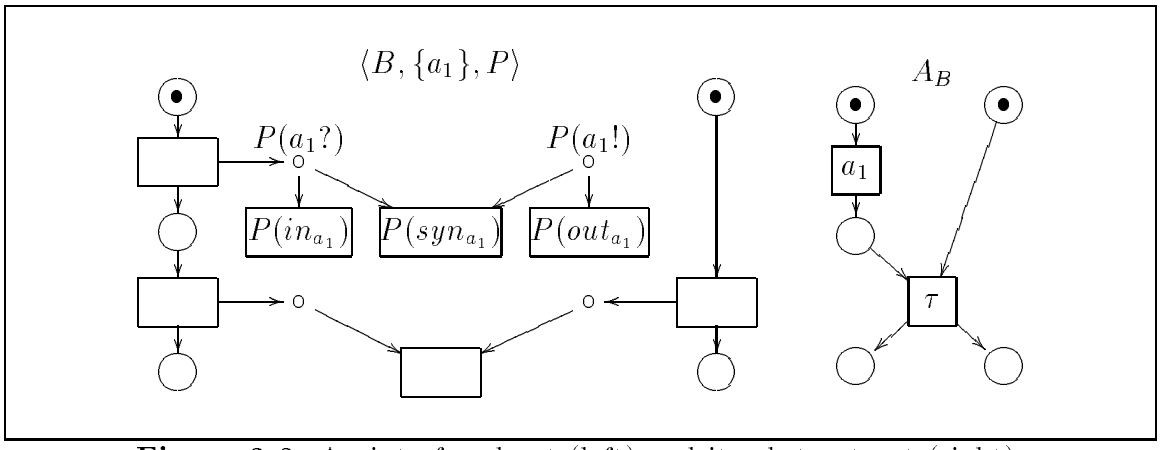


Figure 3.8: An interfaced net (left) and its abstract net (right).

DEFINITION 3.6.2 (LABELS OF TRANSACTIONS)
Let p be an agent. For each (connected) transaction ξ of $[\![p]\!]$, we define

$$\phi(\xi) = \begin{cases} a_i & \text{if } a_i \in fv(p) \text{ and } P(in_{a_i}) \text{ is fired in } \xi \\ \bar{a}_i & \text{if } a_i \in fv(p) \text{ and } P(out_{a_i}) \text{ is fired in } \xi \\ \tau & \text{otherwise} \end{cases}$$

The previous definition is correct because, by Proposition 3.6.2, if ξ contains the firing of $P(in_{a_i})$ for some action $a_i \in fv(p)$, then it can contain neither the firing of $P(in_{a_j})$, nor the firing of $P(out_{a_k})$, for $a_j, a_k \in fv(p)$, $a_j \neq a_i$.

EXAMPLE 3.6.1 *The interfaced net for the agent $(a_1.a_2.nil|\bar{a}_2.nil)\backslash a_2$ is presented in Figure 3.8, together with its (labelled) abstract net.*

It is out of the scope of this presentation to investigate the relationships between the net semantics we propose for SPA agents and those already known in the literature for CCS-like algebras, and we prefer to leave this topic for future works. We just remark that our translation is very linear, and that it provides a reasonable concurrent semantics for the simple class of agents considered here. This last assertion is supported by the following results.

DEFINITION 3.6.3 (BISIMILARITY BETWEEN AGENTS AND MARKINGS)
Let p be an agent, let N be a net whose transitions are labelled by ϕ over the set of actions, and let u be a marking of N. We say that p is bisimilar *to u in N if there exists a relation $\sim$ between agents and markings of N such that $p \sim u$, and:*

1. *for each transition $p \xrightarrow{a} p'$ there exists a firing $u[t\rangle u'$ of N such that $\phi(t) = a$ and $p' \sim u'$;*

2. *for each firing $u[t\rangle u'$ of N with $\phi(t) = a$ there exists a transition $p \xrightarrow{a} p'$ such that $p' \sim u'$.*

PROPOSITION 3.6.3
Let p be an agent, and $[\![p]\!] = \langle B, A, P\rangle$, then p is bisimilar to the initial marking of the abstract net A_B.

Proof (Sketch). After observing that net A_B is acyclic, we proceed by induction on the proof of the transition and on the structure of A_B as obtained by the translation of the agent p into $[\![p]\!]$. □

The restriction operator allows hiding local names w.r.t. some external observer. Therefore, a restricted name has only a local scope, and agents that differ only for the local names (i.e., agents that can be obtained one from the other by α-conversion) are usually considered equivalent. We use the symbol $\_ \equiv_\alpha \_$ to denote such equivalence. For example, we have

$$(a_1.a_2.nil|\bar{a}_2.nil)\backslash a_2 \equiv_\alpha (a_1.a_3.nil|\bar{a}_3.nil)\backslash a_3$$

for any $a_3 \neq a_1$. Proposition 3.6.4 shows that our translation deals very well with α-conversion.

DEFINITION 3.6.4
Two A-interfaced nets $\langle B, A, P\rangle$ and $\langle B', A, P'\rangle$ are isomorphic *if there exists a ZS net isomorphism ψ from B to B' that respects interfaces, i.e., such that $\psi(P(x)) = P'(x)$ for each element $x \in Z(A)$ (either place or transition).*

PROPOSITION 3.6.4
If p and q are agents such that $p \equiv_\alpha q$, then $[\![p]\!]$ and $[\![q]\!]$ are isomorphic.

Proof. It suffices to show that if $p\backslash a \equiv_\alpha p'\backslash b$ where p' is the same as p with all the free occurrences of a (respectively $\bar{a}$) replaced by b (respectively by $\bar{b}$), then $[\![p\backslash a]\!]$ is isomorphic to $[\![p'\backslash b]\!]$. If $a \notin fv(p)$ then $p = p'$ and we have concluded. Otherwise, let $[\![p]\!] = \langle B, A, P\rangle$ and $[\![p']\!] = \langle B, A', P'\rangle$. Then $a \in A$, $b \in A'$, $A \setminus \{a\} = A' \setminus \{b\}$, and P' behaves as P on $Z(A' \setminus \{b\})$ and $P'(Z(\{b\})) = P(Z(\{a\}))$. Therefore, by construction, $[\![p\backslash a]\!] = [\![p'\backslash b]\!]$. □

3.6.1 CSP-like Communications

By slightly modifying the ZS net Z_a associated to each channel a, we can model also CSP-like communications [78], where more than one agent can synchronize at the same time (the encoding becomes very similar to that of the multicasting system).

We consider the agents defined by the grammar

$$p ::= nil \mid a.p \mid p\backslash a \mid p|p.$$

whose operational semantics is given in Table 3.2 using the SOS style.

The modified ZS net $\hat{Z}_a$ is illustrated in Figure 3.9: it contains just one zero place a, representing the channel where agents can only put tokens.

$$\frac{}{a.p \xrightarrow{a} p} \qquad \frac{p \xrightarrow{\lambda} q}{p\backslash a \xrightarrow{\lambda} q\backslash a}\ \lambda \neq a$$

$$\frac{p \xrightarrow{\lambda} q}{p|r \xrightarrow{\lambda} q|r} \qquad \frac{p \xrightarrow{\lambda} q,\ p' \xrightarrow{\lambda} q'}{p|p' \xrightarrow{\lambda} q|q'} \qquad \frac{p \xrightarrow{\lambda} q}{r|p \xrightarrow{\lambda} r|q}$$

Table 3.2: SOS rules for the CSP-like communications, where λ ranges over channel names.

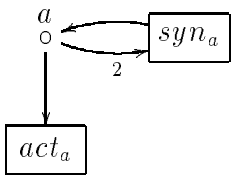


Figure 3.9: The ZS net $\hat{Z}_a$.

The transition syn_a synchronizes two agents, but leaves one token in a, hence other agents are allowed to participate in the communication. The transition act_a can be used to close the pending communication.

Then, the construction of $[\![p]\!]$ is analogous to the one used for SPA agents, and verifies similar properties. Notice, however, that many synchronization trees are possible (likewise the multicasting system), and therefore the *CTph* and the *ITph* yield different abstract nets.

3.7 Summary

In this chapter we have introduced zero-safe nets as a refined model of ordinary nets. Zero-safe nets include an additional simple mechanism for the synchronization of firings. We have given concurrent operational and abstract semantics for zero-safe nets under both the *CTph* and the *ITph*. The differences between the two approaches have been discussed on the basis of an evocative running example, which represents a multicasting system.

We have shown that, independently from the chosen philosophy, the zero-safe net *MS* representing the multicasting system yields an abstract net with infinitely many transitions. The differences between the two abstract nets have been characterized in terms of the feasible one-to-many communications: The *ITph* approach can faithfully model different copy policies, whilst the *CTph* approach can only record the number of agents involved in a communication.

We have also shown that CCS-like and CSP-like *agents* can be easily interpreted by representing the communication channels as zero places, in the style of the multicasting example. Thus, zero-safe nets could offer a suitable framework for a uniform approach to concurrent language translations.

A detailed categorical account of the constructions presented in this chapter is given in Chapter 4.

Chapter 4

Universal Construction

Contents

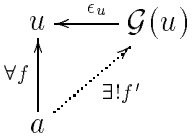


Figure 4.1: Coreflection diagram.

4.1 Motivations

This chapter has a more abstract and mathematical flavour. Its aim is to give evidence that the definitions and the constructions on zero-safe nets presented in Chapter 3 are natural. The tool we use is some elementary category theory. In particular, three concepts are useful here.

The first notion is the *category of models* itself, where the objects are models (in our case zero-safe nets) and the arrows represent some notion of simulation. The choice of arrows is very informative, since they complement and in a sense redefine — in that isomorphic objects are often identified — the meaning of models.

The second notion is *adjunction* (see Definition 2.1.17), which is useful to characterize "natural" constructions. The typical scenario includes two categories C_1 and C_2 - where C_2 has more structure than C_1 - and a (usually obvious) forgetful functor $\mathcal{U} : C_2 \longrightarrow C_1$, which deletes the extra structure. It might happen that $\mathcal{U}$ has a left adjoint $\mathcal{F} : C_1 \longrightarrow C_2$. If this is the case, $\mathcal{F}$ represents the "best" construction for adding the extra structure. In fact the left adjoint is unique (up to isomorphism) and satisfies a key universal property.

The third notion is *coreflection*, which is a special kind of adjunction. Here the scenario includes a category C and a subcategory C' of it. Category C represents "operational" models, while C' defines certain "abstract" models. In addition there is a functor $\mathcal{G} : C \longrightarrow C'$ whose left adjoint is the inclusion functor from C' to C. For every object u of C there is a unique arrow $\epsilon_u : \mathcal{G}(u) \longrightarrow u$ with the universal property that, given any abstract object a of C', for every arrow $f : a \longrightarrow u$ there is a unique arrow $f' : a \longrightarrow \mathcal{G}(u)$ with $f = f'; \epsilon_u$, as the commuting diagram in Figure 4.1 illustrates.

This situation is ideal from a semantic point of view. In fact $\mathcal{G}(u)$ can be understood as an abstraction of model u (e.g., its behaviour), with the additional advantage of being at the same time a model itself. The universal property above means that if we observe models from an abstract point of view (i.e., via morphisms originating from objects in C'), then there is an isomorphism (via left composition with ϵ_u) between observations of u and observations of its abstract counterpart $\mathcal{G}(u)$. Thus in a sense, model u seen from C' is the same as $\mathcal{G}(u)$. Again, if a coreflection exists between C and C' with the inclusion as left adjoint, then it is unique up to isomorphism.

Similar constructions apply to the *CTph* and *ITph* approaches. Both the operational semantics of ZS nets and the derivation of the abstract P/T nets are characterized as two universal constructions, following the *Petri nets are monoids* approach [106]. More precisely, the former can be characterized as an adjunction and the latter as a coreflection. The universal properties of the two constructions witness that they are the "best" feasible choices, thus giving evidence of the naturality of the definitions presented in Chapter 3. Moreover, since left adjoints preserve colimits, our operational semantics preserve several important net compositions (that can be expressed via colimit constructions) on the refined model, e.g., the union of two nets where a subset of places become shared. Similarly, the abstract semantics preserve limit constructions on the refined model.

We also give an alternative presentation of ZS nets in terms of a very simple tile model, whose configurations and observations are the free commutative monoids on stable and zero places respectively.

4.1.1 Structure of the Chapter

After a brief survey of the *Petri nets are monoids* approach given in Sections 4.2.1 and 4.2.2 (for the *CTph* and for the *ITph* respectively), we illustrate the categorical characterization of the operational and abstract semantics of ZS nets under the *CTph* (*ITph*) in Sections 4.3.1 and 4.3.2 (Sections 4.4.1 and 4.4.2). The relationship between ZS nets and the tile model is discussed in Section 4.5. Several related works on the abstraction/refinement of nets are discussed at the end of the chapter.

4.2 Background

4.2.1 Monoidal Structure of Petri Nets

Petri net theory can be profitably developed within category theory. Among the existing approaches we mention [137, 105, 18]. We follow the approach initiated by Meseguer and Montanari [105] (other references are [106, 49, 50, 108, 109]), which focuses on the monoidal structure of Petri nets, where the monoidal operation means parallel composition.

The basic observation is that a P/T Petri net is just a graph $(S^{\oplus}, T, \partial_0, \partial_1)$ where the set of nodes is the free commutative monoid $S^{\oplus}$ over the set of *places* S. (functions $\partial_0, \partial_1 : T \longrightarrow V$ are called *source* and *target*, resp., and we write $t : u \longrightarrow v$, with obvious meaning, to shorten the notation).

Definition 4.2.1 (Category **Petri**)
A Petri net morphism *is a graph morphism* $h = (f : T \longrightarrow T', g : S^{\oplus} \longrightarrow S'^{\oplus})$ *(i.e.,* $g(\partial_i(u)) = \partial'_i(f(u))$ *for* $i = 0, 1$*) where* g *is a monoid homomorphism. Petri nets and Petri net morphisms yield the category* **Petri** *(with the obvious composition and identities).*

$$\frac{u \in S_N^{\oplus}}{id_u : u \longrightarrow u \in \mathcal{F}[N]} \qquad \textbf{(identities)}$$

$$\frac{t : u \longrightarrow v \in T_N}{t : u \longrightarrow v \in \mathcal{F}[N]} \qquad \textbf{(transitions)}$$

$$\frac{a, b \in S_N}{c_{a,b} : a \oplus b \longrightarrow b \oplus a \in \mathcal{F}[N]} \qquad \textbf{(symmetries)}$$

$$\frac{\alpha : u \longrightarrow v,\ \beta : u' \longrightarrow v' \in \mathcal{F}[N]}{\alpha \otimes \beta : u \oplus u' \longrightarrow v \oplus v' \in \mathcal{F}[N]} \qquad \textbf{(parallel composition)}$$

$$\frac{\alpha : u \longrightarrow v,\ \beta : v \longrightarrow w \in \mathcal{F}[N]}{\alpha;\beta : u \longrightarrow w \in \mathcal{F}[N]} \qquad \textbf{(sequential composition)}$$

Table 4.1: The inference rules for $\mathcal{F}[N]$.

It has been shown [106, 49] that it is possible to enrich the algebraic structure of transitions in order to capture some basic constructions on nets. As an example, the forgetful functor from **CMonRPetri** [106] to **Petri** has a left adjoint which associates to each Petri net N its *marking graph* $\mathcal{C}[N]$, which corresponds to the ordinary operational semantics of N (i.e., its arrows are the step sequences of N). The objects of **CMonRPetri** are called *reflexive Petri commutative monoids* (i.e., Petri nets together with a function $id : S^{\oplus} \longrightarrow T$, where T is a commutative monoid[1] $(T, \otimes, 0)$ and ∂_0, ∂_1 and id are monoid homomorphisms), and its arrows are Petri net morphisms preserving identities and the monoidal structures.

4.2.2 Axiomatization of Concatenable Processes

The algebraic structure of processes is well exploited in [49, 50, 126]. These papers show how to associate a free symmetric strict monoidal category (see Definition 2.1.9) $\mathcal{F}[N]$ to each net N in such a way that, under two suitable axioms, it characterizes the concatenable processes of N. This is due to the existence of a left adjoint $\mathcal{F} : \mathbf{Petri} \longrightarrow \mathbf{SSMC}^{\oplus}$ to the forgetful functor from $\mathbf{SSMC}^{\oplus}$ to **Petri**. Later, this axiomatization will be useful to shorten the notation in the sketched proof of Theorem 4.4.6. Given a net N, the category $\mathcal{F}[N]$ has the elements of $S_N^{\oplus}$ as objects, while its arrows are generated by the inference rules in Table 4.1, modulo the axioms expressing that $\mathcal{F}[N]$ is a strict monoidal category, and the ax-

[1] As usual, the notation $(X, op_1, ..., op_n)$ denotes a set X together with its additional structure, which is given by the operations $op_1, ..., op_n$.

ioms stating that the collection $\{c_{u,v}\}_{u,v \in S_N^{\oplus}}$ plays the role of the symmetry natural isomorphism which makes $\mathcal{F}[N]$ into a ssmc (the terms $c_{u,v}$ for $u, v \in S_N^{\oplus}$ denote the arrows obtained composing $c_{a,b}$ for $a, b \in S_N$ by the following recursive rules (that are analogous to axioms (4.5) given in Theorem 4.4.3):

$$\begin{aligned} c_{0,u} &= id_u = c_{u,0}, \\ c_{u \oplus v,w} &= (id_u \otimes c_{v,w}); (c_{u,w} \otimes id_v), \text{ and} \\ c_{u,v \oplus w} &= (c_{u,v} \otimes id_w) * (id_v \otimes c_{u,w}). \end{aligned} \tag{4.1}$$

THEOREM 4.2.1 (CF. [126])
Given a net N, the concatenable processes of N are isomorphic to the arrows of the category $\mathcal{P}[N]$ which is the monoidal quotient of the free ssmc on N ($\mathcal{F}[N]$) modulo the axioms

$$c_{a,b} = id_{a \oplus b} \quad \text{if} \quad a \neq b \in S_N, \tag{4.2}$$

$$s; t; s' = t \quad \text{if} \quad t \in T_N, \text{ and } s, s' \text{ are symmetries.} \tag{4.3}$$

The previous construction provides an algebraic view of net computations which is strictly related to a process understanding of the causal behaviour of a net, but which is not functorial. The main problem is that there exist reasonable morphisms of nets which cannot be extended to a monoidal functor. We report below a convincing example illustrated in [109].

EXAMPLE 4.2.1 *Consider the nets N and N' pictured in Figure 4.2 and the net morphism $f : N \longrightarrow N'$ such that*

- *$f(t_i) = t'_i$ and $f(a_i) = a'$, for $i = 0, 1$ and*
- *$f(b) = b'$ and $f(c) = c'$.*

The morphism f cannot be extended to a functor $\mathcal{P}[f] : \mathcal{P}[N] \longrightarrow \mathcal{P}[N']$. In fact, if such an extension F existed, then

$$F(t_0 \otimes t_1) = F(t_0) \otimes F(t_1) = t'_0 \otimes t'_1$$

by the monoidality of F, and since $t_0 \otimes t_1 = t_1 \otimes t_0$ in $\mathcal{P}[N]$, it would follow that $t'_0 \otimes t'_1 = t'_1 \otimes t'_0$ which is impossible, as the two expressions denote different processes in $\mathcal{P}[N']$.

We will show that this kind of morphisms are reasonably avoided in the category **dZPetri**. Moreover, this choice is justified by the necessity to preserve atomic behaviours through morphisms.

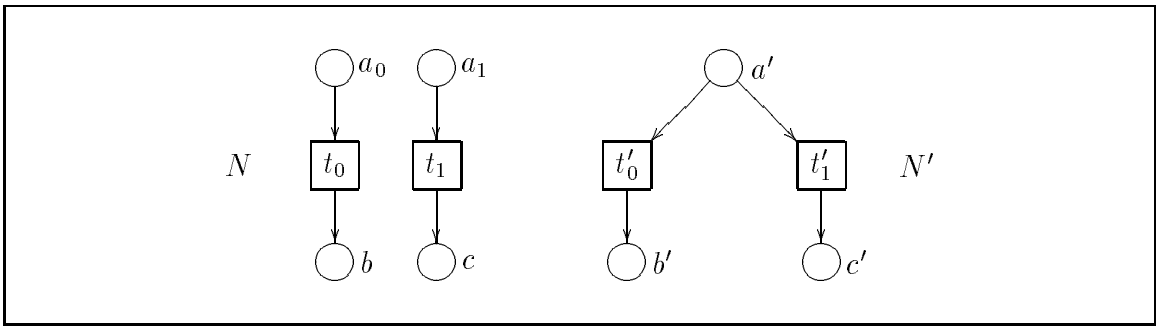


Figure 4.2: Two nets for which there exists a trivial morphism that cannot be extended to a monoidal functor among the corresponding categories of concatenable processes.

4.3 Collective Token Approach

In this section we describe two universal constructions involving zero-safe nets.

The first construction starts from a category **dZPetri** (where zero-safe nets are considered as programs) and exhibits an adjunction to a category **HCatZPetri** consisting of some kind of machines equipped with operations and transitions between states.

It is proved that this adjunction corresponds to the token-pushing semantics of zero-safe nets defined in Section 3.4, in the sense that the transitions of the machine $\mathcal{Z}[B]$ associated to a zero-safe net B are exactly the abstract stable steps of B.

The second construction starts from a different category **ZSN** of zero-safe nets (which however is strictly related to **HCatZPetri**), having the ordinary category **Petri** of P/T nets as a subcategory, and yields a coreflection corresponding exactly to the construction of the abstract net in Definition 3.4.5.

4.3.1 Operational Semantics as Adjunction

Analogously to P/T nets, ZS nets can be seen as graphs whose set of nodes is the free commutative monoid over the set of places $S = L \cup Z$ (partitioned into the sets L and Z, respectively of stable and zero places), and we have the following category of zero-safe nets.

Definition 4.3.1 (Category **dZPetri**)
A ZS net morphism *is a Petri net morphism* $(f, g) : N \longrightarrow N'$ *where* g *is a monoid homomorphism which preserves partitioning of places (i.e., if* $a \in Z$ *then* $g(a) \in Z'^{\oplus}$ *and if* $a \in S \setminus Z$ *then* $g(a) \in (S' \setminus Z')^{\oplus}$*) and satisfies the additional condition of mapping zero places into pairwise disjoint (non-empty) zero markings (i.e.,* $\forall z, z' \in Z_B$ *with* $z \neq z'$*, if* $g_Z(z) = n_1 a_1 \oplus \cdots \oplus n_k a_k$ *and* $g_Z(z') = m_1 b_1 \oplus \cdots \oplus m_l b_l$ *then we have that* $a_i \neq b_j$*, for* $i = 1, \ldots, k$ *and* $j = 1, \ldots, l$*), which is called the* disjoint image *property. ZS nets and their morphisms define the category* **dZPetri**.

REMARK 4.3.1 *Since $S^{\oplus}$ is a free commutative monoid, and since S is partitioned into the sets L and Z of stable and zero places respectively, we can equivalently represent the set of nodes of a ZS net as $L^{\oplus} \times Z^{\oplus}$ (i.e., sources and targets are pairs whose components are elements of the free commutative monoids over stable and zero places respectively). Thus ZS net morphisms become triples of the form $h = (f, g_L, g_Z)$ where both g_L and g_Z are monoid homomorphisms on the monoids of stable and zero places respectively.*

EXAMPLE 4.3.2 *The graph corresponding to the ZS net MS pictured in Figure 3.1 has the following set of arcs: $T_{MS} = \{t_0 : (c,0) \longrightarrow (a \oplus c, 0)$, $t_1 : (a,0) \longrightarrow (b,z)$, $t_2 : (0,z) \longrightarrow (0,2z)$, $t_3 : (a,z) \longrightarrow (b,0)$, $t_4 : (b,0) \longrightarrow (a,0)\}$. Notice that the first component of each source (target) pair represents stable places, whilst the second component represents zero places.*

We introduce the category **HCatZPetri** that is the analogous to **CMonRPetri** for ZS nets. The models of **HCatZPetri** (which we call *ZS graphs*) are more complex than those of **CMonRPetri** since they must be equipped with an operation of composition of arrows to allow for the construction of transactions. Thus **HCatZPetri** is in a sense intermediate between **CMonRPetri** and the category **CatPetri** introduced in [106].

DEFINITION 4.3.2 (CATEGORY **HCatZPetri**)
A ZS graph

$$H = ((L \cup Z)^{\oplus}, (T, \_ \otimes \_, 0, id, \_;\_), \partial_0, \partial_1)$$

is both a ZS net and a reflexive Petri commutative monoid. In addition, it is equipped with a partial function $\_;\_$ called horizontal composition*:*

$$\frac{\alpha : (u,x) \longrightarrow (v,y),\ \beta : (u',y) \longrightarrow (v',z)}{\alpha;\beta : (u \oplus u', x) \longrightarrow (v \oplus v', z)}. \tag{4.4}$$

Horizontal composition is associative and has identities

$$id_{(0,x)} : (0,x) \longrightarrow (0,x)$$

for any $x \in Z^{\oplus}$; in addition, the commutative monoidal operator $\_ \otimes \_$ is functorial w.r.t. horizontal composition, i.e., the axiom

$$(\alpha \otimes \beta);(\alpha' \otimes \beta') = (\alpha;\alpha') \otimes (\beta;\beta')$$

holds whenever the right member is defined.

Given two ZS graphs H and H', a ZS graph morphism *$h = (f, g_L, g_Z) : H \longrightarrow H'$ is both a ZS net morphism and a reflexive Petri monoid morphism such that $f(\alpha;\beta) = f(\alpha);f(\beta)$. ZS graphs and their morphisms (together with the obvious composition and identities) constitute the category* **HCatZPetri**.

Horizontal composition is the key of our approach. It acts as a sequential composition on zero places and as a parallel composition on stable places. This is exactly what we need to model stable steps, because two successive firings in a stable step are allowed if and only if the stable tokens which are needed are already present in the initial marking.

PROPOSITION 4.3.1
If $\alpha : (u, 0) \longrightarrow (v, 0)$ *and* $\alpha' : (u', 0) \longrightarrow (v', 0)$ *are two arrows of a ZS graph then* $\alpha; \alpha' = \alpha \otimes \alpha' = \alpha'; \alpha$.

Proof. It can be easily verified that:

$$\alpha; \alpha' = (\alpha \otimes id_{(0,0)}); (id_{(0,0)} \otimes \alpha') = (\alpha; id_{(0,0)}) \otimes (id_{(0,0)}; \alpha') = \alpha \otimes \alpha'.$$

Then, by commutativity of $\_ \otimes \_$, it follows that $\alpha; \alpha' = \alpha'; \alpha$. □

Next results show that **HCatZPetri** has **CMonRPetri** as a subcategory.

PROPOSITION 4.3.2
The full subcategory of **HCatZPetri** *whose objects are all and only Petri nets (i.e.,* $Z = \emptyset$*) is isomorphic to* **CMonRPetri**.

Proof. Let H be a ZS graph such that Z_H is empty. Then for any $\alpha \in T_H$, there exist $u, v \in L_H^{\oplus}$ such that $\alpha : (u, 0) \longrightarrow (v, 0)$. Thus for all $\alpha, \beta \in T_H$ it follows that $\alpha; \beta = \alpha \otimes \beta$, i.e., the horizontal composition adds no structure. □

The following theorem defines the algebraic semantics of ZS nets by means of a universal property.

THEOREM 4.3.3
Let $\mathcal{U}_C$: **HCatZPetri** $\longrightarrow$ **dZPetri** *be the functor which forgets about the additional structure on arrows, i.e.,*

$$\mathcal{U}_C[((L \cup Z)^{\oplus}, (T, \otimes, 0, id, ;), \partial_0, \partial_1)] = (L^{\oplus} \times Z^{\oplus}, T, \partial_0, \partial_1).$$

The functor $\mathcal{U}_C$ *has a left adjoint* $\mathcal{Z}$: **dZPetri** $\longrightarrow$ **HCatZPetri**.

Proof (Sketch). Let the functor $\mathcal{Z}$: **dZPetri** $\longrightarrow$ **HCatZPetri** map a ZS net B into the ZS graph defined by the inference rules in Table 4.2, where transitions form a commutative monoid (i.e., $\alpha \otimes \beta = \beta \otimes \alpha$, $(\alpha \otimes \beta) \otimes \delta = \alpha \otimes (\beta \otimes \delta)$, and $id_{(0,0)} \otimes \alpha = \alpha$, for any $\alpha, \beta, \delta \in \mathcal{Z}[B]$); where the horizontal composition $\_;\_$ is associative and has identities (i.e., $(\alpha; \beta); \delta = \alpha; (\beta; \delta)$ and $\alpha; id_{(0,y)} = \alpha = id_{(0,x)}; \alpha$, whenever such compositions are defined); and where the monoidal operator $\_ \otimes \_$ is functorial (i.e., $id_{(u,x)} \otimes id_{(v,y)} = id_{(u \oplus v, x \oplus y)}$ and $(\alpha \otimes \alpha'); (\beta \otimes \beta') = (\alpha; \beta) \otimes (\alpha'; \beta')$, where the latter holds whenever the rightmost member of the equation is defined). It is easy to verify that mapping $\mathcal{Z}$ extends to a functor which is a right adjoint to functor $\mathcal{U}_C$. □

$$\frac{(u,x) \in L_B^{\oplus} \times Z_B^{\oplus}}{id_{(u,x)} : (u,x) \longrightarrow (u,x) \in \mathcal{Z}[B]} \quad \textbf{(identities)}$$

$$\frac{t : (u,x) \longrightarrow (v,y) \in T_B}{t : (u,x) \longrightarrow (v,y) \in \mathcal{Z}[B]} \quad \textbf{(transitions)}$$

$$\frac{\alpha : (u,x) \longrightarrow (v,y),\ \beta : (u',x') \longrightarrow (v',y') \in \mathcal{Z}[B]}{\alpha \otimes \beta : (u \oplus u', x \oplus x') \longrightarrow (v \oplus v', y \oplus y') \in \mathcal{Z}[B]} \quad \textbf{(par. comp.)}$$

$$\frac{\alpha : (u,x) \longrightarrow (v,y),\ \beta : (u',y) \longrightarrow (v',z) \in \mathcal{Z}[B]}{\alpha;\beta : (u \oplus u', x) \longrightarrow (v \oplus v', z) \in \mathcal{Z}[B]} \quad \textbf{(seq. comp.)}$$

Table 4.2: The inference rules for $\mathcal{Z}[B]$.

The following theorem shows that the algebraic semantics of ZS nets is an extension of the ordinary semantics of P/T nets.

THEOREM 4.3.4
When restricted to P/T nets, functor $\mathcal{Z}$ coincides with $\mathcal{C}$.

Proof. The result immediately follows from Proposition 4.3.1, because if $Z_B = \emptyset$ then for all $t : (u,x) \longrightarrow (v,y) \in T_B$ is $x = y = 0$. □

EXAMPLE 4.3.3 *Let MS be the ZS net of our running example whose set of arcs is defined in Example 4.3.2. For instance the arrow $t_1; t_3 \in \mathcal{Z}[MS]$ has source $(2a, 0)$ and target $(2b, 0)$. Instead, notice that the arrow $(t_1 \otimes id_{(a,0)}); (id_{(b,0)} \otimes t_3)$ goes from $(3a \oplus b, 0)$ to $(a \oplus 3b, 0)$.*

As another example, the following expressions are identified, in $\mathcal{Z}[MS]$, i.e., they all denote the same arrow:

$$\begin{aligned} t_1; t_2; (t_2 \otimes id_{(0,z)}); (t_3 \otimes id_{(0,2z)}); (t_3 \otimes id_{(0,z)}); t_3 &= \\ t_1; t_2; (t_2 \otimes id_{(0,z)}); (t_3 \otimes t_3 \otimes t_3) &= \\ t_1; t_2; (t_2 \otimes id_{(0,z)}); (id_{(0,2z)} \otimes t_3); (t_3 \otimes t_3) &= \\ t_1; t_2; (t_2 \otimes t_3); (t_3 \otimes t_3). & \end{aligned}$$

DEFINITION 4.3.3 (PRIME ARROW)
An arrow $\alpha : (u,0) \longrightarrow (v,0)$ of a ZS graph H is prime *if and only if α cannot be expressed as the monoidal composition of non-trivial arrows (i.e., $\forall \beta, \gamma \in H$ such that $\alpha = \beta \otimes \gamma$, then either $\beta = id_{(0,0)}$, or $\gamma = id_{(0,0)}$).*

THEOREM 4.3.5
Given a ZS net B, the arrows $\alpha : (u,0) \longrightarrow (v,0) \in \mathcal{Z}[B]$ are in bijective correspondence with the abstract stable steps of B. Moreover, if such an arrow is prime then the corresponding abstract stable step is an abstract stable transaction.

Proof. Given (the equivalence class of) a generic stable step

$$s = u \oplus u_0[t_1\rangle u \oplus u_1 \oplus x_1[t_2\rangle \cdots u \oplus u_{n-1} \oplus x_{n-1}[t_n\rangle u \oplus u_n$$

where each multiset x_i contains (all) the zero tokens at the i-th stage of the step, u is the multiset of the stable tokens which stay idle in s, and $t_i : (w_i, y_i) \longrightarrow (v_i, z_i)$ for $i = 1, \ldots, n$, then the corresponding arrow is

$$\alpha_s = (t_1 \otimes id_{(u,x'_1)}); (t_2 \otimes id_{(0,x'_2)}); \cdots; (t_n \otimes id_{(0,x'_n)})$$

with $x'_1 = 0$ and $x'_i \oplus y_i = x_{i-1}$, for $i = 2, \ldots, n$, where y_i is the multiset of zero places in the source of t_i (see above). The correctness of our definition follows immediately, simply noticing that each diamond transformation s' of s is mapped into an arrow $\alpha_{s'}$ which can be proved equal to α_s thanks to the functoriality axiom. In fact, since s' is a diamond transformation of s, then there exists k, with $1 \leq k \leq n-1$, such that t_k and t_{k+1} are concurrently enabled, i.e., $y_{k+1} \leq x'_k$. Thus (for generic stable markings v and v'):

$$\begin{aligned}(t_k \otimes id_{(v,x'_k)}); (t_{k+1} \otimes id_{(v',x'_{k+1})}) &= t_k \otimes t_{k+1} \otimes id_{(v \oplus v', x')} \\ &= (t_{k+1} \otimes id_{(v', x' \oplus y_k)}); (t_k \otimes id_{(v, x' \oplus z_{k+1})})\end{aligned}$$

where $x' \oplus y_{k+1} = x'_k$. Then, it can be easily checked that the axioms given in the proof of Theorem 4.3.3 identify equivalent steps only.

For the converse correspondence, let

$$(t_1 \otimes id_{(u_1,x_1)}); (t_2 \otimes id_{(u_2,x_2)}); \cdots; (t_n \otimes id_{(u_n,x_n)})$$

be any (arbitrarily chosen) *linearization* of a given α, with $t_j : (w_j, y_j) \longrightarrow (v_j, z_j)$. Then, the sequence

$$s = u'_0[t_1\rangle u'_1 \cdots u'_{k-1}[t_k\rangle u'_k[t_{k+1}\rangle u'_{k+1} \cdots u'_{n-1}[t_n\rangle u'_n$$

with $u'_0 = u$ is a stable step.

Now, suppose that α is prime and that $[\![s]\!]$ is not a stable transaction. Then, $\exists s' \in [\![s]\!]$ such that $s' = u[t_{i_1}\rangle p_1 \ldots p_{n-1}[t_{i_n}\rangle v$ with $p_k \in L_B^{\oplus}$ for a certain index k ($\sum_{j=1}^{n} pre(t_{i_j})(a) = u(a)$ for any stable place a, because α is prime). Then, $\alpha = \beta; \gamma$ with $\beta : (q,0) \longrightarrow (q',0)$ and $\gamma : (r,0) \longrightarrow (r',0)$ for some (non-trivial) arrows β and γ with $u = q \oplus r$, $v = q' \oplus r'$ and $q' \oplus r = p_k$. This is contradictory, since $\alpha = \beta; \gamma = \beta \otimes \gamma$ while α is prime by hypothesis. □

Theorem 4.3.5 shows that the operational and algebraic semantics of ZS nets coincide, in the sense that we have a precise algebraic characterization of the basic moves allowed by the operational model.

EXAMPLE 4.3.4 *In our running example the prime arrows of* $\mathcal{Z}[MS]$ *are*

$$\begin{aligned}
\tau_0 &= t_0,\\
\tau_4 &= t_4,\\
\alpha_1 &= t_1; t_3,\\
\alpha_2 &= t_1; t_2; (t_3 \otimes id_{(0,z)}); t_3,\\
&\vdots\\
\alpha_i &= t_1; \beta_i; \delta_i,\\
&\vdots\\
\textit{with}\ \ \beta_i &= t_2; (t_2 \otimes id_{(0,z)}); \cdots; (t_2 \otimes id_{(0,(i-2)z)})\\
\delta_i &= (t_3 \otimes id_{(0,(i-1)z)}); \cdots; (t_3 \otimes id_{(0,z)}); t_3
\end{aligned}$$

The correspondence with the abstract stable transactions of MS, which are given in Example 3.4.3, is the intuitive one. As a further example, some more compact notations for defining α_i *are either* $\delta_i = t_3 \otimes \cdots \otimes t_3$ *where* t_3 *is repeated exactly* i *times or* $\alpha_i = t_1; t_2; (t_2 \otimes t_3); \cdots; (t_2 \otimes t_3); (t_3 \otimes t_3)$ *where the expression* $(t_2 \otimes t_3)$ *appears exactly* $i-2$ *times.*

4.3.2 Abstract Semantics as Coreflection

We now present the universal construction yielding the abstract semantics of our nets. To this purpose we define a category **ZSN** of ZS nets whose morphisms can map a transition into a transaction. This construction is reminiscent of the construction of **ImplPetri** in [106].

DEFINITION 4.3.4 (ABSTRACT TRANSITION)
An abstract transition *of a given ZS net* B *is either a prime arrow of* $\mathcal{Z}[B]$ *or a transition of* B *(seen as an arrow in* $\mathcal{Z}[B]$*).*

DEFINITION 4.3.5 (REFINEMENT MORPHISM)
Given two ZS nets B *and* B'*, a* refinement morphism $h : B \longrightarrow B'$ *is a ZS net morphism* $(f, g_L, g_Z) : B \longrightarrow \mathcal{Z}[B']$ *such that function* f *maps transitions into abstract transitions.*

LEMMA 4.3.6
Given a refinement morphism $h : B \longrightarrow B'$*, let* $\widetilde{h}$ *be its unique extension in* **HCatZPetri**. *Then,* $\widetilde{h}$ *preserves prime arrows.*

Proof. We want to show that if α is prime in $\mathcal{Z}[B]$, then also $\widetilde{h}(\alpha)$ is prime in $\mathcal{Z}[B']$. Now let $u[t_1\rangle u_1 \cdots u_{n-1}[t_n\rangle u_n$ be a firing sequence corresponding to a generic linearization of α. We can distinguish two cases:

- If $n = 1$ then $\widetilde{h}(\alpha) = \widetilde{h}(t_1) = h(t_1)$ which is prime.
- If $n > 1$ then we proceed by contradiction. Suppose that $\widetilde{h}(\alpha)$ is not prime; this implies that $\exists \beta_1, \beta_2 \in \mathcal{Z}[B']$ with $\widetilde{h}(\alpha) = \beta_1 \otimes \beta_2$. Since α is prime, then each t_i involves at least a zero place. Hence, by Definition 4.3.4, each $h(t_i)$ is a transition.

  This induces a corresponding linearization for $\widetilde{h}(\alpha)$, given by

  $$h(u)[h(t_1)\rangle h(u_1) \cdots h(u_{n-1})[h(t_n)\rangle h(u_n).$$

  Then, let $v[s_1\rangle v_1 \cdots v_{k-1}[s_k\rangle v_k$ and $w[s_{k+1}\rangle w_1 \cdots w_{l-1}[s_n\rangle w_l$, with $k + l = n$, be some firing sequences corresponding to β_1 and β_2 respectively.

  Now, suppose that a diamond transformation at position i can be applied to $\widetilde{h}(\alpha)$, i.e., $h(t_i)$ and $h(t_{i+1})$ are both enabled at $h(u_{i-1})$.

  The disjoint image property of h allows to infer that also t_i and t_{i+1} are both enabled at u_{i-1}, so a diamond transformation can also be applied to α at position i.

  Iteratively applying to α the *specular* diamond transformations needed to reach the sequence $h(u)[s_1\rangle u''_1 \cdots u''_{n-1}[s_n\rangle u''_n$ starting from $\widetilde{h}(\alpha)$, we obtain the sequence $u[t_{i_1}\rangle u'_1 \cdots u'_{n-1}[t_{i_n}\rangle u'_n$, with $u'_n = u_n$ and where $h(u'_j) = u''_j$ and $h(t_{i_j}) = s_j$ for $j = 1, \ldots, n$. It is immediate to show that the sequences $v'[t_{i_1}\rangle v'_1 \cdots v'_{k-1}[t_{i_k}\rangle v'_k$ and $w'[t_{i_{k+1}}\rangle w'_1 \cdots w'_{l-1}[t_{i_n}\rangle w'_l$ (with $v' \oplus w' = u$, $h(v') = v$, $h(v'_j) = v_j$ for $j = 1, \ldots, k$, $h(w') = w$ and $h(w'_j) = w_j$ for $j = 1, \ldots, l$) define two arrows α_1 and α_2 such that $\alpha = \alpha_1 \otimes \alpha_2$, thus contradicting the assumption that α is prime.

  □

REMARK 4.3.5 *The disjoint image property on zero places required for the morphisms in* **dZPetri** *is necessary for Lemma 4.3.6 to hold. As an example, consider ZS nets B and B' in Figure 4.3. Then morphism $h : B \longrightarrow B'$, merging both zero places x' and x'' into x and leaving unchanged the rest, maps the prime arrow $\alpha = (t_1 \otimes t_2); (t_3 \otimes t_4) \in \mathcal{Z}[B]$ into the arrow*

$$h(\alpha) = (t_1 \otimes t_2); (t_3 \otimes t_4) = (t_1; t_3) \otimes (t_2; t_4)$$

which is not prime. In fact, the morphism h does not respect the disjoint image property.

DEFINITION 4.3.6 (CATEGORY **ZSN**)
The category **ZSN** *has ZS nets as objects and refinement morphisms as arrows. The composition between two refinement morphisms $h : B \longrightarrow B'$ and $h' : B' \longrightarrow B''$ is defined as the ZS net morphism $h; \widetilde{h}' : B \longrightarrow \mathcal{Z}[B'']$, where $\widetilde{h}'$ is the unique extension of h' to a morphism in* **HCatZPetri**. *Notice that Lemma 4.3.6 guarantees that $h; \widetilde{h}'$ is a refinement morphism.*

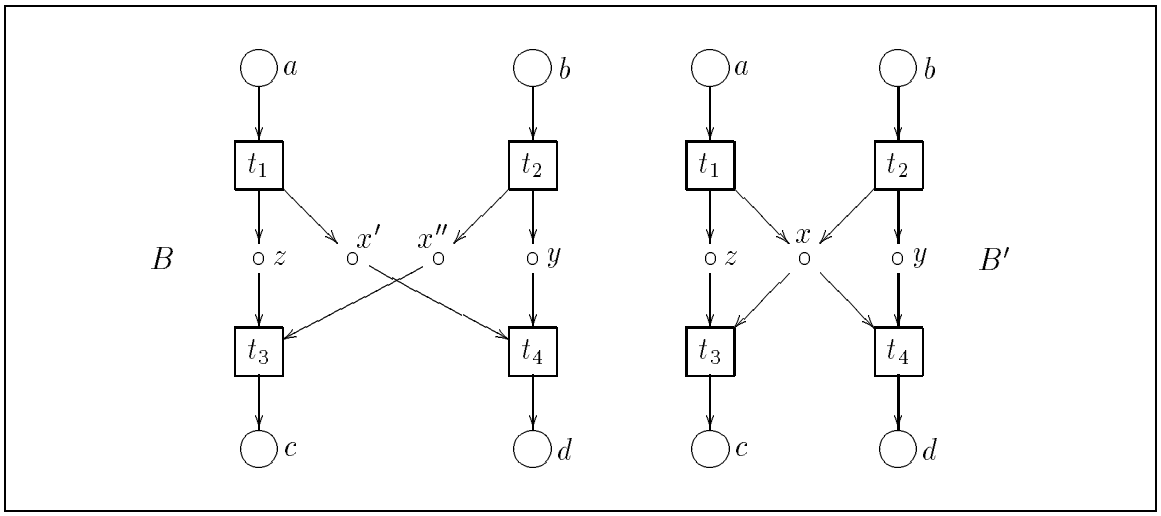


Figure 4.3: Two ZS nets that are used to illustrate the importance of the disjoint image property (see Remark 4.3.5).

THEOREM 4.3.7
Category **Petri** *is embedded into* **ZSN** fully *and* faithfully *as a coreflective subcategory. Furthermore the functor* $\mathcal{A}[\_]$*, which is the right adjoint of the coreflection, maps every ZS net* B *into its abstract net* A_B *(see Definition 3.4.5), i.e.,* $\mathcal{A}[B] = A_B$.

Proof. We start by defining the functor $\mathcal{D}[\_] : \mathbf{Petri} \longrightarrow \mathbf{ZSN}$ by letting

$$\mathcal{D}[(S^{\oplus}, T, \partial_0, \partial_1)] = (S^{\oplus} \times \{0\}, T, (\partial_0, 0), (\partial_1, 0)),$$

i.e., $\mathcal{D}[N]$ is the ZS net generated by N whose nodes are renamed as pairs having the second component equal to 0. The abstract stable transactions (i.e., prime arrows, by Theorem 4.3.5) of $\mathcal{D}[N]$ are all and only its transitions. Thus, a refinement morphism $h : \mathcal{D}[N] \longrightarrow \mathcal{D}[N']$ maps transitions into transitions.

We extend $\mathcal{D}[\_]$ to a functor by defining $\mathcal{D}[(f, g)] = (f, g, 0)$.

Next we want to prove that $\mathcal{D}[\_] \dashv \mathcal{A}[\_] : \mathbf{ZSN} \longrightarrow \mathbf{Petri}$ where $\mathcal{A}[\_]$ maps each ZS net B into its *abstract net* A_B.

Consider a refinement morphism $h = (f, g_L, g_Z) : B \longrightarrow B'$. Let $\widetilde{h}$ be the unique extension of h in **HCatZPetri**. Morphism $\widetilde{h}$ preserves prime arrows (by Lemma 4.3.6). Then mapping $\mathcal{A}[\_]$ extends to a functor by defining $\mathcal{A}[h] = (f', g_L)$ with $f'(\sigma) = \widetilde{h}(\sigma)$ for any $\sigma \in \Upsilon_B$ (we recall that Υ_B denotes the set of all the abstract stable transactions of B).

It follows that the unit component η^C_N of the adjunction is the identity and the counit component ϵ^C_B maps transitions of the abstract net into appropriate abstract transactions. □

The previous theorem is very important, because the universal property of the coreflection confirms that the abstract counterpart A_B of a zero-safe net B (see Definition 3.4.5) is the P/T net that better approximates the behaviour of B according to the *CTph* interpretation.

4.4 Individual Token Approach

The aim of this section is to propose an algebraic characterization of the definitions and the constructions presented in Section 3.5. In doing this, we will follow the outline of the previous section using analogous concepts of category theory. Once again, the *category of models* (objects are models and arrows represent some notion of simulation) is the category **dZPetri** presented in Definition 4.3.1. Then we consider a construction that exhibits an *adjunction* from **dZPetri** to a category **ZSCGraph**. It is proved that this adjunction is strictly related to the operational semantics of ZS nets as defined in Section 3.5. Our second construction starts from a category **ZSC** of ZS nets and more complex morphisms, having the ordinary category **Petri** of P/T nets as a subcategory, and yields a *coreflection* corresponding exactly to the construction of the causal abstract net in Definition 3.5.6.

4.4.1 Operational Semantics as Adjunction

We present here the universal construction yielding the operational semantics of zero-safe nets under the *ITph* interpretation.

The disjoint image property plays a very important rôle here, avoiding the awkward situation arising from Example 4.2.1. Moreover, if we identified two different zero places via a (non-disjoint) morphism then the behaviour of the abstract model might dramatically change. Since we use zero places to specify a suitable synchronization mechanism, it is important to ensure that this mechanism is always preserved.

The next definition introduces a category of more structured models (called *ZS causal graphs*) which is reminiscent of the constructions of the marking graph $\mathcal{C}[N]$ and of the free ssmc $\mathcal{F}[N]$ (see Sections 4.2.1 and 4.2.2).

DEFINITION 4.4.1 (CATEGORY **ZSCGraph**)
A ZS causal graph $E = ((L \cup Z)^{\oplus}, (T, \otimes, 0, id, *), \partial_0, \partial_1)$ *is both a ZS net and a reflexive Petri monoid. In addition, it comes equipped with a partial function* $\_ * \_$ *called* horizontal composition*:*

$$\frac{\alpha : (u, x) \longrightarrow (v, y),\ \beta : (u', y) \longrightarrow (v', y')}{\alpha * \beta : (u \oplus u', x) \longrightarrow (v \oplus v', y')}$$

and a family of horizontal swappings $\{e_{x,y} : (0, x \oplus y) \longrightarrow (0, y \oplus x)\}_{x,y \in Z^{\oplus}}$.

Horizontal composition is associative and has identities $id_{(0,x)}$ *for any* $x \in Z^{\oplus}$. *The monoidal operator* $\_ \otimes \_$ *is functorial w.r.t. horizontal composition. The (horizontal)* naturality *axiom*

$$e_{x,x'} * (\beta \otimes \alpha) = (\alpha \otimes \beta) * e_{y,y'}$$

holds for any $\alpha : (u, x) \longrightarrow (v, y)$ *and* $\beta : (u', x') \longrightarrow (v', y')$.

Moreover, the following coherence *axioms are satisfied for any* $x, y, y' \in Z^{\oplus}$*:*

$$e_{x,y} * e_{y,x} = id_{(0,x\oplus y)} \qquad e_{x,y\oplus y'} = (e_{x,y} \otimes id_{(0,y')}) * (id_{(0,y)} \otimes e_{x,y'}).$$

A morphism h *between two ZS causal graphs* E *and* E' *is a ZS net morphism which in addition respects horizontal composition and swappings, i.e. such that* $h(\alpha * \beta) = h(\alpha) *' h(\beta)$ *and* $h(e_{x,y}) = e'_{h(x),h(y)}$.

This defines the category **ZSCGraph**.

Horizontal composition is the key feature of our approach. It behaves like sequential composition on zero places and like the ordinary parallel composition on stable places. This is necessary to avoid the construction of steps which reuse stable tokens. Swappings are used to specify the causality relation among produced and consumed zero tokens.

PROPOSITION 4.4.1
If $\alpha : (u, 0) \longrightarrow (v, 0)$ *and* $\alpha' : (u', 0) \longrightarrow (v', 0)$ *are two arrows of a ZS causal graph then* $\alpha \otimes \alpha' = \alpha' \otimes \alpha$ *and* $\alpha * \alpha' = \alpha \otimes \alpha'$.

Proof. It can be easily verified that:

$$\begin{aligned} \alpha \otimes \alpha' &= (\alpha \otimes id_{(0,0)}) * (id_{(0,0)} \otimes \alpha') = \alpha * \alpha' \\ &= (id_{(0,0)} \otimes \alpha) * (\alpha' \otimes id_{(0,0)}) = \alpha' \otimes \alpha. \end{aligned}$$

□

PROPOSITION 4.4.2
The full subcategory of **ZSCGraph** *whose objects are Petri nets (i.e.,* $Z = \emptyset$*) is isomorphic to* **CMonRPetri**.

Proof. Let E be a ZS causal graph such that Z_E is empty. Then for all $\alpha \in T_E$, there exist $u, v \in L_E^{\oplus}$ such that $\alpha : (u, 0) \longrightarrow (v, 0)$. Thus $\_*\_$ is no longer partial and it can be applied to any pair of arrows: $\forall \alpha, \beta \in E$ it follows that $\alpha * \beta = \alpha \otimes \beta$ (by Proposition 4.4.1), i.e., the horizontal composition adds no structure and the parallel composition is commutative. Eventually, observe that the collection of swappings is empty. □

Next theorem defines the algebraic semantics of zero-safe nets by means of a universal property.

THEOREM 4.4.3
The obvious forgetful functor $\mathcal{U}_{\mathrm{I}}$: **ZSCGraph** $\longrightarrow$ **dZPetri** *has a left adjoint* $\mathcal{CG}$: **dZPetri** $\longrightarrow$ **ZSCGraph**.

Proof. Let the functor $\mathcal{CG}$: **dZPetri** $\longrightarrow$ **ZSCGraph** map a ZS net B into the ZS causal graph $\mathcal{CG}[B]$ whose arrows are generated by the inference rules in Table 4.3, modulo the axioms expressing that the arrows form a (strict) monoid:

$$\frac{(u,x) \in L_B^{\oplus} \times Z_B^{\oplus}}{id_{(u,x)} : (u,x) \longrightarrow (u,x) \in \mathcal{CG}[B]} \quad \textbf{(identities)}$$

$$\frac{t : (u,x) \longrightarrow (v,y) \in T_B}{t : (u,x) \longrightarrow (v,y) \in \mathcal{CG}[B]} \quad \textbf{(transitions)}$$

$$\frac{z, z' \in Z_B}{d_{z,z'} : (0, z \oplus z') \longrightarrow (0, z' \oplus z) \in \mathcal{CG}[B]} \quad \textbf{(symmetries)}$$

$$\frac{\alpha : (u,x) \longrightarrow (v,y),\ \beta : (u',x') \longrightarrow (v',y') \in \mathcal{CG}[B]}{\alpha \otimes \beta : (u \oplus u', x \oplus x') \longrightarrow (v \oplus v', y \oplus y') \in \mathcal{CG}[B]} \quad \textbf{(par. comp.)}$$

$$\frac{\alpha : (u,x) \longrightarrow (v,y),\ \beta : (u',y) \longrightarrow (v',z) \in \mathcal{CG}[B]}{\alpha * \beta : (u \oplus u', x) \longrightarrow (v \oplus v', z) \in \mathcal{CG}[B]} \quad \textbf{(seq. comp.)}$$

Table 4.3: The inference rules for $\mathcal{CG}[B]$.

$$(\alpha \otimes \beta) \otimes \delta = \alpha \otimes (\beta \otimes \delta), \text{ and } \ id_{(0,0)} \otimes \alpha = \alpha = \alpha \otimes id_{(0,0)}$$

(for any arrows $\alpha, \beta, \delta \in \mathcal{CG}[\mathcal{B}]$); the axioms expressing that horizontal composition $\_ * \_$ is associative and has identities:

$$(\alpha * \beta) * \delta = \alpha * (\beta * \delta), \text{ and } \ \alpha * id_{(0,y)} = \alpha = id_{(0,x)} * \alpha$$

(whenever such compositions are well-defined); the functoriality axioms

$$(\alpha \otimes \alpha') * (\beta \otimes \beta') = (\alpha * \beta) \otimes (\alpha' * \beta'), \text{ and } \ id_{(u,x)} \otimes id_{(v,y)} = id_{(u \oplus v, x \oplus y)}$$

(the former whenever the rightmost member of the equation is defined); finally, the axioms expressing that the collection of swappings $d_{x,y}$ plays the role of the "horizontal" natural isomorphism:

$$d_{x,x'} * (\beta \otimes \alpha) = (\alpha \otimes \beta) * d_{y,y'}, \text{ and } \ d_{z,z'} * d_{z',z} = id_{z \oplus z'}$$

(for any arrows $\alpha : (u,x) \longrightarrow (v,y), \beta : (u',x') \longrightarrow (v',y') \in \mathcal{CG}[B]$, and for any $z, z' \in Z_B$), where $d_{x,y}$ for $x, y \in Z_B^{\oplus}$ denotes any term obtained from the basic symmetries by applying recursively the rules:

$$\begin{aligned} d_{0,x} &= id_{(0,x)} = d_{x,0}, \\ d_{z \oplus x, y} &= (id_{(0,z)} \otimes d_{x,y}) * (d_{z,y} \otimes id_{(0,x)}), \text{ and} \\ d_{x, y \oplus z} &= (d_{x,y} \otimes id_{(0,z)}) * (id_{(0,y)} \otimes d_{x,z}). \end{aligned} \tag{4.5}$$

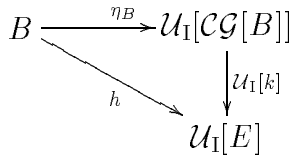


Figure 4.4: Universality of η_B in **dZPetri**.

It follows directly from the definition that $\mathcal{CG}[B]$ is a ZS causal graph. We must show that it is exactly the *free* ZS causal graph on B. Let $\eta_B : B \longrightarrow \mathcal{U}_{\mathrm{I}}[\mathcal{CG}[B]]$ be the ZS net morphism which is the identity on places and the obvious injection on transitions. We show that η_B is universal, i.e., that for any ZS causal graph E and for any ZS net morphism $h = (f, g_L, g_Z) : B \longrightarrow \mathcal{U}_{\mathrm{I}}[E]$, there exists a unique ZS causal graph morphism $k : \mathcal{CG}[B] \longrightarrow E$ such that the diagram in Figure 4.4 commutes in **dZPetri**. For such diagram to commute, morphisms k and h must agree on the generators of $\mathcal{CG}[B]$ and the extension of k to tensor and horizontal composition is uniquely determined by its definition on the generators, namely:

- $k((u,x)) = (g_L(u), g_Z(x))$,
- $k(t) = f(t)$,
- $k(id_{(u,x)}) = id_{(g_L(u),g_Z(x))}$,
- $k(d_{z,z'}) = e_{g_Z(z),g_Z(z')}$,
- $k(\alpha * \beta) = k(\alpha) * k(\beta)$, and
- $k(\alpha \otimes \beta) = k(\alpha) \otimes k(\beta)$.

To conclude the proof it remains to show that k is well-defined. This can be done by observing that k preserves the axioms which generate $\mathcal{CG}[B]$. □

The following theorem shows that the algebraic semantics of ZS nets is an extension of the ordinary semantics of P/T nets.

THEOREM 4.4.4
When restricted to P/T nets, functor $\mathcal{CG}$ coincides with $\mathcal{C}$.

Proof. From Proposition 4.4.1, because if $Z_B = \emptyset$ then for all $t : (u,x) \longrightarrow (v,y)$ in T_B it is the case that $x = y = 0$. □

The ZS causal graph $\mathcal{CG}[B]$ is still too concrete w.r.t. the operational semantics of ZS nets defined in the Section 3.5.1. More precisely we need two more axioms (which are the transposition of axioms (4.2) and (4.3) of Theorem 4.2.1).

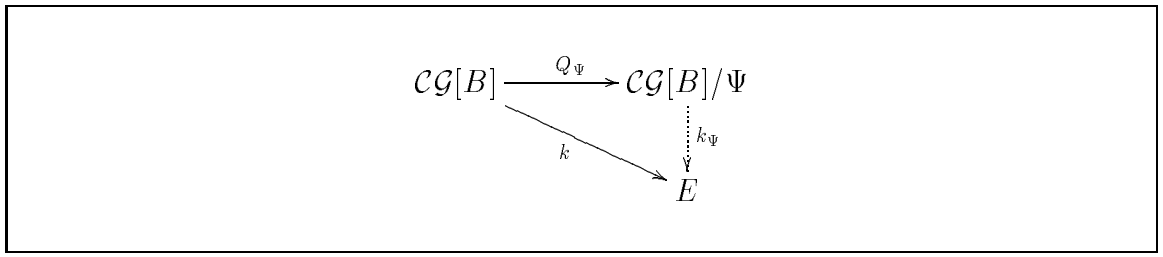


Figure 4.5: Quotient diagram in **ZSCGraph**.

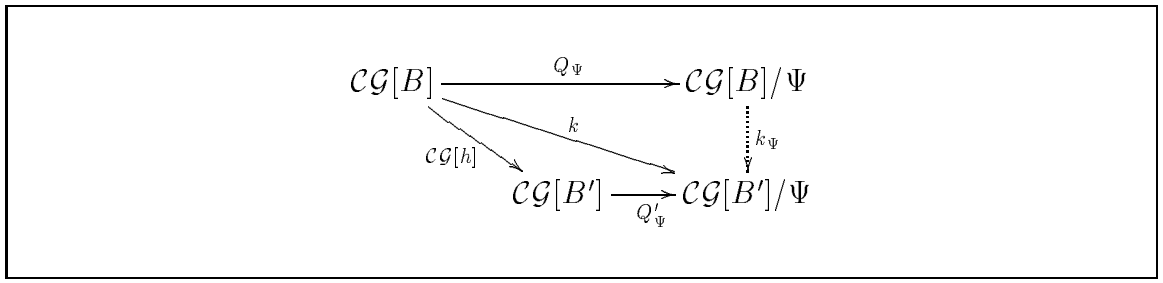


Figure 4.6: The unique extension of morphism $h : B \longrightarrow B'$ in **ZSCGraph**.

DEFINITION 4.4.2 (QUOTIENT Ψ)
Given a ZS net B, let us denote by $\mathcal{CG}[B]/\Psi$ the quotient of the free ZS causal graph $\mathcal{CG}[B]$ generated by B in **ZSCGraph** *modulo the axioms*

$$d_{z,z'} = id_{(0,z\oplus z')} \quad \textit{if} \quad z \neq z' \in Z_B, \tag{4.6}$$

$$d * t * d' = t \quad \textit{if} \quad t \in T_B, \textit{ and } d, d' \textit{ are swappings.} \tag{4.7}$$

The quotient $\mathcal{CG}[B]/\Psi$ of $\mathcal{CG}[B]$ is such that for any ZS causal graph morphism $k : \mathcal{CG}[B] \longrightarrow E$ which respects axioms (4.6) and (4.7) (i.e., $k(d_{z,z'}) = id_{(0,k(z)\oplus k(z'))}$ for any $z \neq z' \in Z_B$ and $k(d*t*d') = k(t)$ for any transition t and swappings d, d' in $\mathcal{CG}[B]$), there exists a unique arrow k_Ψ such that the diagram in Figure 4.5 commutes in **ZSCGraph**, where Q_Ψ is the obvious ZS causal graph morphism associated to the (least) congruence generated by the imposed axiomatization.

PROPOSITION 4.4.5
For any morphism $h : B \longrightarrow B'$ in the category **dZPetri** *there exists a unique extension $\hat{h} : \mathcal{CG}[B]/\Psi \longrightarrow \mathcal{CG}[B']/\Psi$ of h in* **ZSCGraph**.

Proof. Let h be as in the hypothesis and take

$$k = \mathcal{CG}[h]; Q'_\Psi : \mathcal{CG}[B] \longrightarrow \mathcal{CG}[B']/\Psi.$$

It can be easily verified that k respects axioms (4.6) and (4.7) (because h verifies the disjoint image property and maps transitions to transitions), thus k_Ψ is uniquely determined. Then, take $\hat{h} = k_\Psi = (\mathcal{CG}[h]; Q'_\Psi)_\Psi$ (see Figure 4.6). □

EXAMPLE 4.4.1 *Let MS be the ZS net of our running example whose set of arcs is defined in Example 4.3.2. For instance the arrow* $t_1 * t_3 \in \mathcal{CG}[MS]/\Psi$ *has source* $(2a, 0)$ *and target* $(2b, 0)$. *Instead, notice that the arrow* $(t_1 \otimes id_{(a,0)}) * (id_{(b,0)} \otimes t_3)$ *goes from* $(3a \oplus b, 0)$ *to* $(a \oplus 3b, 0)$.

As another example, the following expressions are all identified, i.e., they all denote the same arrow in $\mathcal{CG}[MS]/\Psi$*:*

$$\begin{aligned} t_1 * t_2 * (t_2 \otimes t_3) * (t_3 \otimes t_3) &= t_1 * t_2 * (t_2 \otimes id_{(0,z)}) * (t_3 \otimes t_3 \otimes t_3) \\ &= t_1 * t_2 * d_{z,z} * (t_2 \otimes id_{(0,z)}) * (t_3 \otimes t_3 \otimes t_3) \\ &= t_1 * t_2 * (id_{(0,z)} \otimes t_2) * (t_3 \otimes t_3 \otimes t_3) \\ &= t_1 * t_2 * (t_3 \otimes t_2) * (t_3 \otimes t_3). \end{aligned}$$

To give the expected correspondence between algebraic and operational semantics we rephrase in the current setting the definition of prime arrows given for the category **HCatZPetri**. Since the two notions are essentially the same, we hope that the reader will not be confused by the "terminology overloading."

DEFINITION 4.4.3 (PRIME ARROW II)
An arrow $\alpha : (u, 0) \longrightarrow (v, 0)$ *of a ZS causal graph* E *is* prime *if and only if* α *cannot be expressed as the monoidal composition of non-trivial arrows.*

THEOREM 4.4.6
Given any zero-safe net B*, there is a one-to-one correspondence between arrows* $\alpha : (u, 0) \longrightarrow (v, 0) \in \mathcal{CG}[B]/\Psi$ *and the connected steps of* B*. Moreover, if such an arrow is prime (and it is not an identity) then the corresponding connected step is a connected transaction.*

Proof. For any arrow β of $\mathcal{CG}[B]/\Psi$ we define inductively on the structure of β a concatenable process $C(\beta)$ of N_B as follows:

- $C(t) = t$,
- $C(id_{(u,x)}) = id_u \otimes id_x$,
- $C(d_{z,z}) = c_{z,z}$,
- $C(\beta' \otimes \beta'') = C(\beta') \otimes C(\beta'')$, and
- $C(\beta' * \beta'') = (C(\beta') \otimes u'); (v \otimes C(\beta''))$, where $\beta' : (u, x) \longrightarrow (v, y)$ and $\beta'' : (u', x') \longrightarrow (v', y')$.

Different expression denoting β yields the same result, because all the axioms are verified by the algebra of processes. Moreover, if $\alpha : (u, 0) \longrightarrow (v, 0) \in \mathcal{CG}[B]/\Psi$ then the process obtained from $C(\alpha)$ by forgetting the label-indexed ordering functions of origins and destinations denotes a connected step of B.

Conversely, let $\xi = [\![\omega]\!]_{\approx}$ (for some causal firing sequence ω) be a connected step. The concatenable process $pr(\omega)$ of N_B can be denoted algebraically as the sequential and parallel composition of transitions, identities and symmetries. Moreover, we can always take an equivalent process C without stable symmetries, i.e., such that it can be expressed as $\alpha_1; \ldots; \alpha_n$ where $\alpha_i = \beta_i \otimes u_i \otimes x_i$ with $u_i \in L_B^{\oplus}$, $x_i \in Z_B^{\oplus}$ and $\beta_i \in T_B \cup \{c_{kz,z}\}_{z \in Z_B, k \in \mathbb{N}}$. Then, take $\alpha' = \alpha'_1 * \ldots * \alpha'_n$ where $\alpha'_i = \beta'_i \otimes x_i$ and $\beta'_i = \beta_i$ if $\beta_i \in T_B$ and $\beta'_i = d_{kz,z}$ if $\beta_i = c_{kz,z}$ for some zero place z and integer k. Hence, $\alpha = \alpha' \otimes u'$ where u' is the multiset of idle tokens.

To show the correspondence between prime arrows and connected transaction we proceed by contradiction. Suppose that β is prime, but $C(\beta)$ is either not connected or not full. If $C(\beta)$ is not connected then it is the parallel composition of two processes, thus contradicting the hypothesis that β is prime. If $C(\beta)$ is not full, then there is some idle token, yet contradicting the hypothesis that β is prime. □

EXAMPLE 4.4.2 *In our running example, some prime arrows of* $\mathcal{CG}[MS]$ *are* t_0, $t_1 * t_3$, *and* $t_1 * t_2 * (t_2 \otimes t_2) * (t_3 \otimes t_2 \otimes t_3 \otimes t_3) * (t_3 \otimes t_3)$. *As a counterexample, the arrow* $(t_1 \otimes t_1) * d_{z,z} * (t_2 \otimes t_3) * (t_3 \otimes t_3)$ *is not prime, because*

$$\begin{aligned}(t_1 \otimes t_1) * d_{z,z} * (t_2 \otimes t_3) * (t_3 \otimes t_3) &= (t_1 \otimes t_1) * (t_2 \otimes t_3) * (t_3 \otimes t_3) \\ &= (t_1 * t_2 * (t_3 \otimes t_3)) \otimes (t_1 * t_3).\end{aligned}$$

4.4.2 Abstract Semantics as Coreflection

We conclude this section by presenting the universal construction which corresponds to the abstract semantics of ZS nets in the *ITph*. We make use of a category **ZSC** whose objects are ZS nets and whose morphisms allow for the refinement of a transition into an abstract connected transaction.

DEFINITION 4.4.4 (CAUSAL ABSTRACT TRANSITION)
Given the ZS net B, *a* causal abstract transition *of a* $\mathcal{CG}[B]/\Psi$ *is either a prime arrow of* $\mathcal{CG}[B]/\Psi$ *or a transition of* B *(seen as an arrow in* $\mathcal{CG}[B]/\Psi$*).*

DEFINITION 4.4.5 (CAUSAL REFINEMENT MORPHISM)
Given two ZS nets B *and* B', *a* causal refinement morphism $h : B \longrightarrow B'$ *is a ZS net morphism* $h = (f, g_L, g_Z)$ *from* B *to (the image through the forgetful functor of)* $\mathcal{CG}[B']/\Psi$ *such that function* f *maps transitions into causal abstract transitions.*

Since morphism h verifies the disjoint image property, a transition can be refined into a transaction if and only if both its preset and its postset are stable. Transitions involving zero places can only be mapped to transitions.

LEMMA 4.4.7
Each causal refinement morphism $h : B \longrightarrow B'$ *uniquely extends to a morphism* $\hat{h} : \mathcal{CG}[B]/\Psi \longrightarrow \mathcal{CG}[B']/\Psi$ *in* **ZSCGraph**, *and such extension preserves prime arrows.*

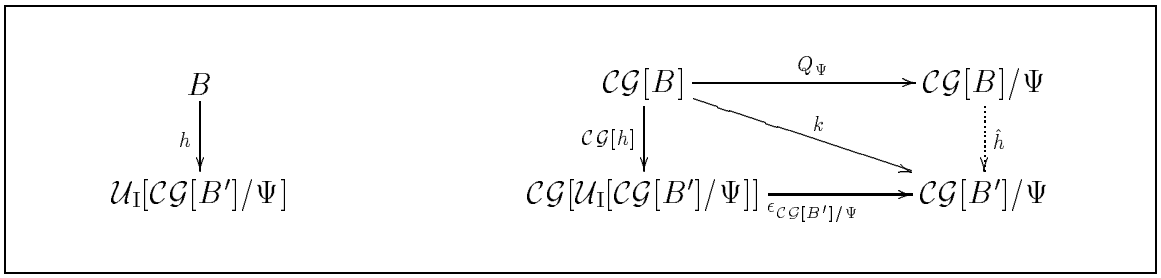


Figure 4.7: The unique extension of morphism $h : B \longrightarrow \mathcal{U}_{\mathrm{I}}[\mathcal{CG}[B']/\Psi]$ in **ZSCGraph**.

Proof. Let $\epsilon_E : \mathcal{CG}[\mathcal{U}_{\mathrm{I}}[E]] \longrightarrow E$ denote the *counit* of the adjunction between **dZPetri** and **ZSCGraph** defined in Theorem 4.4.3, then we can assume $\hat{h} = k_\Psi$ for $k = \mathcal{CG}[h]; \epsilon_{\mathcal{CG}[B']/\Psi}$ (see Figure 4.7).

Preservation of prime arrows can be easily observed at the level of the corresponding processes (the properties of connectedness and fullness are obviously preserved). □

Definition 4.4.6 (Category **ZSC**)
The category **ZSC** *has ZS nets as objects and causal refinement morphisms as arrows, their composition being defined through the extension in* **ZSCGraph** *given by Lemma 4.4.7.*

Theorem 4.4.8
Category **Petri** *is embedded in* **ZSC** *fully and faithfully as a coreflective subcategory. Furthermore, the functor $\mathcal{I}[\_]$, which is the right adjoint of the coreflection, maps every ZS net B into its causal abstract net I_B (see Definition 3.5.6).*

Proof (Sketch). The connected transactions (i.e., prime arrows, by Theorem 4.4.6) of a P/T net are all and only its transitions. Thus a causal refinement morphism $h : N \longrightarrow N'$ maps transitions into transitions.

Next we want to prove that the obvious inclusion functor from **Petri** to **ZSC** has a right adjoint $\mathcal{I}[\_] :$ **ZSC** $\longrightarrow$ **Petri** such that $\mathcal{I}[\_]$ maps each ZS net B into its causal abstract net I_B.

We verify that $\mathcal{I}[\_]$ extends to a functor. Consider a causal refinement morphism $h = (f, g_L, g_Z) : B \longrightarrow B'$. Let $\hat{h} : \mathcal{CG}[B]/\Psi \longrightarrow \mathcal{CG}[B']/\Psi$ be the unique extension of h in **ZSCGraph**. Morphism $\hat{h}$ preserves prime arrows (by Lemma 4.4.7). Then we define $\mathcal{I}[h] = (f', g)$ with $f'(\xi) = \hat{h}(\xi)$ for any $\xi \in \Xi_B$ and $g(a) = g_L(a)$ for any $a \in L_B$.

It follows that the unit component η^I_N of the adjunction is the identity and the counit component ϵ^I_B maps each transition of the abstract net into the appropriate connected transaction. □

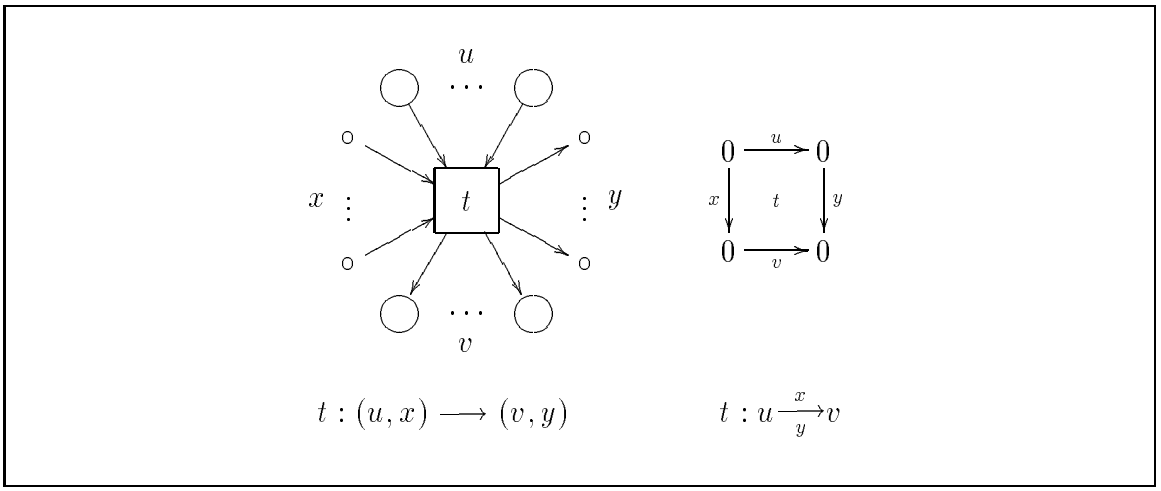


Figure 4.8: A generic ZS net transition t and its associated tile.

4.5 Tiles and Zero-Safe Nets

We want to mention a connection between ZS nets and the tile model [69]. We recall that tiles are rewrite rules, similar to SOS inference rules, equipped with three operations of composition: horizontal, vertical and parallel. Horizontal, vertical, and parallel compositions build tiles corresponding respectively to synchronized steps, to sequentialized steps and to concurrent steps. Tiles can be exactly interpreted as double cells in suitable monoidal double categories, and provide an expressive and clean metalanguage to define a variety of models of computation.

ZS nets represent the simple case where basic tiles are net transitions. The idea is to represent the commutative monoids of stable and zero places as two suitable monoidal categories with only one object called 0, which is the same in both representations and it is obviously the unit element of the tensor product. Notice that sequential composition is equivalent to parallel composition in both categories. In fact for any $f, g : 0 \longrightarrow 0$

$$f; g = (f \otimes 0); (0 \otimes g) = (f; 0) \otimes (0; g) = f \otimes g.$$

Then, a generic ZS net can be thought of as the computad of a (symmetric) strict monoidal double category [21, 69], with only one object 0, and whose horizontal and vertical arrows consist of the elements of the free commutative monoid of stable and zero places respectively (see Figure 4.8). In this setting, the horizontal composition of tiles corresponds to the horizontal composition of arrows in the category of ZS (causal) graphs, and the vertical composition of (prime) tiles yields step sequences (see Definition 3.4.1).

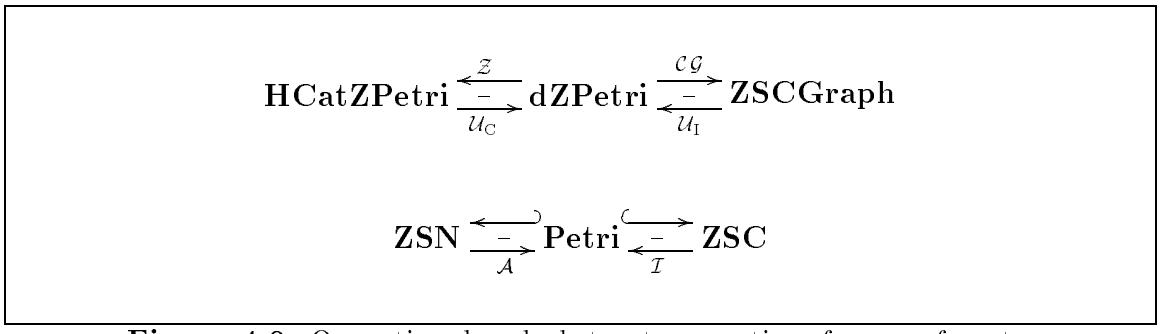


Figure 4.9: Operational and abstract semantics of zero-safe nets.

4.6 Summary

In this chapter we have presented both the *CTph* and *ITph* categorical semantics for zero-safe nets. Correspondingly, for the *CTph* approach, our categorical models are based on monoidal graphs equipped with an operation of horizontal composition, and for the *ITph* approach, we have introduced an additional collection of special transitions, called *swappings*, to represent the permutations of tokens which are all present at the same time into the same place, following the style of [49].

The categorical semantics (summarized by the four adjunctions illustrated in Figure 4.9) recover the operational and abstract semantics proposed in Chapter 3, introducing an algebraic characterization of the whole framework (e.g., prime arrows define the transactions).

Moreover, since left (right) adjoints preserve colimits (limits), it follows that several constructions on the refined model are preserved by the operational and abstract semantics. For instance, the algebraic model of the *pushout* of two nets — which is often useful for combining two nets merging some of their places — is the pushout of their semantics.

We have also sketched the correspondence with the tile model, which motivated the idea of the zero-safe net framework. Indeed, the definition of zero-safe net arise from the study of a simple tile model whose configurations and observations are Petri net markings (over different set of places). In our opinion, this means that the analysis of different tile formats could open several interesting research directions also in well-studied fields.

Related Works on Net Abstraction/Refinement

Several notions of net abstraction/refinement have been proposed in the literature. An extensive comparison of the different approaches can be found in [17, 70]. Most of the times they find application in a top-down design procedure of a concurrent system, where one starts with a very abstract model of the system and then replaces some components by more detailed structures, until the desired level of abstraction is reached.

Typically such refinement process takes several steps to be completed, where at each step a single transition (let say t) of the actual net N is refined into a suitable subnet M, yielding the net $N[t \to M]$ (the net M is usually equipped with initial and final transitions that, in $N[t \to M]$, become connected to the pre- and post-set of t by some standard mechanism, e.g. suitable place merging). We can call *structural* such kind of refinement.

Obviously, a minimal requirement in structural refinement is that the behaviour of the refined net is in some sense consistent with the one of the net to be refined. In general some constraints must be assumed on the "daughter" net M to accomplish the expected behavioural consistency, as e.g. in [134, 132, 131]. In [135], the refinement of places (as opposed to the refinement of transitions) is considered in the context of a general synchronization operator on nets, which composes nets by merging transitions.

A different and appealing idea is elaborated in [83, 84], where transitions can be interpreted as procedure calls. According to this view, the firing of a transition creates an instance of its corresponding subnet, and such calls can also be recursive.

Our approach is more "*behaviour* oriented," in the style of [137, 106, 126]. In particular, for the first time we apply the idea sketched in [106] and then hinted at in successive works, of refining transitions with *behaviours* of a more concrete net, using a completely algebraic approach. Speaking in terms of categories, since the behaviour of a net can be specified by a free construction, we take a suitable subcategory of the Kleisli category associated to such construction (the choice of the subcategory being justified by the atomicity requirements imposed on transactions).

This is clearly different from the structural approach described above, in that we refine all the transitions of the net N by computations of another (zero-safe) net. In particular, the refinements of two transitions t_1 and t_2 of N might use the same kind of resources (e.g., place z of our running example), whereas in the structural approaches, the nets M_1 and M_2 that refine t_1 and t_2 are disjoint (except for places already shared by the pre- and post-sets of t_1 and t_2). Moreover, the coreflections of Theorems 4.3.7 and 4.4.8 yield the notion of abstract net associated to the refined net, which is missing in all structural approaches.

To conclude, we only want to mention [54], where abstraction of nets, according to feasible equivalence relations that define the coarsening, is taken as a primitive notion (as opposed to refinement), and the works on *Petri boxes* (see e.g. [10, 9]).

Part II

Mapping Tile Logic into Rewriting Logic

Abstract of Part II

In a similar way as 2-categories can be regarded as a special case of double categories, *rewriting logic* (in the unconditional case) can be embedded into the more general *tile logic*, where also side effects and rewriting synchronization are considered. Since rewriting logic is the semantic basis of several language implementation efforts, it is useful to map tile logic back into rewriting logic in a conservative way, to obtain executable specifications of tile systems.

We extend the results of earlier work by Meseguer and Montanari [107], focusing on some interesting cases where the mathematical structures representing *configurations* (i.e., states) and *observations* are very similar, in the sense that they have in common some auxiliary structure (e.g., for tupling, projecting, etc.). In particular, we give in full detail the description of two such cases where (net) *process-like* and ordinary *term* structures are employed. Corresponding to these two cases, we introduce two categorical notions, namely, *symmetric strict monoidal double category* and *cartesian double category with consistently chosen products*, which seem to offer an adequate semantic setting for process and term tile systems.

The new model theory of *2EVH-categories*, required to relate the categorical models of tile logic and rewriting logic, is presented making use of *partial membership equational logic*. Consequently, symmetric strict monoidal and cartesian classes of double categories and 2-categories are compared through their embedding in the corresponding versions of 2EVH-categories. As a result of this comparison we obtain a correct rewriting implementation of tile logic.

The results presented in this part are joint work with José Meseguer and Ugo Montanari.

Category theory has been called "abstract nonsense"
by both its friends and its detractors.

— JOSEPH A. GOGUEN, A categorical manifesto

Chapter 5

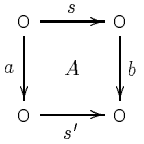
Process and Term Tile Logic

Contents

5.1 Motivations

In this chapter we characterize suitable semantic frameworks for two interesting classes of tile systems, where the similarities between the mathematical structures employed to model system states and their observable evolutions are exploited systematically.

Tile systems [66, 69] yield a formalism for modular descriptions of concurrent systems, extending both the SOS approach [121] and context systems [90] to a framework where the rules, called *tiles*, have a more general format. As summarized in the Introduction of the thesis (see also Section 2.4), a tile can be represented as a two-dimensional structure



also written $A : s \overset{a}{\underset{b}{\longrightarrow}} s'$, and states that that the *initial configuration* s can evolve to the *final configuration* s' producing an observable *effect* b, provided that the subcomponents of s evolve to the subcomponents of s', yielding the observation a, which is the *trigger* for the application of the tile A. Tiles can be composed horizontally (synchronizing the effect of the first rule with the trigger of the second rule), vertically (computational evolutions of a certain component), and in parallel (concurrent steps). These three operations are denoted by $\_ * \_$, $\_ \cdot \_$ and $\_ \otimes \_$ respectively, and satisfy particular composition properties.

Although the tile model can be equipped with a purely logical presentation (by analogy with *rewriting logic* [98, 99]), where the tiles are considered as special sequents subject to certain inference rules (see, e.g., Section 2.4), tile systems can be more generally seen as (*strict*) *monoidal double categories*. In this case tiles are cells, the configurations are arrows of the horizontal 1-category, and the observations are arrows of the vertical 1-category, where objects are just variables used to connect the "static" horizontal category with the "dynamic" vertical evolution. It follows that monoidal double categories are suitable semantic models for tile systems, where the three notions of composition between tiles are represented in a natural and intuitive way. A minimal requirement for modelling a system with tiles is that its configurations and observations define two suitable monoidal categories having the same objects. It might be the case that the mathematical structures employed in the horizontal and vertical dimensions are very similar, in the sense that they are generated from two (different) sets of basic constructors, adding the same *auxiliary structure*.

For example, Ferrari and Montanari [59, 60] have defined tile models for the (asynchronous) π-calculus [113] and for the representation of both the operational and the abstract semantics of CCS with locations [15], where both configurations and observations are *term graphs* [55].

We recall that term graphs are a reference-oriented generalization of the ordinary value-oriented notion of term. They are very useful to model the notion of *name sharing* that is essential for several applications (we can think of names as links to communication channels, or to objects, or to locations, or to remote shared resources, or, also, to some *cause* in the event history of the system). We can also think of term graphs as suitable net-process-like structures (enriched with a sharing mechanism), i.e., as arrows of suitable *symmetric strict monoidal categories*.

Also ordinary term structures can be used to describe both configurations and effects; in this case they can be modelled by *cartesian categories* (*with chosen products*) rather than by just (symmetric) monoidal categories. Thanks to the work of Lawvere relating *algebraic theories* and cartesian categories, and to classical results on cartesianity as enriched monoidality, the auxiliary structure which allows to generate the term algebra starting from a signature is characterized by three natural transformations called *symmetries*, *duplicators*, and *dischargers*.

Consequently, the models of these richer tile systems must rely on monoidal double categories equipped with some additional structure. Such structure must relate the auxiliary structures of configurations and observations and must verify some coherence properties. In this chapter, we introduce two original versions of tile logic, called *process tile logic*, and *term tile logic*.

In process tile logic, configurations and observations are defined in terms of a subclass of directed, acyclic hyper-graphs, where each node has at most one entering (exiting) arc. The "process" terminology is taken from *net theory*, due to the characterization of *concatenable (deterministic) processes* of P/T nets via symmetric strict monoidal categories [50]. Here configurations may model states of a great variety of distributed systems (at a certain level of abstraction), and observations may exactly model causal dependencies between the resources consumed and generated by concurrent/cooperative evolutions of agents. The models proposed in [59, 60] are essentially process tile logic models equipped with "ad-hoc" notions of sharing and garbage collection. Auxiliary tiles for process tile logic are essentially tiles for consistent permutations of interfaces in the two dimensions.

Term tile logic is the natural generalization of term rewriting logic. Here, both configurations and observations are elements of suitable term algebras. Auxiliary tiles of term tile logic are the consistent generalization of symmetries, duplicators and dischargers w.r.t. the two dimensions of tile systems.

The definition of symmetric and of cartesian double categories based on internal category constructions (that is, based on models of the sketch of categories in suitable structured categories, see Section 2.1.8) can only lead to asymmetric models, where the auxiliary structure is fully exploited in one dimension only. We believe that this should not be the case, as suggested by the kind of applications that we are developing as well as by conceptual reasons; therefore, we propose in this chapter the new notions of symmetric monoidal and cartesian double category that behave in the same way both in the horizontal and in the vertical dimension, and that fit very well our concurrency applications.

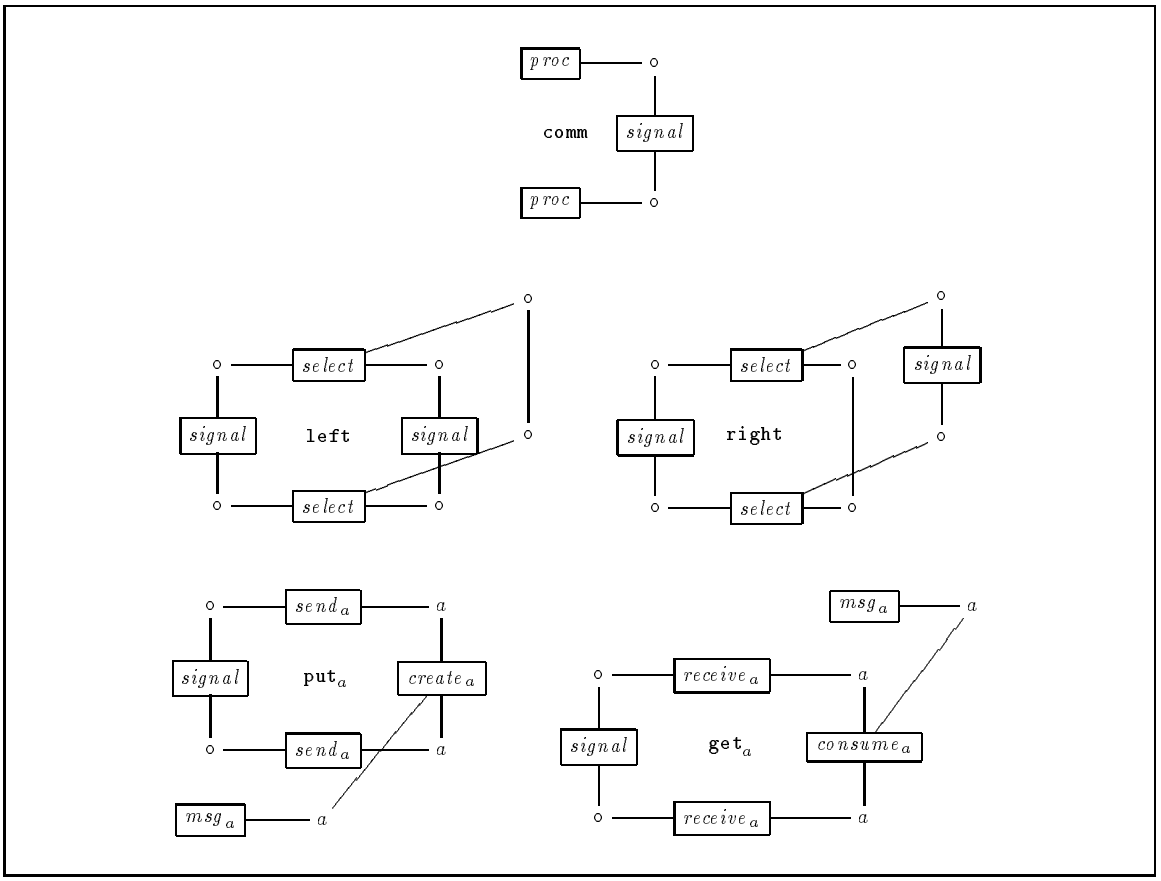


Figure 5.1: A graphical presentation of the tile system $\mathcal{R}_{SAMP}$.

5.1.1 A Message Passing Example

To illustrate with a simple example the nature of the auxiliary structures that we are investigating, let us consider the tile system $\mathcal{R}_{SAMP}$, whose basic tiles are depicted in Figure 5.1 using the "wire and box" notation, where labelled boxes represent the operators and wires represent the connections between the various components, the sequential composition $\_;\_$ merges the output interface of the first argument and the input interface of the second argument, and the parallel composition $\_ \otimes \_$ is represented picturing its arguments side-by-side.

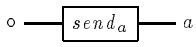
$\mathcal{R}_{SAMP}$ defines a computational model for *simple asynchronous message passing* systems, yielding the causality information on past communications. The basic idea is that each interface represents collections of communicating subsystems, and that observations give information about occurred communications.

Let $?$ be a given set of communication channels, ranged over by a. A *SAMP process* is a collection of *ports* dedicated to sending or receiving messages on some channels. We consider as interfaces the elements of $(? \cup \{\circ\})^*$, i.e., the "type" strings on the set of communication channels plus a special symbol $\circ$ for those processes that are not yet associated to some port. We use λ to denote the empty type string.

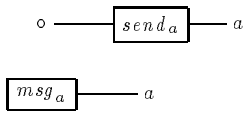
The configurations are the arrows of the symmetric strict monoidal category freely generated by the basic constructors $proc : \lambda \longrightarrow \circ$ (basic empty process), $msg_a : \lambda \longrightarrow a$ (generic message on the channel a), $send_a : \circ \longrightarrow a$ (representing a port dedicated to sending messages on the channel a), $receive_a : \circ \longrightarrow a$ (representing a port dedicated to receiving messages that have been posted on channel a), $select : \circ \longrightarrow \circ \otimes \circ$ (representing a binary mutual exclusion operator for the selection of I/O ports).

The observations are the arrows of the free symmetric strict monoidal category freely generated by the basic constructors $signal : \circ \longrightarrow \circ$ (the capability to communicate, either sending or receiving), $create_a : a \longrightarrow a \otimes a$ (posting of a message on channel a), and $consume_a : a \otimes a \longrightarrow a$ (retrieving a produced message from channel a).

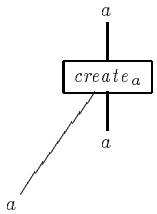
Graphically, the symmetric strict monoidal category structure of configurations and effects can be equivalently represented using acyclic wire and box diagrams where each node has at most one entering wire and at most one exiting wire (to avoid confusion, we remark that "boxes" are not nodes). Due to the parallel composition operation, some of the wire and box representations of the tiles defining $\mathcal{R}_{SAMP}$ require a three-dimensional interpretation of the pictures. For example the tile $\texttt{put}_a$ rewrites the initial configuration



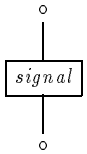
into the final configuration



producing the effect



but only if its subcomponents can provide the capability for the communication, which is the trigger



$$\texttt{comm} : proc \xrightarrow[signal]{} proc$$

$$\texttt{left} : select \xrightarrow[signal \otimes id_\circ]{signal} select \qquad \texttt{right} : select \xrightarrow[id_\circ \otimes signal]{signal} select$$

$$\texttt{put}_a : send_a \xrightarrow[create_a]{signal} msg_a \otimes send_a \qquad \texttt{get}_a : receive_a \otimes msg_a \xrightarrow[consume_a]{signal} receive_a$$

Table 5.1: The tile system $\mathcal{R}_{SAMP}$ presented using the algebraic notation.

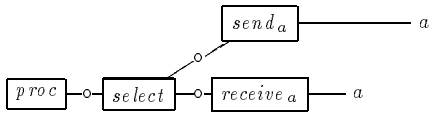


Figure 5.2: The configuration $P = proc; select; (receive_a \otimes send_a)$.

Adopting the algebraic notation for configurations and observations, the basic tiles of $\mathcal{R}_{SAMP}$ can be written as illustrated in Table 5.1.

The first tile, called `comm`, states that each basic process can decide to start a new communication, creating its own communication capability (represented by the effect *signal*).

The two tiles for the configuration *select*, labelled by `left` and `right`, implement the nondeterministic propagation of the communication capability to the ports of the process connected to the left or to the right output of the *select*.

The tile $\texttt{put}_a$ for the configuration $send_a$ states that the communication capability can be used to execute the send, creating a new message on the channel a and yielding the effect $create_a$. Analogously the tile $\texttt{get}_a$ models the receiving of a message. Notice that the "cyclic" behaviour of the ports makes this example very simple and suitable for illustrating our informal presentation of process tile systems.

For example, let us consider the following simple configuration which is pictured in Figure 5.2:

$$P = proc; select; (receive_a \otimes send_a),$$

The rule `comm` can be applied to the box *proc* of P only if the rest of P (i.e., the configuration $select; (receive_a \otimes send_a)$) can be rewritten using as trigger the observation *signal* produced by the application of `comm`. If one tries to use the rule `left` to propagate the capability *signal* to the $receive_a$ box, then rewriting gets stuck, because no message has been produced on the channel a, and the rule $\texttt{get}_a$ cannot be applied. However, the rule `right` can be employed to propagate the capability *signal* to the $send_a$ box. Then, the rule $\texttt{put}_a$ can be used to conclude the transaction (in parallel with the idle rewriting of the box $receive_a$).

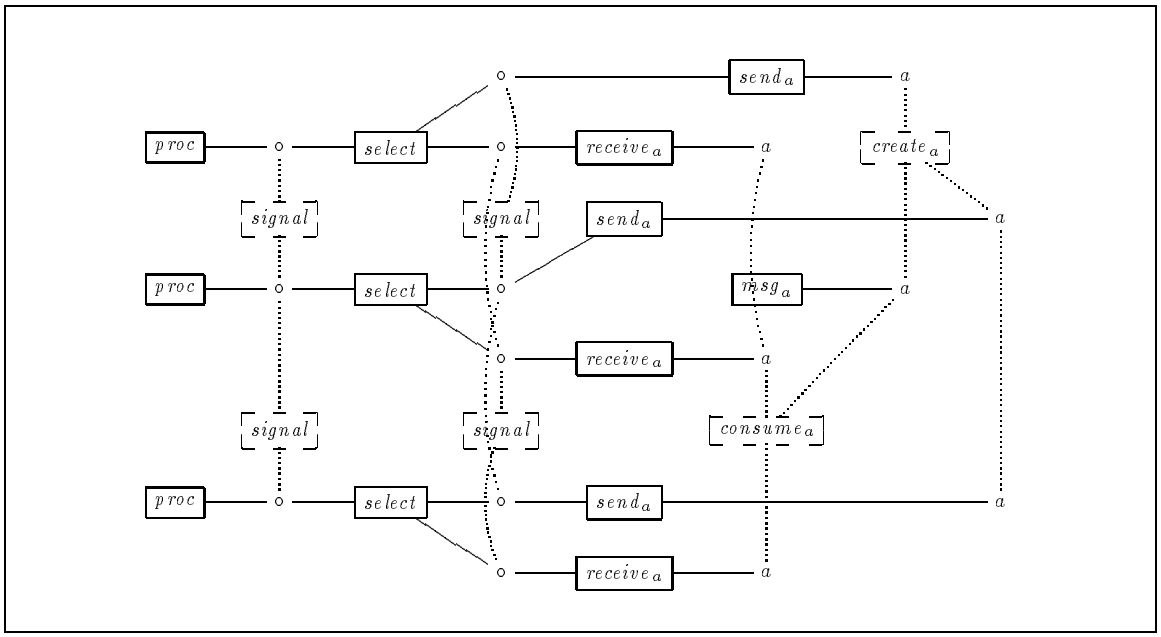


Figure 5.3: The wire and box representation of a tile computation for the configuration P. We have drawn observation diagrams using dotted wires and dashed boxes, for better readability.

At the successive step, the presence of a message on the channel a enables the application of the rule $\texttt{get}_a$ synchronizing it with the applications of rules `comm`, `left` and the idle rewriting of the box $send_a$. The resulting computation is fully illustrated in Figure 5.3.

Denoting the vertical identity of a generic configuration C by C itself, the proof of the first step of the computation is represented by the term

$$S_1 = \texttt{comm} * \texttt{right} * (receive_a \otimes \texttt{put}_a)$$

in the algebra of tiles (with some abuse of notation, we have used $receive_a$ to denote the vertical identity tile for the receive configuration). Similarly, the second step of the computation is denoted by the term $S_2 = \texttt{comm} * \texttt{left} * (\texttt{get}_a \otimes send_a)$. The whole computation is just the vertical composition $S_1 \cdot S_2$ of the two steps.

In the wire and box notation, symmetries are represented by crossing of wires. Vertical symmetries can be used to swap configurations; for example they are necessary to put a message produced elsewhere in the system closer to an enabled $receive_a$ port, making the specification more flexible (otherwise the tile $\texttt{put}_a$ should take into consideration the fact that the message to be consumed is not necessarily contiguous to the enabled $receive_a$).

The vertical symmetries swapping a message and a generic configuration $C : x \longrightarrow y$ (i.e., any parallel and sequential composition of identities and basic constructors) with $x, y \in (? \cup \{\circ\})^{\otimes}$ (i.e., x and y are elements of the monoid freely generated by $? \cup \{\circ\}$) are graphically illustrated in Figure 5.4.

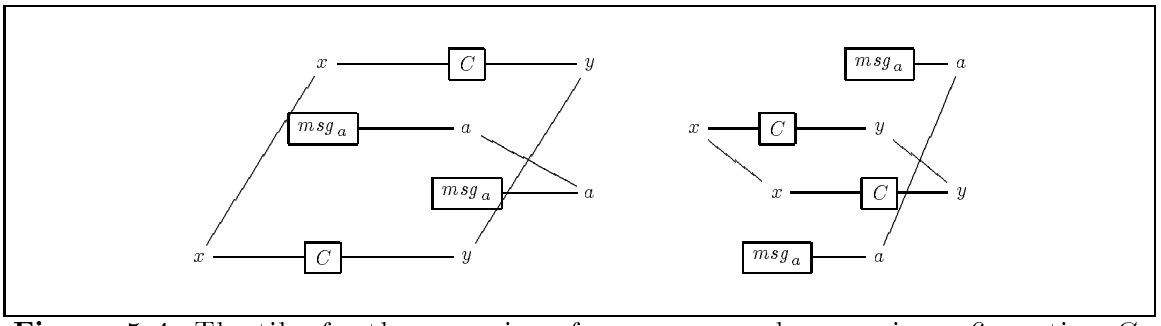


Figure 5.4: The tiles for the swapping of a *message* and a generic configuration C.

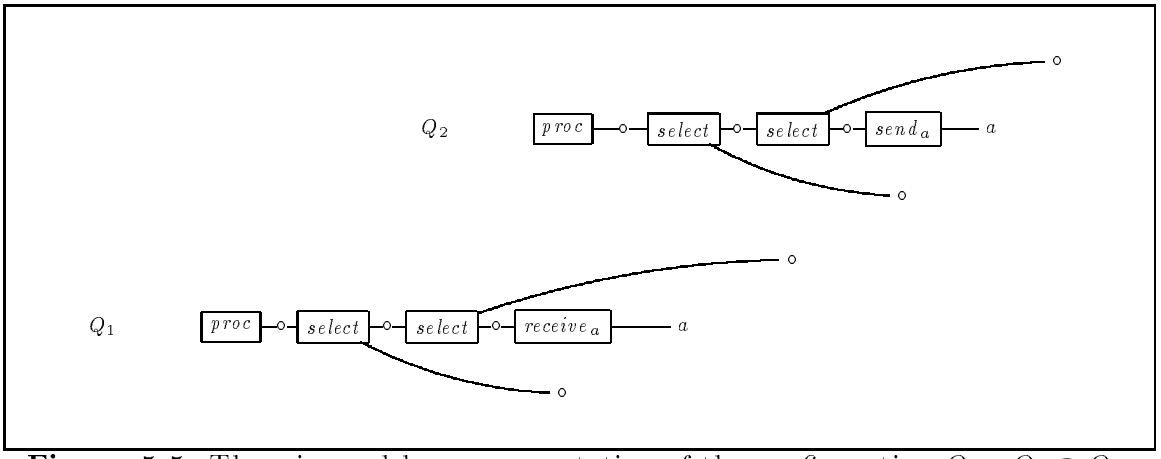


Figure 5.5: The wire and box representation of the configuration $Q = Q_1 \otimes Q_2$.

The tile on the left states that the configuration $msg_a \otimes C$ can be rewritten into $C \otimes msg_a$ yielding the symmetry that swaps the interfaces a and y as effect. The tile on the right defines the inverse rewriting (indeed, the vertical composition of the two tiles yields the identity).

Notice that the triggers of such tiles are identities, therefore they can be horizontally composed with the vertical identity of each configuration whose output interface matches the initial interface x of the configuration C. Horizontal symmetries are not needed in this case, because the vertical symmetries are expressive enough to represent all the possible communications.

For example, let us consider the configuration graphically represented in Figure 5.5, $Q = Q_1 \otimes Q_2$ with

$$\begin{aligned} Q_1 &= proc; select; (id_\circ \otimes (select; (receive_a \otimes id_\circ))) \\ Q_2 &= proc; select; (id_\circ \otimes (select; (send_a \otimes id_\circ))). \end{aligned}$$

It can be easily verified that the configuration Q_2 can produce a message on the channel a (the proof is given by horizontally composing the tiles `comm`, `right`, `left` and $\texttt{put}_a$ using also some parallel composition with vertical identities when needed). However, the box $receive_a$ of Q_1 can only consume adjacent messages.

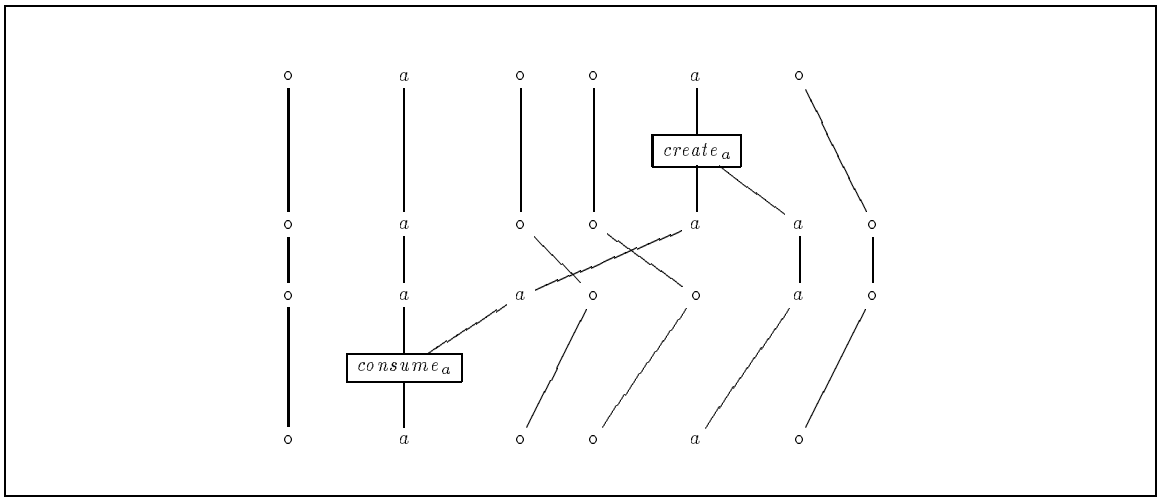


Figure 5.6: The effect resulting from the sending and the retrieving of a message on the channel a in the configuration Q.

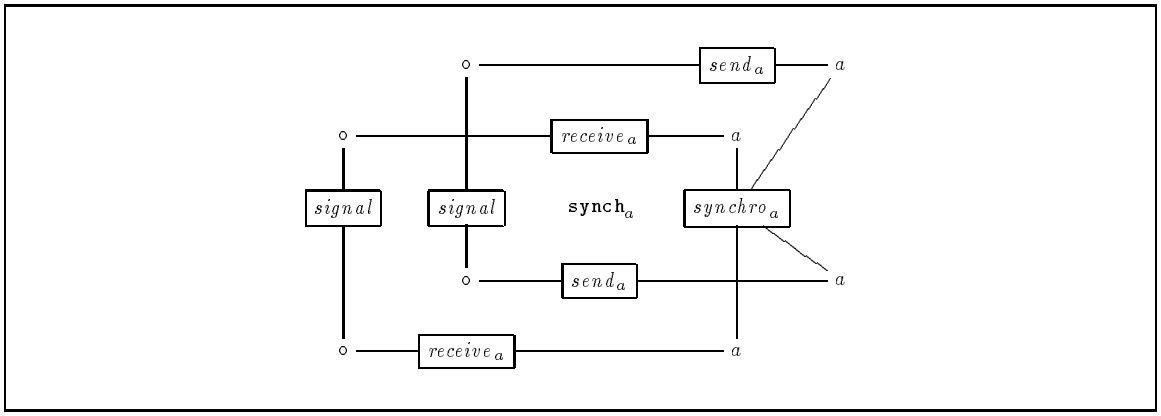


Figure 5.7: The tile $\texttt{synch}_a$ for the synchronization of a send and a receive operations.

The task of bringing the message closer to the receiving port can be accomplished by the vertical swapping of the message and the two wires ($id_{\circ\otimes\circ}$) that separate the message from the *receive* component.

The resulting effect is graphically illustrated in Figure 5.6 (where the "orthographic view" is preferred to the deformed "perspective view").

If we introduce an explicit synchronization mechanism in our system using the tile $\texttt{synch}_a$ represented in Figure 5.7, then also horizontal symmetries become necessary, as well as tiles mixing horizontal and vertical symmetries (in a consistent way).

For example, the interested reader could try to synchronize the $send_a$ and the $receive_a$ operations in the configuration Q (Figure 5.5) without using any horizontal symmetries (it is impossible, because the *select* operator does not allow the vertical symmetries as side-effects). A solution, involving horizontal and vertical symmetries, is illustrated at the end of Section 5.3.1. The idea is to combine horizontal and vertical symmetries using tiles as those pictured in Figure 5.8.

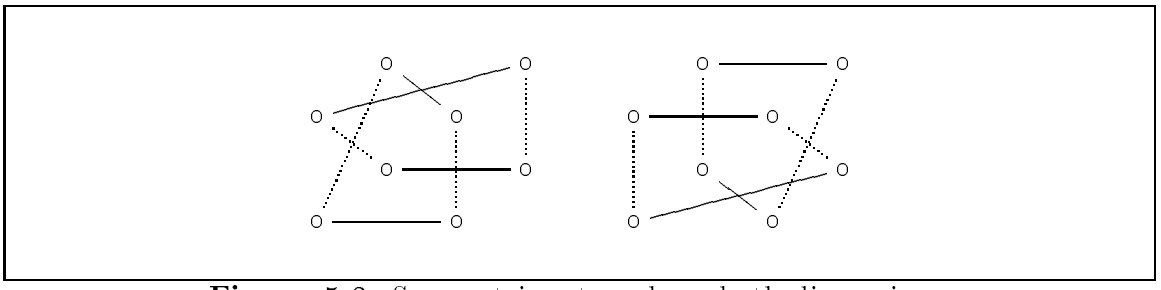


Figure 5.8: Symmetries at work on both dimensions.

For example, the tile on the right in Figure 5.8 can be horizontally composed with any idle configuration (its trigger is the identity) and perform a transformation on the spatial representation of the system, reaching a configuration where the two components of the interface are presented in reverse order. However, this creates a side-effect that must be communicated to the environment, because the protocol to interact with the transformed configuration must be adapted w.r.t. such change.

5.1.2 Examples on Format Expressiveness

The simple message passing system example illustrates very nicely the necessity of tiles with symmetries on their border, which allow moving around the distributed components of a system, and to put the parts that want to interact closer. This has been explained using a concrete graphical representation of the system. In this section, we try to give a more abstract motivation, by showing that similar enrichments of both configurations and observations yield expressive rule formats.

In [69] (see Section 2.4.2) it is shown that the algebraic tile logic (configurations are open terms over a certain signature and observations are the arrows of the free strict monoidal category generated by a set of unary basic actions) yields a rule format, called *algebraic format*, which is more general, e.g., than the *DeSimone format* [127] because the rules can rewrite a context instead of just a basic operator. However, the algebraic format is less general than, e.g., the *GSOS* format [11], which allows the use of multiple (negative and positive) premises for each subcomponent. For example, the rule

$$\frac{P \xrightarrow{a} Q,\ P \xrightarrow{\bar{a}} R}{fork(P) \xrightarrow{\tau} Q|R} \tag{5.1}$$

is in the positive GSOS format, but it is not in the algebraic format. The problem is that we must express two triggers for the same component of the initial input interface (i.e., the variable P). If we consider the basic actions (e.g., a, $\bar{a}$, τ) as unary operators, the trigger can be written as the pair $\langle a(P), \bar{a}(P)\rangle$, which highlights the (vertical) sharing hidden in the premises.

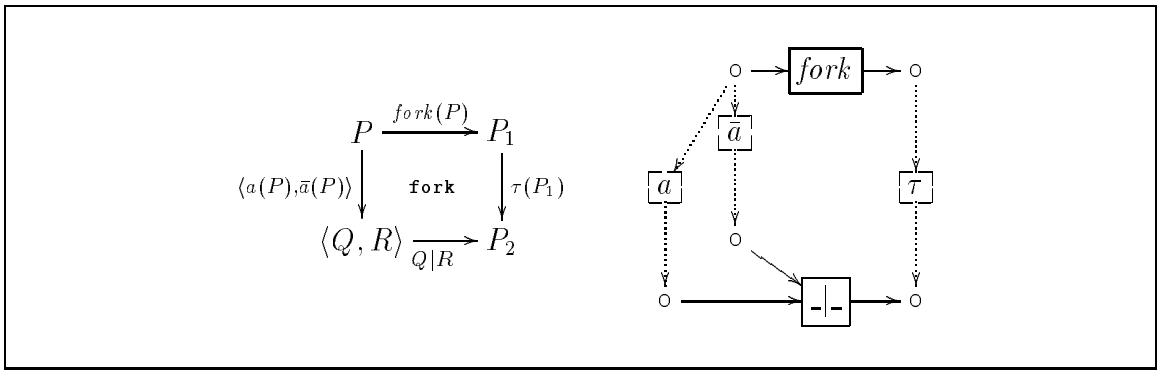


Figure 5.9: The tile corresponding to the GSOS rule (5.1): formally (on the left) and graphically (on the right).

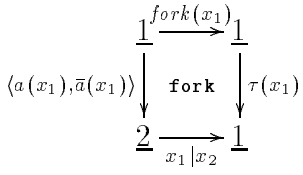


Figure 5.10: The canonical presentation of the tile `fork`.

A cartesian (term) structure on the vertical dimension of tiles (rather than only on the horizontal dimension) can deal with the rule (5.1), as illustrated in Figure 5.9.

Notice the introduction of two placeholders P_1 and P_2 that represent $fork(P)$ and $Q|R$ respectively. Since the choice of names for the variables in the interfaces adds an arbitrary element in the representation of the rules, we can abstract away from such details by choosing a canonical set of names for the variable in each interface. Since interfaces can be viewed as ordered sets of elements, we use the standard naming $x_1, ..., x_n$ for denoting the components of interfaces (of length n), with the obvious meaning that each variable x_i represents the i-th component. In this way, we can just specify the number of components for each interface (e.g., as it is done for algebraic theories, see Section 2.2), and then use the same set of variables for all the terms decorating the border.[1] The representation of the tile `fork` under these assumptions is illustrated as an example in Figure 5.10. A more detailed account on this presentation style is given in Section 5.3.2.

As another example, let us consider the rule for the *replication* operator[2] "!_" of several process algebras, which can be given in the GSOS style as below.

[1] This approach deals very nicely with one-sorted signatures. For the many-sorted case, some confusion might arise when different types are assigned to the same variable in different interfaces, indeed, there is no global typing for variables, but only local typing to each interface. We refer the reader to Section 5.3.2 for more information on this topic.

[2] Just to be consistent with the literature we use the same symbol "!" to denote both the replicator and the discharger, but we remark that they should not be confused.

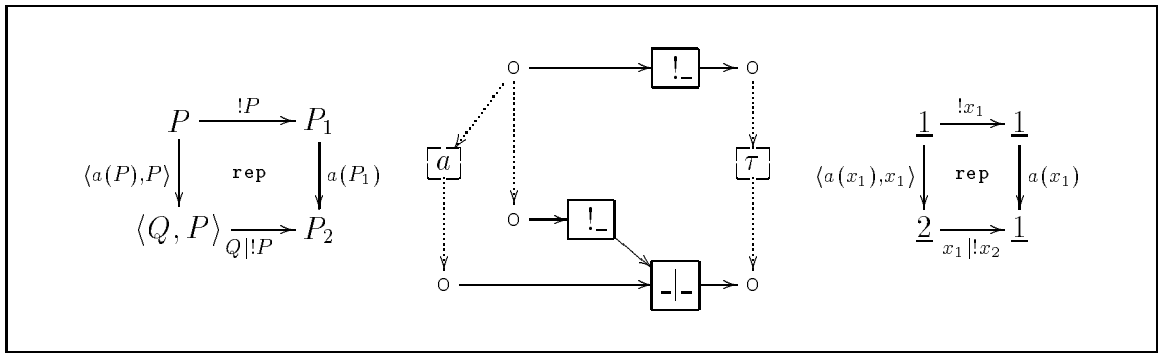


Figure 5.11: The tile corresponding to the GSOS rule (5.2): formally (left), graphically (middle), and in the canonical style (right).

$$\frac{P \xrightarrow{a} Q}{!P \xrightarrow{a} Q|!P} \tag{5.2}$$

In this case, the algebraic tile system cannot deal with the rule because the variable P is used in the conclusions after being tested in the premises (and therefore "forgotten"). Instead, the term tile format can deal with such a case by propagating a copy of the variable P to the final input interface, as illustrated in Figure 5.11.

5.1.3 Defining Symmetric Strict Monoidal and Cartesian Double Categories

We have already mentioned that the introduction of symmetries in two dimensions in the categorical models cannot arise from an internal construction. The first problem is that the internal definition of natural transformation is misleading: since double categories have two different notions of sequential composition, it is not clear which of them to use for a general notion of double natural transformation.

Ehresmann noticed another way of expressing natural transformations, in terms of functors toward higher fold categories [57]. The key point is that a natural transformation is *a functorial collection of commuting squares* in the target category (also called *quartets*), i.e., a functor from a category to a suitable double category (w.r.t. one of the two possible sequential compositions). This notion can be generalized to n-fold categories by constructing the $2n$-fold category of quartets of quartets... (n times) in all the different n dimensions. Once a notion of *multiple functor* between categories of different folds has been given, the notion of *hypertransformation* arises naturally as a multiple functor between the source n-fold category and the $2n$-fold category, which is generated by the target n-fold category. This means that a hypertransformation between n-fold categories involves 2^n n-fold functors (we refer to Appendix C for a more detailed description of the subject).

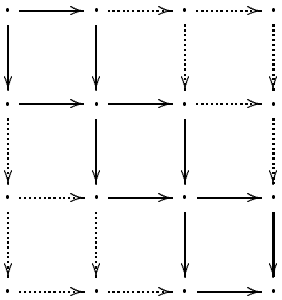


Figure 5.12: Example of diagonal composition of cells: Dotted cells are just suitable identities.

Since double categories are 2-fold categories, this means that we need to define the 4-fold category of horizontal quartets of vertical quartets, and that the hyper-transformations are defined upon $2^2 = 4$ double functors. This yields a definition of transformations which act on both dimensions, asserting the correctness of the two ways of transforming the structure (first horizontally and then vertically, or vice versa).

In this chapter, we promote the equivalent definition of *generalized natural transformation* as a more concrete rephrasing of the hyper-approach for the 2-fold case. A generalized natural transformation involves four double functors, two "horizontal" natural transformations and two "vertical" natural transformations.

Then we instantiate this notion of generalized natural transformation to deal with symmetries, duplicators, and dischargers (as special instances of the general notion). This study involves the complete case analysis of the possible combinations. As an interesting result, we find out that each natural transformation induces two different notions of generalized natural transformations. However, the two possible generalized symmetries are shown to be equivalent (the reason is that symmetries are isomorphisms).

The axiomatization of the basic components of generalized natural transformations in order to state their *coherence* is a subtler problem. The solution that we propose relies on the characterization of suitable *diagonal categories*. We start by considering two particular subclasses of cells, having either both sources or both targets equal to identities. Then, in addition to the horizontal and vertical compositions, we define two *diagonal* compositions (one for each subclass) between cells A and B such that the "upper-left" vertex of B is equal to the "lower-right" vertex of A (see Figure 5.12). These operations are defined in any double category, and allow expressing the coherence axioms for our generalized natural transformations as the rephrasing of the well-known Kelly-MacLane coherence axioms. Moreover, the axiomatization of symmetric monoidal and cartesian double categories seems to precisely characterize concurrent tile computations.

As an important remark, we always assume the monoidal category structure to be strict. Although this assumption leads us to consider models more abstract than most concrete structures (where the associativity and the identity axioms for the monoidal operator are satisfied only up to natural isomorphisms), the notation becomes much simpler, making the whole "tile framework" easier to use. In the conclusion we suggest how our ideas can be extended to the non-strict case.

An alternative definition of cartesian double category could arise by taking the notion of limit in double categories proposed by Grandis and Paré [76]. However, since our research direction investigates a much tighter notion of double products (that we have called "consistently chosen products") rather than the one employed by Grandis and Paré, we have preferred to reserve the analysis of the precise relationships between the two approaches to future investigations.

The definition of cartesian double categories adopted in Section 5.4.2 imposes strong restrictions on the way the various products are chosen. However, these restrictions are not arbitrarily imposed. They are motivated by the observation that symmetries, duplicators and dischargers are in some sense shared between the two dimensions, and thus must be chosen in a consistent way.

5.1.4 Structure of the Chapter

In Section 5.2 we introduce the preliminary definitions regarding double categories, and the notation that will be used in the rest of the thesis. In particular, in Section 5.2.3 we formalize and explain the notion of *generalized natural transformation*, which is essential for the understanding of the concepts discussed in Sections 5.4.1 and 5.4.2, and in Section 5.2.4 we introduce the notion of *diagonal category* that will be used in Sections 5.4.1 and 5.4.2 to simplify the formulation of coherence axioms.

In Sections 5.3.1 and 5.3.2 we introduce the original tile models based on process- and term-like structures of configurations and observations. Each model is first presented in its flat version (Sections 5.3.1.1 and 5.3.2.1), then it is equipped with an algebra of proofs (Sections 5.3.1.2 and 5.3.2.2), and eventually equivalent proof terms are equated to characterize the natural semantic framework of the logic (Sections 5.3.1.3 and 5.3.2.3).

In Section 5.4, we introduce suitable categorical models for process and term tile logic, by exploiting the notion of *generalized transformation* and *diagonal category* to deal with symmetries, duplicators and dischargers. In Sections 5.4.1 and 5.4.2 we incrementally enrich the basic monoidal structure of cells, first with *generalized symmetries* and then with *generalized dischargers* and *generalized duplicators*, presenting also their complete axiomatization. As a result, we propose a precise characterization of *symmetric strict monoidal double categories* and *cartesian double categories with chosen products*.

Although the notion of generalized natural transformation has already appeared (with a different formulation, see Appendix C) in the work of Ehresmann, the material presented in this chapter (i.e., the definition of symmetric strict monoidal and

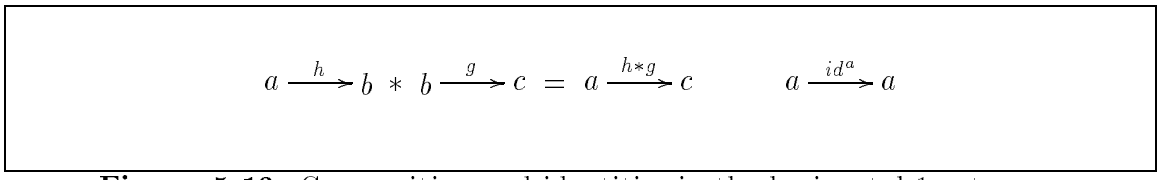


Figure 5.13: Composition and identities in the horizontal 1-category.

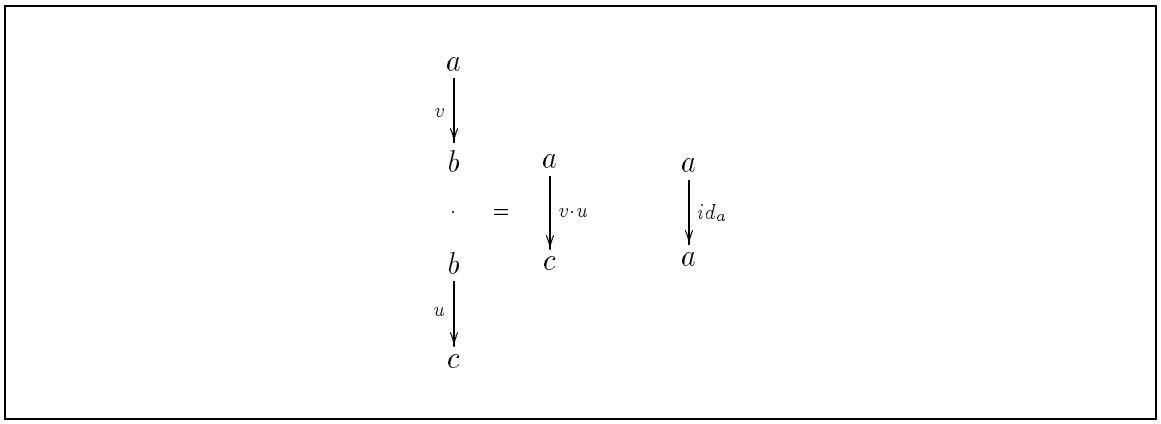


Figure 5.14: Composition and identities in the vertical 1-category.

cartesian with consistently chosen products double categories, and the coherence axioms in particular) is, to our knowledge, original. Moreover, the mathematical structures developed in Sections 5.4.1 and 5.4.2 provide a suitable concurrent semantic framework for tile systems.

5.2 Background

5.2.1 Double Categories

A *double category* is an internal category in **Cat**. Due to the specific structure of **Cat**, double categories admit the following *naïve* presentation, adapted from [82].

Definition 5.2.1 (Double Category)
A double category $\mathcal{D}$ *consists of a collection* $a, b, c, ...$ *of* objects *(also called* 0-cells*), a collection* $h, g, f, ...$ *of* horizontal arrows *(also called* horizontal 1-cells*), a collection* $v, u, w, ...$ *of* vertical arrows *(also called* vertical 1-cells*) and a collection* $A, B, C, ...$ *of* double cells *(also called* cells*).*

Objects and horizontal arrows form the horizontal 1-category $\mathcal{H}$ *(see Figure 5.13), with identity* id^a *for each object* a*, and composition* $\_ * \_$.

Objects and vertical arrows form also a category, called the vertical 1-category $\mathcal{V}$ *(see Figure 5.14), with identity* id_a *for each object* a*, and composition* $\_ \cdot \_$ *(sometimes we will refer to both* id^a *and* id_a *either with the object name* a *or with* id_a*, to shorten the notation, and because, when we consider lower-dimensional objects to be included in higher-dimensional ones the notions indeed coincide).*

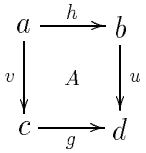


Figure 5.15: Graphical representation of a cell.

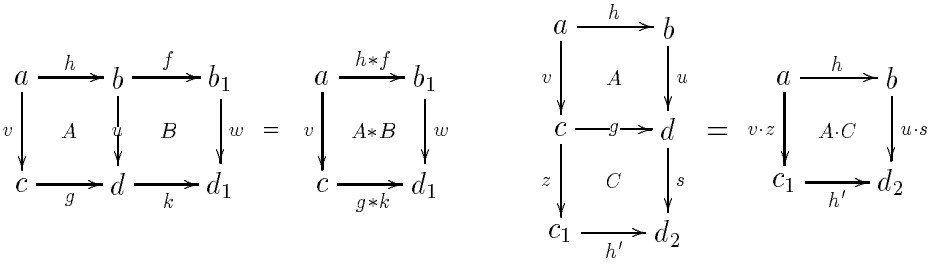


Figure 5.16: Horizontal and vertical composition of cells.

Cells are assigned horizontal source *and* target *(which are vertical 1-cells, i.e., arrows in the vertical 1-category) and* vertical source *and* target *(which are horizontal 1-cells, i.e., arrows in the horizontal 1-category); furthermore sources and targets must be* compatible, *in the sense that, given a cell A, with vertical source h, vertical target g, horizontal source v, and horizontal target u, then h and v have the same source, g and u have the same target, the target of h is equal to the source of u, and the target of v is equal to the source of g. These constraints can be represented by the diagram in Figure 5.15, for which we use the notation $A : h \xrightarrow[u]{v} g$.*

In addition, cells can be composed both horizontally $(\_ * \_)$ and vertically $(\_ \cdot \_)$ as follows: given $A : h \xrightarrow[u]{v} g$, $B : f \xrightarrow[w]{u} k$, and $C : g \xrightarrow[s]{z} h'$, then $A * B : (h * f) \xrightarrow[w]{v} (g * k)$, and $A \cdot C : h \xrightarrow[u \cdot s]{v \cdot z} h'$ are cells. Both compositions can be pictured by pasting the diagrams in Figure 5.16. Moreover, given a fourth cell $D : k \xrightarrow[t]{s} f'$, horizontal and vertical compositions verify the following exchange law *(see also Figure 5.17):*

$$(A \cdot C) * (B \cdot D) = (A * B) \cdot (C * D)$$

Under these rules, cells form both a horizontal category $\mathcal{D}^*$ and a vertical category $\mathcal{D}^{\cdot}$, with respective identities $1_v : a \xrightarrow[v]{v} c$ and $1^h : h \xrightarrow[b]{a} h$. Given $1^h : h \xrightarrow[b]{a} h$ and $1^g : g \xrightarrow[c]{b} g$, the equation $1^h * 1^g = 1^{h*g}$ must hold (and similarly for vertical composition of horizontal identities), as illustrated in Figure 5.18. Furthermore, horizontal and vertical identities of identities coincide, i.e., $1_{id_a} = 1^{id^a}$ and are denoted by the simpler notation 1_a (or just a).

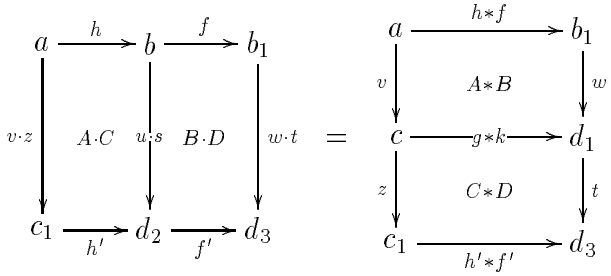


Figure 5.17: The exchange law.

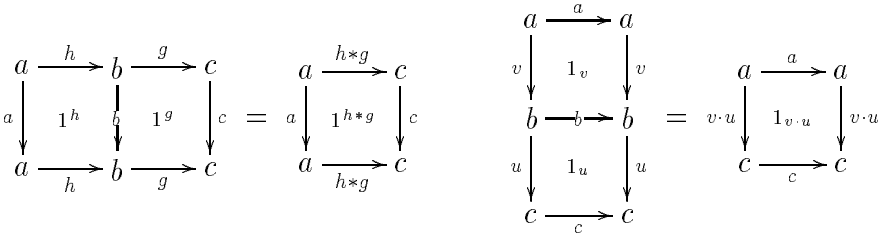
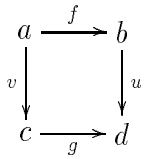


Figure 5.18: Composition of identities.

REMARK 5.2.1 *As a matter of notation, sometimes we will use* $\_;\_$ *to denote the composition on both the horizontal and vertical 1-categories.*

EXAMPLE 5.2.2 (QUARTET CATEGORY) *Given a category* $\mathcal{C}$*, we denote by* $\square\mathcal{C}$ *the* category of quartets *of* $\mathcal{C}$*: its objects are the objects of* $\mathcal{C}$*, its horizontal and vertical arrows are the arrows of* $\mathcal{C}$*, and its cells are the commuting square diagrams of arrows in* $\mathcal{C}$



where $f, v, u, g \in \mathcal{C}$ *and* $f;u = v;g$ *in* $\mathcal{C}$*. The horizontal and vertical categories of* $\square\mathcal{C}$ *are usually denoted by* $\square\square\mathcal{C}$ *and* $\begin{smallmatrix}\square\\\square\end{smallmatrix}\mathcal{C}$ *respectively. Notice that* $\mathcal{C}$ *is both the horizontal and the vertical 1-category of* $\square\mathcal{C}$*.*

A double category $\mathcal{D}$ has two possible interpretations as an internal category in **Cat**. In fact, thanks to the symmetric rôle played by the horizontal and vertical dimensions of a double category, it is possible to adopt a *transposed* approach in the internal construction:

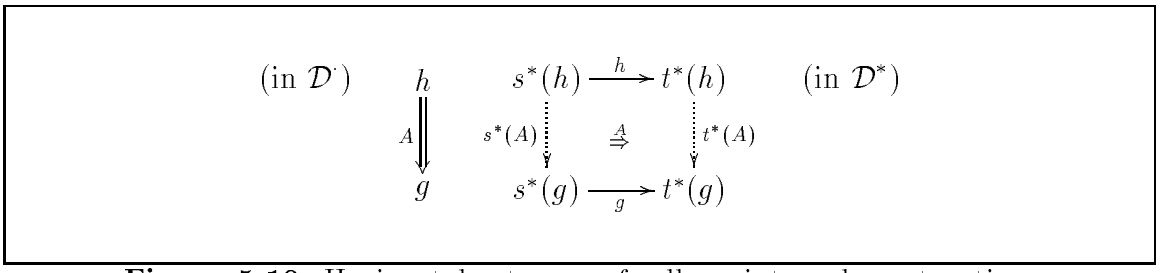


Figure 5.19: Horizontal category of cells as internal construction.

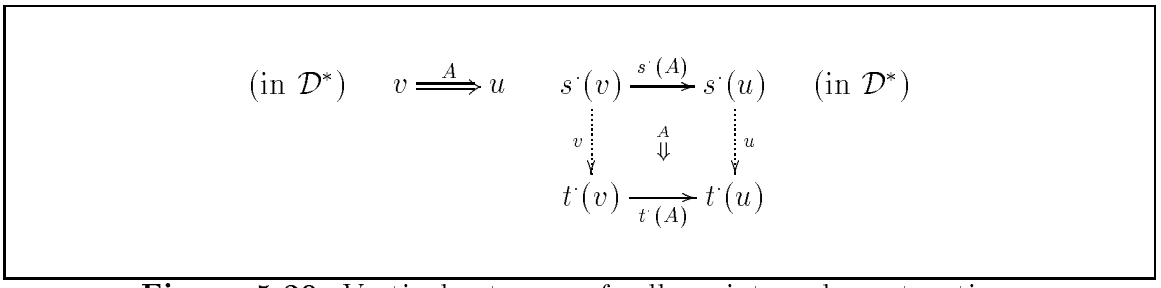


Figure 5.20: Vertical category of cells as internal construction.

1. As the internal category $(\mathcal{V}, \mathcal{D}^{\cdot}, s^*, t^*, \_*\_, i^*)$, where[3] the functors s^* and t^* map each arrow onto its corresponding horizontal source and target respectively, the functor $\_*\_$ defines horizontal composition of cells, and the functor i^* maps each (vertical) arrow of $\mathcal{V}$ onto its (horizontal) identity cell. This corresponds to picturing a generic cell $A : h \longrightarrow g$ of $\mathcal{D}^{\cdot}$ as shown in Figure 5.19.

2. As the internal category $(\mathcal{H}, \mathcal{D}^*, s^{\cdot}, t^{\cdot}, \_\cdot\_, i^{\cdot})$ where the functors $s^{\cdot}$ and $t^{\cdot}$ map each arrow of $\mathcal{D}^*$ to its corresponding horizontal source and target, the functor $\_\cdot\_$ defines vertical composition of cells according to the source and target projections $s^{\cdot}$ and $t^{\cdot}$, and $i^{\cdot}(h) = 1^h$ for each horizontal arrow $h \in \mathcal{H}$ (see Figure 5.20)

DEFINITION 5.2.2 (DOUBLE FUNCTOR)
Given two double categories $\mathcal{D}$ and $\mathcal{E}$, a double functor $F : \mathcal{D} \longrightarrow \mathcal{E}$ *is a 4-tuple of functions mapping objects to objects, horizontal and vertical arrows to horizontal and vertical arrows, and cells to cells, preserving identities and compositions of all kinds.*

Notice that, since a double category is a cat-object in **Cat**, a double functor can be equivalently defined as a pair (F_0, F_1) of functors satisfying the conditions of internal functoriality. The two notions coincide because each functor in (F_0, F_1) is a pair of mappings on objects and arrows preserving the category structure. We denote by **DCat** the category of double categories and double functors.

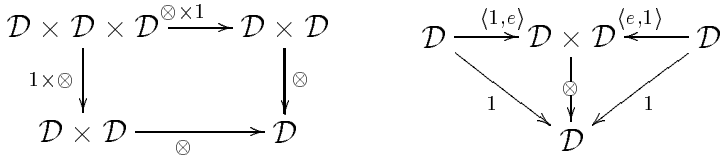
[3]Remember that the category $\mathcal{V}$ has objects in $\mathcal{O}$ and arrows in $\mathcal{V}$ equipped with composition $\_\cdot\_$, and that the objects of the category $\mathcal{D}^{\cdot}$ are horizontal arrows, while its morphisms are cells equipped with vertical composition $\_\cdot\_$.

A *monoidal double category* is an internal category in **MonCat** (the category of monoidal categories and monoidal functors), or equivalently, is an internal monoidal category in **Cat**. For the strict case, we have also the following detailed definition.

DEFINITION 5.2.3 (STRICT MONOIDAL DOUBLE CATEGORY)
A strict monoidal double category, *sMD in the following, is a triple* $(\mathcal{D}, \otimes, e)$, *where:*

- $\mathcal{D}$ *is the underlying double category,*
- $\otimes : \mathcal{D} \times \mathcal{D} \longrightarrow \mathcal{D}$ *is a double functor called the* tensor product, *and*
- e *is an object of* $\mathcal{D}$ *called the* unit object,

such that the following diagrams commute:



where double functor $1 : \mathcal{D} \longrightarrow \mathcal{D}$ *is the identity on* $\mathcal{D}$, *the double functor* $e : \mathcal{D} \longrightarrow \mathcal{D}$ *(with some abuse of the notation) is the constant double functor which associates the object* e *and identities on* e *respectively to each object and each morphism/cell of* $\mathcal{D}$, *and* $\langle \_, \_ \rangle$ *denotes the pairing of double functors induced by the cartesian product of double categories. These equations state that the tensor product* $\_ \otimes \_$ *is associative on both objects, arrows and cells, and that* e *is the unit for* $\_ \otimes \_$.

A monoidal double functor *is a double functor which preserves tensor product and unit object. We denote by* **sMDCat** *the category of monoidal double categories and monoidal double functors.*

5.2.2 Inverse

Since double categories have two operations of composition, the definition of the inverse of a cell is not straightforward. We propose the following:

DEFINITION 5.2.4 (GENERALIZED INVERSE)
Let $A : h \xrightarrow[u]{v} g$ *be a cell in a double category* $\mathcal{D}$. *We say that cell* A *has a* $*$*-inverse if and only if there exists a cell* A^* *such that* $A * A^* = 1_v$, *and* $A^* * A = 1_u$ *(i.e.,* A^* *is the inverse of* A *w.r.t. the* horizontal *composition* $*$, *and this implies the existence of inverses of the horizontal arrows on the border of* A*). Similarly, the* $\cdot$*-inverse* $A^{\cdot}$, *if it exists, satisfies the equations* $A \cdot A^{\cdot} = 1^h$, $A^{\cdot} \cdot A = 1^g$ *(this implies the existence of inverses of the vertical arrows on the border of* A*).*

Then, A *has a* generalized inverse *if and only if* A *has both a* $*$*-inverse and a* $\cdot$*-inverse, and there exists a cell* A^{-1} *such that:*

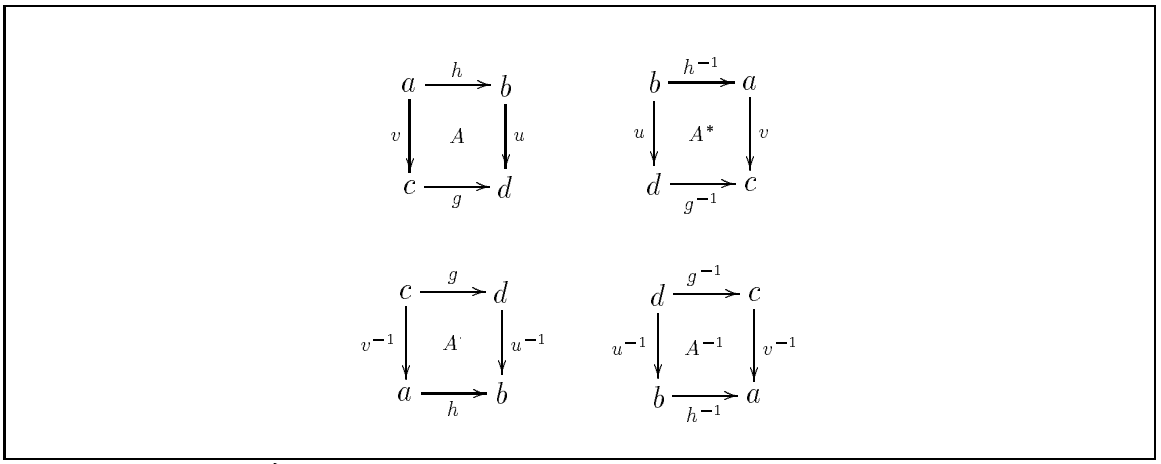


Figure 5.21: Generalized inverse for the cell A.

- $A^{-1} \cdot A^* = 1^{g^{-1}}$,
- $A^* \cdot A^{-1} = 1^{h^{-1}}$ *(i.e., A^{-1} is the $\cdot$-inverse of A^*),*
- $A^{-1} * A^{\cdot} = 1_{u^{-1}}$, *and*
- $A^{\cdot} * A^{-1} = 1_{v^{-1}}$ *(i.e., A^{-1} is also the $*$-inverse of $A^{\cdot}$).*

This definition can be summarized by saying that the $*$-inverse $(A^{\cdot})^*$ of the $\cdot$-inverse of cell A is equal to the $\cdot$-inverse $(A^*)^{\cdot}$ of the $*$-inverse of A, and it is denoted by A^{-1} (see Figure 5.21).

For instance, it follows that $(A \cdot A^{\cdot}) * (A^* \cdot A^{-1}) = (A * A^*) \cdot (A^{\cdot} * A^{-1}) = 1_a$, and that $(A^{-1})^{-1} = A$.

5.2.3 Natural Transformations

Let $F, G : \mathcal{D} \longrightarrow \mathcal{E}$ be two double functors. Following the internal construction approach, an internal natural transformation is an arrow in **Cat** which verifies the naturality conditions w.r.t. one composition and which is functorial w.r.t. the other composition. Thus, it is essential to specify what the *internal* representations of $\mathcal{D}$ and $\mathcal{E}$ are.

In [57, 4, 56] the notion of *hypertransformation* is presented as the generalization of natural transformations to the n-fold case (see Appendix C). We propose the following definition of *generalized natural transformation* as a more concrete rephrasing of the hyper-view for double categories (whereas e.g., in [56] emphasis was given to showing that the category **MCat** of multiple functors is cartesian closed).

As a matter of notation, we call natural *comp*-transformation a transformation which satisfies the naturality requirement w.r.t. the composition operator *comp* and which is functorial w.r.t. the remaining composition operator.

A generalized natural transformation is the key to express relationships between natural $*$-transformations and natural $\cdot$-transformations (which are the two possible notions of internal natural transformation suggested by the internal category viewpoint).

DEFINITION 5.2.5 (GENERALIZED NATURAL TRANSFORMATION)
Let $\mathcal{D}$ and $\mathcal{E}$ be double categories. Given a 4-tuple $(F_{00}, F_{10}, F_{01}, F_{11})$ of double functors from $\mathcal{D}$ to $\mathcal{E}$, a generalized natural transformation *is a 5-tuple*

$$(\alpha_{0\_}, \alpha_{1\_}, \beta_{\_0}, \beta_{\_1}, \eta),$$

which is pictured as the cell

$$\begin{array}{ccc} F_{00} & \xrightarrow{\alpha_{0\_}} & F_{01} \\ {\scriptstyle \beta_{\_0}}\downarrow & \eta & \downarrow{\scriptstyle \beta_{\_1}} \\ F_{10} & \xrightarrow[\alpha_{1\_}]{} & F_{11} \end{array}$$

where:

- *for $i = 0, 1$, the symbol $\alpha_{i\_}$ denotes a natural $*$-transformation from F_{i0} to F_{i1}, i.e., $\alpha_{i\_}$ is also a functor from the category $\mathcal{V}$ of vertical arrows of $\mathcal{D}$ (that is, the objects of $\mathcal{D}^*$) to the category $\mathcal{E}^{\cdot}$,*
- *for $i = 0, 1$, the symbol $\beta_{\_i}$ denotes a natural $\cdot$-transformation from F_{0i} to F_{1i}, i.e., $\beta_{\_i}$ defines also a functor from the category $\mathcal{H}$ of horizontal arrows of $\mathcal{D}$ (that is, the objects of $\mathcal{D}^{\cdot}$) to the category $\mathcal{E}^*$, and*
- *the symbol η denotes both a natural transformation from $\alpha_{0\_}$ to $\alpha_{1\_}$ (seen as functors from $\mathcal{V}$ to $\mathcal{E}^{\cdot}$) and also a natural transformation from $\beta_{\_0}$ to $\beta_{\_1}$ (seen as functors from $\mathcal{H}$ to $\mathcal{E}^*$).*

To shorten the notation, we will denote the generalized natural transformation just by η, using a figure to represent its other components.

We now explain in more detail the previous definition. First, consider the double functors $F_{00}, F_{01} : \mathcal{D} \longrightarrow \mathcal{E}$. A natural $*$-transformation $\alpha_{0\_} : F_{00} \Rightarrow F_{01} : \mathcal{D} \longrightarrow \mathcal{E}$ is a functor (i.e., an arrow in **Cat**) from the category $\mathcal{V}$ of objects of $\mathcal{D}^*$ to $\mathcal{E}^{\cdot}$, which satisfies the equations of internal natural transformations. Thus, the functor $\alpha_{0\_}$ is a natural transformation from F_{00} to F_{01} w.r.t. horizontal composition, i.e., for each cell $A : h \xrightarrow[u]{v} g$ of $\mathcal{D}$ we have the naturality law $F_{00}A * \alpha_{0\_,u} = \alpha_{0\_,v} * F_{01}A$.

$$\begin{array}{ccc} a & \xrightarrow{h} & b \\ {\scriptstyle v}\downarrow & A & \downarrow{\scriptstyle u} \\ c & \xrightarrow[g]{} & d \end{array} \qquad \begin{array}{ccccc} F_{00}a & \xrightarrow{F_{00}h} & F_{00}b & \xrightarrow{\alpha_{0\_,b}} & F_{01}b \\ {\scriptstyle F_{00}v}\downarrow & F_{00}A & \downarrow{\scriptstyle F_{00}u} & \alpha_{0\_,u} & \downarrow{\scriptstyle F_{01}u} \\ F_{00}c & \xrightarrow[F_{00}g]{} & F_{00}d & \xrightarrow[\alpha_{0\_,d}]{} & F_{01}d \end{array} = \begin{array}{ccccc} F_{00}a & \xrightarrow{\alpha_{0\_,a}} & F_{01}a & \xrightarrow{F_{01}h} & F_{01}b \\ {\scriptstyle F_{00}v}\downarrow & \alpha_{0\_,v} & \downarrow{\scriptstyle F_{01}v} & F_{01}A & \downarrow{\scriptstyle F_{01}u} \\ F_{00}c & \xrightarrow[\alpha_{0\_,c}]{} & F_{01}c & \xrightarrow[F_{01}g]{} & F_{01}d \end{array}$$

Hence, for any horizontal arrow $h : a \longrightarrow b$ in $\mathcal{D}$, we have $F_{00}h * \alpha_{0\_,b} = \alpha_{0\_,a} * F_{01}h$, i.e., the object component of functor $\alpha_{0\_}$ defines a natural transformation between the components of F_{00} and F_{01} on the horizontal 1-category.

Consider the double functors $F_{10}, F_{11} : \mathcal{D} \longrightarrow \mathcal{E}$ and a natural $*$-transformation $\alpha_{1\_} : F_{10} \Rightarrow F_{11} : \mathcal{D} \longrightarrow \mathcal{E}$ between them. Then, $\alpha_{1\_}$ also defines a functor from $\mathcal{V}$ to $\mathcal{E}^{\cdot}$ and satisfies the naturality equation w.r.t. horizontal composition. Notice that the functors $\alpha_{0\_}$ and $\alpha_{1\_}$ have the same source and target categories. The generalized natural transformation η acts as a natural transformation $\eta : \alpha_{0\_} \Rightarrow \alpha_{1\_} : \mathcal{V} \longrightarrow \mathcal{E}^{\cdot}$ between them. Thus, the natural transformation η associates to each object a of $\mathcal{V}$ (i.e., an object of $\mathcal{D}$) an arrow η_a of $\mathcal{E}^{\cdot}$ (i.e., a cell of $\mathcal{E}$) in such a way that the equation $\alpha_{0\_,v} \cdot \eta_c = \eta_a \cdot \alpha_{1\_,v}$ holds for each arrow $v : a \longrightarrow c$ in $\mathcal{V}$.

$$
\begin{array}{c} a \\ {\scriptstyle v}\downarrow \\ c \end{array}
\qquad
\begin{array}{ccc}
F_{00}a & \xrightarrow{\alpha_{0\_,a}} & F_{01}a \\
{\scriptstyle F_{00}v}\downarrow & \alpha_{0\_,v} & \downarrow{\scriptstyle F_{01}v} \\
F_{00}c & \xrightarrow{\alpha_{0\_,c}} & F_{01}c \\
{\scriptstyle s^*(\eta_c)}\downarrow & \eta_c & \downarrow{\scriptstyle t^*(\eta_c)} \\
F_{10}c & \xrightarrow[\alpha_{1\_,c}]{} & F_{11}c
\end{array}
\quad = \quad
\begin{array}{ccc}
F_{00}a & \xrightarrow{\alpha_{0\_,a}} & F_{01}a \\
{\scriptstyle s^*(\eta_a)}\downarrow & \eta_a & \downarrow{\scriptstyle t^*(\eta_a)} \\
F_{10}a & \xrightarrow{\alpha_{1\_,a}} & F_{11}a \\
{\scriptstyle F_{10}v}\downarrow & \alpha_{1\_,v} & \downarrow{\scriptstyle F_{11}v} \\
F_{10}c & \xrightarrow[\alpha_{1\_,c}]{} & F_{11}c
\end{array}
$$

This implies that $F_{00}v \cdot s^*(\eta_c) = s^*(\eta_a) \cdot F_{10}v$, and $F_{01}v \cdot t^*(\eta_c) = t^*(\eta_a) \cdot F_{11}v$, i.e., $s^*(\eta)$ (respectively, $t^*(\eta)$) is a natural transformation between the projections on the vertical 1-category of functors F_{00} and F_{10} (F_{01} and F_{11}, respectively).

A similar reasoning can be applied to the orthogonal internal representations of $\mathcal{D}$ and $\mathcal{E}$, defining two natural $\cdot$-transformations $\beta_{\_0} : F_{00} \Rightarrow F_{10} : \mathcal{D} \longrightarrow \mathcal{E}$ and $\beta_{\_1} : F_{01} \Rightarrow F_{11} : \mathcal{D} \longrightarrow \mathcal{E}$, which are functors from the category $\mathcal{H}$ of objects of $\mathcal{D}^{\cdot}$ to $\mathcal{E}^*$, and satisfy the equations $F_{00}A \cdot \beta_{\_0,g} = \beta_{\_0,h} \cdot F_{10}A$ (see the picture below), and $F_{01}A \cdot \beta_{\_1,g} = \beta_{\_1,h} \cdot F_{11}A$ for each cell A of $\mathcal{D}$.

$$
\begin{array}{ccc}
a & \xrightarrow{h} & b \\
{\scriptstyle v}\downarrow & A & \downarrow{\scriptstyle u} \\
c & \xrightarrow[g]{} & d
\end{array}
\qquad
\begin{array}{ccc}
F_{00}a & \xrightarrow{F_{00}h} & F_{00}b \\
{\scriptstyle F_{00}v}\downarrow & F_{00}A & \downarrow{\scriptstyle F_{00}u} \\
F_{00}c & \xrightarrow{F_{00}g} & F_{00}d \\
{\scriptstyle \beta_{\_0,c}}\downarrow & \beta_{\_0,g} & \downarrow{\scriptstyle \beta_{\_0,d}} \\
F_{10}c & \xrightarrow[F_{10}g]{} & F_{10}d
\end{array}
\quad = \quad
\begin{array}{ccc}
F_{00}a & \xrightarrow{F_{00}h} & F_{00}b \\
{\scriptstyle \beta_{\_0,a}}\downarrow & \beta_{\_0,h} & \downarrow{\scriptstyle \beta_{\_0,b}} \\
F_{10}a & \xrightarrow{F_{10}h} & F_{10}b \\
{\scriptstyle F_{10}v}\downarrow & F_{10}A & \downarrow{\scriptstyle F_{10}u} \\
F_{10}c & \xrightarrow[F_{10}g]{} & F_{10}d
\end{array}
$$

The generalized natural transformation η also defines a natural transformation from $\beta_{\_0}$ to $\beta_{\_1}$. Thus, for each arrow $h : a \longrightarrow b$ of $\mathcal{H}$ we get:

$$
a \xrightarrow{h} b
\qquad
\begin{array}{ccccc}
F_{00}a & \xrightarrow{F_{00}h} & F_{00}b & \xrightarrow{s^{\cdot}(\eta_b)} & F_{01}b \\
{\scriptstyle \beta_{\_0,a}}\downarrow & \beta_{\_0,h} & {\scriptstyle \beta_{\_0,b}}\downarrow & \eta_b & \downarrow{\scriptstyle \beta_{\_1,b}} \\
F_{10}a & \xrightarrow[F_{10}h]{} & F_{10}b & \xrightarrow[t^{\cdot}(\eta_b)]{} & F_{11}b
\end{array}
\quad = \quad
\begin{array}{ccccc}
F_{00}a & \xrightarrow{s^{\cdot}(\eta_a)} & F_{01}a & \xrightarrow{F_{01}h} & F_{01}b \\
{\scriptstyle \beta_{\_0,a}}\downarrow & \eta_a & {\scriptstyle \beta_{\_1,a}}\downarrow & \beta_{\_1,h} & \downarrow{\scriptstyle \beta_{\_1,b}} \\
F_{10}a & \xrightarrow[t^{\cdot}(\eta_a)]{} & F_{11}a & \xrightarrow[F_{11}h]{} & F_{11}b
\end{array}
$$

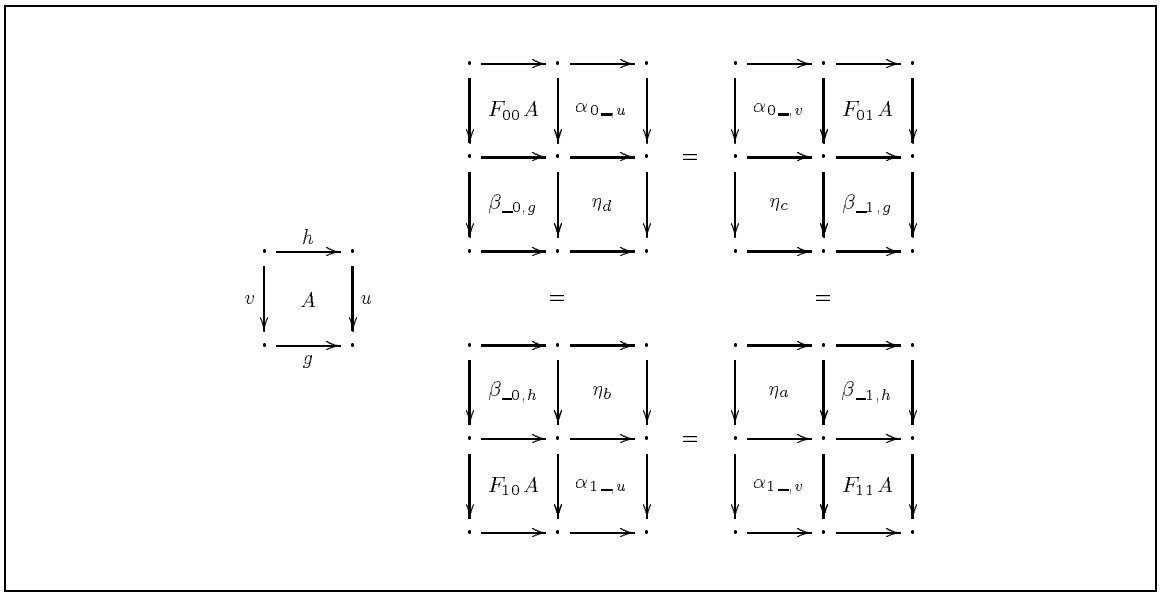


Figure 5.22: Some properties of generalized natural transformations.

It follows that, for each object $a \in \mathcal{O}$, the shape of the cell η_a is

$$\begin{array}{ccc} F_{00}a & \xrightarrow{\alpha_{0\_a}} & F_{01}a \\ {\scriptstyle \beta_{\_0,a}}\downarrow & \eta_a & \downarrow{\scriptstyle \beta_{\_1,a}} \\ F_{10}a & \xrightarrow[\alpha_{1\_a}]{} & F_{11}a \end{array}$$

As an example, it follows directly from the definition that, given a cell $A : h \xrightarrow[u]{v} g$, all the pastings of cells pictured in Figure 5.22 yield identical results.

All the naturality equations are faithfully represented by the commuting hypercube in Figure 5.23 (to make the interpretation easier, vertical arrows are drawn using dotted lines). The hypercube contains 16 vertices, 24 faces, and 8 cubes. Each vertex is the image of one of the four corner objects of cell A through one of the four functors under consideration.

There are eight *empty* faces whose border involves either only vertical or only horizontal arrows. All the other 16 faces are cells of the double category $\mathcal{E}$. Four cells are the image of A w.r.t. the four different functors (see Figure 5.23). Four cells are the components at h and g of the natural $*$-transformations $\alpha_{0\_}$ and $\alpha_{1\_}$. Four cells are the components at v and u of the natural $\cdot$-transformations $\beta_{\_0}$ and $\beta_{\_1}$. The remaining four cells are the components at the objects of the generalized natural transformation η. Each cube has two empty faces. The other four faces commute, in the sense that they express a naturality equation. It follows that the hypercube yields eight equations for each cell A. However, the naturalities of η are both replicated for the two components of each transformation, therefore, there are six distinct equations. The functoriality axioms are given by composing the hypercubes, either one below the other, or one in front of the other.

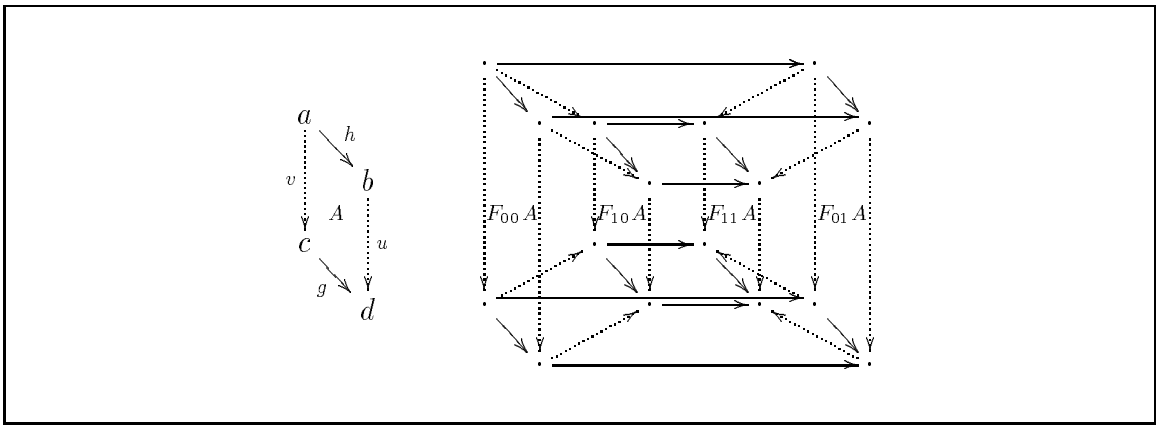


Figure 5.23: The commuting hypercube.

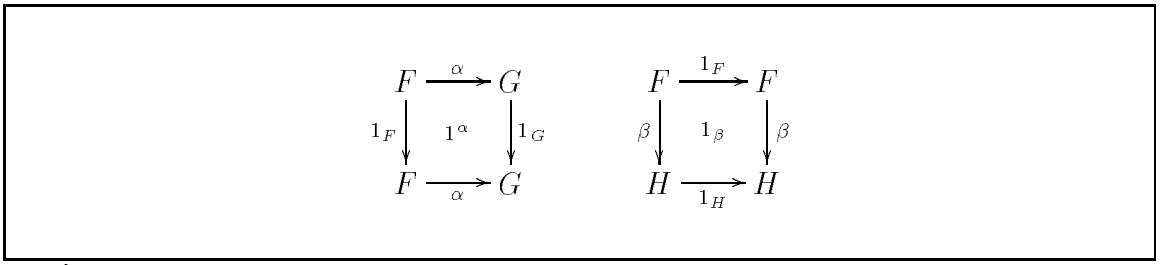


Figure 5.24: Natural $*$- and $\cdot$-transformations as generalized transformations.

REMARK 5.2.3 *Notice that the notion of generalized natural transformation generalizes natural $*$- and $\cdot$-transformations. Indeed, a natural $*$-transformation $\alpha : F \Rightarrow G : \mathcal{D} \longrightarrow \mathcal{E}$ yields a corresponding generalized natural transformation 1^α. Analogously, a natural $\cdot$-transformation $\beta : F \Rightarrow H : \mathcal{D} \longrightarrow \mathcal{E}$ yields a corresponding generalized natural transformation 1_β (see Figure 5.24).*

REMARK 5.2.4 *Natural $*$- and $\cdot$-transformations are instances of a more general pattern. Notice that, if we consider only two generic double functors F and G, then the allowed generalized natural transformations, where the four double functors are chosen from the set $\{F, G\}$, have $2^4 = 16$ possible shapes. Only six of them do not involve transformations from G to F (see Figure 5.25).*

In Sections 5.4.1 and 5.4.2 we instantiate the notion of generalized natural transformation to characterize the auxiliary structures of process and tile systems.

5.2.4 Diagonal Categories

Sometimes, due to the particular kind of cells involved in a complex composition, it is possible to adopt a more concise and convenient notation. This fact follows by observing that, for any double category, it is always possible to characterize two suitable *diagonal* sub-categories.

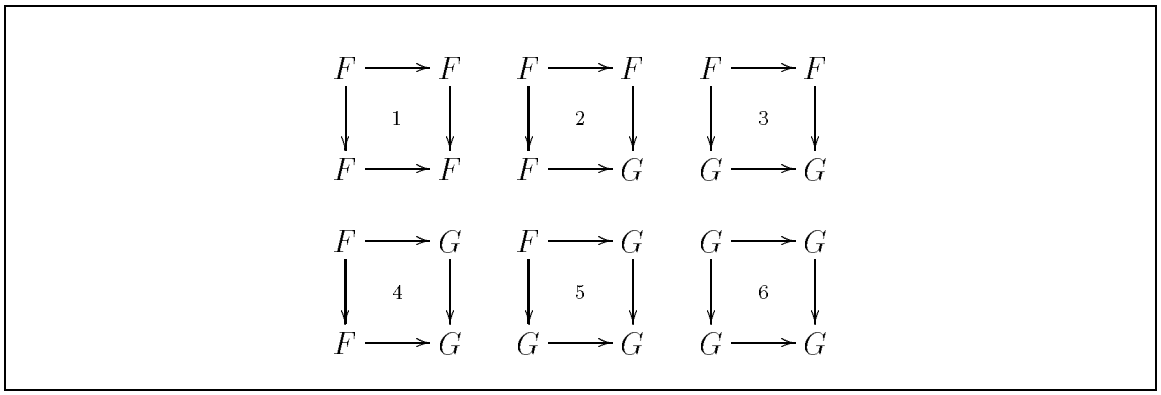
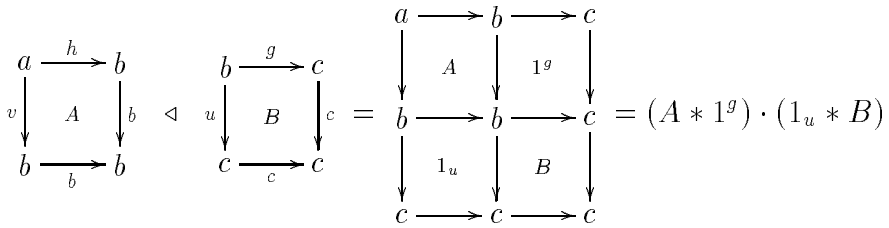
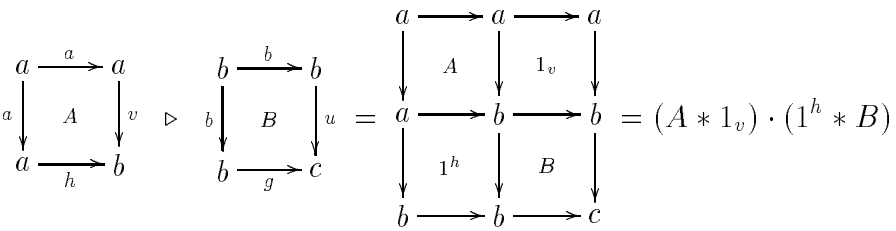


Figure 5.25: The possible shapes for transforming F to G.

In fact, those cells having identities as both horizontal and vertical target are the arrows of a *diagonal* category $\mathcal{D}^{\triangleleft}$ whose composition $\_ \triangleleft \_$ is uniquely defined as depicted below.



In a similar way, we could also define a diagonal composition $\_ \triangleright \_$ for those cells having identities as both horizontal and vertical source (yielding the second diagonal category $\mathcal{D}^{\triangleright}$):



5.3 Process and Term Tile Logic

In process and term tile logic, configurations and observations have in common the same auxiliary structure, i.e., the possibility of re-arranging the interfaces in any consistent way (in the sense that given any auxiliary tile $s \xrightarrow[b]{a} s'$, the composition of the transformation induced by s followed by the one induced by b should yield the same result as the transformation induced by a followed by the one induced by s'). An important requirement is that there should be a *unique* auxiliary tile for each possible bidimensional transformation, i.e., all the possible decompositions of the proof terms of auxiliary tiles yielding the same border should be equivalent.

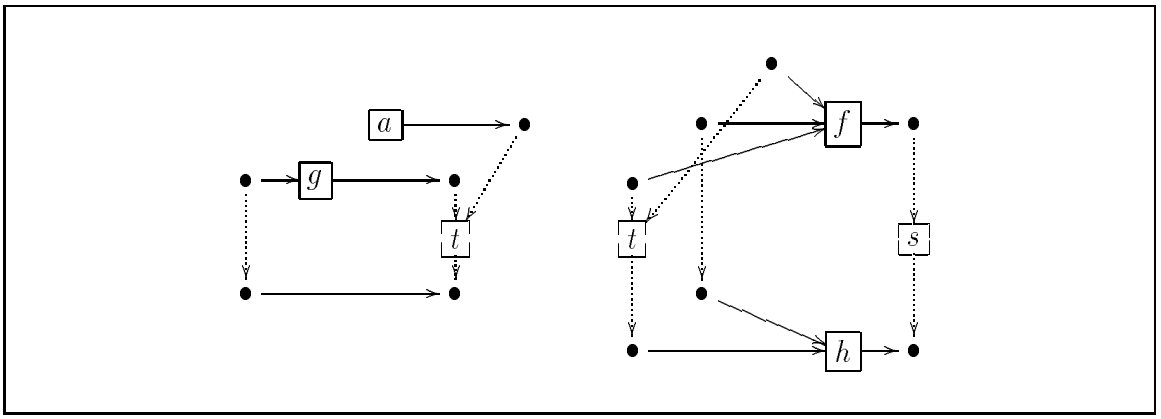


Figure 5.26: Two tiles based on wire and box structures.

Notice that, although auxiliary tiles for process and term tile logic are introduced in this section, their characterization, and in particular the axioms we propose, are based on the research concerning generalized transformations, which is the subject of Section 5.4.2. However, for an easier presentation, and to give a better intuitive understanding of the main ideas with the minimum machinery possible, we have chosen to reverse the order of the two presentations.

5.3.1 Process Tile Logic

In this section we introduce the formal basis of *process tile logic*, whose configurations and observations yield symmetric monoidal categories, and where all the consistent overlappings of wires (see Section 5.1.1) are introduced for free.

In Section 5.1.1 we have used a detailed example to demonstrate the expressiveness of process tile logic. To illustrate how auxiliary tiles can be used, let us consider the following simpler example.

EXAMPLE 5.3.1 *Let us consider the tile system over the signature*

$$\Sigma = \{a : 0 \longrightarrow 1, b : 0 \longrightarrow 1, g : 1 \longrightarrow 1, h : 2 \longrightarrow 1, f : 3 \longrightarrow 1\}$$

of configurations, and the signature $\Sigma' = \{s : 1 \longrightarrow 1, t : 2 \longrightarrow 1\}$ *of observations, consisting of the two tiles in Figure 5.26 (as usual, we draw dotted lines for the vertical dimension to avoid confusion with the horizontal structure).*

Then, one can expect that the configuration $f(g(x_1), x_2, a)$ *(see Figure 5.27) is able to evolve to* $h(x_1, x_2)$*, producing an effect* s *(as the result of the horizontal composition, or synchronization, of the two tiles). However, the tiles cannot be composed in the obvious way without a previous rearrangement of their interfaces, because the arguments of trigger* t *are separated by a component in the initial input interface of the second tile (notice the overlapping of wires), while the effect of the first tile applies only to adjacent arguments. However, the configuration in Figure 5.27 is equivalent to the one in Figure 5.28, where a horizontal symmetry has been introduced.*

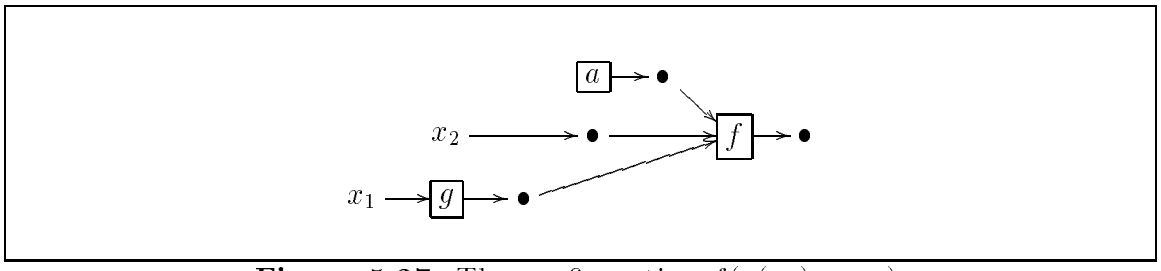


Figure 5.27: The configuration $f(g(x_1), x_2, a)$.

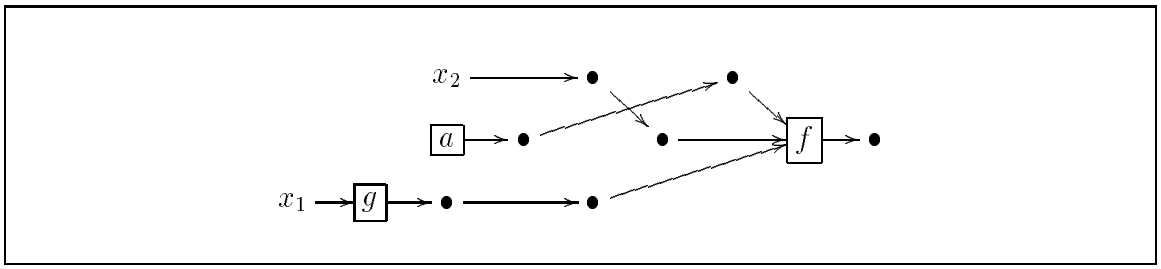


Figure 5.28: Alternative presentation of the configuration $f(g(x_1), x_2, a)$.

What is needed to complete the rewriting synchronization is the tile in Figure 5.29. The synchronization scheme is pictured in Figure 5.30, where:

- *the tile A is obtained as the vertical identity for the parallel composition of g with a and with a wire,*
- *the tile B is the parallel composition of the vertical identity of a wire with the tile illustrated in Figure 5.29,*
- *the tile C is the parallel composition of the first tile in Figure 5.26 with the vertical identity of a wire, and*
- *the tile D is the second tile in Figure 5.26.*

The main limitation of algebraic tile logic is that configurations and effects can share only objects, even when their structure is very similar. As we have shown using the wire and box notation, sometimes it is essential to have a model where consistent rearrangements of interfaces are allowed by default.

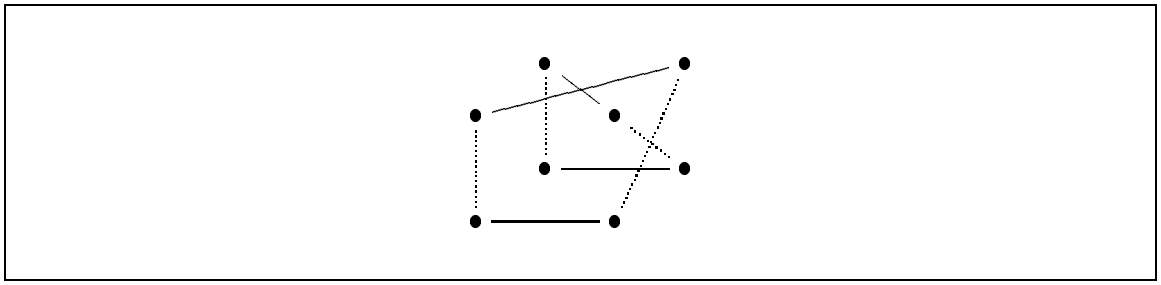


Figure 5.29: A symmetry working on both dimensions.

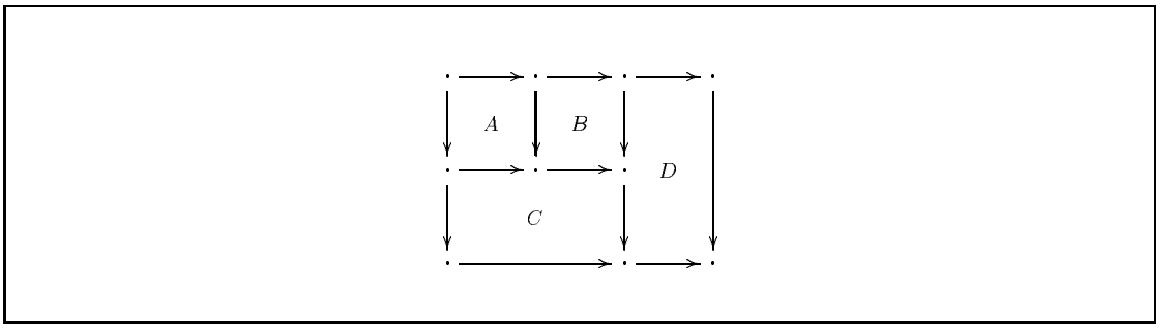


Figure 5.30: The rewriting scheme for the configuration $f(g(x_1), x_2, a)$.

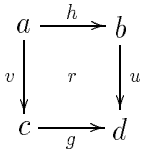
We formalize here this kind of situations, introducing proof terms for the sequents entailed by the logic, together with suitable axioms, which identify intuitively equivalent tile deductions (in particular, all the auxiliary tiles with the same border are identified).

DEFINITION 5.3.1 (PROCESS TILE SYSTEM)
A process tile system *(*PTS*)* $\mathcal{R}$ *is a 4-tuple* $\langle \Sigma_H, \Sigma_V, N, R \rangle$*, where* Σ_H *and* Σ_V *are hypersignatures over the same sorts,* N *is a set of rule names and* R *is a function*

$$R : N \longrightarrow \mathbf{S}(\Sigma_H) \times \mathbf{S}(\Sigma_V) \times \mathbf{S}(\Sigma_V) \times \mathbf{S}(\Sigma_H),$$

such that for all $r \in N$*, if* $R(r) = \langle h, v, u, g \rangle$*, then the arrows* h *and* v *have the same source, the arrows* g *and* u *have the same target, the source of* u *is equal to the target of* h*, and the source of* g *is equal to the target of* v *(i.e., they can correctly compose a tile).*

We will write a generic rule r s.t. $R(r) = \langle h, v, u, g \rangle$ either as the tile



(for appropriate objects a, b, c and d) or as the sequent $r : h \xrightarrow[u]{v} g$. As an example, it is now possible to define a PTS for the system of Example 5.3.1 presented using the wire and box notation. Indeed, let us consider the PTS where (notice that a symmetry appears in the trigger of r_2):

$$\begin{aligned}
\Sigma_H &= \{a : 0 \longrightarrow 1, g : 1 \longrightarrow 1, h : 2 \longrightarrow 1, f : 3 \longrightarrow 1\}, \\
\Sigma_V &= \{s : 1 \longrightarrow 1, t : 2 \longrightarrow 1\}, \\
N &= \{r_1, r_2\}, \\
&\quad r_1 : g \otimes a \xrightarrow[t]{id_1} id_1, \quad r_2 : f \xrightarrow[s]{(id_1 \otimes \gamma_{1,1});(t \otimes id_1)} h \ .
\end{aligned}$$

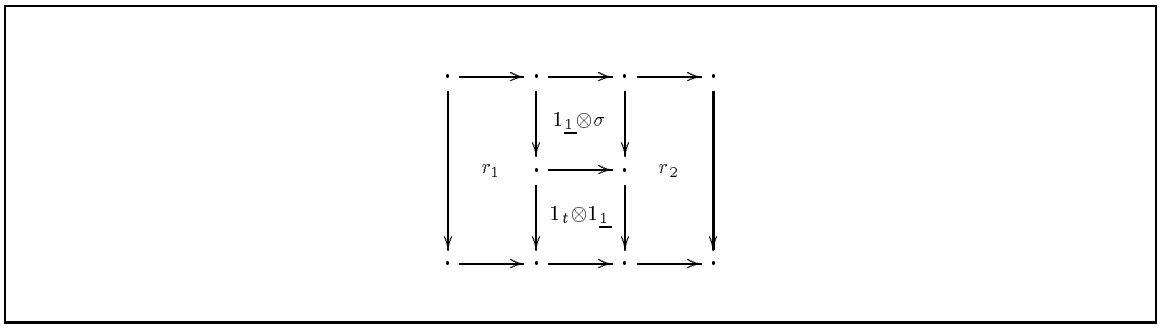


Figure 5.31: The rewriting of configuration $f(g(x_1), x_2, a)$ in process tile logic.

If an auxiliary tile $\sigma : \gamma_{1,1} \xrightarrow[\gamma_{1,1}]{id_2} id_2$ is also provided, then the three tiles can be composed for rewriting the configuration $(g \otimes id_1 \otimes a); f : \underline{2} \longrightarrow \underline{1}$ (corresponding to $f(g(x_1), x_2, a)$), into $h : \underline{2} \longrightarrow \underline{1}$ (i.e., $h(x_1, x_2)$) with identity trigger and yielding an effect $s : \underline{1} \longrightarrow \underline{1}$. We can use the composition scheme of Figure 5.30, but also the scheme in Figure 5.31.

5.3.1.1 The Inference Rules for Process Tile Logic

The rules of the PTS $\mathcal{R}$ can be interpreted as labelled sequents in an adequate *logic of tiles*. Starting from a PTS, it is then possible to derive all the tiles obtained by (finite) application of some *inference rules*, in the same way as it happens for rewriting logic and for algebraic tile logic.

The inference rules define the *free* compositions of the basic tiles in $\mathcal{R}$ according to the three operations (parallel composition, horizontal composition, and vertical composition) of tiles. Moreover, auxiliary tiles must be added for allowing consistent reorderings of configurations and observations.

First, let us consider a one-sorted hypersignature Σ. We denote by Sym (ranged over by $s, s', s_1, \ldots$) the subcategory of $\mathbf{S}(\Sigma)$ having exactly the possible compositions (parallel and sequential) of identities and symmetries as arrows. It can be easily noticed that, for any arrow $s \in Sym$, the source and target of s coincide.

Given $n \in \mathbb{N}$, we denote by Sym_n the subcategory $Sym[\underline{n}, \underline{n}]$ of arrows from $\underline{n}$ to $\underline{n}$ in Sym.

Since Sym does not depend on the signature Σ, we can assume that Sym is a subcategory of both $\mathbf{S}(\Sigma_H)$ and $\mathbf{S}(\Sigma_V)$. It follows that tiles which perform rearrangements of interfaces can be generated by the inference rule

$$\frac{n \in \mathbb{N},\ s_1, s_2, s_3, s_4 \in Sym_n,\ s_1; s_2 = s_3; s_4}{\mathcal{R} \vdash s_1 \xrightarrow[s_3]{s_2} s_4} \tag{5.3}$$

However, it is possible to obtain the same result starting from a reduced set of auxiliary sequents and using the inference rules for tile composition.

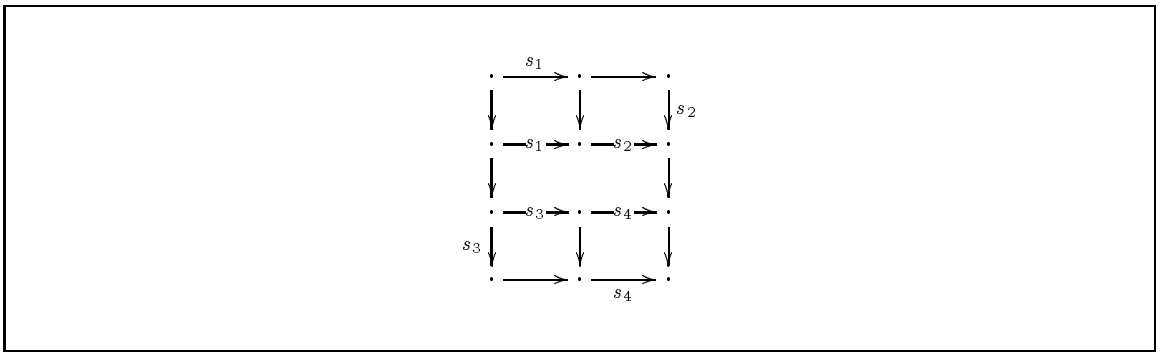


Figure 5.32: The composition scheme used in the proof of Lemma 5.3.1.

LEMMA 5.3.1
Parallel, horizontal and vertical composition of tiles, together with identities, allow us to replace the rule (5.3) by two simpler rules:

$$(swap)\ \frac{n \in \mathbb{N},\ s \in Sym_n}{\mathcal{R} \vdash s \overset{s}{\underset{id_n}{\longrightarrow}} id_n} \qquad (swap')\ \frac{n \in \mathbb{N},\ s \in Sym_n}{\mathcal{R} \vdash id_n \overset{id_n}{\underset{s}{\longrightarrow}} s}$$

Proof. Let $n \in \mathbb{N}$, and consider any $s_1, s_2, s_3, s_4 \in Sym_n$, such that $s_1; s_2 = s_3; s_4$. Since $Sym_n \subseteq \mathbf{S}(\Sigma_H)$, the vertical identities $s_1 \overset{id_n}{\underset{id_n}{\longrightarrow}} s_1$, $s_4 \overset{id_n}{\underset{id_n}{\longrightarrow}} s_4$, and

$$s_1; s_2 \overset{id_n}{\underset{id_n}{\longrightarrow}} s_1; s_2 = s_1; s_2 \overset{id_n}{\underset{id_n}{\longrightarrow}} s_3; s_4$$

are entailed in $\mathcal{R}$. The two rules $(swap)$ and $(swap')$ yield respectively the sequents $s_3 \overset{s_3}{\underset{id_n}{\longrightarrow}} id_n$ and $id_n \overset{id_n}{\underset{s_2}{\longrightarrow}} s_2$. Therefore, we can derive the sequent $s_1 \overset{s_3}{\underset{s_2}{\longrightarrow}} s_4$ generated by rule (5.3) by composing the sequents above using the composition scheme in Figure 5.32. We then observe that the two rules are instances of rule (5.3), and thus we can conclude that they generate exactly the same class of sequents. □

It is possible to do much better though: taking advantage of the compositional structure of Sym, we can have the following finite characterization of basic auxiliary tiles, consisting of only two auxiliary sequents:

$$(swap)\ \frac{}{\mathcal{R} \vdash \gamma_{1,1} \overset{\gamma_{1,1}}{\underset{id_2}{\longrightarrow}} id_2} \qquad (swap')\ \frac{}{\mathcal{R} \vdash id_2 \overset{id_2}{\underset{\gamma_{1,1}}{\longrightarrow}} \gamma_{1,1}}$$

The proof is given in the next section, using decorated sequent (analogous results hold for many-sorted hypersignatures). Hence, we have the following definition.

DEFINITION 5.3.2 (PROCESS TILE SEQUENTS)
Let $\mathcal{R} = \langle \Sigma_H, \Sigma_V, N, R \rangle$ be a PTS*, and let S denote the set of sorts $S_{\Sigma_H} = S_{\Sigma_V}$. We say that $\mathcal{R}$* entails *the class $S_p(\mathcal{R})$ of* flat process sequents *obtained by finitely many applications of the* inference rules *in Table 5.2. For any sequent $h \overset{v}{\underset{u}{\longrightarrow}} g \in S_p(\mathcal{R})$ we write $\mathcal{R} \vdash_{fp} h \overset{v}{\underset{u}{\longrightarrow}} g$, to be read as "$\mathcal{R}$ entails the flat process sequent $h \overset{v}{\underset{u}{\longrightarrow}} g$."*

Basic Sequents. *Generators and Identities:*

$$(gen)\ \frac{r : h \xrightarrow[u]{v} g \in R(N)}{h \xrightarrow[u]{v} g \in S_p(\mathcal{R})}$$

$$(v\text{-}ref)\ \frac{v : a \longrightarrow c \in \mathbf{S}(\Sigma_V)}{id_a \xrightarrow[v]{v} id_c \in S_p(\mathcal{R})} \qquad (h\text{-}ref)\ \frac{h : a \longrightarrow b \in \mathbf{S}(\Sigma_H)}{h \xrightarrow[id_b]{id_a} h\ \in S_p(\mathcal{R})}$$

Auxiliary Sequents. *Symmetries:*

$$(swap)\ \frac{a, b \in S}{\gamma_{a,b} \xrightarrow[id_{b\otimes a}]{\gamma_{a,b}} id_{b\otimes a}\ \in S_p(\mathcal{R})} \qquad (swap')\ \frac{a, b \in S}{id_{a\otimes b} \xrightarrow[\gamma_{a,b}]{id_{a\otimes b}} \gamma_{a,b}\ \in S_p(\mathcal{R})}$$

Composition Rules. *Parallel, Horizontal and Vertical compositions:*

$$(par)\ \frac{h \xrightarrow[u]{v} g \in S_p(\mathcal{R}),\ h' \xrightarrow[u']{v'} g' \in S_p(\mathcal{R})}{h \otimes h' \xrightarrow[u\otimes u']{v\otimes v'} g \otimes g'\ \in S_p(\mathcal{R})}$$

$$(hor)\ \frac{h \xrightarrow[u]{v} g \in S_p(\mathcal{R}),\ h' \xrightarrow[u']{u} g' \in S_p(\mathcal{R})}{h; h' \xrightarrow[u']{v} g; g' \in S_p(\mathcal{R})}$$

$$(vert)\ \frac{h \xrightarrow[u]{v} g \in S_p(\mathcal{R}),\ g \xrightarrow[u']{v'} g' \in S_p(\mathcal{R})}{h \xrightarrow[u;u']{v;v'} g'\ \in S_p(\mathcal{R})}$$

Table 5.2: Inference rules for flat process sequents.

An interesting result is that flat sequents allow both horizontal swapping of observations (as the one in algebraic tile logic) and vertical swapping of configurations.

PROPOSITION 5.3.2
Let $\mathcal{R} = \langle \Sigma_H, \Sigma_V, N, R \rangle$ be a PTS. *For all observations $v : a \longrightarrow c$, $u : b \longrightarrow d$ in $\mathbf{S}(\Sigma_V)$, and for all configurations $h : a \longrightarrow b$, $g : c \longrightarrow d$ in $\mathbf{S}(\Sigma_H)$, the sequents $\gamma_{a,b} \xrightarrow[u \otimes v]{v \otimes u} \gamma_{c,d}$ and $h \otimes g \xrightarrow[\gamma_{b,d}]{\gamma_{a,c}} g \otimes h$ are both entailed by $\mathcal{R}$.*

The proof is given in the next section by exploiting the algebraic structure of decorated sequents. We remark that the class of sequents $S_p(\mathcal{R})$ is *flat*, in the sense that we are not able to recover how a certain sequent has been entailed.

5.3.1.2 Proof Terms for Process Tile Logic

A more concrete version of process tile logic can be defined by decorating the sequents with *proof terms*.

Then, proof terms can be axiomatized in order to capture equivalent proofs according to the symmetric structure. However, the resulting equivalence classes make fewer identifications than those induced by the flat version (where two sequents having the same border are always identified).

We remark that the resulting *logic* is the same as before (see Proposition 5.3.3), but proof terms are now made explicit.

DEFINITION 5.3.3 (PROCESS TILE LOGIC)
Let $\mathcal{R} = \langle \Sigma_H, \Sigma_V, N, R \rangle$ be a PTS, *and let S denote the set of sorts $S_{\Sigma_H} = S_{\Sigma_V}$. We say that $\mathcal{R}$* entails *the class $P_p(\mathcal{R})$ of* decorated process sequents *obtained by a finite number of applications of the inference rules given in Table 5.3. For any sequent $\alpha : h \xrightarrow[u]{v} g \in P_p(\mathcal{R})$, we write $\mathcal{R} \vdash_p \alpha$.*

With proof terms decorating the sequents of the logic, it is possible to use an algebraic notation to subsume complex entailment in the logic.

EXAMPLE 5.3.2 *As an example, consider the following recursive definition (we consider a one-sorted signature, but the many-sorted case is analogous):*

$$\begin{aligned} \sigma_{0,k} &= 1_{id_k} \\ \sigma_{1,k+1} &= ((\sigma_{1,1} \otimes 1_{id_1}) * 1^{id_1 \otimes \gamma_{1,k}}) \cdot \sigma_{1,k} \\ \sigma_{n+1,k} &= ((1_{id_n} \otimes \sigma_{1,k}) * 1^{\gamma_{n,k} \otimes id_1}) \cdot (\sigma_{n,k} \otimes 1_{id_1}) \end{aligned}$$

It follows that for any $n, m \in \mathbb{N}$, then $\sigma_{n,m} : \gamma_{n,m} \xrightarrow[id_{m \otimes n}]{\gamma_{n,m}} id_{m \otimes n} \in P_p(\mathcal{R})$.

Likewise, we get that $\sigma'_{n,m} : id_{n \otimes m} \xrightarrow[\gamma_{n,m}]{id_{n \otimes m}} \gamma_{n,m} \in P_p(\mathcal{R})$, for any $n, m \in \mathbb{N}$.

Basic Proof Sequents. *Generators and Identities:*

$$\frac{r : h \xrightarrow[u]{v} g \in R(N)}{r : h \xrightarrow[u]{v} g \in P_p(\mathcal{R})} \qquad \frac{v : a \longrightarrow c \in \mathbf{S}(\Sigma_V)}{1_v : id_a \xrightarrow[v]{v} id_c \in P_p(\mathcal{R})} \qquad \frac{h : a \longrightarrow b \in \mathbf{S}(\Sigma_H)}{1^h : h \xrightarrow[id_b]{id_a} h \in P_p(\mathcal{R})}$$

Auxiliary Proof Sequents. *Symmetries:*

$$\frac{a, b \in S}{\sigma_{a,b} : \gamma_{a,b} \xrightarrow[id_{b\otimes a}]{\gamma_{a,b}} id_{b\otimes a} \in P_p(\mathcal{R})} \qquad \frac{a, b \in S}{\sigma'_{a,b} : id_{a\otimes b} \xrightarrow[\gamma_{a,b}]{id_{a\otimes b}} \gamma_{a,b} \in P_p(\mathcal{R})}$$

Composition Rules. *Parallel, Horizontal and Vertical compositions:*

$$\frac{\alpha : h \xrightarrow[u]{v} g \in P_p(\mathcal{R}),\ \alpha' : h' \xrightarrow[u']{v'} g' \in P_p(\mathcal{R})}{\alpha \otimes \alpha' : h \otimes h' \xrightarrow[u\otimes u']{v\otimes v'} g \otimes g' \in P_p(\mathcal{R})}$$

$$\frac{\alpha : h \xrightarrow[u]{v} g \in P_p(\mathcal{R}),\ \beta : h' \xrightarrow[u']{u} g' \in P_p(\mathcal{R})}{\alpha * \beta : h; h' \xrightarrow[u']{v} g; g' \in P_p(\mathcal{R})}$$

$$\frac{\alpha : h \xrightarrow[u]{v} g \in P_p(\mathcal{R}),\ \beta : g \xrightarrow[u']{v'} g' \in P_p(\mathcal{R})}{\alpha \cdot \beta : h \xrightarrow[u;u']{v;v'} g' \in P_p(\mathcal{R})}$$

Table 5.3: Inference rules for decorated process sequents.

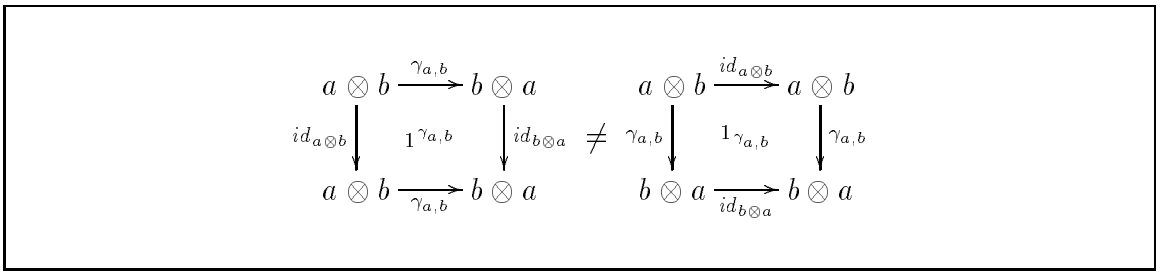


Figure 5.33: $1^{\gamma_{a,b}} \neq 1_{\gamma_{a,b}}$.

Now the proof for Proposition 5.3.2 can be easily constructed as follows:

- for any two effects $v : a \longrightarrow c,\ u : b \longrightarrow d \in \mathbf{S}(\Sigma_V)$ then

$$\gamma_{v,u} = \sigma_{a,b} \cdot 1_{\gamma_{b,a}} \cdot 1_{v\otimes u} \cdot \sigma'_{c,d} :\ \gamma_{a,b} \xrightarrow[u\otimes v]{v\otimes u} \gamma_{c,d}\ \in P_p(\mathcal{R});$$

- for any two configurations $h : a \longrightarrow b,\ g : c \longrightarrow d \in \mathbf{S}(\Sigma_H)$ then

$$\rho_{h,g} = \sigma_{a,c} * 1^{\gamma_{c,a}} * 1^{h\otimes g} * \sigma'_{b,d} :\ h \otimes g \xrightarrow[\gamma_{b,d}]{\gamma_{a,c}} g \otimes h\ \in P_p(\mathcal{R}).$$

We remark that the sequent $1^{\gamma_{a,b}}$ is different from $1_{\gamma_{a,b}}$ (see Figure 5.33).

PROPOSITION 5.3.3
Given a PTS $\mathcal{R}$, *then* $\mathcal{R} \vdash_{fp} h \xrightarrow[u]{v} g \iff \exists \alpha : h \xrightarrow[u]{v} g \in P_p(\mathcal{R})$.

Proof. Immediately follows from the fact that the inference rules in Table 5.2 are exactly the same as in Table 5.3. □

5.3.1.3 Axiomatizing Process Tile Logic

The axiomatization we propose aims at the identification of intuitively equivalent tile computations in process tile logic. As an example, all compositions of auxiliary tiles (and identities) yielding the same flat sequent must be identified. Since the axiomatization is rather long we prefer to sketch here the more interesting properties and to refer the reader to Appendix A for the complete list of axioms.

DEFINITION 5.3.4 (ABSTRACT PROCESS TILE LOGIC)
Let $\mathcal{R} = \langle \Sigma_H, \Sigma_V, N, R \rangle$ *be a* PTS. *We say that* $\mathcal{R}$ entails *the class* $A_p(\mathcal{R})$ *of* abstract process sequents, *whose elements are equivalence classes of proof terms in* $P_p(\mathcal{R})$ *modulo the set of axioms described below (see also Appendix A for the complete list of axioms):*
Associativity Axioms *as in Definition 2.4.3.*
Identity Axioms *as in Definition 2.4.3.*
Monoidality Axioms *as in Definition 2.4.3.*

Functoriality Axioms *for* identities *and* composition *as in Definition 2.4.3.*
Functoriality Axioms *for* derived operators γ *and* ρ*, stating that the swapping of two observations (configurations) respects identities and sequential composition.*
Naturality Axioms *for* derived operators γ *and* ρ *(for all sequents* $\alpha : h \xrightarrow[v]{u} g$ *and* $\alpha' : h' \xrightarrow[v']{u'} g'$ *in* $P_p(\mathcal{R})$*):*

$$(\alpha \otimes \alpha') * \gamma_{u,u'} = \gamma_{v,v'} * (\alpha' \otimes \alpha) \qquad (\alpha \otimes \alpha') \cdot \rho_{g,g'} = \rho_{h,h'} \cdot (\alpha' \otimes \alpha)$$

Uniqueness Axioms *stating that any two compositions of basic auxiliary sequents* ($\sigma_{a,b}$ *and* $\sigma'_{a,b}$) *and identity sequents of configurations and observations* (1^h *and* 1_v) *yielding the same flat sequent are identified. In Appendix A it is shown that these axioms can be partitioned into two main subclasses: naturality axioms for* σ *and* σ'*, and coherence axioms for* γ*,* ρ*,* σ*, and* σ'*.*

Abusing notation, we will write $\mathcal{R} \vdash_p \alpha$ to denote the entailment of the abstract process sequent α (rather than just the entailment of the decorated sequent α).

EXAMPLE 5.3.3 *Let us consider the* PTS $\mathcal{R}_{SMP} = \mathcal{R}_{SAMP} \cup \{\mathtt{synch}_a\}_{a \in \Gamma}$ *defined in Section 5.1.1, whose basic tiles are those presented in Figures 5.1 and 5.7. The algebra* $A_p(\mathcal{R}_{SMP})$ *consists in all the possible sequential and parallel compositions of the basic tiles of the system, the horizontal and vertical identities, and the double symmetries. Then, a possible solution to the synchronization problem proposed for the configuration*

$$\begin{array}{rcl} Q & = & (proc; select; (id_\circ \otimes (select; (receive_a \otimes id_\circ)))) \otimes \\ & \otimes & (proc; select; (id_\circ \otimes (select; (send_a \otimes id_\circ)))) \end{array}$$

can be expressed by the decorated sequent:

$$\begin{array}{rcl} S & = & (1^{R \otimes R} * 1^{id_\circ \otimes receive_a \otimes id_\circ \otimes id_\circ \otimes send_a \otimes id_\circ} * (1_\circ \otimes \sigma^{-1}_{a,\circ \otimes \circ} \otimes 1_{a \otimes \circ})) \cdot \\ & \cdot & (S' \otimes S' * (1_\circ \otimes \gamma_{signal, id_\circ \otimes id_\circ} \otimes 1_{signal} \otimes 1_\circ) * (1_{\circ \otimes \circ \otimes \circ} \otimes \mathtt{synch}_a \otimes 1_\circ)) \cdot \\ & \cdot & (1^{R \otimes R} * (1_\circ \otimes \gamma_{id_a, id_\circ \otimes id_\circ} \otimes 1_{a \otimes \circ}) * 1^{id_\circ \otimes \circ \otimes \circ \otimes receive_a \otimes send_a \otimes id_\circ} * \\ & & *(1_\circ \otimes \sigma^{-1}_{\circ \otimes \circ, a} \otimes 1_{a \otimes \circ})) \end{array}$$

where $R = proc; select; (id_\circ \otimes select)$ *and* $S' = \mathtt{comm} * \mathtt{right} * (1_\circ \otimes \mathtt{left})$*. Moreover, the axioms of* $A_p(\mathcal{R}_{SMP})$ *suffice to show the equivalence between the abstract sequent* S *and the sequent* T *defined below:*

$$\begin{array}{rcl} T & = & (1^{R \otimes R} * 1^{id_\circ \otimes receive_a \otimes id_\circ \otimes id_\circ \otimes send_a \otimes id_\circ} * (1_{\circ \otimes a} \otimes \sigma^{-1}_{\circ \otimes \circ, a} \otimes 1_\circ)) \cdot \\ & \cdot & (S' \otimes S' * (1_\circ \otimes 1_{signal} \otimes \gamma_{id_\circ \otimes id_\circ, signal} \otimes 1_\circ) * (1_\circ \otimes \mathtt{synch}_a \otimes 1_{\circ \otimes \circ \otimes \circ})) \cdot \\ & \cdot & (1^{R \otimes R} * (1_{\circ \otimes a} \otimes \gamma_{id_\circ \otimes id_\circ, id_a} \otimes 1_\circ) * 1^{id_\circ \otimes receive_a \otimes send_a \otimes id_{\circ \otimes \circ \otimes \circ}} * \\ & & *(1_{\circ \otimes a} \otimes \sigma^{-1}_{a, \circ \otimes \circ} \otimes 1_\circ)) \end{array}$$

5.3.2 Term Tile Logic

In this section we introduce a tile format whose configurations and observations are (tuples of) terms over two distinct signatures (over the same sorts), and their composition is defined by term substitution.

When configurations and observations are terms over two distinct (one-sorted) signatures Σ_H and Σ_V, we can assume that a generic basic tile has the form (already discussed by means of examples in Section 5.1.2):

$$\begin{array}{ccc} \underline{n} & \xrightarrow{\vec{h}} & \underline{m} \\ {\scriptstyle \vec{v}}\downarrow & & \downarrow{\scriptstyle u} \\ \underline{k} & \xrightarrow[g]{} & \underline{1} \end{array}$$

with $\vec{h} \in T_{\Sigma_H}(X_n)^m$, $g \in T_{\Sigma_H}(X_k)$, $\vec{v} \in T_{\Sigma_V}(X_n)^k$, and $u \in T_{\Sigma_V}(X_m)$, where $X_i = \{x_1, ..., x_i\}$ is a chosen set of variables, totally ordered by $x_{j_1} < x_{j_2}$ if $j_1 < j_2$, and $T_\Sigma(X)^n$ denotes the n-tuples of terms over the signature Σ and variables in X. The idea is that each interface represents an ordered sequence (i.e., a tuple) of variables; therefore each variable is completely identified by its position in the tuple, and a standard naming $x_1, ..., x_n$ of the variables can be assumed. For example, if the variable x_i appears in the effect u of the above rule, this means that the effect u depends on the i-th component $\vec{h}_i$ of the initial configuration $\vec{h}$. Analogously for the remaining connections. We will present the tiles more concisely using the notation

$$n \triangleright \langle \vec{h} \rangle \xrightarrow[\langle u \rangle]{\langle \vec{v} \rangle} \langle g \rangle$$

where the number of variables in the "upper-left" corner of the tile is made explicit (the values m and k can be easily recovered from the lengths of $\vec{h}$ and $\vec{v}$ respectively). When confusion does not arise, we omit the brackets that delimit term tuples, writing $n \triangleright \vec{h} \xrightarrow[u]{\vec{v}} g$

REMARK 5.3.4 *Notice that the same variable x_i denotes the i-th element of different interfaces when used in each of the four border-arrows of the tile (in particular, only the occurrences of x_i in $\vec{h}$ and in $\vec{v}$ refer to the same element of the initial input interface $\underline{n}$).*

As argued in Section 5.1.2, this notation could become confusing when dealing with many-sorted signatures. This is because different types could be assigned to the same variable in different interfaces (e.g., the variable x_1 could have type a in the initial input interface and type b in the final input interface).

A first solution consists in adopting the notation of algebraic theories, dropping the more familiar term-notation.

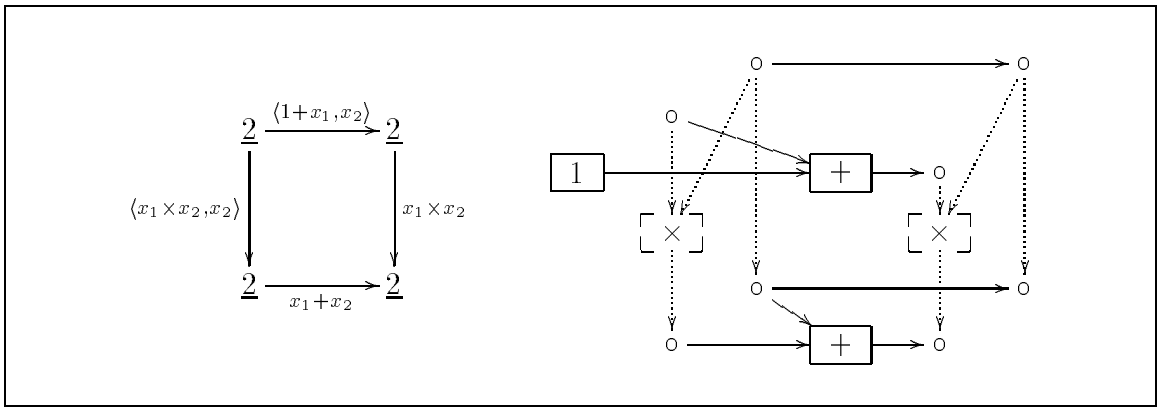


Figure 5.34: A tile for computing $(1+n)\times m$ (left) and its wire and box representation (right).

An alternative consists in presenting tuples of typed terms with variables via syntactic judgments of the form $? \triangleright t : \sigma$, where $?$ is an environment, and σ is a type on a type system equipped with product types over a first order signature Σ.

To improve readability, we will consider one-sorted signatures. However, the extension to the many-sorted case does not present any particular difficulty (apart from the more complex notation).

DEFINITION 5.3.5 (TERM TILE SYSTEM)
A term tile system *(*TTS*)* $\mathcal{R}$ *is a 4-tuple* $\langle \Sigma_H, \Sigma_V, N, R\rangle$*, where* Σ_H *and* Σ_V *are one-sorted signatures,* N *is a set of rule names, and* R *is a function*

$$R : N \longrightarrow \bigcup_{n,m,k\in\mathbb{N}} (T_{\Sigma_H}(X_n))^m \times (T_{\Sigma_V}(X_n))^k \times T_{\Sigma_V}(X_m) \times T_{\Sigma_H}(X_k)$$

where $X_l = \{x_1, ..., x_l\}$ *is a fixed (totally ordered by* $x_i < x_j$ *iff* $i < j$*) set of variables.*

Abusing notation, we denote by λ the empty vector of terms over $T_{\Sigma_H}(X_n)$ and $T_{\Sigma_V}(X_n)$ for each $n \in \mathbb{N}$.

Analogously to PTS, rules in $\mathcal{R}$ can be interpreted as labelled sequents in a *logic of tiles*. Starting from a TTS, it is possible to derive all the tiles obtained by finite application of some *inference rules*, which define the *free* composition of the basic tiles in $\mathcal{R}$ according to the three operations (parallel composition, horizontal composition, and vertical composition) of tile systems. Moreover, some auxiliary tiles are added to allow performing consistent reorderings, duplications, and discharging of variables, state components, and observations.

EXAMPLE 5.3.5 *Suppose that the vertical signature consists of a binary operator* $\_\times\_$ *representing the product of natural numbers, and that the horizontal signature consists of a binary operator* $\_+\_$ *representing the sum of natural numbers. The cell in Figure 5.34 represents a possible way of computing* $(1+n)\times m$*, for generic input arguments* $\langle n, m\rangle$*.*

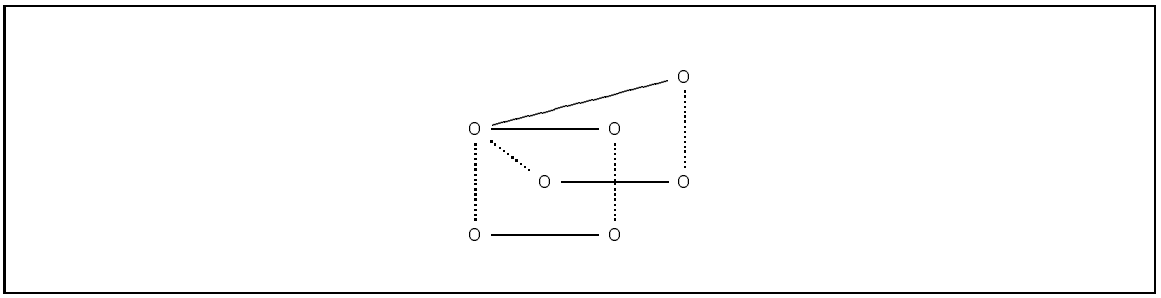


Figure 5.35: Consistent duplication of the initial input interface.

The idea is that to compute the effect $(1+n)\times m$ *from the initial configuration* $\langle 1+n, m\rangle$ *we can sum (see the final configuration) the couple* $\langle n\times m, m\rangle$ *computed by some subcomponent and provided as the trigger of the tile.*

Indeed, the effect applied to the initial configuration (i.e., the horizontal-vertical path along the border of the tile) yields $x_1 \times x_2[1+x_1/x_1, x_2/x_2] = (1+x_1)\times x_2$*, which is clearly arithmetically equivalent to the result given from the application of the final configuration to the trigger (i.e., the vertical-horizontal path)* $x_1 + x_2[x_1\times x_2/x_1, x_2/x_2] = (x_1\times x_2)+x_2$.

EXAMPLE 5.3.6 *Consider the tile defined in Example 5.3.5, and suppose that we must compute* $(1+n)\times n$*, which has only one argument.*

We do not want to redefine an instantiation of our rule for the particular case $n=m$*, because this would undermine the modularity of the specification.*

The simplest solution consists in allowing input duplication, provided that this happens consistently in both the horizontal and the vertical dimension. This can be done by introducing the auxiliary tile below (see Figure 5.35).

$$1 \triangleright\ x_1, x_1 \xrightarrow[x_1, x_2]{x_1, x_1} x_1, x_2$$

Informally, both its initial configuration and its trigger (term vectors $\langle x_1, x_1\rangle$*) duplicate a variable, and both its final configuration and its effect (term vectors* $\langle x_1, x_2\rangle$*) consider the resulting copies as distinct variables.*

To obtain the expected sequent, we vertically compose the auxiliary sequent with the horizontal identity $2 \triangleright\ x_1, x_2 \xrightarrow[x_1\times x_2, x_2]{x_1\times x_2, x_2} x_1, x_2$ *associated to* $\langle x_1\times x_2, x_2\rangle$*, thus obtaining the sequent* $1 \triangleright\ x_1, x_1 \xrightarrow[x_1\times x_2, x_2]{x_1\times x_1, x_1} x_1, x_2$ *, which can be horizontally composed with the sequent in Figure 5.34. The result is the expected sequent*

$$1 \triangleright\ 1 + x_1, x_1 \xrightarrow[x_1\times x_2]{x_1\times x_1, x_1} x_1 + x_2\ .$$

5.3.2.1 The Inference Rules for Term Tile Logic

DEFINITION 5.3.6 (TERM TILE SEQUENTS)
Let $\mathcal{R} = \langle \Sigma_H, \Sigma_V, N, R \rangle$ be a TTS *(over one-sorted signatures). We say that $\mathcal{R}$* entails *the class $S_t(\mathcal{R})$ of* flat term sequents *obtained by a finite number of applications of the* deduction rules *given in Table 5.4, where the notation $\vec{s}\,[t_i/x_i]_{i=1}^n$ denotes the simultaneous substitution of the variables $x_1, \ldots, x_n$ by the corresponding terms $t_1, \ldots, t_n$ in all the terms of the tuple $\vec{s}$.*

For any sequent $n \triangleright \vec{h} \xrightarrow[\vec{u}]{\vec{v}} \vec{g} \in S_t(\mathcal{R})$ we write $\mathcal{R} \vdash_{ft} n \triangleright \vec{h} \xrightarrow[\vec{u}]{\vec{v}} \vec{g}$, to be read as "$\mathcal{R}$ entails the (flat) sequent $n \triangleright \vec{h} \xrightarrow[\vec{u}]{\vec{v}} \vec{g}$."

We briefly comment on the inference rules in Table 5.4, showing also some examples of deduction in the style of Example 5.3.6. The first rule (*gen*) says that $\mathcal{R}$ entails all the flat versions (i.e., without the label) of the rules in $R(N)$. Rules (*v-ref*) and (*h-ref*) define idle vertical and horizontal components of the system respectively (we could have used a more finitary approach by defining (*v-ref*) and (*h-ref*) only for the operators of the signatures and deriving the sequents for generic terms by composing the "basic" auxiliary sequents).

Then, some auxiliary tiles are added "for free." They are necessary to guarantee the completeness of $S_t(\mathcal{R})$ w.r.t. all the permutations, tuplings and projections of configurations and observations, and they are independent from the TTS under consideration. We can divide the auxiliary rules in three sub-classes: symmetries, duplicators, and dischargers. Rules (*swap*) and (*swap'*) define basic consistent swappings of adjacent variables according to the fact that a swapping in one dimension must be simulated in the other dimension. The term tile framework allows using a term-like notation instead of symmetries γ as in Section 5.3.1, but the rules (*swap*) and (*swap'*) define exactly the same basic sequents

$$\gamma_{1,1} \xrightarrow[id_2]{\gamma_{1,1}} id_2 \quad \text{and} \quad id_2 \xrightarrow[\gamma_{1,1}]{id_2} \gamma_{1,1}$$

of process tile logic. More complex swappings can also be entailed by $\mathcal{R}$. As an example, for any $n, m \in \mathbb{N}$, the sequent

$$(n+m) \triangleright\ x_{n+1}, \ldots, x_{n+m}, x_1, \ldots, x_n \xrightarrow[x_1,\ldots,x_{n+m}]{x_{n+1},\ldots,x_{n+m},x_1,\ldots,x_n} x_1, \ldots, x_{n+m}$$

(that swaps the first n variables of the interface with the successive m variables), is in $S_t(\mathcal{R})$. Observations (configurations) swappings can be handled as in process tile logic: for any two observations $v \in T_{\Sigma_V}(X_n)$ and $u \in T_{\Sigma_V}(X_m)$ the sequent

$$(n+m) \triangleright\ x_{n+1}, \ldots, x_{n+m}, x_1, \ldots, x_n \xrightarrow[u, u[x_{m+i}/x_i]_{i=1}^n]{v, u[x_{n+i}/x_i]_{i=1}^m} x_2, x_1$$

is entailed by $\mathcal{R}$.

Basic Sequents. *Generators and Identities:*

$$(gen)\ \frac{r : n \triangleright\ \vec{h} \xrightarrow[u]{\vec{v}} g\ \in R(N)}{n \triangleright\ \vec{h} \xrightarrow[u]{\vec{v}} g\ \in S_t(\mathcal{R})}$$

$$(v\text{-}ref)\ \frac{\vec{v} \in (T_{\Sigma_V}(X_n))^k}{n \triangleright\ x_1, ..., x_n \xrightarrow[\vec{v}]{\vec{v}} x_1, ..., x_k\ \in S_t(\mathcal{R})} \qquad (h\text{-}ref)\ \frac{\vec{h} \in (T_{\Sigma_H}(X_n))^m}{n \triangleright\ \vec{h} \xrightarrow[x_1,...,x_m]{x_1,...,x_n} \vec{h}\ \in S_t(\mathcal{R})}$$

Auxiliary Sequents. *Symmetries, Duplicators and Dischargers:*

$$(swap)\ \frac{}{2 \triangleright\ x_2, x_1 \xrightarrow[x_1,x_2]{x_2,x_1} x_1, x_2\ \in S_t(\mathcal{R})} \qquad (swap')\ \frac{}{2 \triangleright\ x_1, x_2 \xrightarrow[x_2,x_1]{x_1,x_2} x_2, x_1\ \in S_t(\mathcal{R})}$$

$$(dup)\ \frac{}{1 \triangleright\ x_1, x_1 \xrightarrow[x_1,x_2]{x_1,x_1} x_1, x_2\ \in S_t(\mathcal{R})} \qquad (dup')\ \frac{}{1 \triangleright\ x_1 \xrightarrow[x_1,x_1]{x_1} x_1, x_1\ \in S_t(\mathcal{R})}$$

$$(dis)\ \frac{}{1 \triangleright\ \lambda \xrightarrow[\lambda]{\lambda} \lambda\ \in S_t(\mathcal{R})} \qquad (dis')\ \frac{}{1 \triangleright\ x_1 \xrightarrow[\lambda]{x_1} \lambda\ \in S_t(\mathcal{R})}$$

Composition Rules. *Parallel, Horizontal and Vertical compositions:*

$$(par)\ \frac{n \triangleright\ \vec{h} \xrightarrow[\vec{u}]{\vec{v}} \vec{g}\ ,\ n' \triangleright\ \vec{h'} \xrightarrow[\vec{u'}]{\vec{v'}} \vec{g'}\ \in S_t(\mathcal{R}),\ \left|\vec{h}\right| = m, |\vec{v}| = k, \left|\vec{h'}\right| = m', \left|\vec{v'}\right| = k'}{(n+n') \triangleright\ \vec{h}, \vec{h'}[x_{i+n}/x_i]_{i=1}^{n'} \xrightarrow[\vec{u},\vec{u'}[x_{i+m}/x_i]_{i=1}^{m'}]{\vec{v},\vec{v'}[x_{i+n}/x_i]_{i=1}^{n'}} \vec{g}, \vec{g'}[x_{i+k}/x_i]_{i=1}^{k'}\ \in S_t(\mathcal{R})}$$

$$(hor)\ \frac{n \triangleright\ \vec{h} \xrightarrow[\vec{u}]{\vec{v}} \vec{g}\ ,\ m \triangleright\ \vec{h'} \xrightarrow[\vec{u'}]{\vec{u}} \vec{g'}\ \in S_t(\mathcal{R}),\ \left|\vec{h}\right| = m,\ |\vec{u}| = k}{n \triangleright\ \vec{h'}[h_i/x_i]_{i=1}^{m} \xrightarrow[\vec{u'}]{\vec{v}} \vec{g'}[g_i/x_i]_{i=1}^{k}\ \in S_t(\mathcal{R})}$$

$$(vert)\ \frac{n \triangleright\ \vec{h} \xrightarrow[\vec{u}]{\vec{v}} \vec{g}\ ,\ k \triangleright\ \vec{g} \xrightarrow[\vec{u'}]{\vec{v'}} \vec{g'}\ \in S_t(\mathcal{R}),\ |\vec{v}| = k,\ |\vec{u}| = l}{n \triangleright\ \vec{h} \xrightarrow[\vec{u'}[u_i/x_i]_{i=1}^{l}]{\vec{v'}[v_i/x_i]_{i=1}^{k}} \vec{g'}\ \in S_t(\mathcal{R})}$$

Table 5.4: Inference rules for flat term sequents.

The second class of auxiliary rules produces tiles for "making consistent copies." Rules (*dup*) and (*dup'*) duplicate a variable of the interface in both the horizontal and the vertical dimension. An application of rule (*dup*) has been shown in Example 5.3.6. Using the notation of algebraic theories (see Section 2.2), rules (*dup*) and (*dup'*) can be written as

$$\frac{}{\nabla_1 \overset{\nabla_1}{\underset{id_2}{\longrightarrow}} id_2 \in S_t(\mathcal{R})} \qquad \frac{}{id_1 \overset{id_1}{\underset{\nabla_1}{\longrightarrow}} \nabla_1 \in S_t(\mathcal{R})}$$

By suitably composing the basic auxiliary tiles for duplication and swapping with identities we can conclude that, for any $n \in \mathbb{N}$, the TTS $\mathcal{R}$ entails the sequent

$$n \triangleright\ x_1, \ldots, x_n, x_1 \ldots, x_n \xrightarrow[x_1,\ldots,x_{2n}]{x_1,\ldots,x_n,x_1\ldots,x_n} x_1, \ldots, x_{2n} \tag{5.4}$$

(which duplicates an interface of length n). Moreover, $S_t(\mathcal{R})$ contains also the sequents for the duplication of observations and of configurations. We illustrate this result for a generic configuration $h \in T_{\Sigma_H}(X_n)$: from rule (*h-ref*) we get the sequent $n \triangleright\ h \xrightarrow[x_1]{x_1,\ldots,x_n} h$, which can be horizontally composed with the sequent generated by (*dup'*) to obtain the sequent

$$n \triangleright\ h \xrightarrow[x_1,x_1]{x_1,\ldots,x_n} h, h\ . \tag{5.5}$$

Then, we can horizontally compose the sequent (5.4) together with the identity sequent for the parallel composition of h with itself. The resulting sequent is

$$n \triangleright\ h, h \xrightarrow[x_1,x_2]{x_1,\ldots,x_n,x_1,\ldots,x_n} h, h[x_{i+n}/x_i]_{i=1}^{n}\ . \tag{5.6}$$

Finally, the vertical composition of sequents (5.5) and (5.6) yields the sequent

$$n \triangleright\ h \xrightarrow[x_1,x_1]{x_1,\ldots,x_n,x_1,\ldots,x_n} h, h[x_{i+n}/x_i]_{i=1}^{n}\ .$$

EXAMPLE 5.3.7 *Consider a vertical signature that contains a unary operator $v(\_)$. A similar construction to the one illustrated above yields the sequent*

$$1 \triangleright\ x_1, x_1 \xrightarrow[v(x_1),v(x_2)]{v(x_1)} x_1, x_1\ . \tag{5.7}$$

By vertically composing the sequent (5.7) with the sequent $1 \triangleright\ x_1, x_1 \xrightarrow[x_1,x_2]{x_1,x_1} x_1, x_2$ (which is generated by rule (dup)), we derive the sequent $1 \triangleright\ x_1, x_1 \xrightarrow[v(x_1),v(x_2)]{v(x_1),v(x_1)} x_1, x_2$. Making a backward interpretation, such sequent states that, whenever the same effects are required for two distinct components, the system can match the condition with just one component being able to produce the same effect twice.

The last class of auxiliary rules introduces dischargers. Rules (*dis*) and (*dis'*) consistently discharge a variable. Like symmetries and duplicators, they can be composed to get more complex rules for projecting configurations, etc.

The composition rules define a general scheme for combining sequents which are "already" entailed in order to build more complex entailments. The parallel composition defined by the rule (*par*) is a total operation. It describes how to put in parallel any two sequents. Intuitively, they are put side by side, the variables of the "second" sequent being renamed with standard fresh variables (they are dependent upon the variables used in the "first" sequent). For instance, if we put in parallel the elementary idle sequent $1 \triangleright\ x_1 \xrightarrow[x_1]{x_1} x_1$ with itself, we obtain the flat term sequent $2 \triangleright\ x_1, x_2 \xrightarrow[x_1,x_2]{x_1,x_2} x_1, x_2$.

Horizontal composition, described by the rule (*hor*), is partial. It applies only if the effects produced by the first sequent correspond exactly (for instance, no arbitrary renamings of the variables are allowed) to the triggers required by the second sequent. The horizontal composition of two composable sequents yields the sequent obtained by taking the trigger of the first sequent, the effect of the second sequent and by substituting the variables involved in the configurations of the second sequent with the corresponding configurations of the first sequent. Similarly for the vertical composition. These operations have been extensively used in the previous examples.

5.3.2.2 Proof Terms for Term Tile Logic

Likewise $S_p(\mathcal{R})$, the resulting class of sequents $S_t(\mathcal{R})$ is flat. We provide a more concrete inference system by decorating sequents with proof terms. Then proof terms can be axiomatized in order to capture equivalent proofs according to the intuitive cartesian structure. However, the resulting equivalence classes make fewer identifications than those induced by the flat version (where two sequents having the same border are always identified).

DEFINITION 5.3.7 (TERM TILE LOGIC)
Let $\mathcal{R} = \langle \Sigma_H, \Sigma_V, N, R \rangle$ be a TTS *(over one-sorted signatures). We say that $\mathcal{R}$* entails *the class $P_t(\mathcal{R})$ of* decorated term sequents *obtained by a finite number of applications of the inference rules defined in Table 5.5.*

For any sequent $\alpha : n \triangleright\ \vec{l} \xrightarrow[\vec{u}]{\vec{v}} \vec{g} \in P_t(\mathcal{R})$ we say that $\mathcal{R}$ entails α, written either $\mathcal{R} \vdash_t \alpha$ or, more verbosely, $\mathcal{R} \vdash_t \alpha : n \triangleright\ \vec{l} \xrightarrow[\vec{u}]{\vec{v}} \vec{g}$.

The inference rules are the same as those in Table 5.4, but now each sequent is decorated with a proof term, which describes the deduction process entailing that sequent.

Basic Proof Sequents. *Generators and Identities:*

$$\frac{r : n \triangleright \ \vec{h} \xrightarrow[u]{\vec{v}} g \ \in R(N)}{r : n \triangleright \ \vec{h} \xrightarrow[u]{\vec{v}} g \ \in P_t(\mathcal{R})}$$

$$\frac{\vec{v} \in (T_{\Sigma_V}(X_n))^k}{1_{\vec{v}} : n \triangleright \ x_1, ..., x_n \xrightarrow[\vec{v}]{\vec{v}} x_1, ..., x_k \ \in P_t(\mathcal{R})} \qquad \frac{\vec{h} \in (T_{\Sigma_H}(X_n))^m}{1^{\vec{h}} : n \triangleright \ \vec{h} \xrightarrow[x_1,...,x_m]{x_1,...,x_n} \vec{h} \ \in P_t(\mathcal{R})}$$

Auxiliary Proof Sequents. *Symmetries, Duplicators and Dischargers:*

$$\frac{}{\sigma_{1,1} : 2 \triangleright \ x_2, x_1 \xrightarrow[x_1,x_2]{x_2,x_1} x_1, x_2 \ \in P_t(\mathcal{R})} \qquad \frac{}{\sigma'_{1,1} : 2 \triangleright \ x_1, x_2 \xrightarrow[x_2,x_1]{x_1,x_2} x_2, x_1 \ \in P_t(\mathcal{R})}$$

$$\frac{}{\pi_1 : 1 \triangleright \ x_1, x_1 \xrightarrow[x_1,x_2]{x_1,x_1} x_1, x_2 \ \in P_t(\mathcal{R})} \qquad \frac{}{\tau_1 : 1 \triangleright \ x_1 \xrightarrow[x_1,x_1]{x_1} x_1, x_1 \ \in P_t(\mathcal{R})}$$

$$\frac{}{\phi_1 : 1 \triangleright \ \lambda \xrightarrow[\lambda]{\lambda} \lambda \ \in P_t(\mathcal{R})} \qquad \frac{}{\psi_1 : 1 \triangleright \ x_1 \xrightarrow[\lambda]{x_1} \lambda \ \in P_t(\mathcal{R})}$$

Composition Rules. *Parallel, Horizontal and Vertical compositions:*

$$\frac{\alpha : n \triangleright \ \vec{h} \xrightarrow[\vec{u}]{\vec{v}} \vec{g} \ , \alpha' : n' \triangleright \ \vec{h'} \xrightarrow[\vec{u'}]{\vec{v'}} \vec{g'} \ \in P_t(\mathcal{R}), \left|\vec{h}\right| = m, |\vec{v}| = k, \left|\vec{h'}\right| = m', \left|\vec{v'}\right| = k'}{\alpha \otimes \alpha' : (n + n') \triangleright \ \vec{h}, \vec{h'}[x_{i+n}/x_i]_{i=1}^{n'} \xrightarrow[\vec{u},\vec{u'}[x_{i+m}/x_i]_{i=1}^{m'}]{\vec{v},\vec{v'}[x_{i+n}/x_i]_{i=1}^{n'}} \vec{g}, \vec{g'}[x_{i+k}/x_i]_{i=1}^{k'} \ \in P_t(\mathcal{R})}$$

$$\frac{\alpha : n \triangleright \ \vec{h} \xrightarrow[\vec{u}]{\vec{v}} \vec{g} \ , \ \alpha' : m \triangleright \ \vec{h'} \xrightarrow[\vec{u'}]{\vec{u}} \vec{g'} \ \in P_t(\mathcal{R}), \ \left|\vec{h}\right| = m, \ |\vec{u}| = k}{\alpha * \alpha' : n \triangleright \ \vec{h'}[h_i/x_i]_{i=1}^{m} \xrightarrow[\vec{u'}]{\vec{v}} \vec{g'}[g_i/x_i]_{i=1}^{k} \ \in P_t(\mathcal{R})}$$

$$\frac{\alpha : n \triangleright \ \vec{h} \xrightarrow[\vec{u}]{\vec{v}} \vec{g} \ , \ \alpha' : k \triangleright \ \vec{g} \xrightarrow[\vec{u'}]{\vec{v'}} \vec{g'} \ \in P_t(\mathcal{R}), \ |\vec{v}| = k, \ |\vec{u}| = l}{\alpha \cdot \alpha' : n \triangleright \ \vec{h} \xrightarrow[\vec{u'}[u_i/x_i]_{i=1}^{l}]{\vec{v'}[v_i/x_i]_{i=1}^{k}} \vec{g'} \ \in P_t(\mathcal{R})}$$

Table 5.5: Inference rules for decorated term sequents.

PROPOSITION 5.3.4
Given a TTS $\mathcal{R}$, *then* $\mathcal{R} \vdash_{ft} n \triangleright \vec{h} \xrightarrow[\vec{u}]{\vec{v}} \vec{g} \iff \exists \alpha : n \triangleright \vec{h} \xrightarrow[\vec{u}]{\vec{v}} \vec{g} \in P_t(\mathcal{R})$.

Proof terms provide a tool for the algebraic definition of some interesting classes of sequents. For instance $\sigma_{n,m}$, $\sigma'_{n,m}$, $\gamma_{v,u}$, and $\rho_{h,g}$ can be defined exactly as in process tile logic (see page 150). Similarly, the sequents

$$\pi_n : n \triangleright x_1, \ldots, x_n, x_1 \ldots, x_n \xrightarrow[x_1,\ldots,x_{2n}]{x_1,\ldots,x_n,x_1\ldots,x_n} x_1, \ldots, x_{2n}$$

$$\tau_n : n \triangleright x_1, \ldots, x_n \xrightarrow[x_1,\ldots,x_n,x_1\ldots,x_n]{x_1,\ldots,x_n} x_1, \ldots, x_n, x_1 \ldots, x_n$$

for the duplication of interfaces can be defined as follows:

$$\begin{array}{rcl} \pi_0 & = & \tau_0 \; = \; 1_{id_0} \\ \pi_{n+1} & = & ((\pi_n \otimes \pi_1) * (1_{id_n \otimes \gamma_{n,1} \otimes id_1})) \cdot (1_{id_n} \otimes \sigma_{n,1} \otimes 1_{id_1}) \\ \tau_{n+1} & = & ((\tau_n \otimes \tau_1) * (1_{\nabla_1 \otimes \nabla_n})) \cdot (1_{id_n} \otimes \sigma_{n,1} \otimes 1_{id_1}) \end{array}$$

Furthermore, the sequents for duplicating configurations and observations can be constructed through the following expressions:

- for any observation $\vec{v} \in (T_{\Sigma_V}(X_n))^k$, then $\nabla_{\vec{v}} = (1_{\vec{v}} \cdot \tau_k) * (\pi_n \cdot 1_{\vec{v} \otimes \vec{v}})$, with

$$\nabla_{\vec{v}} : n \triangleright x_1, ..., x_n, x_1, ..., x_n \xrightarrow[\vec{v},\vec{v}[x_{n+i}/x_i]_{i=1}^n]{\vec{v}} x_1, ..., x_k, x_1, ..., x_k$$

- for any configuration $\vec{h} \in (T_{\Sigma_H}(X_n))^m$, then

$$\delta_{\vec{h}} = (1^{\vec{h}} * \tau_m) \cdot (\pi_n \cdot 1^{\vec{h} \otimes \vec{h}}) : n \triangleright \vec{h} \xrightarrow[x_1,\ldots,x_m,x_1,\ldots,x_m]{x_1,\ldots,x_n,x_1,\ldots,x_n} \vec{h}, \vec{h}[x_{n+i}/x_i]_{i=1}^n$$

Dischargers also admit a similar generalization, yielding the sequents:

$$\phi_n : n \triangleright \lambda \xrightarrow[\lambda]{\lambda} \lambda \qquad \psi_n : n \triangleright x_1, \ldots, x_n \xrightarrow[\lambda]{x_1,\ldots,x_n} \lambda$$

$$!_{\vec{v}} = (1_{\vec{v}} \cdot \psi_k) * \phi_n : n \triangleright \lambda \xrightarrow[\lambda]{\vec{v}} \lambda \qquad \dagger_{\vec{h}} = (1^{\vec{h}} * \psi_m) \cdot \phi_n : n \triangleright \vec{h} \xrightarrow[\lambda]{\lambda} \lambda$$

where $\phi_0 = \psi_0 = 1_{id_0}$, $\phi_{n+1} = \phi_n \otimes \phi_1$, and $\psi_{n+1} = \psi_n \otimes \psi_1$.

5.3.2.3 Axiomatizing Term Tile Logic

The class $P_t(\mathcal{R})$ turns out to be too concrete, in the sense that sequents that intuitively should represent the same rewriting may have different representations.

The following axiomatization identifies intuitively equivalent tile computations in term tile logic. In particular, all the compositions of auxiliary tiles (and identities) yielding the same flat sequents are identified. However, the list of axioms is quite long, so that we prefer to give an informal description of the more interesting properties. We refer the interested reader to Appendix B for the complete axiomatization.

DEFINITION 5.3.8 (ABSTRACT TERM TILE LOGIC)
Let $\mathcal{R} = \langle \Sigma_H, \Sigma_V, N, R \rangle$ be a TTS. *We say that $\mathcal{R}$* entails *the class $A_t(\mathcal{R})$ of* abstract term sequents, *whose elements are equivalence classes of proof terms in $P_t(\mathcal{R})$ modulo the set of axioms described below (see also Appendix B for the complete list of axioms):*
Associativity Axioms *as in Definition 2.4.3.*
Identity Axioms *as in Definition 2.4.3.*
Monoidality Axioms *as in Definition 2.4.3.*
Functoriality Axioms *for* identities *and* composition *as in Definition 2.4.3.*
Functoriality Axioms *for* derived operators γ *and* ρ, *stating that the swapping of two observations (configurations) respects identities and sequential composition.*
Functoriality Axioms *for* derived operators ∇ *and* δ, *stating that the duplication of observations (configurations) respects identities and sequential composition.*
Functoriality Axioms *for* derived operators $!$ *and* $\dagger$, *stating that the discharging of observations (configurations) respects identities and sequential composition.*
Naturality Axioms *for* derived operators γ *and* ρ *(for all sequents* $\alpha : h \xrightarrow[v]{u} g$ *and* $\alpha' : h' \xrightarrow[v']{u'} g'$ *in* $P_t(\mathcal{R})$*):*

$$(\alpha \otimes \alpha') * \gamma_{u,u'} = \gamma_{v,v'} * (\alpha' \otimes \alpha) \qquad (\alpha \otimes \alpha') \cdot \rho_{g,g'} = \rho_{h,h'} \cdot (\alpha' \otimes \alpha)$$

Naturality Axioms *for* derived operators ∇ *and* δ *(for any sequent* $\alpha : h \xrightarrow[v]{u} g$ *in* $P_t(\mathcal{R})$*):*

$$\alpha * \nabla_u = \nabla_v * (\alpha \otimes \alpha) \qquad \alpha \cdot \delta_g = \delta_h \cdot (\alpha \otimes \alpha)$$

Naturality Axioms *for* derived operators $!$ *and* $\dagger$ *(for any sequent* $\alpha : h \xrightarrow[v]{u} g$ *in* $P_t(\mathcal{R})$*):*

$$\alpha * !_u = !_v \qquad \alpha \cdot \dagger_g = \dagger_h$$

Uniqueness Axioms, *stating that any two compositions of basic auxiliary sequents ($\sigma_{1,1}$, $\sigma'_{1,1}$, π_1, τ_1, ϕ_1 and ψ_1) and identity sequents of configuration and observations (1^h and 1_v) yielding the same flat sequent are identified. In Appendix B it is shown that these axioms can be partitioned into two main subclasses: naturality axioms, and coherence axioms.*

Abusing the notation, we will write $\mathcal{R} \vdash_t \alpha$ to denote the entailment of the abstract term sequent of α and not just of the decorated sequent α.

The comparison of tile logic with rewriting logic suggested that we look at cartesian double categories as the basis on which to interpret the algebraic structures of the models. Unfortunately, the notion of cartesian double categories is not present (at least to our knowledge) in the literature. Thus, an exploration of the subject has been necessary as part of this research. If we assume the existence of a double category $\mathcal{L}_{\mathcal{D}}(\mathcal{R})$ with chosen double products, where the objects are underlined natural numbers, the horizontal (vertical) arrows are the terms over the horizontal (vertical) signature and the cells are (equivalence classes of) proof terms, then a very general

notion of *term tile model* can be given in terms of double-product-preserving double functor from $\mathcal{L}_{\mathcal{D}}(\mathcal{R})$ to a generic cartesian double category. Keeping this concept in mind, we will explore a suitable definition of cartesian double categories with chosen products (Section 5.4.2) and its relationship to cartesian 2-categories (Chapter 6). We will eventually show how it is possible to derive the double category $\mathcal{L}_{\mathcal{D}}(\mathcal{R})$ via a free construction (see Definition 6.5.2 and Theorem 6.5.1).

5.3.2.4 Format Expressiveness

The format of term tiles is very general. In particular, it extends the *positive GSOS* format [11], where the generic rule has the form

$$\frac{\{x_i \xrightarrow{a_{ij}} y_{ij} \mid 1 \leq i \leq k, 1 \leq j \leq n_i\}}{f(x_1, ..., x_k) \xrightarrow{a} C[x_1, y_{11}, ..., y_{1n_1}, ..., x_k, y_{k1}, ..., y_{kn_k}]}$$

where the variables are all distinct, f is a k-ary operator of the (configuration) signature, $n_i \geq 0$, a_{ij} and a are actions and C is a context. Indeed, this becomes the term tile

$$k \triangleright\ f(x_1, ..., x_k) \xrightarrow[a(x_1)]{\vec{a}} C[x_1 ... x_N]\ ,$$

where $N = \sum_{i=1}^{k}(n_i + 1)$, and $\vec{a} = x_1, a_{11}(x_1), .., a_{1n_1}(x_1), ..., x_k, a_{k1}(x_k), .., a_{kn_k}(x_k)$ is the vector of triggers (for each argument of f, $\vec{a}$ contains the idle trigger plus all the actions that are tested in the premises of the GSOS rule for that argument).

Notice that the positive GSOS format cannot be handled by the tile systems defined by using the internal construction, e.g., where the duplicator is not allowed in both dimensions (see [69]). Finally notice that term tile logic can also handle rules with *lookahead* as the one defined in [77]:

$$\frac{x \xrightarrow{a^+} y,\ y \xrightarrow{a^-} z}{\text{combine}(x) \xrightarrow{a} \text{combine}(z)}$$

which, in term tile format, becomes $1 \triangleright\ \text{combine}(x_1) \xrightarrow[a(x_1)]{a^-(a^+(x_1))} \text{combine}(x_1)$.

Therefore, term tile format appears very expressive and this encourages us to provide an executable framework for its specification (see Chapter 7).

5.4 Symmetric Monoidal and Cartesian Double Categories

Process and term tile logic represent two wide classes of systems whose configurations and observations can both be described as *net-process*-like structures and *terms* over a given signature.

The fundamental concept that gives the basis of both logics is the introduction of suitable *auxiliary* tiles, modelling all the consistent rearrangements of the interfaces. In this section we propose a characterization of "double" auxiliary structures in terms of suitable generalized transformations, introducing original definitions of *symmetric strict monoidal* and *cartesian* double categories.

In particular, the axiomatization of proof sequents for process tile logic makes the algebra $A_p(\mathcal{R})$ of proof sequents into a symmetric strict monoidal double category. In this sense, the models of process tile logic could be adequately represented as symmetries-preserving monoidal double functors from $A_p(\mathcal{R})$ to generic SsMD's.

Similarly, the axiomatization of proof sequents for term tile logic makes the algebra $A_t(\mathcal{R})$ of proof sequents into a cartesian double category, defining a suitable initial model of $\mathcal{R}$ for term tile logic.

5.4.1 Symmetric Strict Monoidal Double Categories

Let $X : \mathcal{D} \times \mathcal{D} \longrightarrow \mathcal{D} \times \mathcal{D}$ be the double functor which swaps the arguments, i.e., such that for each $A, B \in \mathcal{D}$, $X(A, B) = (B, A)$.

In the 1-dimensional case, a *symmetry* is a natural isomorphism between the tensor product ${}_{-1}\otimes_{-2}$ (the functor $\otimes$) and the swapped tensor product ${}_{-2}\otimes_{-1}$ (the functor $X;\otimes$), which verifies some additional coherence axioms [93] (see Section 2.1.3).

A *double symmetry* is a generalized natural transformation, with a generalized inverse, and it verifies some similar axioms.

REMARK 5.4.1 *For notation reasons, in what follows we favour the horizontal dimension, by using the common symbols associated to ordinary symmetries, duplicators and dischargers to denote natural $*$-transformations rather than natural $\cdot$-transformations. However, all such notions, being based on that of generalized natural transformation, treat the horizontal and vertical dimensions in the same way.*

DEFINITION 5.4.1 (SYMMETRIC STRICT MONOIDAL DOUBLE CATEGORIES) *A* symmetric strict monoidal double category, *SsMD for short, is a tuple $(\mathcal{D}, \otimes, e, \sigma)$ such that the triple $(\mathcal{D}, \otimes, e)$ is a sMD, and σ is the generalized natural transformation pictured below.*

$$\begin{array}{ccc} \otimes & \xrightarrow{\gamma} & X;\otimes \\ {\scriptstyle\rho}\downarrow & \sigma & \downarrow{\scriptstyle 1} \\ X;\otimes & \xrightarrow[1]{} & X;\otimes \end{array}$$

This means that all the following equations have to be satisfied:

- Naturality *of γ and ρ:*

  For any pair of cells $A : h \xrightarrow[u]{v} g$, $A' : h' \xrightarrow[u']{v'} g'$ in $\mathcal{D}$,

  $$(A \otimes A') * \gamma_{u,u'} = \gamma_{v,v'} * (A' \otimes A), \qquad (A \otimes A') \cdot \rho_{g,g'} = \rho_{h,h'} \cdot (A' \otimes A).$$

- Functoriality *of* γ *and* ρ*:*

  For any vertical arrows $v : a \longrightarrow c$, $v' : a' \longrightarrow c'$, $w : c \longrightarrow d$, $w' : c' \longrightarrow d'$,

  $$\gamma_{v \cdot w, v' \cdot w'} = \gamma_{v,v'} \cdot \gamma_{w,w'}.$$

  For any horizontal arrows $h : a \longrightarrow b$, $h' : a' \longrightarrow b'$, $f : b \longrightarrow c$, $f' : b' \longrightarrow c'$,

  $$\rho_{h*f,h'*f'} = \rho_{h,h'} * \rho_{f,f'}.$$

  For any pair of objects a *and* a' *in* $\mathcal{D}$,

  $$\gamma_{id_a,id_{a'}} = 1^{\gamma_{a,a'}}, \qquad \rho_{id_a,id_{a'}} = 1_{\rho_{a,a'}}.$$

- Naturality *of* σ*:*

  For any vertical arrows $v : a \longrightarrow c$ *and* $u : b \longrightarrow d$ *in* $\mathcal{D}$,

  $$\gamma_{v,u} \cdot \sigma_{c,d} = \sigma_{a,b} \cdot 1_{u \otimes v}.$$

  For any horizontal arrows $h : a \longrightarrow b$ *and* $g : c \longrightarrow d$ *in* $\mathcal{D}$,

  $$\rho_{h,g} * \sigma_{b,d} = \sigma_{a,c} * 1^{g \otimes h}.$$

- *Moreover, the Kelly-MacLane* coherence axioms *for* γ, ρ *and* σ *also hold:*

  For any vertical arrows $v : a \longrightarrow c$, $u : b \longrightarrow d$, *and* $w : a' \longrightarrow c'$ *in* $\mathcal{D}$,

  $$\gamma_{u \otimes w,v} = (1_u \otimes \gamma_{w,v}) * (\gamma_{u,v} \otimes 1_w), \qquad \gamma_{v,u} * \gamma_{u,v} = 1_{v \otimes u}.$$

  For any horizontal arrows $h : a \longrightarrow b$, $g : c \longrightarrow d$, *and* $f : a' \longrightarrow b'$ *in* $\mathcal{D}$,

  $$\rho_{f \otimes g,h} = (1^f \otimes \rho_{g,h}) \cdot (\rho_{f,h} \otimes 1^g), \qquad \rho_{h,g} \cdot \rho_{g,h} = 1^{h \otimes g}.$$

  For any objects a, b, *and* c *in* $\mathcal{D}$,

  $$\sigma_{a \otimes b,c} = (1_a \otimes \sigma_{b,c}) \triangleleft (\sigma_{a,c} \otimes 1_b), \qquad \sigma_{a,b} \triangleleft \sigma_{b,a} = 1_{a \otimes b}.$$

The generalized inverse of $\sigma_{a,b}$ can be easily defined in terms of σ, γ and ρ:

- the $*$-*inverse* of $\sigma_{a,b}$ is $\sigma^*_{a,b} = \sigma_{b,a} \cdot 1_{\rho_{a,b}}$,
- the $\cdot$-*inverse* of $\sigma_{a,b}$ is $\sigma^{\cdot}_{a,b} = \sigma_{b,a} * 1^{\gamma_{a,b}}$, and
- the *generalized inverse* of $\sigma_{a,b}$ is $\sigma^{-1}_{a,b} = 1_{\rho_{b,a}} \cdot (\sigma_{a,b} * 1^{\gamma_{b,a}}) = 1^{\gamma_{b,a}} * (\sigma_{a,b} \cdot 1_{\rho_{b,a}})$.

REMARK 5.4.2 *The above notation could be somehow misleading, because the generalized inverse $\sigma_{a,b}^{-1}$ of $\sigma_{a,b}$ is not $\sigma_{b,a}$ as one might expect from the second coherence axiom for σ. The fact is that $\sigma_{a,b} * 1^{\gamma_{b,a}} \neq 1_{\rho_{a,b}}$; thus, $1^{\gamma_{b,a}} \neq \sigma_{a,b}^{*}$. In some sense, the cell $\sigma_{b,a}$ is just the diagonal inverse of $\sigma_{a,b}$.*

DEFINITION 5.4.2 (CATEGORY **SsMDCat**)
We denote by **SsMDCat** *the category of SsMD's and monoidal double functors preserving all the symmetries.*

PROPOSITION 5.4.1
The forgetful functor from **SsMDCat** *to* **Set**, *mapping a SsMD into its set of objects has a left adjoint that maps each set S into the* free SsMD *on S (denoted by $DSym_S$) whose objects are the elements of the free monoid $S^{\otimes}$ over S.*

The following *representation theorem* states the correspondence between double symmetries and ordinary symmetries.

THEOREM 5.4.2
For any set S, the double category $DSym_S$ is isomorphic to the double category of quartets over the free symmetric strict monoidal category Sym_S on S.

Proof. Since Sym_S is isomorphic to $\mathbf{S}(\Sigma_S)$, where Σ_S is the empty signature over the set of sorts S, it follows that any arrow $f \in Sym_S$ can be represented as the composition $f_1; \cdots; f_n$, where each f_i has the form $id_{a_i} \otimes \gamma_{b_i,c_i} \otimes id_{d_i}$, for $i = 1, ..., n$, i.e., f_i is the parallel composition of a symmetry with suitable identities, and we call *elementary* all such arrows. Then, for any elementary arrow $f_i : x_i \longrightarrow y_i$ we define $A_{f_i} = 1_{a_i} \otimes \sigma_{b_i,c_i} \otimes 1_{d_i} : f_i \xrightarrow[y_i]{f_i} y_i$ and $B_{f_i} = 1_{a_i} \otimes \sigma'_{b_i,c_i} \otimes 1_{d_i} : x_i \xrightarrow[f_i]{x_i} f_i$. Therefore, for any $f : x \longrightarrow y \in Sym_S$, the cells $A_f = A_{f_1} \triangleleft \cdots \triangleleft A_{f_n} : f \xrightarrow[y]{f} y$ and $B_f = B_{f_1} \triangleright \cdots \triangleright B_{f_n} : x \xrightarrow[f]{x} f$ belong $DSym_S$. Eventually, given any commutative square $h; u = v; g$ in Sym_S, the cell $(1^h * B_u) \cdot (A_v * 1^g) : h \xrightarrow[u]{v} g$ is in $DSym_S$.

For the converse correspondence, first note that all the cells in $DSym_S$ are defined in terms of σ (also the identities), and thus all the cell represent commutative diagrams of Sym_s. To conclude the proof, we must show that any two cells with the same "border" are identified in $DSym_S$ (e.g., this property is obvious for the diagonal categories, due to the coherence axioms). We show that each cell $\alpha : h \xrightarrow[u]{v} g$ admits the following normalized representation in terms of "diagonal" cells:

$$\alpha = (B_h * A_h * B_u) \cdot (A_v * B_g * A_g)$$

Such normal form is indeed preserved by the three compositions of symmetric monoidal double categories. We give the proof for the horizontal composition (parallel composition is trivial and horizontal composition is analogous to the vertical case). Let $\alpha : h \xrightarrow[u]{v} g$ and $\beta : h' \xrightarrow[w]{u} g'$, we want to show that $\alpha * \beta$ is equal to $(B_{h;h'} * A_{h;h'} * B_w) \cdot (A_v * B_{g;g'} * A_{g;g'})$.

$$\begin{aligned}
\alpha * \beta &= ((B_h * A_h * B_u) \cdot (A_v * B_g * A_g)) * ((B_{h'} * A_{h'} * B_w) \cdot (A_u * B_{g'} * A_{g'})) \\
&= (1^h \cdot (B_h * A_h * B_u) \cdot (A_v * B_g * A_g)) * \\
&\quad *((B_{h'} * A_{h'} * B_w) \cdot (A_u * B_{g'} * A_{g'}) \cdot 1^{g'}) \\
&= (1^h * B_{h'} * A_{h'} * B_w) \cdot (B_h * A_h * B_u * A_u * B_{g'} * A_{g'}) \cdot \\
&\quad \cdot (A_v * B_g * A_g * 1^{g'}) \\
&= (1^h * 1^{h'} * B_w) \cdot (1^h * 1^u * 1^{g'}) \cdot (A_v * 1^g * 1^{g'}) \\
&= (1^{h;h'} * B_w) \cdot (1^{h;u;g'}) \cdot (A_v * 1^{g;g'}) \\
&= (B_{h;h'} * A_{h;h'} * B_w) \cdot (A_v * B_{g;g'} * A_{g;g'})
\end{aligned}$$

□

Therefore, we have an interesting characterization of the components of σ as the possible square-shaped decompositions of the arrows of the free symmetric strict monoidal category Sym_S on S. This establishes a correspondence between the first and the second characterizations of auxiliary tiles given in Section 1.2. An analogous result holds for the cartesian case (see Theorem 5.4.5).

It is easy to show that the axiomatization of proof sequents for the process tile logic as given in Definition 5.3.4 makes $A_p(\mathcal{R})$ into a SsMD (where $\sigma'_{n,m} = \sigma^{-1}_{m,n}$). In this sense, the models of process tile logic could be adequately represented as symmetries-preserving monoidal double functors from $A_p(\mathcal{R})$ to generic SsMD's (see Proposition 6.4.1 and the analogous result to the one expressed by Theorem 6.5.1 for the cartesian case).

5.4.2 Cartesian Double Categories

A fairly general notion of double products should require the products to exist according to the four possible compositions that we have seen: horizontal ($\_ * \_$), vertical ($\_ \cdot \_$), and the two diagonals ($\_ \triangleleft \_$ and $\_ \triangleright \_$). Pursuing the analogy with the 1-dimensional case (see Section 2.1), we propose the following definitions.

Definition 5.4.3 (Double products and double terminal object)
Given a double category $\mathcal{D}$, we say that $\mathcal{D}$ has all double (binary) products *if all the categories $\mathcal{D}^*$, $\mathcal{D}^\cdot$, $\mathcal{D}^\triangleleft$, and $\mathcal{D}^\triangleright$ have (binary) products.*

We say that $\mathcal{D}$ has a double terminal object *if all the categories $\mathcal{D}^*$, $\mathcal{D}^\cdot$, $\mathcal{D}^\triangleleft$, and $\mathcal{D}^\triangleright$ have a terminal object.*

Definition 5.4.4 (Cartesian Double Category)
A double category $\mathcal{D}$ is called cartesian *if it has all (binary) double products and a double terminal object.*

However, we are interested in a much tighter notion of product, similar to the choice of a "canonical product."

In fact, the more liberal definition does not establish any correspondence between the same notions on different dimensions, but simply states their existence. Thus, we adopt the convention that not only are the products *chosen* in all the four dimensions, but that they are also *consistently chosen*. For simplicity, in the rest of the thesis, we will consider only this kind of cartesian double categories if not stated otherwise, so that we can safely omit the phrase "consistently chosen products."

REMARK 5.4.3 *In general there are many different kinds of cartesian double categories with* partially chosen products*, where only some of the categories $\mathcal{D}^*$, $\mathcal{D}^\cdot$, $\mathcal{D}^\triangleleft$, and $\mathcal{D}^\triangleright$ have chosen products. In this sense, the more general definition could be called with* least chosen products *and the definition which we are about to discuss could be also called* with most chosen products.

In the 1-dimensional case, the notion of category with chosen products has an equivalent formulation in terms of symmetric monoidal category enriched by two natural transformations called *duplicator* and *discharger*. A *duplicator* is a natural transformation between the identity and the tensor product of two copies of the argument and verifies some additional coherence axioms involving symmetries and dischargers. A *discharger* is a natural transformation between the identity and the constant functor mapping each element into the unit of the tensor product. Thus, *double duplicator* and *double discharger* can be defined as generalized natural transformations verifying similar coherence axioms. Let $\Delta : \mathcal{D} \longrightarrow \mathcal{D} \times \mathcal{D}$ be the diagonal double functor which makes a copy of the argument, i.e., such that for each $A \in \mathcal{D}$, $\Delta(A) = (A, A)$. Then we have the following definition:

DEFINITION 5.4.5 (CARTESIAN DOUBLE CATEGORY WITH C.C.PRODUCTS)
A cartesian double category (with consistently chosen products) *is a tuple* $(\mathcal{D}, \otimes, e, \sigma, \pi, \tau, \phi, \psi)$ *such that* $(\mathcal{D}, \otimes, e, \sigma)$ *is a SsMD enriched with the generalized natural transformations* π, τ, ϕ, *and* ψ *pictured below.*

$$\begin{array}{ccc} 1_{\mathcal{D}} & \xrightarrow{\nabla} & \Delta;\otimes \\ {\scriptstyle\delta}\downarrow & \pi & \downarrow{\scriptstyle 1} \\ \Delta;\otimes & \xrightarrow[1]{} & \Delta;\otimes \end{array} \qquad \begin{array}{ccc} 1_{\mathcal{D}} & \xrightarrow{1} & 1_{\mathcal{D}} \\ {\scriptstyle 1}\downarrow & \tau & \downarrow{\scriptstyle\delta} \\ 1_{\mathcal{D}} & \xrightarrow[\nabla]{} & \Delta;\otimes \end{array} \qquad \begin{array}{ccc} 1_{\mathcal{D}} & \xrightarrow{!} & e \\ {\scriptstyle\dagger}\downarrow & \phi & \downarrow{\scriptstyle 1} \\ e & \xrightarrow[1]{} & e \end{array} \qquad \begin{array}{ccc} 1_{\mathcal{D}} & \xrightarrow{1} & 1_{\mathcal{D}} \\ {\scriptstyle 1}\downarrow & \psi & \downarrow{\scriptstyle\dagger} \\ 1_{\mathcal{D}} & \xrightarrow[!]{} & e \end{array}$$

This means that all the following equations have to be satisfied:

- Naturality *of* ∇, $!$, $\dagger$, *and* δ*:*

  For any cell $A : h \xrightarrow[u]{v} g$ *in* $\mathcal{D}$,

$$\begin{array}{ll} A * \nabla_u = \nabla_v * (A \otimes A), & A * !_u = !_v, \\ A \cdot \dagger_g = \dagger_h, & A \cdot \delta_g = \delta_h \cdot (A \otimes A). \end{array}$$

- Functoriality *of* ∇, $!$, $\dagger$, *and* δ*:*

  For any vertical arrows $v : a \longrightarrow c$ *and* $w : c \longrightarrow d$ *in* $\mathcal{D}$,

  $$\nabla_{v \cdot w} = \nabla_v \cdot \nabla_w, \qquad !_{v \cdot w} = !_v \cdot !_w.$$

  For any horizontal arrows $h : a \longrightarrow b$ *and* $f : b \longrightarrow d$ *in* $\mathcal{D}$,

  $$\delta_{h * f} = \delta_h * \delta_f, \qquad \dagger_{h * f} = \dagger_h * \dagger_f.$$

  For any object a *in* $\mathcal{D}$,

  $$\nabla_{id_a} = 1^{\nabla_a}, \qquad !_{id_a} = 1^{!_a}, \qquad \dagger_{id_a} = 1_{\dagger_a}, \qquad \delta_{id_a} = 1_{\delta_a}.$$

- Naturality *of* π, ϕ, ψ, *and* τ*:*

  For any vertical arrow $v : a \longrightarrow c$ *in* $\mathcal{D}$,

  $$\nabla_v \cdot \pi_c = \pi_a \cdot 1_{v \otimes v}, \qquad !_v \cdot \phi_c = \phi_a, \qquad 1_v \cdot \psi_c = \psi_a, \qquad 1_v \cdot \tau_c = \tau_a \cdot \nabla_v.$$

  For any horizontal arrow $h : a \longrightarrow b$ *in* $\mathcal{D}$,

  $$\delta_h * \pi_b = \pi_a * 1^{h \otimes h}, \qquad \dagger_h * \phi_b = \phi_a, \qquad 1^h * \psi_b = \psi_a, \qquad 1^h * \tau_b = \tau_a * \delta_h.$$

- *Kelly-MacLane* coherence axioms *for* ∇, $!$, δ, *and* $\dagger$*:*

  For any vertical arrows $v : a \longrightarrow c$, $u : b \longrightarrow d$, *and for any horizontal arrows* $h : a \longrightarrow b$, $g : c \longrightarrow d$ *in* $\mathcal{D}$,

  $$\begin{array}{ll}
  \nabla_{v \otimes u} = (\nabla_v \otimes \nabla_u) * (1_v \otimes \gamma_{v,u} \otimes 1_u), & \delta_{h \otimes g} = (\delta_h \otimes \delta_g) \cdot (1^h \otimes \rho_{h,g} \otimes 1^g), \\
  !_{v \otimes u} = !_v \otimes !_u, & \dagger_{h \otimes g} = \dagger_h \otimes \dagger_g, \\
  \nabla_{id_e} = 1_e = !_{id_e}, & \delta_{id_e} = 1_e = \dagger_{id_e}, \\
  \nabla_v * (\nabla_v \otimes 1_v) = \nabla_v * (1_v \otimes \nabla_v), & \delta_h \cdot (\delta_h \otimes 1^h) = \delta_h \cdot (1^h \otimes \delta_h), \\
  \nabla_v * \gamma_{v,v} = \nabla_v, & \delta_h \cdot \rho_{h,h} = \delta_h, \\
  \nabla_v * (1_v \otimes !_v) = 1_v, & \delta_h \cdot (1^h \otimes \dagger_h) = 1^h.
  \end{array}$$

- *Kelly-MacLane* coherence axioms *for* π, ϕ, τ, *and* ψ*:*

  For any objects a *and* b *in* $\mathcal{D}$,

  $$\begin{array}{ll}
  \pi_{a \otimes b} = (\pi_a \otimes \pi_b) \triangleleft (1_a \otimes \sigma_{a,b} \otimes 1_b), & \tau_{a \otimes b} = (\tau_a \otimes \tau_b) \triangleright (1_a \otimes \sigma^{-1}_{a,b} \otimes 1_b), \\
  \phi_{a \otimes b} = \phi_a \otimes \phi_b, & \psi_{a \otimes b} = \psi_a \otimes \psi_b, \\
  \pi_e = 1_e = \phi_e, & \tau_e = 1_e = \psi_e, \\
  \pi_a \triangleleft (\pi_a \otimes 1_a) = \pi_a \triangleleft (1_a \otimes \pi_a), & \tau_a \triangleright (\tau_a \otimes 1_a) = \tau_a \triangleright (1_a \otimes \tau_a), \\
  \pi_a \triangleleft \sigma_{a,a} = \pi_a, & \tau_a \triangleright \sigma^{-1}_{a,a} = \tau_a, \\
  \pi_a \triangleleft (1_a \otimes \phi_a) = 1_a, & \tau_a \triangleright (1_a \otimes \psi_a) = 1_a.
  \end{array}$$

- Double coherence axioms *for* π, ϕ, τ, *and* ψ:

  For any objects a *in* $\mathcal{D}$,

$$\tau_a * \pi_a = \nabla_a, \qquad \tau_a \cdot \pi_a = \delta_a, \qquad \psi_a * \phi_a = !_a, \qquad \psi_a \cdot \phi_a = \dagger_a.$$

Since most of the axioms are either just a rephrasing of those for the 1-dimensional case, or they are due to the definition of generalized natural transformations, we believe that only the last four *double coherence* axioms need some comment. They are needed in order to ensure the coherence of our structure (in the sense that *auxiliary* cells are uniquely identified by looking at their border). We should have also required similar axioms for σ and σ^{-1} in the definition of SsMD's, but since symmetries define isomorphisms there was no need to introduce explicitly σ^{-1}, because its existence was implied by the presence of σ, γ and ρ.

Duplicators and dischargers (in general) are not isomorphisms; thus, we need to introduce both kinds of tiles: π and τ, and ϕ and ψ. In doing this we need to ensure that their compositions do not introduce any unnecessary additional structure. For instance, the vertical composition $\tau \cdot \pi$ returns a generalized natural transformation

$$\begin{array}{ccc} 1_{\mathcal{D}} & \xrightarrow{1} & 1_{\mathcal{D}} \\ {\scriptstyle\delta}\downarrow & {\scriptstyle\tau\cdot\pi} & \downarrow{\scriptstyle\delta} \\ \Delta;\otimes & \xrightarrow[1]{} & \Delta;\otimes \end{array}$$

that is already present as the identity of δ.

Definition 5.4.6 (Category **CDCat**)
We call **CDCat** *the category of cartesian double categories with consistently chosen products and monoidal double functors preserving all the symmetries, duplicators and dischargers.*

Proposition 5.4.3
The category **CDCat** *is isomorphic to the category of cartesian double categories (in the sense of Definition 5.4.4) where all the products are chosen and where the morphisms are product preserving functors on the nose.*

Proposition 5.4.4
The forgetful functor from **CDCat** *to* **Set**, *mapping each cartesian double category into its set of objects, has a left adjoint which maps each set* S *into the* free cartesian double category *on* S *(denoted by* $DCart_S$*) whose objects are the elements of the free monoid* $S^{\otimes}$ *over* S.

Theorem 5.4.5
For any set S, *the double category* $DCart_S$ *is isomorphic to the double category of quartets over the free cartesian category* $Cart_S$ *on* S.

The proof of Theorem 5.4.5 is analogous to that of Theorem 5.4.2 and thus omitted. We remind that this establishes the correspondence between the naïve characterization of auxiliary tiles and the categorical one. Similarly to the case of process tile logic, the axiomatization of proof sequents for term tile logic stated in Definition 5.3.8 makes $A_t(\mathcal{R})$ into a cartesian double category, defining a suitable initial model for term tile logic (see Proposition 6.4.1 and Theorem 6.5.1).

5.5 Summary

We have introduced the notion of *generalized natural transformation* among four double functors, showing that it extends the concept of natural transformation to the double category setting, in a more appropriate way than the internal approach does. Our definition is equivalent to the notion of *hypertransformation* [57] when restricted to the 2-fold case. Then, we have defined the notions of *symmetric strict monoidal double category* and of *cartesian double category* (*with consistently chosen products*) pursuing the analogy with the 1-dimensional case. The generalized natural transformations that we have considered have an interesting characterization in terms of the square-shaped commuting diagrams involving their corresponding 1-dimensional versions. We also remark that the *coherence axioms* can be lifted to our setting without much effort, thanks to the characterization of two suitable *diagonal categories*.

We have shown an interesting theoretical application of symmetric and cartesian double categories related to the *models of computation* area, and in particular to *tile logic*. Specifically, we have extended the model proposed by Gadducci and Montanari in [69] to deal with a more complex auxiliary structure belonging to both configurations and observations.

The categorical models developed here are very important to pursue the comparison with similar models for rewriting logic, as we will show in Chapter 6. In particular, we will define an executable rewriting implementation of process and term tile systems whose correctness relies on a control mechanism over the rewriting engine. In the third part of the thesis, we also show how this mechanism can be specified using *internal strategies* in Maude [34].

In the future, it could be interesting to extend the framework presented here to weaker categories, where some of the naturality axioms for the generalized natural transformations are missing. Other models should also be able to deal with co-duplicators and co-dischargers at the same time. Another possibility might be to extend our investigation to non-strict monoidal structures. Since in the 1-dimensional case the non-strictness is expressed by three natural transformations (isomorphisms), intuitively the lifting to the double case should just take into account suitable generalized natural transformations. Eventually, we are currently studying the possibility of adding higher-order features to the tile model, by taking cartesian closed categories for configurations and observations.

Chapter 6

Mapping Tile Logic into Rewriting Logic

Contents

6.1 Motivations

The evolution of an agent in a concurrent system often depends on the behaviours of other cooperating agents. For example, in some critical state, an agent must have the opportunity to check incoming communications from many sources, so to evolve consistently with other agents. Therefore, the synchronization feature of tile logic makes it more suitable than rewriting logic for the specification of reactive systems. On the other hand, whereas no language has been designed following the tile logic paradigm, several language implementation efforts in different countries (e.g., CafeObj [64], ELAN [13], Maude [35]) not only have rewriting logic as their semantic basis, but also support either executable specifications or parallel programming in rewriting logic. Besides, rewriting logic offers a framework where other logics can be suitably encoded and their deductions simulated in a natural way. Thus, a conservative mapping of tile logic into rewriting logic would facilitate the execution, testing and development of tile specifications.

In this chapter we investigate such connection in the case that both configurations and observations rely on common *auxiliary structures* (e.g., for rearranging interfaces), as it is the case of both process and term tile logic (see Chapter 5). The comparison between tile logic and rewriting logic takes place by embedding their categorical models (e.g., cartesian double categories for term tile logic and cartesian 2-categories for rewriting logic) in a recently developed framework called *partial membership equational logic* (**PMEqtl**) [104]. Indeed, the features of partial membership equational logic (partiality, subsorts, membership assertions) offer a natural framework for the specification of categorical structures, first because the sequential composition of arrows is a partial operation, and secondly because membership predicates over a poset of sorts allow objects to be modelled as a subset of the arrows, and arrows as a subset of the cells. Moreover, the *tensor product construction* illustrated in [107] can be easily formulated in **PMEqtl**, and yields a concise definition of monoidal double categories.

Using **PMEqtl**, it is possible to define an extended version of 2-categories, where also double categories can be conveniently embedded. As illustrated in the Introduction, the idea is to stretch the tiles into rewrite rules (see Figure 1.10) still maintaining the possibility to distinguish horizontal arrows from vertical arrows. This approach was first proposed by Meseguer and Montanari, presenting an extended version of 2-categories, called 2VH-categories, able to include appropriately the structure of double categories [107]. However, their model is not suitable for dealing with some common arrows between the horizontal and the vertical dimension (representing the auxiliary structure).

We introduce an extended version of 2VH-categories, able to include the structure of symmetric monoidal and cartesian double categories (and more generally, double categories with shared structure) by making explicit what the auxiliary common structure is (thus avoiding the numerous axioms presented in Section 5.4.2, that can be recovered by simpler ones expressed in the extended model).

The main result is that 2EVH-categories correspond to an extended rewriting logic where deduction in tile logic can be simulated.

From a practical viewpoint, the mapping from tile logic into rewriting logic becomes effective, provided that "the rewriting engine is able to control rewritings." How this can be obtained is explained in Chapter 7.

6.1.1 Structure of the Chapter

In Section 6.2.1 we introduce partial membership equational logic and the construction of the tensor product of theories in **PMEqtl**. Several examples illustrate the Maude-like presentation style adopted here for the definition of theories in **PMEqtl**.

In Section 6.2.2 we recall the theory of 2VH-categories, proposed by Meseguer and Montanari, and show that it cannot be used to model auxiliary structures shared between the horizontal and the vertical dimension.

The extended model theory of 2EVH-categories is illustrated in Section 6.3, where also the monoidal, symmetric and cartesian extensions are fully discussed.

Section 6.4 presents the notion of computads, which is used in Section 6.5 to relate tile systems and models of theories in **PMEqtl**.

Section 6.5 establishes the correspondence between term tile logic and the extended logic associated to 2EVH-categories.

6.2 Background

6.2.1 Partial Membership Equational Logic

In this section we summarize the basics of *partial membership equational logic* (**PMEqtl**) [104]. This is a logic of partial algebras with subsorts and subsort polymorphism whose sentences are Horn clauses on equations $t = t'$ and membership assertions $t : s$.

We treat here the *one kinded* case, where the poset of sorts has a single connected component. A more detailed exposition for the many-kinded case can be found in [104].

6.2.1.1 Partial Algebras and Membership Equational Theories

Definition 6.2.1 (Partially Ordered Signature)
A partially ordered signature *(*po-signature*) is a triple* $\Omega = (S, \leq, \Sigma)$*, where*

- $(S, \leq)$ *is a poset with a top element* $\top$*, and*
- $\Sigma = \{\Sigma_k\}_{k \in \mathbb{N}}$ *is a family of sets of operators, indexed by natural numbers.*

The poset $(S, \leq)$ is called the *poset of sorts* of Ω.

DEFINITION 6.2.2 (PARTIAL Ω-ALGEBRA)
Given a po-signature $\Omega = (S, \leq, \Sigma)$, a partial Ω-algebra A *assigns:*

1. *to each* $s \in S$ *a set* A_s, *in such a way that whenever* $s \leq s'$, *we have* $A_s \subseteq A_{s'}$;
2. *to each* $f \in \Sigma_k$, $k \geq 0$, *a partial function* $A_f : A_\top^k \multimap A_\top$.

Given two partial Ω*-algebras* A *and* B, *an* Ω-homomorphism *from* A *to* B *is a (total) function* $h : A_\top \longrightarrow B_\top$ *such that:*

1. *for each* $s \in S$, $h(A_s) \subseteq B_s$ *(hence, for each* $s \in S$ *the function* h *restricts to a function* $h|_s : A_s \longrightarrow B_s$*);*
2. *for each* $f \in \Sigma_k$, $k \geq 0$, *and* $\vec{a} \in A_\top^k$, *if* $A_f(\vec{a})$ *is defined, then* $B_f(h^k(\vec{a}))$ *is also defined and equal to* $h(A_f(\vec{a}))$.

This determines a category $\mathbf{PAlg}_\Omega$.

DEFINITION 6.2.3 (DECLARATION, FORMULA AND SENTENCE)
Let $\Omega = (S, \leq, \Sigma)$ *be a po-signature. Given a set of variables* $X = \{x_1, \ldots, x_m\}$, a variable declaration $\tilde{X}$ *is a sequence* $x_1 : \overline{s_1}, \ldots, x_m : \overline{s_m}$, *where for each* $i = 1, \ldots, m$, $\overline{s_i}$ *is a set of sorts* $\{s_{i1}, \ldots, s_{ik_i}\}$.

Atomic Ω-formulas *are either equations* $t = t'$, *where* $t, t' \in T_\Sigma(X)$ *(with* $T_\Sigma(X)$ *the usual free* Σ*-algebra on variables* X*), or membership assertions of the form* $t : s$, *with* $t \in T_\Sigma(X)$ *and* $s \in S$.

General Ω-sentences *are then Horn clauses of either one of the two forms below*

$$\forall \tilde{X} \quad t = t' \;\Leftarrow\; u_1 = v_1 \wedge \ldots \wedge u_n = v_n \wedge w_1 : s_1 \wedge \ldots \wedge w_m : s_m$$
$$\forall \tilde{X} \quad t : s \;\Leftarrow\; u_1 = v_1 \wedge \ldots \wedge u_n = v_n \wedge w_1 : s_1 \wedge \ldots \wedge w_m : s_m$$

where the t, t', u_i, v_i *and* w_j *are all terms in* $T_\Sigma(X)$.

DEFINITION 6.2.4 (THEORY, MODEL)
Given a partial Ω*-algebra* A *and a variable declaration* $\tilde{X}$, *we can define assignments* $a : \tilde{X} \longrightarrow A$ *in the obvious way (if* $x : \overline{s}$ *and* $s \in \overline{s}$, *then we must have* $a(x) \in A_s$*) and then we can define a partial function* $\overline{a} : T_\Sigma(X) \multimap A_\top$, *extending* a *in the obvious way. For atomic sentences we define satisfaction by*

$$A, a \models t = t'$$

meaning that $\overline{a}(t)$ *and* $\overline{a}(t')$ *are both defined and* $\overline{a}(t) = \overline{a}(t')$ *(that is, we take an* existence equation *interpretation) and by*

$$A, a \models t : s$$

meaning that $\overline{a}(t)$ *is defined and* $\overline{a}(t) \in A_s$.

Satisfaction of Horn clauses is then defined in the obvious way. Given a set ? *of* Ω*-sentences, we then define* $\mathbf{PAlg}_{\Omega,\Gamma}$ *as the full subcategory of* $\mathbf{PAlg}_\Omega$ *determined by those partial* Ω*-algebras that satisfy all the sentences in* ?. *In other words, the pair* $T = (\Omega, ?)$ *is a* theory, *and* $\mathbf{PAlg}_T = \mathbf{PAlg}_{\Omega,\Gamma}$ *is the category of its models.*

As an example, we recall the definition of the theory of categories from [107]. The theory is presented in a self-explanatory Maude-like notation [35], which will be used extensively in the rest of the presentation. We refer the interested reader to Appendix D for a more detailed introduction to the syntax of Maude (and to the abbreviations we adopt).

In the following, we will denote theories either by their Maude name (e.g., `CAT`), or by their indexed notation (e.g., $T_{\texttt{CAT}}$). Moreover, we will denote the associated category of models $\mathbf{PAlg}_{\texttt{CAT}}$ by the simpler "boldfaced" notation $\mathbf{Cat}$.

REMARK 6.2.1 *To shorten the notation, we will write*

```
cmb f;g : Arrow iff c(f) = d(g) .
```

as a shorthand for

```
cmb f;g : Arrow if c(f) == d(g) .
ceq c(f) = d(g) if f;g : Arrow .
```

This and most of the other shorthands that we use are summarized in Appendix D.

EXAMPLE 6.2.2 (CATEGORY) *The theory of categories $T_{\texttt{CAT}}$ is a theory in* **PME-qtl**. *Its poset of sorts is* $\{\texttt{Object}, \texttt{Arrow}\}$ *with* $\texttt{Object} \leq \texttt{Arrow}$. *There are two unary* domain *and* codomain *operations* `d(_)` *and* `c(_)`, *and a binary composition operation* `_;_`. *The composition operator is defined if and only if the codomain of the first argument is equal to the domain of the second argument. As usual in many presentations, here objects are identified with the corresponding identity arrows. Then, the theory of categories can be defined as in Table 6.1. It is easy to check that a model of* `CAT` *is exactly a category, and that a* `CAT`*-homomorphism is exactly a functor.*

DEFINITION 6.2.5 (SIGNATURE AND THEORY MORPHISM)
Given two po-signatures $\Omega = (S, \leq, \Sigma)$ and $\Omega' = (S', \leq', \Sigma')$, a signature morphism *$H : \Omega \longrightarrow \Omega'$ is given by: a monotonic function $H : (S, \leq) \longrightarrow (S', \leq')$, and an $\mathbb{N}$-indexed family of functions $\{H_k : \Sigma_k \longrightarrow \Sigma'_k\}_{k \in \mathbb{N}}$.*
Such a signature morphism induces a forgetful functor $U_H : \mathbf{PAlg}_{\Omega'} \longrightarrow \mathbf{PAlg}_{\Omega}$, where for each $A' \in \mathbf{PAlg}_{\Omega'}$ we have:

1. *for each $s \in S$, $U_H(A')_s = A'_{H(s)}$;*
2. *for each[1] $f \in \Sigma_k$, $U_H(A')_f = A'_{H(f)} \cap (\underbrace{A'_{H(\top)} \times \ldots \times A'_{H(\top)}}_{k} \times A'_{H(\top)})$;*
3. *for each Ω'-homomorphism $h' : A' \to B'$, $U_H(h') = h'|_{H(\top)} : A'_{H(\top)} \to B'_{H(\top)}$, which is well-defined as a restriction of h' because h' is sort-preserving.*

[1] Notice that $U_H(A')_f = A'_{H(f)}$ would not be correct in general, since $U_H(A')_\top = A'_{H(\top)}$, where $H(\top)$ is not necessarily $\top$.

```
fth CAT is
  sorts Object Arrow .
  subsort Object < Arrow .
  ops d(_) c(_) _;_ .
  vars f g h : Arrow .
  var  a : Object .
  mbs d(f) c(f) : Object .
  eq  d(a) = a .
  eq  c(a) = a .
  cmb f;g : Arrow iff c(f) = d(g) .
  ceq d(f;g) = d(f) if c(f) = d(g) .
  ceq c(f;g) = c(g) if c(f) = d(g) .
  ceq a;f = f if d(f) = a .
  ceq f;a = f if c(f) = a .
  ceq (f;g);h = f;(g;h) if c(f) = d(g) and c(g) = d(h) .
endfth
```

Table 6.1: The theory of categories in **PMEqtl**.

Given theories $(\Omega, ?)$ *and* $(\Omega', ?')$, a theory morphism $H : (\Omega, ?) \longrightarrow (\Omega', ?')$ *is a signature morphism* $H : \Omega \longrightarrow \Omega'$ *such that* $U_H(\mathbf{PAlg}_{\Omega',\Gamma'}) \subseteq \mathbf{PAlg}_{\Omega,\Gamma}$, *so that* U_H *restricts to a forgetful functor* $U_H : \mathbf{PAlg}_{\Omega',\Gamma'} \longrightarrow \mathbf{PAlg}_{\Omega,\Gamma}$.

We refer to [104] for proof-theoretical conditions on $?$ and $?'$ ensuring that a signature morphism $H : \Omega \longrightarrow \Omega'$ defines a theory morphism $H : (\Omega, ?) \longrightarrow (\Omega', ?')$.

Proposition 6.2.1 (Free Construction from Theory Morphism)
Given a theory morphism $H : (\Omega, ?) \longrightarrow (\Omega', ?')$, *the associated forgetful functor* $U_H : \mathbf{PAlg}_{\Omega',\Gamma'} \longrightarrow \mathbf{PAlg}_{\Omega,\Gamma}$ *has a left adjoint* $F_H : \mathbf{PAlg}_{\Omega,\Gamma} \longrightarrow \mathbf{PAlg}_{\Omega',\Gamma'}$, *i.e.,*

$$(\Omega, ?) \xrightarrow{H} (\Omega', ?') \qquad \text{yields} \qquad \mathbf{PAlg}_{\Omega,\Gamma} \underset{U_H}{\overset{F_H}{\rightleftarrows}} \mathbf{PAlg}_{\Omega',\Gamma'}$$

Definition 6.2.6 (Conservative, complete and persistent morphism)
A theory morphism $H : (\Omega, ?) \longrightarrow (\Omega', ?')$ *is* conservative *(respectively* complete, persistent*) with respect to sort* s *if, for each algebra* $A \in \mathbf{PAlg}_{\Omega,\Gamma}$, *the component*

$$(\eta_A)_s : A_s \longrightarrow (U_H(F_H(A))_s$$

corresponding to s *of the unit of the adjunction associated to* H *is injective (respectively surjective, bijective).*

The morphism H *is* conservative *(respectively* complete, persistent*) if it is conservative (respectively complete, persistent) with respect to all* $s \in S$.

DEFINITION 6.2.7 (SUBALGEBRA)
Given a po-signature $\Omega = (S, \leq, \Sigma)$ and a partial Ω-algebra A, an Ω-subalgebra *B of A is an S-sorted family of subsets $\{B_s \subseteq A_s\}_{s \in S}$ such that:*

1. *it is closed under the operations of Σ, that is, for each $f \in \Sigma_k$, and for each $\vec{b} \in B^k_\top$, if $A_f(\vec{b})$ is defined, then $A_f(\vec{b}) \in B_\top$;*

2. *it is closed under subsorts, in the sense that for each sort $s \in S$ we have $B_s = A_s \cap B_\top$.*

It is clear that B with such operations and sorts is an Ω-algebra itself, and that the inclusion function $B \subseteq A$ is an Ω-homomorphism.

LEMMA 6.2.2
For any set ? of Horn sentences in partial membership equational logic, the category $\mathbf{PAlg}_{\Omega,\Gamma}$ is closed under Ω-subalgebras, i.e., if $A \in \mathbf{PAlg}_{\Omega,\Gamma}$ and B is an Ω-subalgebra of A, then $B \in \mathbf{PAlg}_{\Omega,\Gamma}$.

EXAMPLE 6.2.3 (SUBCATEGORY) *For the theory* CAT *of Example 6.2.2, the subalgebras of a category $\mathcal{C}$ are exactly its subcategories.*

The notion of Ω-subalgebra is strictly stronger than that of Ω-monomorphism. It is easy to check that, in the category $\mathbf{PAlg}_\Omega$, an arrow $m : C \longrightarrow A$ is a monomorphism if and only if the associated function $m : C_\top \longrightarrow A_\top$ is injective. Of course, by taking the smallest image of m, any such monomorphism always factorizes through an isomorphism and an inclusion $C \xrightarrow{\sim} B \hookrightarrow A$, where B is a *weak* subalgebra, as defined below.

DEFINITION 6.2.8 (WEAK SUBALGEBRA)
Given a signature $\Omega = (S, \leq, \Sigma)$ and a partial Ω-algebra A, a weak Ω-subalgebra *of A is a partial Ω-algebra B such that $B_\top \subseteq A_\top$ and such that the inclusion map $B \hookrightarrow A$ is an Ω-homomorphism.*

In general, given a set ? of Horn sentences in **PMEqtl** and a partial algebra $A \in \mathbf{PAlg}_{\Omega,\Gamma}$, a weak subalgebra B of A needs not satisfy the sentences ?. For example, given a nonempty category $\mathcal{C}$, the weak subalgebra with same arrows and objects as $\mathcal{C}$, but with all operations everywhere undefined is not a category. However, for $(\Omega, ?) =$ CAT, the following relationship happens to hold between subalgebras and weak subalgebras.

EXAMPLE 6.2.4 *Given a category $\mathcal{C}$, if $\mathcal{D} \subseteq \mathcal{C}$ is a weak subalgebra and $\mathcal{D}$ itself is a category, then $\mathcal{D} \subseteq \mathcal{C}$ is a subalgebra, that is, a subcategory.*

6.2.1.2 The Tensor Product Construction

Given a signature $\Omega = (S, \leq, \Sigma)$ and a category $\mathcal{C}$ with finite limits and with a suitable poset of canonical inclusions $\mathcal{I}$, we can define the category of partial Ω-algebras in $\mathcal{C}$, denoted by $\mathbf{PAlg}_{\Omega}(\mathcal{C})$ and $\mathbf{PAlg}_{\Omega,\Gamma}(\mathcal{C})$ (see [107]). It is evident that categories $\mathbf{PAlg}_{\Omega}$ and $\mathbf{PAlg}_{\Omega,\Gamma}$ are just the special case $\mathbf{PAlg}_{\Omega}(\mathbf{Set})$ and $\mathbf{PAlg}_{\Omega,\Gamma}(\mathbf{Set})$. Noticing that $\mathbf{PAlg}_{\Omega}$ and $\mathbf{PAlg}_{\Omega,\Gamma}$ are categories with limits, and that Ω-subalgebra inclusions $A \subseteq B$ constitute a poset category of canonical inclusions, it is possible to define the category $\mathbf{PAlg}_T(\mathbf{PAlg}_{T'})$ for any two theories $T = (\Omega, ?)$ and $T' = (\Omega', ?')$. Moreover, in a way analogous to algebraic theories [92, 62], to lim theories [65], and to sketches [89], we can define the construction of a *tensor theory* $T \otimes T'$ in partial membership equational logic such that

$$\mathbf{PAlg}_{T \otimes T'} \simeq \mathbf{PAlg}_T(\mathbf{PAlg}_{T'}) \simeq \mathbf{PAlg}_{T'}(\mathbf{PAlg}_T).$$

Notice that we could have chosen a bigger poset of subalgebra inclusions, yielding a looser definition of $\mathbf{PAlg}_T(\mathbf{PAlg}_{T'})$. A natural choice would have been the set of weak subalgebra inclusions. This would yield a notion of tensor product of theories equivalent to the tensor product of their corresponding sketches. However, as we have already pointed out, the notion of weak subalgebra is too loose, giving rise in general to somewhat unintuitive models; for this reason we favour instead the notion of tensor product associated to subalgebras. Nevertheless, in the special case $T' = \mathtt{CAT}$, because of the property mentioned in Example 6.2.4, the definition of $\mathbf{PAlg}_T(\mathbf{PAlg}_{\mathtt{CAT}})$ is the same whether we choose subalgebras or instead weak subalgebras as canonical inclusions in $\mathbf{PAlg}_{\mathtt{CAT}}$.

The explicit definition of $T \otimes T'$, introduced in [107], is as follows.

Definition 6.2.9 (Tensor Product of Theories)
Let $T = (\Omega, ?)$ and $T' = (\Omega', ?')$ be theories in partial membership equational logic, with $\Omega = (S, \leq, \Sigma)$ and $\Omega' = (S', \leq', \Sigma')$. Then their tensor product *$T \otimes T'$ is the theory with signature $\Omega \otimes \Omega'$ having:*

1. *poset of sorts $(S, \leq) \times (S', \leq')$;*
2. *signature $\Sigma \otimes \Sigma'$, with an operator[2] $f^l \in (\Sigma \otimes \Sigma')_n$ for each $f \in \Sigma_n$, and with an operator $g^r \in (\Sigma \otimes \Sigma')_m$ for each $g \in \Sigma'_m$. In particular, for f a constant in Σ_0 we get a constant f^l in $(\Sigma \otimes \Sigma')_0$.*

The axioms of $T \otimes T'$ are the following:

A. Inherited Axioms.

For each axiom α in $?$ having the form $\forall(x_1 : \overline{s_1}, \ldots, x_m : \overline{s_m})\ \ \varphi(\vec{x}) \Leftarrow c(\vec{x})$ with $\overline{s_i} = \{s_{i1}, \ldots, s_{il_i}\}$, for $1 \leq i \leq m$, we introduce an axiom

$$\alpha^l \ = \ \forall(x_1 : \overline{s_1^l}, \ldots, x_m : \overline{s_m^l})\ \ \varphi^l(\vec{x}) \Leftarrow c^l(\vec{x})$$

[2] Here, superscripts l and r of operators stand respectively for *left* and *right*.

with $\overline{s_i^l} = \{(s_{i1}, \top'), \ldots, (s_{il_i}, \top')\}$, *for* $1 \leq i \leq m$, *and with* φ^l, c^l *the obvious translations of* φ, c *obtained by replacing each* $f \in \Sigma$ *by its corresponding* f^l. *Similarly, we define for each axiom* $\beta \in ?'$ *the axiom* β^r *and impose all these axioms.*

B. Subalgebra Axioms.

For each $f \in \Sigma_n$ *and each* $s' \in S', s' \neq \top'$, *we introduce the axiom:*

$$\forall (x_1 : (\top, s'), \ldots, x_n : (\top, s')) \\ f^l(x_1, \ldots, x_n) : (\top, s') \Leftarrow f^l(x_1, \ldots, x_n) : (\top, \top').$$

For each $g \in \Sigma'_m$ *and each* $s \in S, s \neq \top$, *we introduce the axiom:*

$$\forall (x_1 : (s, \top'), \ldots, x_m : (s, \top')) \\ g^r(x_1, \ldots, x_m) : (s, \top') \Leftarrow g^r(x_1, \ldots, x_m) : (\top, \top').$$

For each $(s, s') \in S \times S'$ *with* $s \neq \top$ *and* $s' \neq \top'$, *we have the axiom:*

$$\forall x : (\top, \top') \;\; x : (s, s') \Leftarrow x : (\top, s') \;\wedge\; x : (s, \top').$$

C. Homomorphism Axioms.

For each $f \in \Sigma_n, g \in \Sigma'_m, n + m \geq 0$, *we introduce the axiom:*

$$\forall \vec{x} \;\; f^l(g^r(\vec{x}_{1\_}), \ldots, g^r(\vec{x}_{n\_})) = g^r(f^l(\vec{x}_{\_1}), \ldots, f^l(\vec{x}_{\_m})) \Leftarrow \\ \Leftarrow \textstyle\bigwedge_{1 \leq i \leq n} g^r(\vec{x}_{i\_}) : (\top, \top') \;\wedge\; \bigwedge_{1 \leq j \leq m} f^l(\vec{x}_{\_j}) : (\top, \top').$$

where

$$\begin{aligned} \vec{x} &= \{x_{ij} : (\top, \top')\}_{1 \leq j \leq m}^{1 \leq i \leq n}, \\ \vec{x}_{i\_} &= \{x_{ij} : (\top, \top')\}_{1 \leq j \leq m}, \quad 1 \leq i \leq n \\ \vec{x}_{\_j} &= \{x_{ij} : (\top, \top')\}_{1 \leq i \leq n}, \quad 1 \leq j \leq m. \end{aligned}$$

The essential property of $T \otimes T'$ is expressed by the following theorem.

Theorem 6.2.3 (Models of the Tensor Product)
Let T, T' *be theories in partial membership equational logic. Then we have the following isomorphisms of categories:*

$$\mathbf{PAlg}_{T \otimes T'} \simeq \mathbf{PAlg}_T(\mathbf{PAlg}_{T'}) \simeq \mathbf{PAlg}_{T'}(\mathbf{PAlg}_T).$$

A useful property of the tensor product of theories is its *functoriality* in the category of theories. Therefore, if $H : T_1 \longrightarrow T_2$ and $G : T_1' \longrightarrow T_2'$ are theory morphisms, we have an associated theory morphism:

$$H \otimes G : T_1 \otimes T_1' \longrightarrow T_2 \otimes T_2'.$$

It can be shown that the tensor product of theories is associative and commutative up to isomorphism, that is, we have natural isomorphisms of theories $T \otimes T' \simeq T' \otimes T$ and $T \otimes (T' \otimes T'') \simeq (T \otimes T') \otimes T''$ giving a symmetric monoidal category structure to the category of theories.

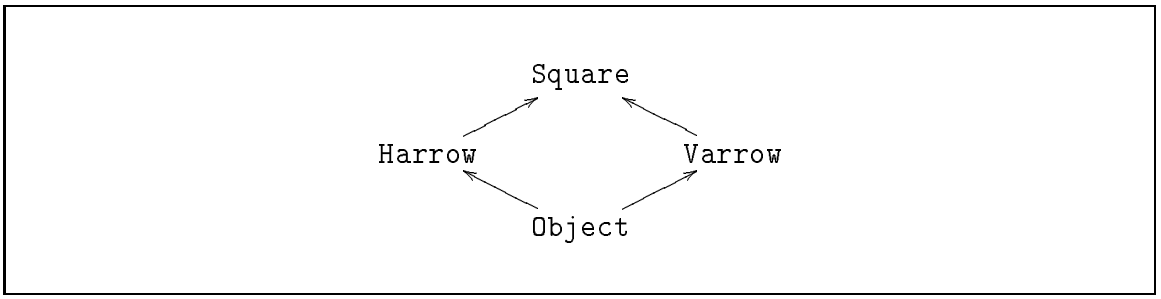


Figure 6.1: The poset of sorts for $T_{\mathtt{DCAT}}$.

EXAMPLE 6.2.5 (DOUBLE CATEGORY) *A* double category *is a category structure on* **Cat**, *that is, an object of* $\mathbf{PAlg}_{\mathtt{CAT}}(\mathbf{PAlg}_{\mathtt{CAT}}) = \mathbf{DCat}$. *The theory* $\mathtt{CAT} \otimes \mathtt{CAT}$ *then axiomatizes double categories in partial membership equational logic. Spelling out the specification of* $T \otimes T'$ *for the case of* $T = T' = \mathtt{CAT}$ *we get the following poset of sorts, where* Square *is the top (see Figure 6.1):*

$$\begin{array}{ll} (\mathtt{Object},\mathtt{Object}) = \mathtt{Object}, & (\mathtt{Arrow},\mathtt{Arrow}) = \mathtt{Square}, \\ (\mathtt{Arrow},\mathtt{Object}) = \mathtt{Harrow}, & (\mathtt{Object},\mathtt{Arrow}) = \mathtt{Varrow}, \\ \mathtt{Object} \leq \mathtt{Harrow} \leq \mathtt{Square}, & \mathtt{Object} \leq \mathtt{Varrow} \leq \mathtt{Square}. \end{array}$$

For the source and target operations in $\mathtt{CAT} \otimes \mathtt{CAT}$ *we adopt the intuitive "north, east, west, south" notation:*

$$\mathtt{d}^l(\_) = \mathtt{w}(\_), \quad \mathtt{c}^l(\_) = \mathtt{e}(\_), \quad (\_;\_)^l = \_*\_,$$
$$\mathtt{d}^r(\_) = \mathtt{n}(\_), \quad \mathtt{c}^r(\_) = \mathtt{s}(\_), \quad (\_;\_)^r = \_\cdot\_.$$

The presentation of double categories in Maude-like notation is given in Table 6.2. Notice that we do not present the literal instances of the axioms, but equivalent forms. For example, we get

```
w(h) : Object .
```

from `w(h): Varrow` *(by inherited axioms), plus* `w(h): Harrow` *(by the subalgebra axiom properly speaking), plus the subalgebra axiom forcing*

$$\mathtt{Harrow} \cap \mathtt{Varrow} = \mathtt{Object}.$$

In the following, we enrich our Maude-like notation with the tensor product construction. The presentation of double categories thus becomes much simpler:

```
fth DCAT is CAT ⊗ CAT renamed by (
  sorts (Object,Object) to Object . (Arrow,Arrow) to Square .
        (Arrow,Object) to Harrow . (Object,Arrow) to Varrow .
  ops d(_) left to w(_) . d(_) right to n(_) .
      c(_) left to e(_) . c(_) right to s(_) .
      _;_ left to _*_ . _;_ right to _·_ ) .
endfth
```

```
fth DCAT is
  sorts Object Harrow Varrow Square .
  subsorts Object < Harrow Varrow < Square .
  ops  n(_) e(_) w(_) s(_) _*_ _·_ .
  vars f h : Harrow .
       u v : Varrow .
       A B C D : Square .
  *** Inherited Axioms: Horizontal
  mbs w(A) e(A) : Varrow .
  eq  w(v) = v .   eq  e(v) = v .
  cmb A*B : Square iff e(A) = w(B) .
  ceq w(A*B) = w(A) if e(A) = w(B) .
  ceq e(A*B) = e(B) if e(A) = w(B) .
  ceq v*A = A if w(A) = v .
  ceq A*v = v if e(A) = v .
  ceq (A*B)*C = A*(B*C) if e(A) = w(B) and e(B) = w(C) .
  *** Inherited Axioms: Vertical
  mbs n(A) s(A) : Harrow .
  eq  n(h) = h .   eq  s(h) = h .
  cmb A·B : Square iff s(A) = n(B) .
  ceq n(A·B) = n(A) if s(A) = n(B) .
  ceq s(A·B) = s(B) if s(A) = n(B) .
  ceq h·A = A if n(A) = h .
  ceq A·h = h if s(A) = h .
  ceq (A·B)·C = A·(B·C) if s(A) = n(B) and s(B) = n(C) .
  *** Subalgebra Axioms
  cmb A : Object if A : Harrow and A : Varrow .
  mbs w(h) e(h) n(v) s(v) : Object .
  mb  f*h : Harrow .   mb  u·v : Varrow .
  *** Homomorphism Axioms
  eq  n(w(A)) = w(n(A)) .   eq  n(e(A)) = e(n(A)) .
  eq  s(w(A)) = w(s(A)) .   eq  s(e(A)) = e(s(A)) .
  ceq w(A·B) = w(A)·w(B) if s(A) = n(B) .
  ceq e(A·B) = e(A)·e(B) if s(A) = n(B) .
  ceq n(A*B) = n(A)*n(B) if e(A) = w(B) .
  ceq s(A*B) = s(A)*s(B) if e(A) = w(B) .
  ceq (A*B)·(C*D) = (A·C)*(B·D)
      if e(A) = w(B) and e(C) = w(D) and
         s(A) = n(C) and s(B) = n(D) .
endfth
```

Table 6.2: The theory of double categories in **PMEqtl**.

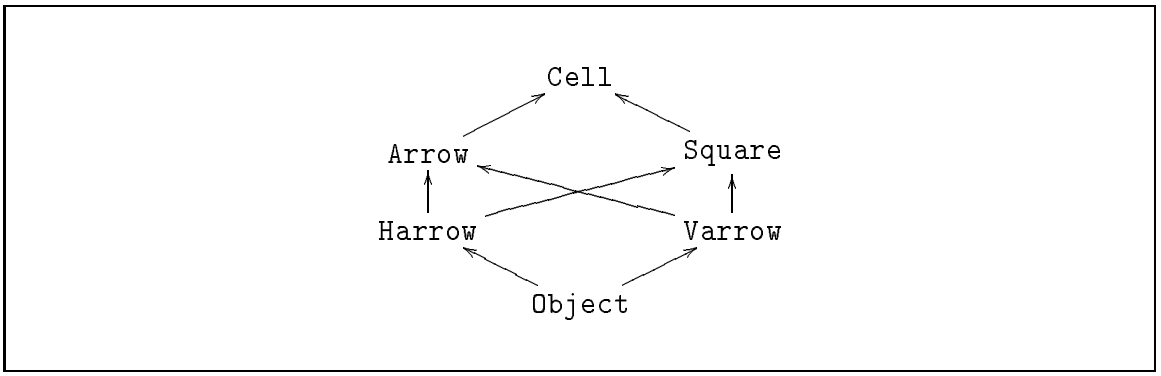


Figure 6.2: The poset of sorts for $T_{\texttt{2VHCAT}}$.

2-categories [82] (see also Section 2.1.7) are probably the best known kind of enriched category. In particular, they yield models of rewriting logic in a very natural way [98]. It should be clear that they can be considered as the special case of double categories whose vertical arrows coincide with objects. In 2-categories, squares are called cells, and horizontal arrows are called arrows. Moreover, the north and south source and target of a cell A are denoted by `d(`A`)` and `c(`A`)`, while the west and east source and target become `l(`A`)` and `r(`A`)`. Also, horizontal composition is denoted `_;_` and vertical composition is denoted `_o_`. The explicit Maude-like definition of 2-categories [107] will be useful in the following.

```
fth 2CAT is including DCAT renamed by (
  sorts Square to Cell . Harrow to Arrow . Varrow to Object .
  ops w(_) to l(_) . e(_) to r(_) .
      n(_) to d(_) . s(_) to c(_) .
      _*_ to _;_ .   _·_ to _o_ ) .
endfth
```

6.2.2 2VH-Categories

The extended version of a 2-category proposed in [107], called 2VH-category, includes the double category structure and has the poset of sorts shown in Figure 6.2. The idea is that the theory `2CAT` is imported in `2VHCAT` as such, without any renaming. In addition, new sorts `Harrow`, `Varrow` and `Square` are introduced, which correspond to the homonymous sorts of double categories. The basic intuition is that, if we are given a 2-category with subcategories `Harrow` and `Varrow` of `Arrow`, such that they are disjoint except for objects, and such that the horizontal and vertical components can be recovered from their composition, then we can form a double category by considering squares with horizontal and vertical sides, and we can define their horizontal and vertical composition by using the already existing cell composition of the 2-category.

Table 6.3 illustrates the complete description of the theory $T_{\texttt{2VHCAT}}$ as originally given by Meseguer and Montanari in [107].

```
fth 2VHCAT is including 2CAT
  sorts Harrow Varrow Square .
  subsorts Object < Harrow Varrow < Arrow Square < Cell .
  ops n(_) e(_) w(_) s(_) _*_ _·_ .
  vars f h : Harrow .
       u v : Varrow .
       t : Arrow .
       p q : Square .
       A B : Cell .
  cmb t : Object  if  t : Harrow and t : Varrow .
  cmb n(A) : Cell iff A : Square and n(A) : Harrow .
  cmb e(A) : Cell iff A : Square and e(A) : Varrow .
  cmb w(A) : Cell iff A : Square and w(A) : Varrow .
  cmb s(A) : Cell iff A : Square and s(A) : Harrow .
  ceq n(q) = h if d(q) = h;v .
  ceq e(q) = v if d(q) = h;v .
  ceq w(q) = v if c(q) = v;h .
  ceq s(q) = h if c(q) = v;h .
  eq  d(q) = n(q);e(q) .
  eq  c(q) = w(q);s(q) .
  cmb f;h : Harrow  iff r(f) = l(h) .
  cmb u;v : Varrow  iff r(u) = l(v) .
  cmb p*q : Square iff e(p) = w(q) .
  cmb A*B : Cell
      iff A : Square and  B : Square and e(A) = w(B) .
  cmb p·q : Square iff s(p) = n(q) .
  cmb A·B : Cell
      iff A : Square and B : Square and s(A) = n(B) .
  ceq p*q = (n(p);q)∘(p;s(q))  if  e(p) = w(q) .
  ceq p·q = (p;e(q))∘(w(p);q)  if  s(p) = n(q) .
endfth
```

Table 6.3: The theory of 2VH-categories.

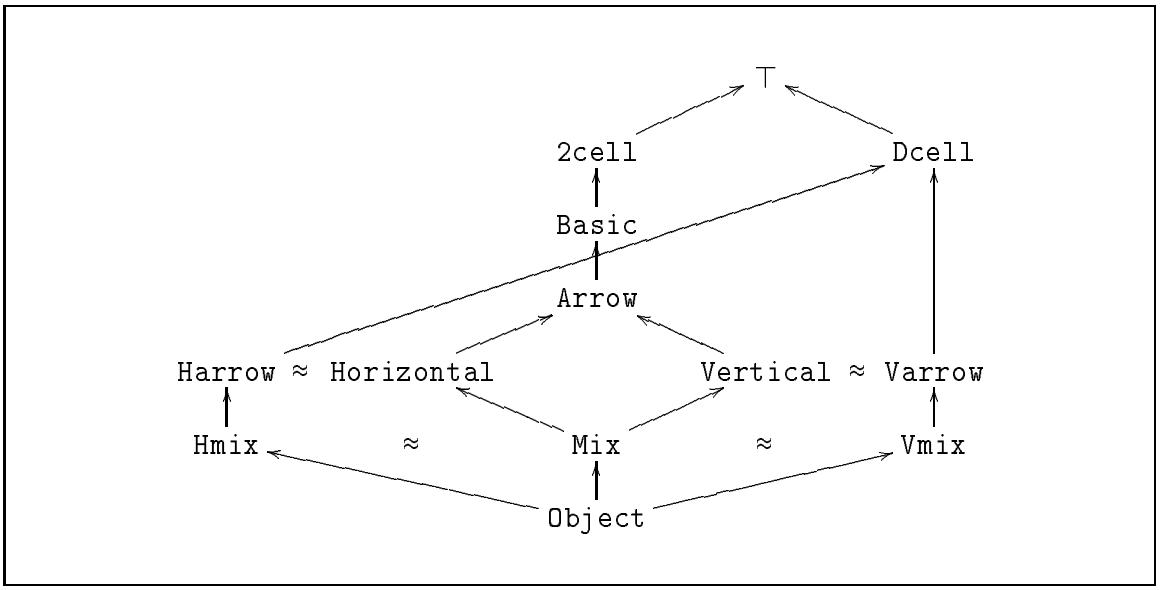


Figure 6.3: The poset of sorts for $T_{\texttt{2EVHCAT}}$.

We focus on a particular axiom of the specification, namely

```
var t : Arrow .
cmb t : Object  if  t : Harrow and t : Varrow .
```

As a consequence, no shared structure between the horizontal and vertical dimensions can be defined, except for objects. In the cases of symmetric double categories and cartesian double categories this is clearly too restrictive. We are therefore led to the definition of a more flexible theory of 2EVH-categories, where the problem is elegantly solved. Then, analogous results to those presented in [107] can be proved in the extended setting.

6.3 Extended 2VH-Categories

Basically, we could think of auxiliary constructors (i.e., symmetries, duplicators, and dischargers) as structure shared between the horizontal and vertical categories. In this sense it would be natural to introduce a subsort of both `Harrow` and `Varrow` containing the auxiliary constructors (and the objects). However this solution contradicts the subalgebra axiom stating that the only arrows which are both horizontal and vertical are the identities. The fact is that we need to represent two disjoint copies of auxiliary constructors, one for each dimension considered.

The poset of sorts that we propose is shown in Figure 6.3. The sort `Mix` includes the auxiliary structure which is thus shared between the sorts `Horizontal` and `Vertical`. The sort `Arrow` is the union of the two. The sorts `Harrow` and `Hmix` (respectively `Varrow` and `Vmix`) are isomorphic copies of sorts `Horizontal` and `Mix` (respectively `Vertical` and `Mix`).

The representation of the poset given in Figure 6.3 suggests us to call *internal* the sorts `Mix`, `Horizontal`, and `Vertical`, and to call *lateral* or *external* their isomorphic copies `Harrow`, `Varrow`, `Hmix`, and `Vmix`. The sort `Basic` contains identity cells and possibly the cells of some *2-computads* (see Section 6.4). The intuition is that, if we are given a cartesian 2-category such that the subcategories `Horizontal` and `Vertical` of `Arrow` are disjoint, except for objects and auxiliary arrows, then we can construct a cartesian double category by considering double cells whose horizontal and vertical sides are isomorphic to the two partitions of `Arrow`. Moreover, it is possible to define their horizontal and vertical composition in terms of the existing cell composition of 2-categories.

Since the sorts `Horizontal` and `Vertical` share `Mix` arrows, it follows that more than one double cell can correspond to the same 2-cell representation, namely, when they differ only in the way the source and target arrows of the cell are decomposed into the composition of `Horizontal` and `Vertical` arrows. Moreover there are some cells that do not generate any double cell. Thus, it is possible to define a total mapping from `Dcell` onto `2cell`, but in general this mapping is neither injective, nor surjective. However, the isomorphisms between lateral copies and internal sorts easily follow from the definition of the more general mapping.

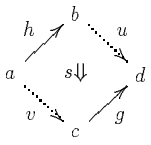
We present the Maude-like definition of the theory $T_{\texttt{2EVHCAT}}$, alternating the source code with some explanations and examples. We start by giving the formal translation of Figure 6.3 and fixing the variable notation for each subsort.

```
fth 2EVHCAT is
  including 2CAT renamed by (sort Cell to 2cell) .
  sorts Mix Horizontal Vertical Basic ⊤ .
  sorts Hmix Harrow Vmix Varrow Dcell .
  subsorts Object < Hmix Mix Vmix .
           Mix < Horizontal Vertical < Arrow < Basic < 2cell .
           Hmix < Harrow . Vmix < Varrow .
           Harrow Varrow < Dcell .
           2cell Dcell < ⊤ .
  vars a : Object .
       m : Mix .
       h h' h'' g g' g'' f : Horizontal .
       v v' v'' u u' u'' w : Vertical .
       t t' : Arrow .
       s : Basic .
       hm : Hmix .
       vm : Vmix .
       ha : Harrow .
       va : Varrow .
       p : Dcell .
       l l' l'' : 2cell .
```

The sequential composition of two `Horizontal` (respectively `Vertical`) arrows is also a `Horizontal` (respectively `Vertical`) arrow. Notice that the sort `Arrow` contains all the existing compositions among `Horizontal` and `Vertical` arrows. Arrows having sort `Mix` can act as either `Horizontal` or `Vertical` arrows, depending on the circumstances. The only arrows that are both `Horizontal` and `Vertical` are those of sort `Mix`.

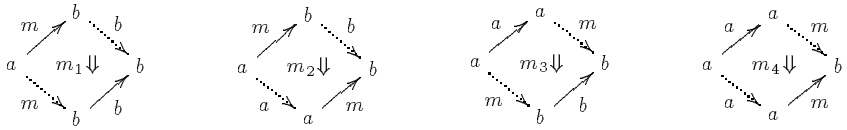
```
cmb t;t' : Horizontal
    iff r(t) = l(t') and t : Horizontal and t' : Horizontal .
cmb t;t' : Vertical
    iff r(t) = l(t') and t : Vertical and t' : Vertical .
cmb t : Mix iff t : Horizontal and t : Vertical .
```

We define a mechanism to construct double cells starting from `Basic` cells. Informally, we want to distinguish between all the possible double cells which are generated by different decompositions of the border of the same 2-cell. The partial operation `mk(_:_,_,_,_)` solves this problem. Its first argument is a `2cell` element, its second and last arguments are `Horizontal` arrows, and the remaining arguments are `Vertical` arrows. If s is the `Basic` cell $s : h;u \Rightarrow v;g$



where h and g are `Horizontal`, and u and v are `Vertical` (we use dotted arrows just to emphasize vertical arrows), then `mk(`s`:`h`,`u`,`v`,`g`)` is a `Dcell`. Since the sort `Basic` contains all the identity cells (which are in sort `Arrow`), and `Horizontal` and `Vertical` can share more structure than just `Object` (the sort `Mix`), it follows that the decomposition of many cells is not unique.

As an example, let $m : a \longrightarrow b$ be an arrow having sort `Mix`, then there are at least four possible ways in which the identity cell on m can be decomposed along the correct horizontal-vertical pattern, each corresponding to a (different, if m is not also an `Object`) `Dcell`, namely $m_1 =$ `mk(`m`:`m`,`b`,`m`,`b`)`, $m_2 =$ `mk(`m`:`m`,`b`,`a`,`m`)`, $m_3 =$ `mk(`m`:`a`,`m`,`m`,`b`)`, and $m_4 =$ `mk(`m`:`a`,`m`,`a`,`m`)` pictured below.



To shorten the notation, we will use two additional derived operators, `mkh(_)` and `mkv(_)`, which are specialized versions of the more general `mk(_:_,_,_,_)`. The intuition is that, given a `Horizontal` arrow h, then `mkh(`h`)` is the `Dcell` representing the vertical identity of h, and similarly for a `Vertical` arrow v then `mkv(`v`)` denotes the corresponding `Dcell`.

In the example above $m_2 = \texttt{mkh}(m)$ and $m_3 = \texttt{mkv}(m)$. The other two decompositions return cells having the same `Mix` arrows as both horizontal and vertical sources (targets), which are the basic ingredients for the construction of generalized transformations as defined in Section 5.4.2. We remark that all the isomorphisms between internal and lateral sorts described earlier are given by `mkh(_)` and `mkv(_)`. The inverses can be easily defined using a projection operator π`cell(_)`, returning the `2cell` associated to a `Dcell`.

```
ops mk(_:_,_,_,_) mkh(_) mkv(_) πcell(_) .
cmb mk(s:h,u,v,g) : Dcell iff h;u = d(s) and v;g = c(s) .
eq  mkh(h) = mk(h:h,r(h),l(h),h) .
eq  mkv(v) = mk(v:l(v),v,v,r(v)) .
mb  mkh(h) : Harrow .
mb  mkv(v) : Varrow .
mb  mkh(m) : Hmix .
mb  mkv(m) : Vmix .
mb  mk(a:a,a,a,a) : Object .
eq  mk(a:a,a,a,a) = a .
ceq πcell(mk(l:h,u,v,g)) = l iff mk(l:h,u,v,g) : Dcell .
mb  πcell(p) : 2cell .
mb  πcell(ha) : Horizontal .
mb  πcell(va) : Vertical .
mb  πcell(hm) : Mix .
mb  πcell(vm) : Mix .
cmb mk(πcell(p):h,u,v,g) : Dcell
    iff h;u = d(πcell(p)) and v;g = c(πcell(p)).
```

Next, we define the double category structure over the cells generated via the operation `mk(_:_,_,_,_)`. We begin by defining the sort of the four projections for each kind of arrow (we omit some axioms because they are redundant). Then, we explicitly define which are the sources and targets of the `Dcell` obtained via a `mk(_:_,_,_,_)` construction. Eventually we formalize the correctness of the previous definitions w.r.t. the operator π`cell(_)` and the sources and targets of the `2cell`s.

```
ops n(_) e(_) w(_) s(_) .
mb  n(p) : Harrow .
mb  s(p) : Harrow .
mb  w(p) : Varrow .
mb  e(p) : Varrow .
mb  w(ha) : Object .
mb  e(ha) : Object .
mb  n(va) : Object .
mb  s(va) : Object .
```

```
  ceq n(mk(l:h,u,v,g)) = mkh(h) if mk(l:h,u,v,g) : Dcell .
  ceq s(mk(l:h,u,v,g)) = mkh(g) if mk(l:h,u,v,g) : Dcell .
  ceq w(mk(l:h,u,v,g)) = mkv(v) if mk(l:h,u,v,g) : Dcell .
  ceq e(mk(l:h,u,v,g)) = mkv(u) if mk(l:h,u,v,g) : Dcell .
  eq  d(πcell(p)) = πcell(n(p));πcell(e(p)) .
  eq  c(πcell(p)) = πcell(w(p));πcell(s(p)) .
  ceq mk(πcell(p):h,u,v,g) = p
      iff h = πcell(n(p)) and u = πcell(e(p)) and
          v = πcell(w(p)) and g = πcell(s(p)) .
```

Notice that, once either the source `d(_)` or the target `c(_)` of a `2cell` has been defined, the axioms of 2-categories give also the value of the source `l(_)` and target `r(_)` of the `2cell` under consideration.

We are now ready to define horizontal ($\_ * \_$) and vertical ($\_ \cdot \_$) compositions between composable `Dcell`'s in terms of suitable compositions between underlying `2cell`'s. As it has been done for sources and targets, we first express the membership axioms, and then give the equational definitions.

```
  ops _*_ _·_ .
  cmb mk(l':h',u,v,g')*mk(l'':h'',w,u,g'') : Dcell
      if mk(l':h',u,v,g') : Dcell and mk(l'':h'',w,u,g'') : Dcell .
  cmb mk(l':h,u',v',g)·mk(l'':g,u'',v'',f) : Dcell
      if mk(l':l':h,u',v',g) : Dcell and mk(l'':g,u'',v'',f) : Dcell .
  cmb mkh(h')*mkh(h'') : Harrow if r(h') = l(h'') .
  cmb mkv(v')·mkv(v'') : Varrow if r(v') = l(v'') .
  ceq mk(l':h',u,v,g')*mk(l'':h'',w,u,g'') =
      mk((h';l'')∘(l';g''):(h';h''),w,v,(g';g''))
      if mk(l':h',u,v,g') : Dcell and mk(l'':h'',w,u,g'') : Dcell .
  ceq mk(l':h,u',v',g)·mk(l'':g,u'',v'',f) =
      mk((l';u'')∘(v';l''):h,(u';u''),(v';v''),f)
      if mk(l':h,u',v',g) : Dcell and mk(l'':g,u'',v'',f) : Dcell .
endfth
```

This concludes the presentation of the theory $T_{\texttt{2EVHCAT}}$.

6.3.1 Monoidal Theories

For the expressiveness of our approach it is essential to add a monoidal structure to the model presented above. In fact, the shared structures we focus on rely on the notions of symmetry, duplicator and discharger, and therefore on the notion of monoidal categories. Nevertheless, this extension is almost effortless, thanks to the tensor product construction (see Definition 6.2.9). It suffices to introduce a theory `MON` of (strict) monoids (see Table 6.4) and then to apply the tensor construction.

```
fth MON is
  sort Monoid .
  ops 1 _⊗_ [assoc id: 1] .
  vars i j : Monoid .
  mb  i⊗j : Monoid .
endfth
```

Table 6.4: The theory of monoids in **PMEqtl**.

We remark that `1` is implicitly defined as a constant of sort `Monoid` (which is the only one defined) and that we exploit the possibility given by Maude of declaring the associativity and identity axioms as attributes of the $\_\otimes\_$ operator. As explained in Appendix D, if a binary operator `f(_,_)` is declared to be associative, then the Maude engine matches equations regardless of how parentheses are left- or right-associated, and the simpler syntax $\texttt{f}(t_1,t_2,\ldots,t_n)$ can be used for any $n \in \mathbb{N}$. The membership assertion $\texttt{mb}\ i\otimes j\ :\ \texttt{Monoid}$ just specifies that the monoidal operation is total.

Then, the theory of (strict) monoidal 2EVH-categories `MON2EVHCAT` can be expressed in a Maude-like notation as in Table 6.5. In the same way, it is easy to define the monoidal theories of categories, 2-categories and double categories by applying very similar tensor product constructions between `MON` and `CAT`, `2CAT` and `DCAT`, called `MONCAT`, `MON2CAT` and `MONDCAT` respectively (see Table 6.6).

6.3.2 Symmetric Theories

The first shared structure that we want to introduce in our model is given by the permutations over the arguments of a monoidal product, called symmetries.

For each pair a, b of `Object`'s we introduce an arrow $\texttt{sym}(a,b)$ of sort `Mix`, which plays the rôle of the symmetry for a and b. The naturality axiom schema is defined for each pair of `Arrow`'s t and t', and the coherence axioms equate all the different compositions of symmetries leading to the same final result. The theory `SYMCAT` of symmetric strict monoidal categories (see Definition 2.1.9) is illustrated in Table 6.7.

A similar construction applies to 2-categories. In this case, the Maude-like definition includes `MON2CAT` instead of `MONCAT`, the source and target functions `d(_)` and `c(_)` are renamed by `l(_)` and `r(_)`, and the naturality axiom involves two variables of sort `Cell`, l and l'.

Notice that, apart from the renaming of some operators, the difference between the definitions of `SYMCAT` and `SYM2CAT` is given by the naturality axiom, which applies to generic arrows in `SYMCAT` and to generic cells in `SYM2CAT`. Since every arrow of `SYM2CAT` is also a cell, it follows that, in `SYM2CAT`, the naturality of the symmetries still holds for generic arrows. Hence, we adopt the convenient notation given in Table 6.8 for the definition of `SYM2CAT`.

```
fth MON2EVHCAT is MON ⊗ 2EVHCAT renamed by (
 sorts (Monoid,Object) to Object . (Monoid,Mix) to Mix .
       (Monoid,Hmix) to Hmix . (Monoid,Horizontal) to Horizontal .
       (Monoid,Vmix) to Vmix . (Monoid,Vertical) to Vertical .
       (Monoid,Harrow) to Harrow . (Monoid,Varrow) to Varrow .
       (Monoid,Arrow) to Arrow . (Monoid,Basic) to Basic .
       (Monoid,2cell) to 2cell . (Monoid,Dcell) to Dcell .
       (Monoid,⊤) to ⊤ .
 ops  1 left to 1 . _⊗_ left to _⊗_ .
      d(_) right to d(_) . c(_) right to c(_) .
      l(_) right to l(_) . r(_) right to r(_) .
      _;_ right to _;_ . _∘_ right to _∘_ .
      n(_) right to n(_) . s(_) right to s(_) .
      w (_)right to w(_) . e(_) right to e(_) .
      _*_ right to _*_ . _·_ right to _·_ .
      mk(_:_,_,_,_) right to mk(_:_,_,_,_) .
      mkh(_) right to mkh(_) . mkv(_) right to mkv(_) .
      πcell(_) right to πcell(_) . ) .
endfth
```

Table 6.5: The theory of monoidal 2EVH-categories in **PMEqtl**.

In a certain sense, using this notation, we are able to define the union of the two imported axiomatizations. Apart from a more compact description of the resulting theory, a very important feature of this approach consists in focusing on the conceptual extension w.r.t. previously defined theories. In `SYM2CAT` the only axiom which really needs to be added is the naturality on cells. Notice that such an extension is consistent with the naturality axiom on arrows which is imported from `SYMCAT`, but makes it redundant.

The case of symmetric monoidal double categories (the theory `SYMDCAT`) is more involved and requires all the axioms presented in Section 5.4.1 for the generalized symmetries, plus some additional axioms induced by the membership logic.

The Maude-like specification of the theory $T_{\texttt{SYMDCAT}}$ is given in Table 6.9. Notice that the diagonal composition `_◁_` used in Section 5.4.1 for expressing the coherence axioms for σ is here represented in terms of horizontal and vertical compositions.

The (strict monoidal) extended version of 2VH-categories gives the opportunity of representing symmetries as arrows of sort `Mix`, and then defining the generalized symmetries via the `mk(_:_,_,_,_)` operation. As before, we can avoid repeating the axiomatization for the symmetries by importing it from `SYMCAT`, the only difference is that now the symmetries have sort `Mix`. Then, we can also define the induced transformations γ`(_,_)`, ρ`(_,_)`, and σ`(_,_)` acting on `Dcell`'s via the `mk` operation (see Table 6.10).

```
fth MONCAT is MON ⊗ CAT renamed by (
  sorts (Monoid,Object) to Object .
        (Monoid,Arrow) to Arrow .
  ops  1 left to 1 . _⊗_ left to _⊗_ .
       d(_) right to d(_) . c(_) right to c(_) .
       _;_ right to _;_ . ) .
endfth

fth MON2CAT is MON ⊗ 2CAT renamed by (
  sorts (Monoid,Object) to Object .
        (Monoid,Arrow) to Arrow .
        (Monoid,Cell) to Cell .
  ops  1 left to 1 . _⊗_ left to _⊗_ .
       d(_) right to d(_) . c(_) right to c(_) .
       l(_) right to l(_) . r(_) right to r(_) .
       _;_ right to _;_ . _∘_ right to _∘_ . ) .
endfth

fth MONDCAT is MON ⊗ DCAT renamed by (
  sorts (Monoid,Object) to Object .
        (Monoid,Harrow) to Harrow .
        (Monoid,Varrow) to Varrow .
        (Monoid,Square) to Square .
  ops  1 left to 1 . _⊗_ left to _⊗_ .
       n(_) right to n(_) . s(_) right to s(_) .
       w (_)right to w(_) . e(_) right to e(_) .
       _*_ right to _*_ . _·_ right to _·_ . ) .
endfth
```

Table 6.6: Monoidal theories of categories, 2-categories, and double categories.

```
fth SYMCAT is
  including MONCAT .
  op sym(_,_) .
  vars a a' b b' : Object .
       t t' : Arrow .
  mb  sym(a,b) : Arrow .
  eq  d(sym(a,b)) = a⊗b .
  eq  c(sym(a,b)) = b⊗a .
  ceq (t⊗t');sym(b,b') = sym(a,a');(t'⊗t)
      if d(t) = a and d(t') = a' and c(t) = b and c(t') = b' .
  eq  sym(a,1) = a .
  eq  sym(1,a) = a .
  eq  sym(a⊗b,c) = (a⊗sym(b,c));(sym(a,c)⊗b) .
  eq  sym(a,b);sym(b,a) = a⊗b .
endfth
```

Table 6.7: The theory of symmetric strict monoidal categories.

```
fth SYM2CAT is
  including MON2CAT .
  including SYMCAT renamed by (
            ops d(_) to l(_) . c(_) to r(_) .) .
  vars a a' b b' : Object .
       l l' : Cell .
  ceq (l⊗l');sym(b,b') = sym(a,a');(l'⊗l)
      if l(l) = a and l(l') = a' and r(l) = b and r(l') = b' .
endfth
```

Table 6.8: The theory of symmetric 2-categories in **PMEqtl**.

```
fth SYMDCAT is
  including MONDCAT .
  ops γ(_,_) ρ(_,_) σ(_,_) .
  vars a a' b : Object .
       ha ha' hb hb' : Harrow .
       va va' vb vb' : Varrow .
       A B : Square .
  cmb γ(A,B) : Square iff A : Varrow and B : Varrow .
  cmb ρ(A,B) : Square iff A : Harrow and B : Harrow  .
  cmb σ(A,B) : Square iff A : Object and B : Object .
  mb  γ(a,b) : Harrow .
  mb  ρ(a,b) : Varrow .
  eq  n(γ(va,vb)) = γ(n(va),n(vb)) .
  eq  s(γ(va,vb)) = γ(s(va),s(vb)) .
  eq  w(γ(va,vb)) = va⊗vb .
  eq  e(γ(va,vb)) = vb⊗va .
  eq  n(ρ(ha,hb)) = ha⊗hb .
  eq  s(ρ(ha,hb)) = hb⊗ha .
  eq  w(ρ(ha,hb)) = ρ(w(ha),w(hb)) .
  eq  e(ρ(ha,hb)) = ρ(e(ha),e(hb)) .
  eq  n(σ(a,b)) = γ(a,b) .
  eq  s(σ(a,b)) = b⊗a .
  eq  w(σ(a,b)) = ρ(a,b) .
  eq  e(σ(a,b)) = b⊗a .
  ceq γ(va·va',vb·vb') = γ(va,vb)·γ(va',vb')
      if s(va) = n(va') and s(vb) = n(vb') .
  ceq ρ(ha*ha',hb*hb') = ρ(ha,hb)*ρ(ha',hb')
      if e(ha) = w(ha') and e(hb) = w(hb') .
  eq  (A⊗B)*γ(e(A),e(B)) = γ(w(A),w(B))*(B ⊗ A) .
  eq  (A⊗B)·ρ(s(A),s(B)) = ρ(n(A),n(B))·(B⊗A) .
  eq  γ(va,vb)·σ(s(va),s(vb)) = σ(n(va),n(vb))·(vb⊗va) .
  eq  ρ(ha,hb)*σ(e(ha),e(hb)) = σ(w(ha),w(hb))*(hb ⊗ ha) .
  eq  γ(va⊗va',vb) = (va⊗γ(va',vb))*(γ(va,vb)⊗va') .
  eq  γ(va,vb)*γ(vb,va) = va⊗vb .
  eq  ρ(ha⊗ha',hb) = (ha⊗ρ(ha',hb))·(ρ(ha,hb)⊗ha') .
  eq  ρ(ha,hb)·ρ(hb,ha) = ha⊗hb .
  eq  σ(a⊗a',b) = ((a⊗σ(a',b))*(γ(a,b)⊗a'))·(σ(a,b)⊗a') .
  eq  (σ(a,b)*γ(b,a))·σ(b,a) = a⊗b .
endfth
```

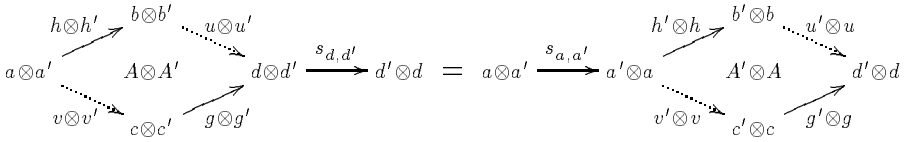
Table 6.9: The theory of symmetric monoidal double categories.

```
fth SYM2EVHCAT is
  including MON2EVHCAT .
  including SYM2CAT .
  ops γ(_,_) ρ(_,_) σ(_,_) .
  vars a a' b b' : Object .
       ha ha'  : Harrow .
       va va'  : Varrow .
  mb  sym(a,b) : Mix .
  ceq ρ(ha,ha') =  mk((πcell(ha)⊗πcell(ha'));sym(b,b'):
                      (πcell(ha)⊗πcell(ha')),sym(b,b'),
                      sym(a,a'),(πcell(ha')⊗πcell(ha)))
      iff w(ha) = a and e(ha) = b and w(ha') = a' and e(ha') = b' .
  ceq γ(va,va') =  mk((πcell(va)⊗πcell(va'));sym(b,b'):
                      sym(a,a'),(πcell(va')⊗πcell(va)),
                      (πcell(va)⊗πcell(va')),sym(b,b'))
      iff n(va) = a and s(va) = b and n(va') = a' and s(va') = b' .
  eq  σ(a,b) = mk(sym(a,b):sym(a,b),(a⊗b),sym(a,b),(a⊗b)) .
endfth
```

Table 6.10: The theory of symmetric 2EVH-categories.

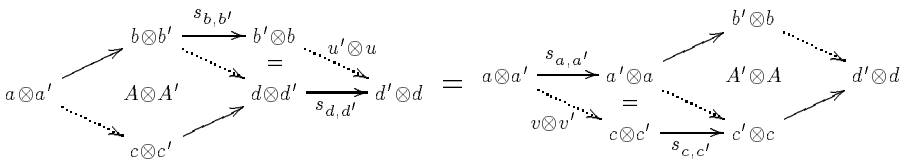
THEOREM 6.3.1
The obvious signature morphism from SYMDCAT *to* SYM2EVHCAT *is a theory morphism.*

Proof (Sketch). We need to prove that the axioms of symmetric monoidal double categories can be derived from the axioms of SYM2EVHCAT. The proof is very long, but trivial. We graphically show the part of the naturality proof, which is the more interesting. From the axiom of symmetry in SYM2EVHCAT

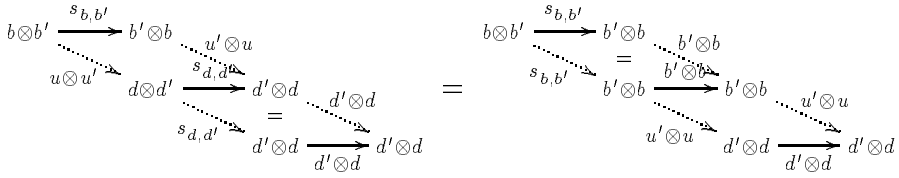


we can derive the following equations of symmetric double categories:

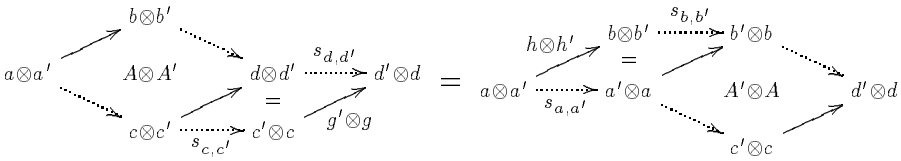
$$(\mathtt{mk}(A:...) \otimes \mathtt{mk}(A':...)) * \gamma(u,u') = \gamma(v,v') * (\mathtt{mk}(A':...) \otimes \mathtt{mk}(A:...))$$



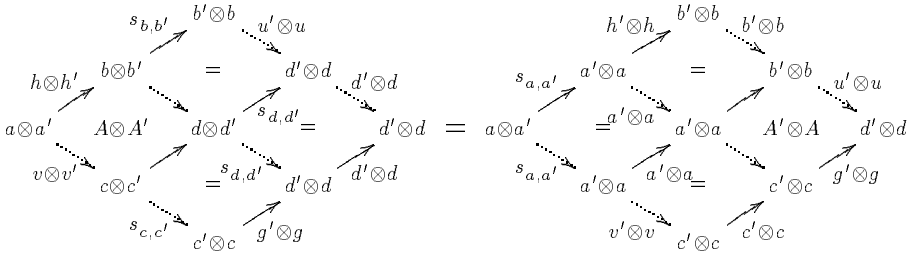
$$\gamma(u,u') \cdot \sigma(d,d') = \sigma(b,b') \cdot (\mathtt{mkv}(u') \otimes \mathtt{mkv}(u))$$



$$(\mathtt{mk}(A:...)\otimes\mathtt{mk}(A':...))\cdot\rho(g,g')=\rho(h,h')\cdot(\mathtt{mk}(A':...)\otimes\mathtt{mk}(A:...))$$



and we can prove the scheme depicted in Figure 5.22.



□

The theory morphism above can be specified in Maude-like notation as:

```
view SE from SYMDCAT to SYM2EVHCAT is
     sort Square to Dcell .
endview
```

This result witnesses the correspondence between the second and the third characterization of auxiliary tiles that we have given in the Introduction of the thesis. Moreover, the following result holds.

THEOREM 6.3.2
The theory morphism `SE` *from* `SYMDCAT` *to* `SYM2EVHCAT` *is persistent w.r.t. sorts* `Objects`, `Harrow`, *and* `Varrow` *and it is complete.*

We conjecture that `SE` is not persistent w.r.t. sort `Square`, i.e., in general, it might be the case that two double cells which are distinguished in `SYMDCAT` are instead identified as the same `Dcell` in `SYM2EVHCAT`.

```
fth CARTCAT is
  including SYMCAT .
  ops dup(_) dis(_) .
  vars a : Object .
       t : Arrow .
  mb  dup(a) : Arrow .
  eq  l(dup(a)) = a .
  eq  r(dup(a)) = a⊗a .
  mb  dis(a) : Arrow .
  eq  l(dis(a)) = a .
  eq  r(dis(a)) = 1 .
  ceq t;dup(b) = dup(a);(t⊗t) if l(t) = a and r(t) = b .
  ceq t;dis(b) = dis(a) if l(t) = a and r(t) = b .
  eq  dup(1) = 1 .
  eq  dis(1) = 1 .
  eq  dup(a⊗b) = (dup(a)⊗dup(b));(a⊗sym(a,b)⊗b) .
  eq  dis(a⊗b) = dis(a)⊗dis(b) .
  eq  dup(a);(dup(a)⊗a) = dup(a);(a⊗dup(a)) .
  eq  dup(a);sym(a,a)  = dup(a) .
  eq  dup(a);(a⊗dis(a)) = a .
endfth
```

Table 6.11: The theory of cartesian categories with chosen products.

6.3.3 Cartesian Theories

The theory `CARTCAT` of cartesian categories (with chosen products and functors preserving products on the nose) is defined in Table 6.11. In particular, for each `Object` a we introduce two arrows `dup(`a`)` and `dis(`a`)` for representing the components at a of the natural duplicator and discharger transformation respectively.

Then, similarly to symmetric 2-categories, the theory `CART2CAT` of cartesian 2-categories can be defined simply adding the naturality axioms on cells (see Table 6.12).

The definition of the theory `CARTDCAT` cartesian double categories is subject to the explicit axiomatization given in Section 5.4.2. Since the description of `CARTDCAT` is very long and analogous to `SYMDCAT`, we leave as an exercise for the really very interested reader to translate the theory of cartesian double categories into Maude-like notation.

Theorem 6.3.4 states that it is possible to derive the axiomatization from the one of cartesian 2EVH-categories given in Table 6.13. Here, duplicators and dischargers are shared arrows (i.e., have sort `Mix`) and the additional double transformations are defined in terms of ordinary duplicators and dischargers via the `mk(_:_,_,_,_)` operator.

```
fth CART2CAT is
  including SYM2CAT .
  including CARTCAT renamed by (
            ops d(_) to l(_) . c(_) to r(_) .) .
  vars a b : Object .
       l : Cell .
  ceq l;dup(b) = dup(a);(l⊗l) if l(l) = a and r(l) = b .
  ceq l;dis(b) = dis(a) if l(l) = a and r(l) = b .
endfth
```

Table 6.12: The theory of cartesian 2-categories.

```
fth CART2EVHCAT is
  including SYM2EVHCAT .
  including CART2CAT .
  ops ∇(_) δ(_) !(_) †(_)
      π(_) τ(_) φ(_) ψ(_) .
  vars a b : Object .
       ha : Harrow .
       va : Varrow .
  mb  dup(a) dis(a): Mix .
  ceq ∇(ha) =  mk((πcell(ha);dup(b)):
                    πcell(ha),dup(b),dup(a),(πcell(ha)⊗πcell(ha)))
      iff w(ha) = a and e(ha) = b .
  ceq δ(va) =  mk((πcell(va);dup(b)):
                    dup(a),(πcell(va)⊗πcell(va)),πcell(va),dup(b))
      iff n(va) = a and s(va) = b .
  eq  π(a) = mk(dup(a):dup(a),(a⊗a),dup(a),(a⊗a)) .
  eq  τ(a) = mk(dup(a):a,dup(a),a,dup(a)) .
  ceq !(ha) =  mk((πcell(ha);dis(b)):πcell(ha),dis(b),dis(a),1)
      iff w(ha) = a and e(ha) = b .
  ceq †(va) =  mk((πcell(va);dis(b)):dis(a),1,πcell(va),dis(b))
      iff n(va) = a and s(va) = b .
  eq  φ(a) = mk(dis(a):dis(a),1,dis(a),1) .
  eq  ψ(a) = mk(dis(a):a,dis(a),a,dis(a)) .
endfth
```

Table 6.13: The theory of cartesian 2EVH-categories.

THEOREM 6.3.3
The obvious signature morphism between `CARTDCAT` *and* `CART2EVHCAT` *is a theory morphism.*

The proof is analogous to that of Theorem 6.3.1 and thus omitted. The theory morphism above yields the correspondence between the second and the third characterization of auxiliary duplicators and dischargers given in Section 1.2 and can be specified in Maude-like notation as:

```
view CE from CARTDCAT to CART2EVHCAT is
     sort Square to Dcell .
endview
```

THEOREM 6.3.4
The theory morphism `CE` *from* `CARTDCAT` *to* `CART2EVHCAT` *is persistent w.r.t. sorts* `Objects`, `Harrow`, *and* `Varrow` *and it is complete.*

6.4 Computads

The notion of *computad* [129, 130] allows for a compact presentation of double categories which are freely generated from a finitary structure (i.e., from the computad). From the point of view of presenting a specification this is very relevant, because it is only necessary to deal with a finite set of rules, which can then be composed in all possible ways to derive the more structured rules, but still maintaining a modular approach to the system description. Therefore, in a certain sense, the computad plays the rôle of a signature that is used to generate the term algebra.

DEFINITION 6.4.1 (COMPUTAD)
A computad *is a triple* $\langle H, V, T\rangle$*, where* H *and* V *are categories with the same set of objects* O*, and* T *is a set of cells, each of which has assigned two pairs of compatible arrows, in* H *and* V*, as vertical and horizontal source and target, respectively.*

Given two computads $\langle H, V, T\rangle$ *and* $\langle H', V', T'\rangle$*, a* c-morphism *from* $\langle H, V, T\rangle$ *to* $\langle H', V', T'\rangle$ *is a triple* $\langle F_h, F_v, F_d\rangle$ *such that* $F_h : H \longrightarrow H'$ *and* $F_v : V \longrightarrow V'$ *are functors which agree on objects, and* $F_d : T \longrightarrow T'$ *is a function such that for each rule* $s \in T$ *the horizontal (resp. vertical) source and target of* $F_d(s)$ *are the images through*[3] F_v *(resp.* F_h*) of the horizontal (resp. vertical) source and target of* s*.*

A computad is symmetric *(resp.* cartesian*) if both* H *and* V *are symmetric monoidal categories (resp. cartesian categories) with symmetries* $\gamma = \{\gamma_{a,b}\}_{a,b\in O}$ *in* H *and* $\rho = \{\rho_{a,b}\}_{a,b\in O}$ *in* V *(resp. also with duplicators* $\nabla = \{\nabla_a\}_{a\in O}$ *in* H*,* $\delta = \{\delta_a\}_{a\in O}$ *in* V*, and with dischargers* $! = \{!_a\}_{a\in O}$ *in* H *and* $\dagger = \{\dagger_a\}_{a\in O}$ *in* V*); c-morphisms are then required to preserve the additional symmetric monoidal (resp. cartesian) structure.*

[3] We remind the reader that the horizontal source and target of a rule s are arrows in V, whereas the vertical source and target of s are arrows in H.

The Maude-like definition of the theory of symmetric (cartesian) computads in Table 6.14 (Table 6.15) is obtained by replacing `MONCAT` with `SYMCAT` (`CARTCAT`) in the theory `CTD` defined in [107]. We import two separated symmetric (cartesian) structures for both horizontal and vertical arrows. At this level, only objects are shared. In particular, notice that the two monoidal operators are different, except when applied to objects. However, they will be identified when generating the associated symmetric (cartesian) double category. Moreover, the operations apply to elements of sort `Rule` if and only if those elements belong to some subsort of `Rule`. Many of the results presented in [107] for the monoidal case can then be extended to the symmetric and cartesian cases.

PROPOSITION 6.4.1
Let `SD` *be the signature morphism from* `SYMCTD` *to* `SYMDCAT` *mapping the sort* `Rule` *to the sort* `Square`*, the operator* $\oplus$ *to the operator* $\otimes$*, and for the rest relating homonymous sorts and operators, and, analogously, let* `CD` *be the signature morphism from* `CARTCTD` *to* `CARTDCAT` *mapping the sort* `Rule` *to the sort* `Square`*, the operator* $\oplus$ *to the operator* $\otimes$*, and for the rest relating homonymous sorts and operators. Then, both* `SD` *and* `CD` *are theory morphisms.*

The theory morphisms above may be represented in Maude-like notation as follows:

```
view SD from SYMCTD to SYMDCAT is
     sort Rule to Square .
     op _⊕_ to _⊗_ .
endview

view CD from CARTCTD to CARTDCAT is
     sort Rule to Square .
     op _⊕_ to _⊗_ .
endview
```

We may compose `SD` with `SE`, and `CD` with `CE` to get theory morphisms from `SYMCTD` to `SYM2EVHCAT` and from `CARTCTD` to `CART2EVHCAT`, respectively.

```
view SVH from SYMCTD to SYM2EVHCAT is SD ; SE
endview

view CVH from CARTCTD to CART2EVHCAT is CD ; CE
endview
```

The following propositions summarize the results. Their proofs follow from Proposition 6.2.1 and from the properties of adjunctions.

```
fth SYMCTD is
 including SYMCAT renamed by (
           sort Arrow to Harrow .
           ops d(_) to w(_) . c(_) to e(_) . _;_ to _*_ .
               sym(_,_) to γ(_,_) .) .
 including SYMCAT renamed by (
           sort Arrow to Varrow .
           ops d(_) to n(_) . c(_) to s(_) . _;_ to _·_ .
               _⊗_ to _⊕_ . sym(_,_) to ρ(_,_) .) .
 sort Rule .
 subsorts Harrow Varrow < Rule .
 vars A B : Rule .
      a b : Object .
      h g : Harrow .
      v u : Varrow .
 mbs w(A) e(A) : Varrow .
 mbs n(A) s(A) : Harrow .
 eq  n(h) = h .
 eq  s(h) = h .
 eq  w(v) = v .
 eq  e(v) = v .
 eq  n(w(A)) = w(n(A)) .
 eq  n(e(A)) = e(n(A)) .
 eq  s(w(A)) = w(s(A)) .
 eq  s(e(A)) = e(s(A)) .
 mb  h⊗g : Harrow .
 mb  v⊕u : Harrow .
 eq  a⊕b = a⊗b .
 cmb A⊗B : Rule iff A : Harrow and B : Harrow .
 cmb A⊕B : Rule iff A : Varrow and B : Varrow .
 cmb A*B : Rule iff A : Harrow and B : Harrow and e(A) = w(B) .
 cmb A·B : Rule iff A : Varrow and B : Varrow and s(A) = n(B) .
 cmb A : Object if A : Harrow and A : Varrow .
 cmb γ(A,B) : Rule iff A : Object and B : Object .
 cmb ρ(A,B) : Rule iff A : Object and B : Object .
endfth
```

Table 6.14: The theory of symmetric computads.

```
fth CARTCTD is
 including CARTCAT renamed by (
           sort Arrow to Harrow .
           ops d(_) to w(_) . c(_) to e(_) . _;_ to _*_ .
               sym(_,_) to γ(_,_) .
               dup(_) to ∇(_,_) . dis(_) to !(_) .) .
 including CARTCAT renamed by (
           sort Arrow to Varrow .
           ops d(_) to n(_) . c(_) to s(_) . _;_ to _·_ .
               _⊗_ to _⊕_ . sym(_,_) to ρ(_,_) .
               dup(_) to δ(_,_) . dis(_) to †(_) .) .
 sort Rule .
 subsorts Harrow Varrow < Rule .
 vars A B : Rule .
      a b : Object .
      h g : Harrow .
      v u : Varrow .
 mbs w(A) e(A) : Varrow .
 mbs n(A) s(A) : Harrow .
 eq  n(h) = h .   eq  s(h) = h .
 eq  w(v) = v .   eq  e(v) = v .
 eq  n(w(A)) = w(n(A)) .
 eq  n(e(A)) = e(n(A)) .
 eq  s(w(A)) = w(s(A)) .
 eq  s(e(A)) = e(s(A)) .
 mb  h⊗g : Harrow .
 mb  v⊕u : Harrow .
 eq  a⊕b = a⊗b .
 cmb A⊗B : Rule iff A : Harrow and B : Harrow .
 cmb A⊕B : Rule iff A : Varrow and B : Varrow .
 cmb A*B : Rule iff A : Harrow and B : Harrow and e(A) = w(B) .
 cmb A·B : Rule iff A : Varrow and B : Varrow and s(A) = n(B) .
 cmb A : Object if A : Harrow and A : Varrow .
 cmb γ(A,B) : Rule iff A : Object and B : Object .
 cmb ∇(A) : Rule iff A : Object .
 cmb !(A) : Rule iff A : Object .
 cmb ρ(A,B) : Rule iff A : Object and B : Object .
 cmb δ(A) : Rule iff A : Object .
 cmb †(A) : Rule iff A : Object .
endfth
```

Table 6.15: The theory of cartesian computads.

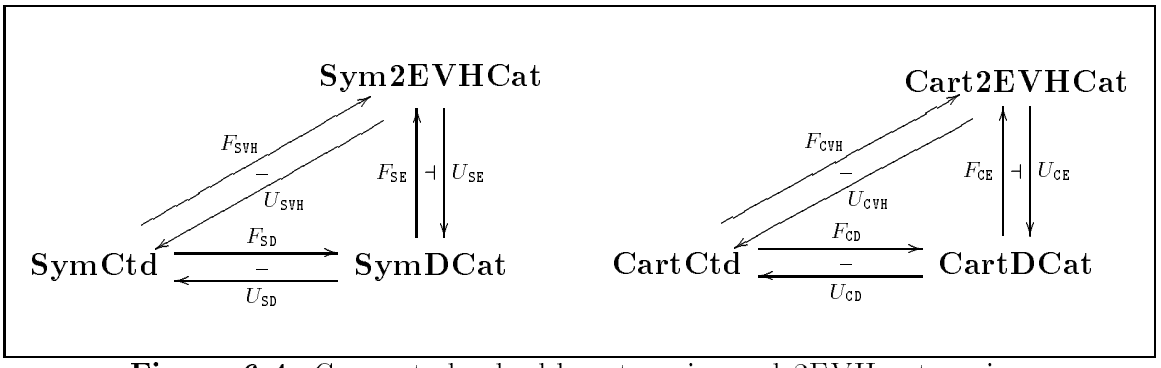


Figure 6.4: Computads, double categories and 2EVH-categories.

PROPOSITION 6.4.2
The forgetful functor $U_{\mathtt{SD}}$: **SymDCat** $\longrightarrow$ **SymCtd** *associated to the theory morphism* SD *(see Proposition 6.4.1) has a left adjoint* $F_{\mathtt{SD}}$: **SymCtd** $\longrightarrow$ **SymDCat**.

Similarly, the forgetful functor $U_{\mathtt{SVH}}$: **Sym2EVHCat** $\longrightarrow$ **SymCtd** *has a left adjoint* $F_{\mathtt{SVH}}$: **SymCtd** $\longrightarrow$ **Sym2EVHCat**.

Furthermore, $F_{\mathtt{SVH}}$ *is given by the composition of the functor* $F_{\mathtt{SD}}$ *with the left adjoint* $F_{\mathtt{SE}}$ *to the forgetful functor* $U_{\mathtt{SE}}$: **Sym2EVHCat** $\longrightarrow$ **SymDCat**.

PROPOSITION 6.4.3
The forgetful functor $U_{\mathtt{CD}}$: **CartDCat** $\longrightarrow$ **CartCtd** *associated to the theory morphism* CD *(see Proposition 6.4.1) has a left adjoint* $F_{\mathtt{CD}}$: **CartCtd** $\longrightarrow$ **CartDCat**.

Similarly, the forgetful functor $U_{\mathtt{CVH}}$: **Cart2EVHCat** $\longrightarrow$ **CartCtd** *has a left adjoint* $F_{\mathtt{CVH}}$: **CartCtd** $\longrightarrow$ **Cart2EVHCat**.

Furthermore, $F_{\mathtt{CVH}}$ *is given by the composition of the functor* $F_{\mathtt{CD}}$ *with the left adjoint* $F_{\mathtt{CE}}$ *to the forgetful functor* $U_{\mathtt{CE}}$: **Cart2EVHCat** $\longrightarrow$ **CartDCat**.

Figure 6.4 summarizes the results.

6.4.1 VH-computads

Taking advantage of the sort `Basic` in $T_{\mathtt{2EVHCAT}}$, it is possible to follow an alternative construction, still obtaining an analogous result. The idea is to reduce each symmetric (cartesian) computad to a suitable symmetric (cartesian) 2-computad, called *VH-computad*, which can then be used to freely generate the associated symmetric (cartesian) 2EVH-category.

DEFINITION 6.4.2 (VH-COMPUTAD)
A VH-computad *is a 4-tuple* $\langle A, H, V, D\rangle$*, where* H *and* V *are lluf*[4] *subcategories of the category* A *(i.e.,* H*,* V*, and* A *have exactly the same objects), and* D *is a set of cells. Each cell has assigned a pair of compatible arrows in* A *as vertical source and target, respectively.*

[4]We remind that a *lluf* subcategory of a category $\mathcal{C}$ is just a subcategory of $\mathcal{C}$ having exactly the same objects as $\mathcal{C}$.

```
fth SYMVHCTD is
  including SYMCAT renamed by (
            ops d(_) to l(_) . c(_) to r(_) .) .
  sorts Mix Horizontal Vertical HV VH 2rule .
  subsorts Object < Mix < Horizontal Vertical < Arrow < 2rule .
  ops d(_) c(_) .
  vars A A' : 2rule .
       h : Horizontal .
       v : Vertical .
  mb sym(a,b) : Mix .
  eq d(h) = h .
  eq c(h) = h .
  eq d(v) = v .
  eq c(v) = v .
  eq l(d(A)) = l(A) .
  eq l(c(A)) = l(A) .
  eq r(d(A)) = r(A) .
  eq r(c(A)) = r(A) .
  cmb A : Mix iff A : Horizontal and A : Vertical .
endfth
```

Table 6.16: The theory of symmetric VH-computads.

Given two computads $\langle A, H, V, D\rangle$ *and* $\langle A', H', V', D'\rangle$, a VH-morphism *is a pair* $\langle F, F_d\rangle$ *such that* $F : A \longrightarrow A'$ *is a functor with* $F(H) \subseteq H'$ *and* $F(V) \subseteq V'$, *and* $F_d : D \longrightarrow D'$ *is a function such that for each rule* $d \in D$ *the horizontal (vertical) source and target of* $F_d(d)$ *are the images through* F *of the (vertical) source and target of* d.

A VH-computad is symmetric *(resp.* cartesian*) if* A, H, *and* V *are symmetric (resp. cartesian) categories.*

The Maude-like definition of the theories of symmetric (cartesian) VH-computads is given in Tables 6.16 and 6.17, respectively. Notice that the theories `SYMVHCTD` and `CARTVHCTD` only differ for the imported theories.

Proposition 6.4.4
Let `S2VH` *be the signature morphism from* `SYMVHCTD` *to* `SYM2EVHCAT` *mapping the sort* `2rule` *into the sort* `Basic`, *and for the rest relating homonymous sorts and operators. Likewise, let* `C2VH` *be the signature morphism from* `CARTVHCTD` *to* `CART2EVHCAT` *mapping the sort* `2rule` *into the sort* `Basic`, *and for the rest relating homonymous sorts and operators. Then,* `S2VH` *and* `C2VH` *are theory morphisms.*

Theorem 6.5.3 in the next section establishes the relevance of these alternative constructions.

```
fth CARTVHCTD is
  including CARTCAT renamed by (
            ops d(_) to l(_) . c(_) to r(_) .) .
  sorts Mix Horizontal Vertical HV VH 2rule .
  subsorts Object < Mix < Horizontal Vertical < Arrow < 2rule .
  ops d(_) c(_) .
  vars A A' : 2rule .
       h : Horizontal .
       v : Vertical .
  mb sym(a,b) : Mix .
  mb dup(a) : Mix .
  mb dis(a) : Mix .
  eq d(h) = h .
  eq c(h) = h .
  eq d(v) = v .
  eq c(v) = v .
  eq l(d(A)) = l(A) .
  eq l(c(A)) = l(A) .
  eq r(d(A)) = r(A) .
  eq r(c(A)) = r(A) .
  cmb A : Mix iff A : Horizontal and A : Vertical .
endfth
```

Table 6.17: The theory of cartesian VH-computads.

6.5 Term Tile Systems and Computads

In this section we establish the correspondence between term tile logic and the free cartesian double category model which is its natural interpretation. We start by explaining how to translate a generic TTS into a suitable computad. As an important result the free cartesian double category arising from the computad entails the same flat sequents of the term tile logic associated to the term tile system. Then, we show that the *extended logic* defined upon the same computad in the theory of cartesian 2EVH-categories also coincides with the cartesian tile logic when considering their flat version (instead, the same is not necessarily true whenever proof terms are considered). We point out that completely analogous results hold for process tile logic. To shorten the notation, we present the results for one-sorted signatures only, but they can be extended to the many-sorted case without any problem.

DEFINITION 6.5.1 (CARTESIAN COMPUTAD OF A TTS)
Let $\mathcal{R} = \langle \Sigma_H, \Sigma_V, N, R\rangle$ be a TTS. *The associated cartesian computad $Ctd(\mathcal{R})$ is the triple $\langle T_{\Sigma_H}(X), T_{\Sigma_V}(X), T_R\rangle$, where the set of tiles T_R is such that*

$$\begin{array}{ccc} \underline{n} & \xrightarrow{\vec{h}} & \underline{m} \\ {\scriptstyle \vec{v}}\downarrow & r & \downarrow{\scriptstyle u} \\ \underline{k} & \xrightarrow[g]{} & \underline{1} \end{array} \in T_R \qquad \Longleftrightarrow \qquad r : n \triangleright \vec{h} \xrightarrow[u]{\vec{v}} g \in \mathcal{R}$$

DEFINITION 6.5.2 (CARTESIAN TILE LOGIC)
Given a term tile system $\mathcal{R}$, the cartesian tile logic *of $\mathcal{R}$ is the cartesian double category $\mathcal{L}_{\mathcal{D}}(\mathcal{R}) = F_{\mathtt{CE}}(Ctd(\mathcal{R}))$ freely generated from the computad $Ctd(\mathcal{R})$ by the left adjoint functor $F_{\mathtt{CE}}$ described in Proposition 6.4.3. For $\alpha : h \xrightarrow[u]{v} g$ in $F_{\mathtt{CE}}(Ctd(\mathcal{R}))_{\mathtt{Square}}$ we also write $\mathcal{R} \vdash_c \alpha$ ($\mathcal{R} \vdash_{fc} h \xrightarrow[u]{v} g$ for flat sequents).*

The following theorem shows that cartesian tile logic and term tile logic coincide.

THEOREM 6.5.1
Given a TTS *$\mathcal{R} = \langle \Sigma_H, \Sigma_V, N, R\rangle$, then $\mathcal{R} \vdash_c \alpha \Longleftrightarrow \mathcal{R} \vdash_t \alpha$.*

DEFINITION 6.5.3 (EXTENDED LOGIC)
Given a TTS *$\mathcal{R} = \langle \Sigma_H, \Sigma_V, N, R\rangle$ the* extended logic *of $\mathcal{R}$ is the cartesian 2EVH-category $F_{\mathtt{CVH}}(Ctd(\mathcal{R})) = F_{\mathtt{CE}}(F_{\mathtt{CD}}(Ctd(\mathcal{R})))$ freely generated from $Ctd(\mathcal{R})$ by the left adjoint functor $F_{\mathtt{CVH}}$ defined in Proposition 6.4.3. For $\beta : h \xrightarrow[u]{v} g \in F_{\mathtt{CVH}}(Ctd(\mathcal{R}))_{\mathtt{Dcell}}$ we also write $\mathcal{R} \vdash_e \beta$ ($\mathcal{R} \vdash_{fe} h \xrightarrow[u]{v} g$ for flat sequents).*

COROLLARY 6.5.2
Given a TTS *$\mathcal{R} = \langle \Sigma_H, \Sigma_V, N, R\rangle$, then $\mathcal{R} \vdash_{ft} h \xrightarrow[u]{v} g \Longleftrightarrow \mathcal{R} \vdash_{fe} h \xrightarrow[u]{v} g$.*

The relevance of this result is that we can use an implementation of rewriting logic to deduce the same flat sequents which are entailed in term tile logic.

Since there are several available languages designed for dealing with rewriting logic specifications, we can actually build tools which work with term tile logic as well. From this perspective, the following result introduces a further step in the translation from tile logic to rewriting logic. In fact, it shows that it is possible to start with a suitable 2-computad in place of the double computad and the result does not change.

DEFINITION 6.5.4 (CARTESIAN VH-COMPUTAD OF A TTS)
Let $\mathcal{R} = \langle \Sigma_H, \Sigma_V, N, R\rangle$ be a TTS. *The associated cartesian VH-computad $Cvh(\mathcal{R})$ is the 4-tuple $\langle T_{\Sigma_H \cup \Sigma_V}(X), T_{\Sigma_H}(X), T_{\Sigma_V}(X), D_R\rangle$, where the set of basic cells D_R is such that*

$$D_R = \left\{ \hat{r} : \underline{n} \overset{u(\vec{h})}{\underset{g(\vec{v})}{\Downarrow}} \underline{1} \;\middle|\; r : n \triangleright \vec{h} \xrightarrow[u]{\vec{v}} g \in R \right\}$$

An important property of $Cvh(\mathcal{R})$ is that the source of each cell in D_R is representable as the sequential composition of an arrow in H and an arrow in V, and the target is the sequential composition of an arrow in V and an arrow in H.

DEFINITION 6.5.5 (CARTESIAN VH-LOGIC)
Given a term tile system $\mathcal{R}$, the cartesian VH-logic *of $\mathcal{R}$ is the cartesian 2EVH-category $F_{\mathtt{C2VH}}(Cvh(\mathcal{R}))$ freely generated from $Cvh(\mathcal{R})$ by the left adjoint functor associated to the theory morphism of Proposition 6.4.4. For $\beta : h \xrightarrow[u]{v} g$ in $F_{\mathtt{C2VH}}(Ctd(\mathcal{R}))_{\mathtt{Dcell}}$ we also write $\mathcal{R} \vdash_{cVH} \beta$ ($\mathcal{R} \vdash_{fcVH} h \xrightarrow[u]{v} g$ for flat sequents).*

THEOREM 6.5.3
Given a TTS *$\mathcal{R} = \langle \Sigma_H, \Sigma_V, N, R\rangle$, then $\mathcal{R} \vdash_{ft} h \xrightarrow[u]{v} g \iff \mathcal{R} \vdash_{fcVH} h \xrightarrow[u]{v} g$.*

The constraint imposed on the extended logics that the computation must be a `Square` can be enforced at the meta-level of the rewriting system by means of a particular internal strategy. Moreover, if — as it is the case of the examples that we have studied — the tile system is *uniform*, then the internal strategy becomes very simple and can be inserted in a standard way directly in the specification layer (see Section 7.4.2).

DEFINITION 6.5.6 (UNIFORM SYSTEMS)
A cartesian (symmetric strict monoidal) double category $\mathcal{D}$ is uniform *if the 2EVH-category $F_{\mathtt{CE}}(\mathcal{D})$ (respectively $F_{\mathtt{SE}}(\mathcal{D})$) satisfies the following conditional membership axiom:*

$$\forall\, A : \mathtt{2cell},\ h, g : \mathtt{Horizontal},\ v, u : \mathtt{Vertical}$$
$$\mathtt{mk}(A{:}h,u,v,g) : \mathtt{Dcell} \Leftarrow \mathtt{d}(A) = h;u \ \wedge\ \mathtt{c}(A) = v;g.$$

A TTS (PTS) *$\mathcal{R}$ is* uniform *if its associated cartesian (symmetric strict monoidal) double category is uniform.*

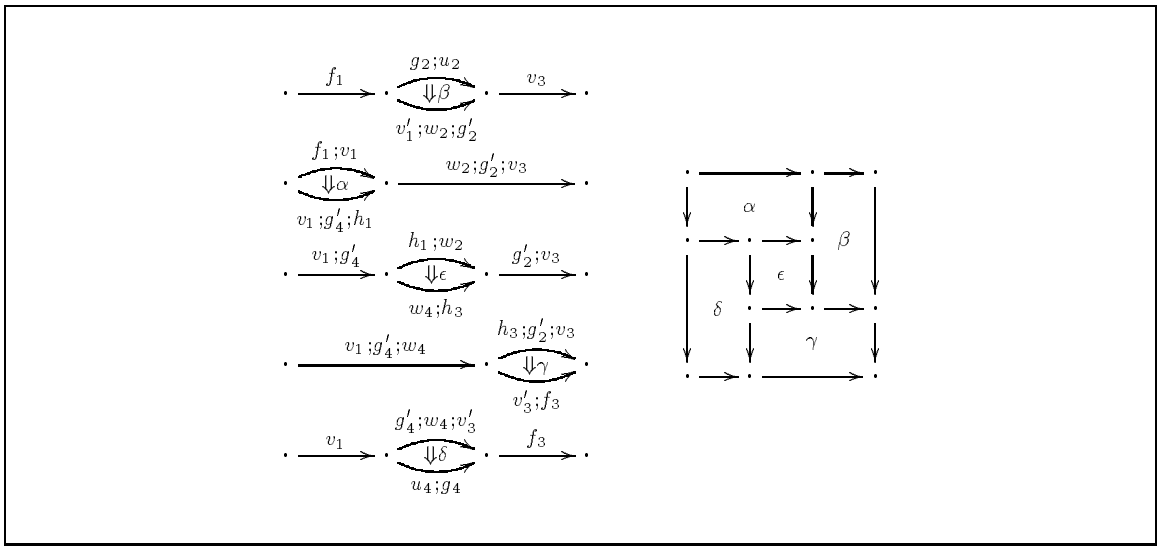


Figure 6.5: The "four bricks (and a square)" counterexample.

In general it is not enough to check that the domain (resp. codomain) arrow of a `2cell` is the composition of a horizontal and a vertical arrow (resp. a vertical and a horizontal arrow) to make sure that the cell is a square. The following well-known "four bricks" tile system provides a counterexample (see e.g., [107]).

EXAMPLE 6.5.1 (FOUR BRICKS COUNTEREXAMPLE) *Let $\mathcal{R}$ be a (one-sorted) tile system, where all constructors, both horizontal and vertical, have $\underline{1}$ as source and target. The set of rules is as follows:*

$$\alpha : f_1 \xrightarrow[v_1']{v_1} g_4'; h_1, \quad \beta : g_2 \xrightarrow[u_2]{v_1';w_2} g_2', \quad \epsilon : h_1 \xrightarrow[w_2]{w_4} h_3, \quad \gamma : h_3; g_2' \xrightarrow[v_3]{v_3'} f_3, \quad \delta : g_4' \xrightarrow[w_4;v_3']{u_4} g_4.$$

It is easy to see that the stretched cells can be composed as follows, but the result is not a `Dcell` *(see also Figure 6.5).*

$$f_1;\beta;v_3 \;\circ\; \alpha;w_2;g_2',v_3 \;\circ\; v_1;g_4';\epsilon;g_2';v_3 \;\circ\; v_1;g_4';w_4;\gamma \;\circ\; v_1;\delta;f_3$$

In fact, even if domain and codomain arrows are of the right form, this cell cannot be obtained from the rules just employing tile horizontal and vertical compositions $\_*\_$ and $\_\cdot\_$.

REMARK 6.5.2 *Dawson [47] presented an interesting result based on the diagram in Figure 6.5 (called* pinwheel *in his terminology). Indeed, Dawson shows that any (rectangular shaped) assembly of tiles that cannot be composed in the algebra of tiles contains a pinwheel (where some of the α, β, γ, δ and ϵ can be obtained as the tile composition of other cells).*

This result is based on a general associativity law *of tile compositions noticed in [48] and it will be useful in Chapter 7 to prove that the implemented tile systems are uniform.*

6.6 Summary

The work presented in this chapter is a natural continuation of the line of research presented by Meseguer and Montanari in [107]. The aim is to investigate the relationship between two very general models of computation: rewriting logic and tile logic. Rewriting logic extends the algebraic semantics approach to concurrent systems with state changes and is the basis of several existing languages. On the other hand, tile logic relies on very general notions of state and observable effect, giving rise to a plethora of compositional models of computation, depending on the chosen structures of configurations and observations.

To compare tile logic and rewriting logic, we have employed **PMEqtl** as a foundational framework. As suggested in the conclusion of [107], we have developed that part of the theory related to the case of cartesian, and symmetric shared structures, corresponding to term tile logic and process tile logic. Our results show that: (a) rewriting logic derivation of `Dcell`'s within 2EVH-categories coincides with tile logic derivation within double categories, (b) the three characterizations that we have given in Section 1.2 for the auxiliary tiles (naïve, categorical, and constructive) coincide (see Theorems 5.4.2, 5.4.5, 6.3.1, and 6.3.4), (c) in general we need a meta-level to control the correctness of the result (i.e., to ensure that the sequent corresponds to some `Dcell`), and (d) the last constraint becomes much simpler to satisfy if the tile system is uniform, because we have only to consider the "border" of each sequent and not the whole proof.

Part III

Implementing Tile Logic

Abstract of Part III

Tile logic extends *rewriting logic* by taking into account side effects and rewriting synchronization. Therefore a correct rewriting implementation of tile logic requires the development of a meta-layer to control rewritings, i.e., to discard computations that do not correspond to any deduction in tile logic so that only the results of tile computations are returned.

By exploiting the *reflective* capabilities of the rewriting logic language *Maude*, such meta-layer can be specified as a kernel of *internal strategies*. It turns out that the required strategies are very general and can be reformulated in terms of search algorithms for non-confluent systems that are equipped with a notion of success. We formalize such strategies, giving their detailed description in Maude, and showing their application to the modelling of *uniform* tile systems.

In particular, we show how such methodology can be applied to *process* and *term tile systems* that extend a wide class of well-known SOS formats, presenting several implementations of CCS-like process calculi.

The results presented in this part are joint work with José Meseguer and Ugo Montanari.

He tried it six more times. "Well," he said, lying back, ceasing.
"Try it again, next time it'll work," she said.
"It's no use," he said. "Something's wrong."
"Well, you've got to try it once more."
He tried it once more.

— RAY BRADBURY, The Next in Line

Chapter 7

Implementing Tile Systems for Process Calculi

Contents

7.1 Motivations

In this section we will show how the language Maude — thanks to its reflective capabilities and, in particular, thanks to the possibility of defining internal strategy languages — can be used to prototype and execute tile rewriting systems.

The evolution of a process in a concurrent system can depend on the behaviours of other cooperating processes. A specification language for concurrent systems cannot leave out of consideration some mechanism for expressing guarded choices in the body of a process. Moreover, such a mechanism should allow local choices to be coordinated, affecting the global evolution of the system.

The tile paradigm looks very appealing from this point of view, however a crucial question has concerned how to give an implementation to tile logic. Systems based on rewriting logic are a natural choice, due to the great similarity of rewriting logic with the more general framework of tiles.

The results presented in Chapter 6 (and in [107]) show that tile logic can be conservatively mapped to rewriting logic provided that "the rewriting engine is able to control rewritings." Specifically, we have seen that we can compute tiles in rewriting logic, provided that we have some mechanism to acknowledge correct tile proofs (i.e., rewritings in sort `Dcell`). However, if we activate the rewriting of a generic configuration, we could end as well in a computation that does not correspond to any tile computation. This is often due to the implicit nondeterminism in the specification, and cannot be avoided. Therefore, we need a methodological approach which could drive the computation along the correct paths.

To achieve this meta-control of the computation, we make use of the *reflective* capabilities [37, 38, 33] of the Maude language [35, 34] to define suitable *internal strategies* [39], which allow the user to acknowledge and to collect the possible results.

To give some examples of the implementation mechanism, we instantiate the general idea to the case studies of *full CCS* (whose presentation requires the term tile format because of the replicator constructor), *finite CCS* (where the computational perspectives are reversed in a counterintuitive way to allow collecting the evolution of a process instead of just testing it) and *located CCS* (presented in the process tile format).

7.1.1 Structure of the Chapter

In Sections 7.2.1 and 7.2.2 we present the reflective capabilities of rewriting logic and explain how they are supported by the language Maude. In Section 7.3 we address the issue of non-confluent rewrite systems, and propose a meta-layer of internal strategies, written in a self-explanatory Maude-like notation, for collecting results and embedding tile systems. In Section 7.4 we present two specific approaches to the implementation of uniform tile systems. Finally, in Sections 7.5 and 7.6 we formalize their application to uniform tile systems, and in particular to the the term tile systems for finite and full CCS, and to the process tile system for located CCS.

7.2 Background

7.2.1 Internal Strategies in Rewriting Logic

Given a logical theory T in a logic, a *strategy* is any computational way of looking for certain proofs of some theorems of T. In particular, we assume the existence of a strategy language $S(T)$ associated with T in which strategies controlling deductions in T can be defined. If such a language is external to the logic, then control becomes an extralogical feature of the system. If strategies can be defined *inside* the logic that they control, we are in a much better situation, since formal reasoning within the system can be applied to the strategies themselves.

As an example, consider a meta-circular interpreter with a fixed strategy. If such strategy remains outside the logic, this will make such an interpreter less flexible, and will complicate formal reasoning about its correctness, whereas strategies defined within the same logic can be represented and can be reasoned about at the object level.

Thus, an *internal strategy language* [37, 33] is a theory-transforming function S that sends each theory T to another theory $S(T)$ *in the same logic*, whose deductions simulate controlled deductions of T. In our opinion, reflective logics are intrinsically suitable for defining internal strategy languages of this kind, since control statements at the meta-level may be expressed within the logic.

Definition 7.2.1 (Reflective logic, universal theory)
Given a logic, we say that it is reflective *[38, 39] relative to a class $\mathcal{C}$ of theories if we can find inside $\mathcal{C}$ a* universal theory U *where all the other theories in the class $\mathcal{C}$ can be simulated, in the sense that there exists a* representation *function*

$$\overline{(\_ \vdash \_)} : \bigcup_{T \in \mathcal{C}} \{T\} \times s(T) \longrightarrow s(U),$$

where $s(T)$ denotes the set of meaningful sentences in the language of a theory T, such that for each $T \in \mathcal{C}$ and $\varphi \in s(T)$,

$$T \vdash \varphi \Longleftrightarrow U \vdash \overline{T \vdash \varphi}.$$

Since U itself is representable ($U \in \mathcal{C}$), the representation can be iterated, so that we get a *reflective tower*

$$T \vdash \varphi \iff U \vdash \overline{T \vdash \varphi} \iff U \vdash \overline{U \vdash \overline{T \vdash \varphi}} \cdots.$$

If a reflective logic has an internal strategy language, then the strategies $S(U)$ for the universal theory are particularly important, since they represent, at the object level, strategies for computing in the universal theory. A meta-circular interpreter for such a language can then be regarded as the implementation of a particular strategy in $S(U)$, and reasoning about the properties of such an interpreter can then be carried out inside the logic itself.

The class of *finitely presentable rewrite theories* has universal theories (in the precise sense that there is a finitely presented rewrite theory U that can simulate all other finitely presented rewrite theories, including itself), making rewriting logic reflective [37, 33].

A rewrite theory T consists of a signature Σ of operators, a set E of equations, and a set of labelled rewrite rules (see Section 2.3). The deductions of T are rewrites modulo E using such rules (also the *proofs* of the deductions could be taken into account, but we restrict ourselves to the simpler case). Moreover, since the meaningful sentences in the language of a rewrite theory T are rewrite sequents $t \Rightarrow t'$, where t and t' are Σ-terms, the general notion of reflection presented above may be restated in the following form.

The class $\mathcal{C}$ is that of finitely presentable rewrite theories. Let U be a universal finitely presentable rewrite theory. The representation function used in [37, 33] $\overline{(\_ \vdash \_)}$ encodes a pair consisting of a rewrite theory T in $\mathcal{C}$ and a sentence $t \Rightarrow t'$ in T as a sentence $\langle \overline{T}, \overline{t} \rangle \Rightarrow \langle \overline{T}, \overline{t'} \rangle$ in U, in such a way that

$$T \vdash t \Rightarrow t' \iff U \vdash \langle \overline{T}, \overline{t} \rangle \Rightarrow \langle \overline{T}, \overline{t'} \rangle,$$

where the function $\overline{(\_)}$ recursively defines the representation of rules, terms, etc. as terms in the universal theory U.

7.2.2 Internal Strategies in Maude

Maude [34] is a logical language based on rewriting logic. For our present purposes the key point is that the Maude implementation supports an arbitrary number of levels of reflection and gives the user access to important reflective capabilities, including the possibility of defining and using internal strategy languages, their implementation and proof of correctness relying on the notion of a basic *reflective kernel*, that is, some basic functionality provided by the universal theory U.

The idea is to first define a *strategy language kernel* as a function *Meta-Level* of rewrite theories, which sends T to a definitional extension of U that defines how rewriting in T is accomplished at the meta-level.

For instance, a typical semantic definition that one wants to have in *Meta-Level*(T) is that of *meta-apply*$(\overline{t}, \overline{l})$, which simulates at the meta-level one step of rewriting at the top of a term t using the rule labelled l in T. Proving the correctness of such a small strategy language kernel is then easily done, by using the correctness of U itself as a universal theory.

The next step is to define a strategy language of choice, say *Strategy*, as a function sending each theory T to a theory that extends *Meta-Level*(T) by additional strategy expressions and corresponding semantic rules, all of which are recursive definitional extensions of those in the kernel in an appropriate sense, so that their correctness can then be reduced to that of the kernel.

7.2.2.1 The Kernel

The Maude implementation supports meta-programming of internal strategies via a module `META-LEVEL` defined in [34, 39], but for efficiency reasons the module `META-LEVEL` is built-in. In particular, `META-LEVEL` provides sorts `Term` and `Module`, so that the representations $\overline{t}$ and $\overline{T}$ of a term t and of a module T have sorts $\overline{t}$: `Term` and $\overline{T}$: `Module` respectively. Then the declaration

`protecting META-LEVEL[`T`]` .

imports the module `META-LEVEL`, declares a new constant `T` of sort `Module`, and adds an equation making `T` equal to the representation of T in `META-LEVEL`. Therefore, we can regard `META-LEVEL` as a module-transforming operation that maps a module T to another module `META-LEVEL[`T`]` that is a definitional extension of U. Here, for simplicity, we adopt a restricted version of such meta-level (e.g., we are not interested in partial instantiation of the rules to be applied during the meta-rewriting). In particular the following operations are defined:

- `meta-reduce(`$\overline{t}$`)` takes the meta-representation $\overline{t}$ of a term t and evaluates as follows: (a) first $\overline{t}$ is converted to the term it represents; (b) then this term is fully reduced using the equations in T; (c) the resulting term t_r is converted to a meta-term $\overline{t}_r$ which is returned as a result.

- `meta-apply(`$\overline{t}$`,`$\overline{l}$`,`n`)` takes the meta-representation of a term t and of a rule label l, and a natural number n and is evaluated as follows: (a) first $\overline{t}$ is converted to the term it represents; (b) then this term is fully reduced using the equations in T; (c) the resulting term t_r is matched against all rules with label l (discarding the matches that fail to satisfy the condition of their rule); (d) the first n successful matches are discarded; (e) if there is an $(n+1)$-th match, its rule is applied using that match; otherwise, `{error*,empty}` is returned; (f) if a rule is applied, the resulting term t' is fully reduced using the equations in T; (g) the resulting term t'_r is converted to a meta-term $\overline{t}'_r$ that is returned as a result, paired with the match used in the reduction (the operator `{_,_}` is used to construct the term).

To make the notation easier, we have used a simpler syntax than the one of the Maude implementation, where `meta-reduce` has an additional argument representing the module T (in the meta-notation) whose equations are used to reduce the term t, and where `meta-apply` has two additional arguments: (1) the meta-representation of the module T as for `meta-reduce` and (2) a set of assignments (possibly empty) defining a partial substitution σ for the variables in the rules of T labelled by l (σ is applied before the matching with the term to be rewritten).

`META-LEVEL` can be considered in our terminology as a *kernel internal strategy language for rewriting logic*. We describe in Table 7.1 the part of the signature of the module `META-LEVEL` that is relevant for our presentation (e.g., omitting equations).

Since in all the applications that we consider only the meta-level of one module is necessary, we give here a parametric definition of `META-LEVEL`, assuming that all the operations (e.g., `meta-reduce`, `meta-apply`, etc.) are instantiated by the parameter T of sort `Module`.

Some examples on the use of the meta-notation are presented in Appendix D, together with the description of our Maude-like notation and the main differences with the Maude syntax. We refer the reader to [34] for an extensive introduction to the subject.

7.3 Maude as a Semantic Framework

In this section we illustrate in detail the problems arising in a non-Church-Rosser system, and how they can be solved by means of internal strategies in reflective languages. In particular we will develop our strategies using the language Maude, defining a general layer to be placed on top of the (nondeterministic) specification layer.

The importance of similar mechanisms is well-known, and other languages based on rewriting logic (e.g., ELAN [13]) have built-in constructs that can deal with general forms of nondeterminism, and also provide powerful mechanisms to drive rewritings [12]. Nevertheless, our approach is rather general (it is parametric w.r.t. a user-definable success predicate) and allows the application of several visiting policies, different from the depth-first (with backtracking) algorithms that are usually preferred in a built-in implementation for efficiency reasons, but that could diverge also in the presence of solutions. Moreover, Maude supports specification in partial membership equational logic, and therefore facilitates the translation of the approach presented in Chapter 6.

7.3.1 Nondeterministic Rewriting Systems

In rewriting logic, nondeterminism can arise whenever multiple rewritings are enabled for the same term. For example the rewriting rules

$$
\begin{array}{l}
\texttt{crl t}(\vec{x})\ \texttt{=>}\ \texttt{t}_1(\vec{x})\ \texttt{if}\ \texttt{G}_1(\vec{x})\ . \\
\texttt{crl t}(\vec{x})\ \texttt{=>}\ \texttt{t}_2(\vec{x})\ \texttt{if}\ \texttt{G}_2(\vec{x})\ . \\
\vdots \\
\texttt{crl t}(\vec{x})\ \texttt{=>}\ \texttt{t}_n(\vec{x})\ \texttt{if}\ \texttt{G}_n(\vec{x})\ .
\end{array}
$$

describe a system in which the terms matching $\texttt{t}(\vec{x})$ can be nondeterministically rewritten into n different terms (in what follows the conditions $\texttt{G}_i(\vec{x})$ are called *guards*, and we say that a guard is *satisfied* if it is evaluated to `true` and that it *fails* if it is evaluated to `false`).

If several guards among the $\texttt{G}_i(\vec{x})$ are satisfied, then the rule to be applied is chosen by the rewrite engine accordingly to some general policy.

```
mod META-LEVEL[T :: Module] is
  sorts Qid Term TermList Label Nat.
  sorts Assignment Substitution ResultPair .
  subsorts Qid < Term < TermList .
  subsorts Assignment < Substitution .
  op 0 : -> Nat .
  op suc(_) : Nat -> Nat .
  op pred(_) : Nat -> Nat .
  op _[_] : Qid TermList -> Term .
  op _,_ : TermList TermList -> TermList [assoc] .
  op error* : -> Term .
  op _<-_ : Qid Term -> Assignment .
  op empty : -> Substitution .
  op _;_ : Substitution Substitution -> Substitution
           [assoc comm id: empty] .
  op {_,_} : Term Substitution -> ResultPair .
  op extTerm : ResultPair -> Term .
  op extSubs : ResultPair -> Substitution .
  op meta-apply : Term Label Nat -> ResultPair .
  op meta-reduce : Term  -> Term .
  ⋮
endm
```

Table 7.1: Partial description of the module META – LEVEL.

We can distinguish between three nondeterministic mechanisms:

- **Conditional choice**: the guards are sequentially processed according to their listing order. The first satisfied guard, say $\texttt{G}_i(\vec{x})$, selects its corresponding i-th rule for the rewriting. Using conditional choice the programmer knows which rule will be chosen if more than one is satisfied. On the other hand, due to the explicit *priority* ordering on the clauses, the use of conditional choice can result in a non-fair policy.

- **Don't care (dc) nondeterminism**: in this case, if any (not necessarily the first) of the guards is satisfied, then the corresponding rule can be applied. Here the main assumption is that whatever choice will be selected, the system will continue to behave correctly. For instance, this approach is well suited for Church-Rosser systems. At the semantic level, dc nondeterminism can overcome the drawback of conditional choice, but the programmer has less control over the computation flow.

- **Don't know (dk) nondeterminism**: sometimes it is not enough to explore just one branch of the nondeterministic computations, because many problems (e.g., in Artificial Intelligence or in Operations Research) are currently solvable only by resorting to some sort of search. In this case the nondeterminism leads to a parallel exploration of the enabled branches. However, performance considerations suggest alternative visiting policies (e.g., depth first with backtracking instead of breadth first). Under some assumption (e.g., finiteness of the tree) the user may explore all the branches, and collect all the solutions. According to dk nondeterminism, if only one clause, say $\texttt{G}_i(\vec{x})$, is satisfied, then the rewriting is said to be *determinate* and the i-th rule is applied. If more than one clause are satisfied, then the statement is said to be *nondeterminate* and the alternative paths are explored concurrently.

For efficiency reasons, only dc nondeterminism is implemented in Maude's default interpreter. This means that whenever multiple reductions are possible the system arbitrarily executes one of them in a fair top-down fashion and ensures fairness in the choice of the rule to be applied, but the user has virtually no control over computations, because the order in which the clauses are listed is not important, or more generally, because although execution paths can be traced, it is difficult to understand how the default strategy determines each choice in a complex example (moreover, the rules can be applied also to proper subterms, and not only at the "top" of the tree-like structure of terms). However, since Maude is a reflective language, it is possible to overcome this limitation by importing the meta-level of some specification, and controlling the computation with suitable (meta-programmed) strategies [39]. We are mostly interested in strategies for dk nondeterminism.

In particular, let us assume that a predicate `ok(_)` over terms defines a suitable notion of success/failure. We say that a term t is *final* if $\texttt{ok}(t) \in \{\texttt{true},\texttt{false}\}$.

```
mod ND-SEM[T :: Module] is protecting META-LEVEL[T] .
  sort TermSequence .
  subsort Term < TermSequence .
  op nilSeq : -> TermSequence .
  op seq : TermSequence TermSequence -> TermSequence
           [assoc id: nilSeq] .
  op first : Term Label Nat -> TermSequence .
  op firstAux : Term Label Nat Nat -> TermSequence .
  vars  t : Term .
        l : Label .
        n m : Nat .
  eq first(t,l,0) = nilSeq .
  eq first(t,l,suc(n)) = firstAux(t,l,suc(0),suc(n)) .
  ceq firstAux(t,l,n,m) = nilSeq if n > m .
  ceq firstAux(t,l,suc(n),m) =
      if meta-apply(t,l,n) == {error*, empty} then nilSeq
      else seq(extTerm(meta-apply(t,l,n)),
               firstAux(t,l,suc(suc(n)),m)) fi
      if n < m .
  op last : Term Label Nat -> TermSequence .
  op allRew : Term Label -> TermSequence .
  eq last(t,l,n) =
     if meta-apply(t,l,n) == {error*, empty} then nilSeq
       else seq(extTerm(meta-apply(t,l,n)),last(t,l,suc(n))) fi .
  eq allRew(t,l) = last(t,l,0) .
endm
```

Table 7.2: The module $\texttt{ND}-\texttt{SEM}[T]$.

A computation c of the system is *successful* if c reaches a final term t such that $\texttt{ok}(t) = \texttt{true}$ and for every term t' visited by c $\texttt{ok}(t') \neq \texttt{false}$, and c is *failing* if $\texttt{ok}(t') = \texttt{false}$ for some term t' visited by c. Obviously, all the failing computations must be discarded (e.g., as soon as a failure is detected the system stops, and a new run with different choices is considered).

7.3.2 Collective Strategies in Maude

In many cases we need to have good ways of controlling the rewriting inference process — which in principle could go in many undesired directions — by means of adequate strategies. Maude offers the possibility of making these strategies *internal* to the logic, i.e., they can be defined by rewrite rules, and can be reasoned about as rules in any other theory.

We illustrate this idea by partially specifying a basic internal strategy language which is able to support dk nondeterministim. In Maude it becomes a module-transforming operation `ND-SEM`, which maps a module T to another module `ND-SEM[`T`]` (see Table 7.2) that extends the strategy kernel `META-LEVEL`.

We define three different functionalities, whose correctness can be easily derived from the correctness of `meta-apply`. The first functionality, called `first`, takes as arguments the meta-representations of a term t and of a label l, and a natural number n, and it evaluates to the sequence[1] of terms containing the first n successful rewritings of t in the theory T using rules with label l. If no rewrite is possible then the empty list `nilSeq` is returned. If only m rewritings are possible, with $m < n$, then the sequence contains only the corresponding m terms.

A second functionality, called `last`, is given for collecting an unbounded number of possible rewritings. Since the presentation of the theory T is finite and also the term t that one wants to rewrite is a finite term, it follows that there are always a finite number of possible (one step) rewritings for the term t in T. However, it is common that the number of possible rewritings is unknown by the user, so that the `first` operation does not give much help. We define `last` as a function taking as arguments the meta-representations of a term t of T and of a rule label l, and a natural number n. The evaluation of this construct returns the sequence of terms containing all the successful rewritings of t in T using rules with label l, except the first n ones. This can be immediately generalized (when $n = 0$) to a function `allRew` taking as arguments the meta-representations of t and l, and returning all the successful rewritings of t in T using rules with label l.

We can now define a new layer which includes different policies for visiting the tree of nondeterministic rewritings. Notice that the specification level is not affected by the meta-extensions. We add a transformation `TREE` which maps a module T to another module `TREE[`T`]`, extending `ND-SEM[`T`]` with a breadth-first and a depth-first visit mechanisms for the nondeterministic rewriting trees in T.

A strategy expression [39] has either the form `rewWith(`t`,`S`)`, where S is the rewriting strategy that we wish to compute, or `failure`, which means that something goes wrong. As the computation of a given strategy proceeds, t gets rewritten according to S and S itself is reduced into the remaining strategy to be computed. In case of termination S becomes the trivial strategy `idle`. In what follows, we assume the existence of a user-definable predicate `ok(_)` (at the object level) as described in Section 7.3.1. Table 7.3 contains the initial part of module `TREE[`T`]`, with those declarations that are common to the strategies we are defining.

We briefly comment on the breadth-first algorithm illustrated in Table 7.4. The expression `rewWith(`t`,breadth(`l`))` means that the user wants to rewrite a term t in T using rules with label l, and exploring all the possibilities "in parallel" until a solution is found.

[1] Here we discuss sequences with repetitions. If one is interested (e.g., for efficiency reasons) in *sequences without repetitions*, then the simple axiom `eq seq(`t`,`TL`,`t`) = seq(`t`,`TL`)` should be added for t : `Term` and TL : `TermSequence`.

```
mod TREE[T :: Module] is protecting ND-SEM[T] .
  sort TermSet .
  subsort Term < TermSet .
  op emptySet : -> TermSet .
  op set : TermSet TermSet -> TermSet [assoc comm id: emptySet] .
  op isIn : Term TermSet -> Bool .
  vars t t' : Term .
       TS : TermSet .
  ceq set(t,t') = t if meta-reduce('_==_[t,t']) == 'true .
  eq isIn(t,emptySet) = false .
  ceq isIn(t,set(t',TS)) = true if meta-reduce('_==_[t,t']) == 'true .
  ceq isIn(t,set(t',TS)) = isIn(t,TS)
      if meta-reduce('_=/=_[t,t']) == 'true .
  sorts Strategy StrategyExpression .
  op idle : -> Strategy .
  op rewWith : Term Strategy -> StrategyExpression .
  op failure : -> StrategyExpression .
  vars TL : TermSequence .
       l : Label .
```

Table 7.3: Initial part of module TREE[T].

```
op breadth : Label -> Strategy .
op rewWithBF : TermSequence TermSet Label -> StrategyExpression .
eq rewWith(t,breadth(l)) = rewWithBF(t,emptySet,l) .
eq rewWithBF(nilSeq,TS,l) = failure .
eq rewWithBF(t,TS,l) =
   if isIn(t,TS) then failure
      else (if meta-reduce('ok[t]) == 'true then rewWith(t, idle)
               else (if meta-reduce('ok[t]) == 'false then failure
                        else rewWithBF(allRew(t,l), set(t,TS),l)
                     fi) fi) fi .
eq rewWithBF(seq(t,TL),TS,l) =
   if isIn(t,TS) then rewWithBF(TL,TS,l)
      else (if meta-reduce('ok[t]) == 'true then rewWith(t,idle)
               else (if meta-reduce('ok[t]) == 'false
                        then rewWithBF(TL,set(t,TS),l)
                        else rewWithBF(seq(TL,allRew(t,l)),
                                       set(t,TS),l)
                     fi) fi) fi .
```

Table 7.4: Definition of the breadth-first strategy in module TREE[T].

This corresponds to the evaluation of the expression `rewWithBF(`t`,emptySet,`l`)`. The function `rewWithBF` takes as arguments a sequence of terms TL, a set of terms TS, and a label l. TS represents the set of already visited terms. The sequence TL contains the terms that have not yet been "checked." If the first argument is the empty sequence of terms, then the function evaluates to *failure*, which means that no solution is reachable (i.e., that all the possible computations fail). If there is at least one term t in the sequence, such that $t \notin TS$ and `ok(`t`) = false`, then the possible rewritings of t in T via rules with label l are appended to the rest of the list (i.e., the sequence of terms is managed as a queue). If `ok(`t`) = true` then t is a solution and the evaluation returns `rewWith(`t`,idle)`.

The implementation of the strategy `depth(`l`)` (see Table 7.5) for the depth-first visit of the tree is very similar to the previous one, except that the sequence of terms TL in `rewWithDF(`TL`,`TS`,`l`)` is managed as a stack instead of a queue.

Notice that this solution does not correspond exactly to the classical notion of depth-first visit, because once a term t is selected from the stack, all of its possible rewritings are calculated. To improve the efficiency of the depth-first visit, we propose the following variant (see Table 7.6): the stack contains pairs of the form (t, i), where t is a term and i is an integer. When such a pair is selected, it means that only the first $i - 1$ rewritings of t have been already inspected and the i-th rewriting t_i of t (if any) should be the next. The advantage is that the stack is shorter, because all rewritings are computed by need. We use the name `depthBT` for this strategy, because it implements a sort of backtracking mechanism. Since this strategy yields the same result as the `depth` strategy, in what follows we do not specify which one is used when a depth-first visit is involved.

In both cases (breadth-first or depth-first visits) the solution is processed in a deterministic way, i.e., if multiple solutions are reachable, then each strategy selects only one of them. It is also possible to define a nondeterministic visit of the tree (in the sense that the specification is nondeterministic, not the Maude execution) using a rewrite rule with label `aux` instead of an equation (see Table 7.7).

If we add an appropriate notion of success, then the module `ND-SEM[TREE[`T`]]` would allow collecting the nondeterministic visits to the (nondeterministic) tree of rewritings in T, and the module `TREE[TREE[`T`]]` allows different mechanisms for exploring the resulting tree of nondeterministic applications of the meta-level rule `aux`. For instance, it could be possible to collect *all* the solutions of the initial nondeterministic system (whereas `TREE[`T`]` allows finding only one solution) by defining a predicate `ok` at the meta-level that yields a very simple notion of *meta-success*. The idea is to use one of the strategies at the meta-meta-level to explore all the possible nondeterministic visits of the tree, finding a success if and only if every application of the rule `aux` at the meta-level leads to a meta-success. The notion of meta-success that we are looking for is given by a meta-term of the type `rewWithND(`LT`, `TS`, `l`)` where LT is a `TermList` such that all the terms in LT are successful (and LT is not empty).

```
op depth : Label -> Strategy .
op rewWithDF : TermSequence TermSet Label -> StrategyExpression .
eq rewWith(t,depth(l)) = rewWithDF(t,emptySet,l) .
eq rewWithDF(nilSeq,TS,l) = failure .
eq rewWithDF(t,TS,l) =
   if isIn(t,TS)
      then failure
      else (if meta-reduce('ok[t]) == 'true
               then rewWith(t,idle)
               else (if meta-reduce('ok[t]) == 'false
                        then failure
                        else rewWithDF(allRew(t,l),set(t,TS),l)
                     fi) fi) fi .
eq rewWithDF(seq(t,TL),TS,l) =
   if isIn(t,TS)
      then rewWithDF(TL,TS,l)
      else (if meta-reduce('ok[t]) == 'true
               then rewWith(t,idle)
               else (if meta-reduce('ok[t]) == 'false
                        then rewWithDF(TL,set(t,TS),l)
                        else rewWithDF(seq(allRew(t,l),TL),
                                       set(t,TS),l)
                     fi) fi) fi .
```

Table 7.5: Definition of the depth-first strategy in module TREE[T].

```
sorts Pair PairSequence .
subsort Pair < PairSequence .
op pair : Term Nat -> Pair .
op nilPair : -> Pair .
op seqPair : PairSequence PairSequence -> PairSequence
             [assoc id : nilPair] .
op depthBT : Label -> Strategy .
op rewWithBT : PairSequence TermSet Label -> StrategyExpression .
var PL : PairSequence .
eq rewWith(t,depthBT(l)) = rewWithBT(pair(t,0),emptySet,l) .
eq rewWithBT(nilSeq,TS,l) = failure .
eq rewWithBT(pair(t,n),TS,l) =
   if isIn(t,TS) then failure
   else (if meta-reduce('ok[t]) == 'true
         then rewWith(t,idle)
         else (if meta-reduce('ok[t]) == 'false
               then failure
               else (if meta-apply(t,l,n) == {error*,empty}
                         then failure
                         else rewWithBT(seqPair(pair(
                                 extTerm(meta-apply(t,l,n)),0),
                                 pair(t,suc(n))),set(t,TS),l)
                     fi) fi) fi) fi .
eq rewWithBT(seqPair(pair(t,n),PL),TS,l) =
   if isIn(t,TS) then rewWithDF(PL,TS,l)
   else (if meta-reduce('ok[t]) == 'true
         then rewWith(t,idle)
         else (if meta-reduce('ok[t]) == 'false
               then rewWithBT(PL,set(t,TS),l)
               else (if meta-apply(t,l,n) == error*,empty
                     then rewWithBT(PL,set(t,TS),l)
                     else rewWithBT(seqPair(pair(
                             extTerm(meta-apply(t,l,n)),0),
                             pair(t,suc(n)),PL),set(t,TS),l)
                     fi) fi) fi) fi .
```

Table 7.6: Definition of the depth-first strategy with backtracking in module TREE[T].

```
op nondet : Label -> Strategy .
op rewWithND : TermSequence TermSet Label -> StrategyExpression .
var TL' : TermSequence .
eq rewWith(t,nondet(l)) = rewWithND(t,emptySet,l) .
eq rewWithND(nilSeq,TS,l) = failure .
rl [aux] : rewWithND(seq(TL,t,TL'),TS,l) =>
           if isIn(t,TS)
           then rewWithND(seq(TL,TL'),TS,l)
           else (if meta-reduce('ok[t]) == 'true
                 then rewWith(t,idle)
                 else (if meta-reduce('ok[t]) == 'false
                       then rewWithND(seq(TL,TL'),set(t,TS),l)
                       else rewWithND(seq(TL,allRew(t,l),TL'),
                                      set(t,TS),l)
                       fi) fi) fi .
```

Table 7.7: Definition of the nondeterministic (non-confluent) strategy in module TREE[T].

EXAMPLE 7.3.1 *As an example consider the following module* Ex *defining the (finite) nondeterministic transition system below, where the only states with success are* s(3), s(4) *and* s(10) *(see Figure 7.1).*

```
mod EX is
  sort State .
  op s : Nat -> State .
  op ok : State -> Bool .
  rl [choice] : s(1) => s(2) .
  rl [choice] : s(1) => s(3) .
  rl [choice] : s(1) => s(4) .
  rl [choice] : s(2) => s(5) .
  rl [choice] : s(2) => s(6) .
  rl [choice] : s(3) => s(7) .
  rl [choice] : s(3) => s(8) .
  rl [choice] : s(4) => s(9) .
  rl [choice] : s(6) => s(10) .
  eq ok(s(3))  = true .
  eq ok(s(4))  = true .
  eq ok(s(10)) = true .
endm
```

The query rew s(1) *gives state* s(5) *as a result, which is not a solution. This corresponds to executing a run of the system which can terminate in any final state.*

The meta-queries

```
Maude> rew rewWith('s['1], breadth('choice)) .
Maude> rew rewWith('s['1], depth('choice)) .
Maude> rew rewWith('s['1], nondet('choice)) .
```

yield respectively the answers

```
rewWith('s['3], idle)
rewWith('s['10], idle)
rewWith('s['3], idle)
```

All of them are acceptable solutions, and we can also observe that the nondeterministic strategy for the reductions gives the same result as the depth-first strategy. However, none of these strategies leads to state `s(4)`*, which is a reachable solution. But all the meta-meta-queries (in* `TREE[TREE[EX]]`*)*

```
Maude> rew rewWith('rewWith['_[_]['']s,''1], 'nondet[''choice]],
                   breadth('aux)) .
Maude> rew rewWith('rewWith['_[_]['']s,''1], 'nondet[''choice]],
                   depth('aux)) .
Maude> rew rewWith('rewWith['_[_]['']s,''1], 'nondet[''choice]],
                   nondet('aux)) .
```

give as a result the same meta-meta-term

```
rewWith('rewWithND[('seq[('_[_]['']s, ''10]),
                         ('_[_]['']s, ''3]),
                         ('_[_]['']s, ''4])]),
                   ('set[('_[_]['']s, ''1]),
                         ('_[_]['']s, ''2]),
                         ('_[_]['']s, ''5]),
                         ('_[_]['']s, ''6])]),
                   ''choice],
        idle)
```

collecting all the successful reachable states of the system in the meta-meta-term notation

```
('seq[('_[_]['']s, ''10]), ('_[_]['']s, ''3]), ('_[_]['']s, ''4])])
```

An alternative solution to the problem of collecting the "solutions" of the system can be given by analyzing the nature of the nondeterministic rule `aux`. It is possible to distinguish two cases depending on the selected term t.

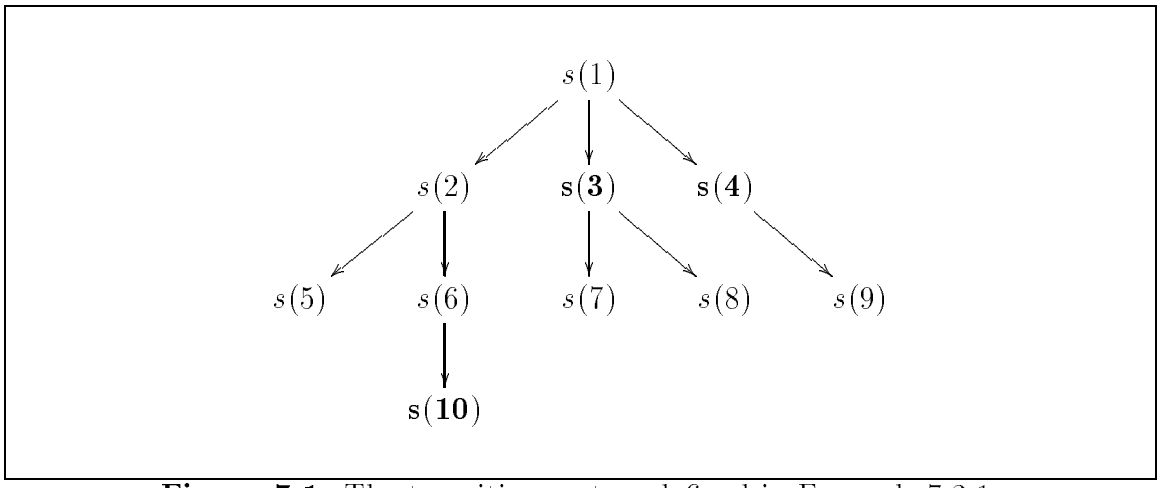


Figure 7.1: The transition system defined in Example 7.3.1.

If t is successful, then the rule discharges all the other possible solutions and chooses t as the final state. If t is not successful (and it is not yet visited) then the computation proceeds by exploring also the rewritings of t.

Whenever we are looking for the whole set of solutions, we need either to apply rule `aux` to a selected term t that is not successful, or to stop as soon as we reach a sequence of successful terms only.

Notice that at this level there is only one solution and that all the computation paths leading to that solution have always the same length. It follows that we can define a simpler specification by using an equation and a rewrite rule.

The strategy that we obtain can be described as:

> "Expand any term that is not a solution, and eventually choose one of the solutions (if it exists)."

Then, at the meta-meta-level, we need only one step of rewriting to find a solution, and the set of meta-solutions can be collected via the function `allRew`. Moreover, the notion of success at the meta-level is simpler and more intuitive. We make use of an auxiliary predicate `okSeq` to acknowledge the sequences of solutions (see Table 7.8).

EXAMPLE 7.3.2 *If we consider the module* `ND-SEM[TREE[EX]]` *where* `EX` *is the module defined in the previous example, then the meta-meta-query*

```
Maude> rew allRew('rewWith['_[_]['']s, ''1],'nondet[''choice]],'aux) .
```

gives as result the sequence of meta-terms

```
seq('rewWith[('_[_]['']s, ''3]), 'idle],
    'rewWith[('_[_]['']s, ''4]), 'idle],
    'rewWith[('_[_]['']s, ''10]), 'idle])
```

```
op okSeq : TermSequence -> Bool .
eq okSeq(nilSeq) = true .
ceq okSeq(t) = true if meta-reduce('ok[t]) == 'true .
ceq okSeq(t) = false if meta-reduce('ok[t]) =/= 'true .
eq okSeq(seq(t,TL)) = okSeq(t) and okSeq(TL) .
eq rewWith(t,nondet(l)) = rewWithND(t,emptySet,l) .
eq rewWithND(nilSeq,TS,l) = failure .
ceq rewWithND(seq(TL,t,TL'),TS,l) =
    if isIn(t,TS)
    then rewWithND(seq(TL,TL'),TS,l)
    else if meta-reduce('ok[t]) == 'false
         then rewWithND(seq(TL,TL'),set(t,TS),l)
         else rewWithND(seq(TL,allRew(t,l),TL'),set(t,TS),l) fi fi
    if meta-reduce('ok[t]) =/= 'true .
crl [aux] : rewWithND(seq(TL,t,TL'),TS,l) => rewWith(t,idle)
    if okSeq(seq(TL,t,TL')) .
op ok : StrategyExpression -> Bool .
eq ok(rewWith(t,idle)) = true .
```

Table 7.8: Nondeterministic strategy revisited.

The results presented in this section can be summarized as follows. Given a nondeterministic rewriting specification T, equipped with a general notion of "success," then:

- the module `ND-SEM[`T`]` allows collecting and analyzing all the possible one-step rewritings of a term (modulo the equations of T);
- the module `TREE[`T`]` allows analyzing one solution among those reachable from a term, depending on the adopted strategy among the three proposed;
- the module `ND-SEM[TREE[`T`]]` allows collecting and analyzing all the (subtree-topmost) solutions reachable from a term. Notice that each solution (if any) is reachable with only one step of meta-rewriting. In Sections 7.5 and 7.6 we will illustrate some applications of this procedure to the executable implementation of tile systems for CCS-like process calculi.

7.3.3 Subterm Rewriting

In order to apply the internal strategies for collecting all the possible rewritings of a term, the definition of the function `allRew` in module `ND-SEM` must be slightly changed. The reason is that the function `meta-apply` applies rewriting only at the top of the term, whereas the rewrite rules could match also proper subterms.

Therefore, we redefine `allRew` so that it evaluates all the possible rewritings of the term and also of every proper subterm, by a recursive exploration of each argument, accomplished by the auxiliary function `allRewAux` (it receives an additional numeric parameter n, indicating that we want to collect all the possible rewritings of the n-th argument of t, if it exists).

```
vars t : Term .
     l : Label .
     n : Nat .
op allRew : Term Qid -> TermSequence .
op allRewAux : TermList Qid Nat -> TermSequence .
eq allRew(t,l) = seq(last(t,l,0), allRewAux(t,l,suc(0))) .
eq allRewAux(t,l,n) =
   if getArgument(t,n) == error* then nilSeq
      else seq(replaceSeq(t,allRew(getArgument(t,n),l),n),
               allRewAux(t,l,suc(n))) fi .
```

Two additional functions `getArgument` and `replaceSeq` must be defined to extract the n-th argument from t and to replace it with all its possible rewritings according to rules labelled by l.

```
op getArgument : Term Nat -> Term .
op getArgumentAux : TermList Nat -> Term .

vars F : Qid .
     RL RL' : TermList .
     t' : Term .

eq  getArgument(F,n) = error* .
eq  getArgument(F[RL],n) = getArgumentAux(RL,n) .
eq  getArgumentAux(t,n) =
    if n == suc(0)
       then t
       else error*
    fi .
eq  getArgumentAux((t,RL),n) =
    if n == suc(0)
       then t
       else (if n > 0
                then getArgumentAux(RL,pred(n))
                else error*
             fi)
    fi .
```

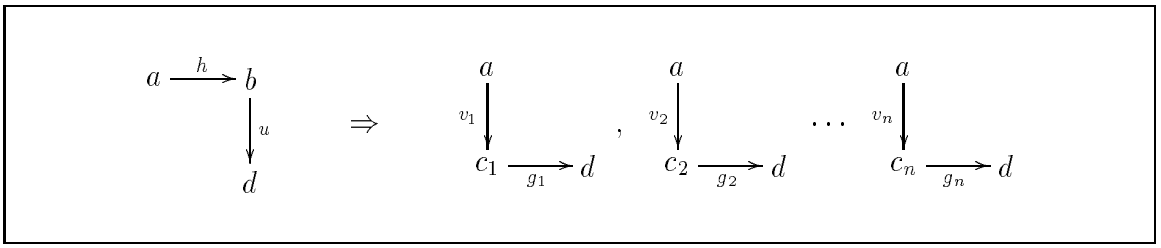


Figure 7.2: A possible tile-query and its answers.

```
op replaceSeq : Term TermSequence Nat -> TermSequence .
op replaceTerm : Term Term Nat -> Term .
op replaceTermAux : TermList Term Nat -> TermList .

eq  replaceSeq(t,nilSeq,n) = nilSeq .
eq  replaceSeq(t,t',n) = replaceTerm(t,t',n) .
eq  replaceSeq(t,seq(t',TL),n) =
    seq(replaceTerm(t,t',n),replaceSeq(t,TL,n)) .
eq  replaceTerm(F,t,n) = error* .
eq  replaceTerm(F[RL],t,n) = F[replaceTermAux(RL,t,n)] .
eq  replaceTermAux(t,t',n) =
    if n == suc(0)
       then t'
       else error*
    fi .
eq  replaceTermAux((t,RL),t',n) =
    if n == suc(0)
       then (t', RL)
       else (if n > 0
                then (t, replaceTermAux(RL,t',pred(n)))
                else error*
             fi)
    fi .
```

7.4 Nondeterminism and Tile Systems

The general strategies that we have presented apply immediately to the translations of tile systems. All we need to specify is the right notion of success, which is user-definable case by case.

For instance, a general notion of success for uniform tile systems consists of checking that we have reached a state which can be decomposed in a vertical arrow followed by a horizontal arrow. Indeed, a typical query in a tile system could be something like "derive all (some of) the tiles with a given horizontal source h and vertical target u" (see Figure 7.2).

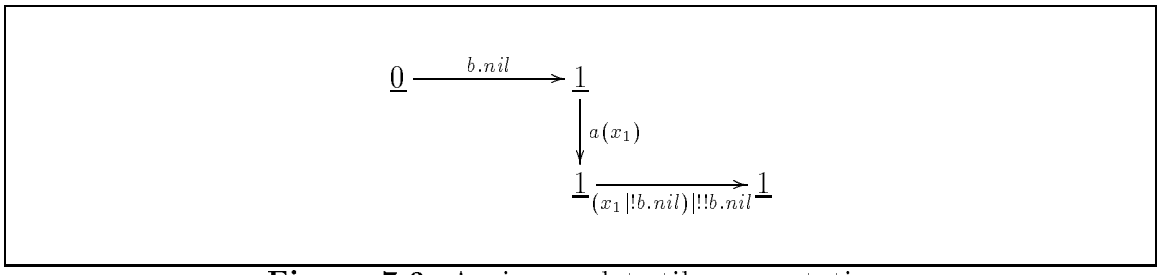


Figure 7.3: An incomplete tile computation.

For example, consider a term tile system that contains the tile for the action prefix (see Figure 1.5 in page 13) and the tile for the replicator (see Figure 5.11 in page 130). The rewriting rules that correspond to the stretched versions of the two tiles are respectively:

$$\begin{aligned} \mu(\mu.P) &\Rightarrow P \\ \mu(!P) &\Rightarrow \mu(P)|!P \end{aligned}$$

The query `rew` $a(!!b.nil)$ can be rewritten into $a(!b.nil)|!!b.nil$ in a first step, and successively into $(a(b.nil)|!b.nil)|!!b.nil$. Then, no more rewriting is possible, but the answer is not in the correct "vertical-horizontal" format, since the tile proof has not been completed (see Figure 7.3). In this case, the meta-layer that we have defined in the previous section is able to check that the computation is not a tile proof, and since no other rewriting is possible, it will answer `failure` to the query. Without the meta-layer, the user cannot be sure that the wrong answer given by the system is the only possible computation, and has to prove this by him/herself.

A surprising thing in the translation of a tile system is that queries start with a horizontal target rather than with a source. The obvious explanation consists in the usual orientation of observation arrows. However in some of the examples that we have considered when developing this approach (e.g. in the case of finite CCS), we realized that the vertical and the horizontal dimension can be swapped in such a way that the intuitive queries are of the kind "derive all one-step transitions for a given agent P." We refer to Section 7.5.3 for a more detailed explanation on this transformation.

7.4.1 Non Uniform Case

If the tile system is not uniform, then also the actual proof term decorating the derivation has to be taken into account. Consequently, the meta-strategies also need to be changed in order to record not only the state, but also the derivation steps that led to that state. This means that the structure of the meta-state would quickly grow huge during the execution, and that the computations would be affected becoming very slow. Since at present we do not have any meaningful example of a non uniform system, we are not really interested in having such an implementation.

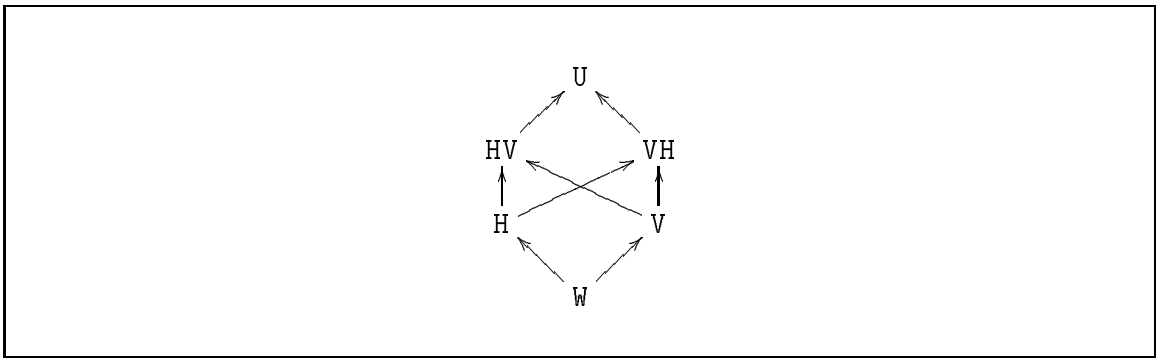


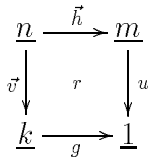
Figure 7.4: The poset of sorts for $\widehat{\mathcal{R}}$.

7.4.2 Uniform Case: Term Tile Logic

In this section we show how it is possible to make use of the membership assertions to model uniform cartesian theories.

7.4.2.1 A First Approach

Let $\mathcal{R} = \langle \Sigma_H, \Sigma_V, N, R \rangle$ be a generic TTS, where Σ_H and Σ_V are two (one-sorted) disjoint signatures and $R(N)$ is a set of rules having the form



where $\vec{h} \in (T_{\Sigma_H}(X_n))^m$, $\vec{v} \in (T_{\Sigma_V}(X_n))^k$, $u \in T_{\Sigma_V}(X_m)$, and $g \in T_{\Sigma_H}(X_k)$, such that the term tile logic of $\mathcal{R}$ (i.e., the cartesian double category freely generated from $Ctd(\mathcal{R})$ by the left adjoint functor described in Proposition 6.4.3) is uniform.

Term tile systems are quite close to the ordinary term rewriting framework, and the membership assertions and subsorting mechanism of Maude can be used (jointly with the internal strategies presented in the previous section) to model uniform term tile systems.

The idea is to define a rewrite theory $\widehat{\mathcal{R}}$ that simulates deductions in $\mathcal{R}$, exploiting the membership mechanism of Maude to acknowledge the correct computations. The theory $\widehat{\mathcal{R}}$ has the poset of sorts illustrated in Figure 7.4.

We briefly comment on their meaning: the sort `W` informally contains the variables of the system as constants; the sort `H` contains the terms over the signature Σ_H and variables in `W` (similarly for the sort `V`); the sort `HV` contains those terms over the signature $\Sigma_{H \cup V}$ and variables in `W` such that they are decomposable as terms over the signature Σ_V applied to terms over Σ_H, i.e., a horizontal arrow followed by a vertical arrow; likewise for `VH`; and the sort `U` contains terms over the signature $\Sigma_{H \cup V}$. As summarized above, we introduce the following operations and membership assertions, for each $h \in \Sigma_H$ and $v \in \Sigma_V$ (with h of arity n and v of arity m):

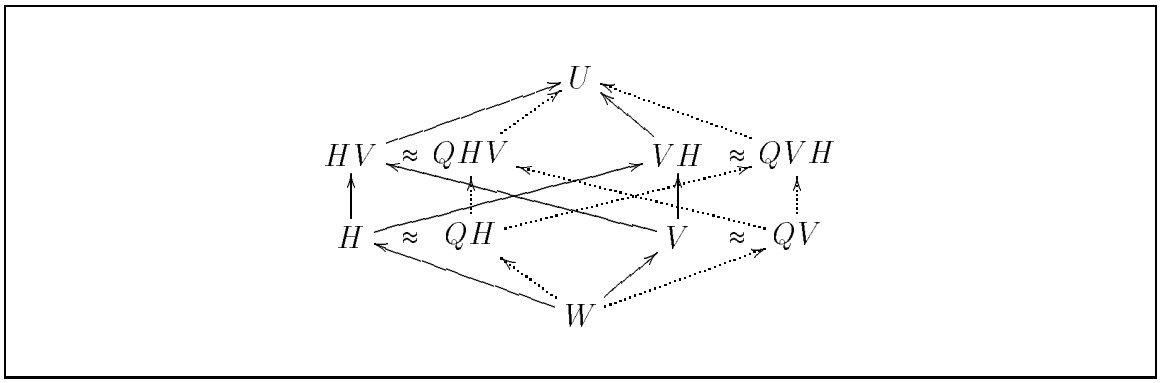


Figure 7.5: The poset of sorts (and "quoted" sorts) for $\hat{\mathcal{R}}$.

```
op h : U^n -> U .
op v : U^m -> U .
vars x_1 ... x_max : U .
cmb h(x_1,...,x_n) : H iff x_1 ... x_n : H .
cmb h(x_1,...,x_n) : VH iff x_1 ... x_n : VH .
cmb v(x_1,...,x_m) : V iff x_1 ... x_m : V .
cmb v(x_1,...,x_m) : HV iff x_1 ... x_m : HV .
```

Eventually, the rewriting rules of $\widehat{\mathcal{R}}$ are the stretched versions $u(\vec{h}) \Rightarrow g(\vec{v})$ of each tile $n \triangleright \vec{l} \xrightarrow[u]{\vec{v}} g$ in $\mathcal{R}$.

```
rl [tile] : u(h⃗) => g(v⃗) .
```

The following result characterizes the correctness of such implementation.

THEOREM 7.4.1
Given a uniform TTS $\mathcal{R}$, *then* $\mathcal{R} \vdash n \triangleright \vec{l} \xrightarrow[u]{\vec{v}} g \iff \widehat{\mathcal{R}} \vdash u(\vec{h}) \Rightarrow g(\vec{v})$.

Proof. The proof follows from Theorem 6.5.3 and from the fact that $\mathcal{R}$ is uniform.□

In Section 7.5.2 we will apply this translation to the TTS for full CCS. However, it might be the case that a term t in `U` but not in `HV` is rewritten into a `VH` configuration. We can extend this approach to prove a stronger relationship between the sequents entailed by tile and rewriting logic.

7.4.2.2 A Stronger Correspondence

This time, we define a rewrite theory $\hat{\mathcal{R}}$, where the poset of its sorts is the one illustrated in Figure 7.5. The sorts `W`, `H`, `V`, `HV` and `VH` are as before. The sorts `QH`, `QV`, `QHV`, and `QVH` are *quoted versions* of the corresponding sorts. We denote the quoted version of a signature Σ_S by $\Sigma_{S'}$, adopting the convention that all the operators of the $\Sigma_{S'}$ are *syntactically* quoted versions of the operators in Σ_S, i.e., $f \in \Sigma_{S,n}$ iff $f' \in \Sigma_{S',n}$).

The sort `U` contains terms over the signature $\Sigma_{H \cup V \cup H' \cup V'}$ and variables in `W`. As summarized above, we introduce the following operations and membership assertions:

```
op h : U^n -> U .
op v : U^m -> U .
op qh : U^n -> U .
op qv : U^m -> U .
vars x_1 ... x_max : U .
cmb h(x_1,...,x_n) : H iff x_1 ... x_n : H .
cmb h(x_1,...,x_n) : VH iff x_1 ... x_n : VH .
cmb v(x_1,...,x_m) : V iff x_1 ... x_m : V .
cmb v(x_1,...,x_m) : HV iff x_1 ... x_m : HV .
cmb qh(x_1,...,x_n) : QH iff x_1 ... x_n : QH .
cmb qh(x_1,...,x_n) : QVH iff x_1 ... x_n : QVH .
cmb qv(x_1,...,x_m) : QV iff x_1 ... x_m : QV .
cmb qv(x_1,...,x_m) : QHV iff x_1 ... x_m : QHV .
```

for each $h \in \Sigma_{H,n}$ and $v \in \Sigma_{V,m}$. After that we add two operations which allow translating a term into its quoted version and viceversa.

```
quote : U -> U .
unquote : U -> U .
cmb quote(x_1) : QH iff x_1 : H .
cmb quote(x_1) : QV iff x_1 : V .
cmb quote(x_1) : QHV iff x_1 : HV .
cmb quote(x_1) : QVH iff x_1 : VH .
cmb quote(x_1) : W iff x_1 : W .
eq quote(h(x_1,...,x_n)) = qh(quote(x_1),...,quote(x_n)) .
eq quote(v(x_1,...,x_m)) = qv(quote(x_1),...,quote(x_m)) .
ceq quote(x_1) = x_1 if x_1 : W .
cmb unquote(x_1) : H iff x_1 : QH .
cmb unquote(x_1) : V iff x_1 : QV .
cmb unquote(x_1) : HV iff x_1 : QHV .
cmb unquote(x_1) : VH iff x_1 : QVH .
cmb unquote(x_1) : W iff x_1 : W .
eq unquote(qh(x_1,...,x_n)) = h(unquote(x_1),...,unquote(x_n)) .
eq unquote(qv(x_1,...,x_m)) = v(unquote(x_1),...,unquote(x_m)) .
ceq unquote(x_1) = x_1 if x_1 : W .
```

The rewriting rules are just the quoted versions of the rules in $\mathcal{R}$:

```
rl [qr] : quote(u(h⃗)) => quote(g(v⃗)) .
```

We then add an operator `top(_)` to indicate the term to be rewritten, and two rules to begin and to end the rewriting computation.

```
top : U -> U .
crl [start] : top(x1) => top(quote(x1)) if x1 : HV .
crl [end] : top(x1) => top(unquote(x1)) if x1 : QVH .
```

The idea is that the quoted sorts are hidden to the user, and the "top" operator ensures that the rewriting process can start only if the query has the correct horizontal-vertical form. The following result can be proved via a simple inspection of the rules in $\hat{\mathcal{R}}$.

THEOREM 7.4.2
Given a uniform TTS $\mathcal{R}$, *then* $\mathcal{R} \vdash_{ft} n \triangleright \vec{h} \xrightarrow[u]{\vec{v}} g \;\Leftrightarrow\; \hat{\mathcal{R}} \vdash \mathtt{top(u}(\vec{\mathtt{h}})\mathtt{)} \Rightarrow \mathtt{top(g}(\vec{\mathtt{v}})\mathtt{)}$.

Moreover, $\hat{\mathcal{R}} \vdash \mathtt{top(t)} \Rightarrow \mathtt{top(t')}$ *implies that there exist* u, $\vec{v}$, g, $\vec{h}$ *and* n *such that* $\mathtt{t} = u(\vec{h})$, $\mathtt{t'} = g(\vec{v})$, *and* $\mathcal{R} \vdash_{ft} n \triangleright \vec{h} \xrightarrow[u]{\vec{v}} g$.

In Section 7.5.3 this translation is applied to the TTS for *finite CCS*, and an example of execution is illustrated in detail.

7.5 Term Tile Logic: Finite and Full CCS

Milner's *Calculus for Communicating Systems* (CCS) [111] is among the better well-known and studied concurrency models. In the recent literature, several ways in which CCS can be conservatively represented in rewriting logic have been proposed [94, 133]. We present here two executable implementations of full CCS and of finite CCS. They are obtained through the translation into Maude of suitable tile systems.

7.5.1 CCS and its Operational Semantics

Let Δ (ranged over by α) be the set of *basic actions*, and $\bar{\Delta}$ the set of *complementary actions* (where $\bar{\_}$ is an involutive function such that $\Delta = \bar{\bar{\Delta}}$ and $\Delta \cap \bar{\Delta} = \emptyset$). We denote by Λ (ranged over by λ) the set $\Delta \cup \bar{\Delta}$. Let $\tau \notin \Lambda$ be a *distinguished* action, and let $Act = \Lambda \cup \{\tau\}$ (ranged over by μ) be the set of CCS actions. Then, a *CCS process* is a term generated by the following grammar:

$$P ::= nil \;\mid\; \mu.P \;\mid\; P\backslash\alpha \;\mid\; P + P \;\mid\; P|P \;\mid\; !P.$$

We let P, Q, R, ... range over the set $Proc$ of CCS processes. Assuming the reader familiar with the notation, we give just an informal description of CCS algebra operators: the constant nil yields the *inactive* process; process $\mu.P$ is a process behaving like P but only after the execution of action μ; process $P\backslash\alpha$ is the process P with actions α and $\bar{\alpha}$ blocked by *restriction* $\_\backslash\alpha$; process $P + Q$ is the *nondeterministic (guarded) sum* of processes P and Q; process $P|Q$ is the *parallel composition* of processes P and Q; finally process $!P$ is the *replicator* of process P. The semantics of CCS can be described by a transition system presented in the SOS style.

$$\frac{}{\mu.P \xrightarrow{\mu} P} \qquad \frac{P \xrightarrow{\mu} Q}{!P \xrightarrow{\mu} Q|!P} \qquad \frac{P \xrightarrow{\mu} Q}{P\backslash\alpha \xrightarrow{\mu} Q\backslash\alpha}\ \mu \notin \{\alpha, \bar{\alpha}\}$$

$$\frac{P \xrightarrow{\mu} Q}{P + R \xrightarrow{\mu} Q} \qquad \frac{P \xrightarrow{\mu} Q}{R + P \xrightarrow{\mu} Q}$$

$$\frac{P \xrightarrow{\mu} Q}{P|R \xrightarrow{\mu} Q|R} \qquad \frac{P \xrightarrow{\mu} Q}{R|P \xrightarrow{\mu} R|Q} \qquad \frac{P \xrightarrow{\lambda} Q,\ P' \xrightarrow{\bar{\lambda}} Q'}{P|P' \xrightarrow{\tau} Q|Q'}$$

Table 7.9: The transition system of full CCS.

DEFINITION 7.5.1 (OPERATIONAL SEMANTICS)
The CCS transition system *is given by the relation* $T \subseteq Proc \times Act \times Proc$ *inductively generated from the set of axioms and inference rules in Table 7.9, where* $P \xrightarrow{\mu} Q$ *stands for* $(P, \mu, Q) \in T$.

REMARK 7.5.1 *To avoid dealing with the meta-level operation of substitution, we have chosen to use the replicator* $!P$ *instead of the ordinary recursive operator* `rec` $x.P$ *of CCS that comes equipped with the rule*

$$\frac{P[\texttt{rec}\ x.P/x] \xrightarrow{\mu} Q}{\texttt{rec}\ x.P \xrightarrow{\mu} Q}$$

Our choice does not affect the expressiveness of the calculus, because it is well known that for each agent `rec` $x.P$ *there exists a weak equivalent agent that can simulate it, namely* $(\alpha_x.nil|!\bar{\alpha}_x.P')\backslash\alpha_x$, *where* α_x *is a new channel name (i.e., not used by* P*) and* P' *is the process obtained by replacing each occurrence of the variable* x *in* P *by* $\alpha_x.nil$.

The operational meaning is that a process P may perform an action μ becoming Q iff it is possible to inductively construct a sequence of rule applications to conclude that $P \xrightarrow{\mu} Q$. More generally, a process P_0 may evolve to a process P_n if there exists a *computation* $P_0 \xrightarrow{\mu_1} P_1 \ldots P_{n-1} \xrightarrow{\mu_n} P_n$.

7.5.2 A Term Tile System for full CCS

In [69] Gadducci and Montanari noticed that the operational semantics of finite CCS fits very well the algebraic tile format. As we have shown in Section 5.1.2, the replicator cannot be directly expressed in the algebraic tile format, but requires instead the term tile format. We adapt their definition to settle the following TTS for the full version of the calculus.

$$\mathtt{act}_\mu : 1 \triangleright\ \mu.x_1 \xrightarrow[\mu(x_1)]{x_1} x_1 \qquad \mathtt{res}_{\mu,\alpha} : 1 \triangleright\ x_1\backslash\alpha \xrightarrow[\mu(x_1)]{\mu(x_1)} x_1\backslash\alpha \quad (\mu \notin \{\alpha, \bar{\alpha}\})$$

$$\mathtt{lsm}_\mu : 2 \triangleright\ x_1 + x_2 \xrightarrow[\mu(x_1)]{\mu(x_1),x_2} x_1 \qquad \mathtt{lpar}_\mu : 2 \triangleright\ x_1|x_2 \xrightarrow[\mu(x_1)]{\mu(x_1),x_2} x_1|x_2$$

$$\mathtt{rsum}_\mu : 2 \triangleright\ x_1 + x_2 \xrightarrow[\mu(x_1)]{x_1,\mu(x_2)} x_2 \qquad \mathtt{rpar}_\mu : 2 \triangleright\ x_1|x_2 \xrightarrow[\mu(x_1)]{x_1,\mu(x_2)} x_1|x_2$$

$$\mathtt{rep}_\mu : 1 \triangleright\ !x_1 \xrightarrow[\mu(x_1)]{\mu(x_1),x_1} x_1|!x_2 \qquad \mathtt{syn}_\lambda : 2 \triangleright\ x_1|x_2 \xrightarrow[\tau(x_1)]{\lambda(x_1),\bar{\lambda}(x_2)} x_1|x_2$$

Table 7.10: The term tile system for CCS with replicator.

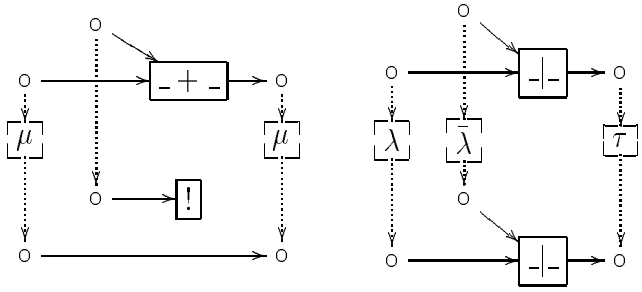


Figure 7.6: The wire and box representation of tiles $\mathtt{suml}_\mu$ and $\mathtt{syn}_\lambda$.

DEFINITION 7.5.2 (TERM TILE SYSTEM FOR CCS)
The TTS $\mathcal{R}_{CCS}$ *has* $\Sigma_A = \{\mu(\_) : 1 \longrightarrow 1 \mid \mu \in Act\}$ *as horizontal signature, the signature* Σ_P *of full CCS processes as vertical signature, and the basic tiles in Table 7.10.*

The tiles for the action prefix and for the replicator have already been discussed (see e.g., pages 13, 130 and 234). As additional examples, we briefly comment the tile $\mathtt{suml}_\mu$ for left nondeterministic choice, and the tile $\mathtt{syn}_\lambda$ for synchronization illustrated in Figure 7.6.

The meaning of the first tile is that the action μ (i.e., the effect $\mu(x_1)$) can be executed by the sum of two subprocesses (i.e., from the initial configuration) if the left subprocess (i.e., the variable x_1 in the initial input interface) can perform the action μ (i.e., the trigger $\mu(x_1)$), evolving to the same subprocess (i.e., the variable x_1 in the final input interface) that will be reached by the nondeterministic sum after such rewriting (i.e., the final configuration x_1). Notice that we can handle the garbaging of the discarded process in the easiest way, using a discharger to throw it away (thanks to the auxiliary structure and tile that were not present in the monoidal system given in [107]). In our notation, this corresponds to not mentioning a variable of the input interface (i.e., the final input interface).

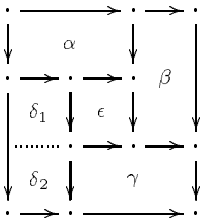


Figure 7.7: An "impure" pinwheel.

$$\frac{}{[\mu, P\rangle : \mu.P \xrightarrow{\mu} P} \qquad \frac{t : P \xrightarrow{\mu} Q}{!t : !P \xrightarrow{\mu} Q|!P} \qquad \frac{t : P \xrightarrow{\mu} Q}{t\backslash\alpha : P\backslash\alpha \xrightarrow{\mu} Q\backslash\alpha}\ \mu \notin \{\alpha, \bar{\alpha}\}$$

$$\frac{t : P \xrightarrow{\mu} Q}{t\langle +R : P + R \xrightarrow{\mu} Q} \qquad \frac{t : P \xrightarrow{\mu} Q}{R+\rangle t : R + P \xrightarrow{\mu} Q}$$

$$\frac{t : P \xrightarrow{\mu} Q}{t\lfloor R : P|R \xrightarrow{\mu} Q|R} \qquad \frac{t : P \xrightarrow{\mu} Q}{R\rfloor t : R|P \xrightarrow{\mu} R|Q} \qquad \frac{t : P \xrightarrow{\lambda} Q,\ t' : P' \xrightarrow{\bar{\lambda}} Q'}{t \| t' : P|P' \xrightarrow{\tau} Q|Q'}$$

Table 7.11: The decorated transition system of full CCS.

The tile for the synchronization of two complementary actions performed by concurrent subprocesses can be read in a similar way. The relevant thing is that both arguments must provide the right triggers, and therefore their evolutions are coordinated.

Analogously to [69], the following result establishes the correspondence between the ordinary SOS semantics for CCS and the sequents entailed by $\mathcal{R}_{CCS}$.

PROPOSITION 7.5.1
The TTS $\mathcal{R}_{CCS}$ *is uniform, and for any CCS agents* P *and* Q*, and action* μ*:*

$$P \xrightarrow{\mu} Q \in T \iff \mathcal{R}_{CCS} \vdash 0 \triangleright P \xrightarrow[\mu(x_1)]{} Q\ .$$

Proof. To prove that $\mathcal{R}_{CCS}$ is uniform, we exploit the result by Dawson on *binarily-composable diagrams* [47], summarized in Remark 6.5.2 (page 209). Basically it states that every non-composable diagram contains a "pinwheel" with cells α, β, γ, δ, and ϵ as in Figure 6.5. Then, we can observe that for every such pinwheel originated by $\mathcal{R}_{CCS}$ the cell δ can always be decomposed as the vertical composition of two cells δ_1 and δ_2, where the horizontal target of δ_1 is the horizontal source of ϵ (see Figure 7.7). Indeed each basic tile of $\mathcal{R}_{CCS}$ has a basic constructor as effect. Therefore $\mathcal{R}_{CCS} \vdash_t ((\alpha \cdot (\delta_1 * \epsilon)) * \beta) \cdot (\delta_2 * \gamma)$.

For the second part of the proposition we proceed by induction on the structure of transition proofs. To facilitate the presentation we decorate the transition systems with proof terms as in Table 7.11.

To prove that $t : P \xrightarrow{\mu} Q \in T \implies \mathcal{R}_{CCS} \vdash 0 \triangleright P \xrightarrow[\mu(x_1)]{} Q$ we map each transition in the corresponding sequent using the function $[\![\_]\!]$, inductively defined as follows:

$$\begin{aligned}
[\![[\mu, P\rangle]\!] &= 1^P * \mathtt{act}_\mu \\
[\![!t]\!] &= (\delta_P \cdot ([\![t]\!] \otimes 1^P)) * \mathtt{rep}_\mu \\
[\![t\backslash\alpha]\!] &= [\![t]\!] * \mathtt{res}_{\mu,\alpha} \\
[\![t\langle +R]\!] &= ([\![t]\!] \otimes 1^R) * \mathtt{lsum}_\mu \\
[\![R+\rangle t]\!] &= (1^R \otimes [\![t]\!]) * \mathtt{rsum}_\mu \\
[\![t\lfloor R]\!] &= ([\![t]\!] \otimes 1^R) * \mathtt{lpar}_\mu \\
[\![R\rfloor t]\!] &= (1^R \otimes [\![t]\!]) * \mathtt{rpar}_\mu \\
[\![t \| t']\!] &= ([\![t]\!] \otimes [\![t']\!]) * \mathtt{syn}_\lambda
\end{aligned}$$

To prove that $t : P \xrightarrow{\mu} Q \in T \impliedby \mathcal{R}_{CCS} \vdash 0 \triangleright P \xrightarrow[\mu(x_1)]{} Q$ we proceed by induction on the process P (we show only a few cases, but the others are very similar):

- $P = nil$: trivial.
- $P = \mu_1.R$: let $\alpha : 0 \triangleright P \xrightarrow[\mu(x_1)]{} Q$, then $\mu = \mu_1$ and $\alpha = \beta * \mathtt{act}_\mu$ for some $\beta : 0 \triangleright R \xrightarrow[x_1]{} Q$, therefore $R = Q$ and $P \xrightarrow{\mu} Q \in T$.
- $P = !R$: let $\alpha : 0 \triangleright P \xrightarrow[\mu(x_1)]{} Q$, then $\alpha = \beta * \mathtt{rep}_\mu$ for some $\beta : 0 \triangleright R \xrightarrow[\mu(x_1),x_1]{} Q$, therefore $(\beta \triangleleft (1_{\underline{1}} \otimes \psi_1)) : 0 \triangleright R \xrightarrow[\mu(x_1)]{} Q$, hence by applying the inductive hypothesis to R we have $R \xrightarrow{\mu} Q \in T$, and by applying the inference rule for the replicator we can conclude that $P \xrightarrow{\mu} Q|P \in T$.

□

By Proposition 7.5.1, it follows immediately that a suitable implementation of $\mathcal{R}_{CCS}$ can be obtained by taking the rewrite theory $\widehat{\mathcal{R}}_{CCS}$ defined in Section 7.4.2.1, and by defining a suitable success predicate for the meta-strategies of Section 7.3.2. Therefore the rules of $\widehat{\mathcal{R}}_{CCS}$ (all labelled by `tile`) are those in Table 7.12, and the success predicate is defined as:

```
ceq ok(t) = true if t : VH .
```

$\mu(\mu.x_1) \Rightarrow x_1$	$\mu(x_1\backslash\alpha) \Rightarrow \mu(x_1)\backslash\alpha \ (\mu \neq \alpha, \bar{\alpha})$
$\mu(x_1 + x_2) \Rightarrow \mu(x_1)$	$\mu(x_1 \mid x_2) \Rightarrow \mu(x_1) \mid x_2$
$\mu(x_1 + x_2) \Rightarrow \mu(x_2)$	$\mu(x_1 \mid x_2) \Rightarrow x_1 \mid \mu(x_2)$
$\mu(!x_1) \Rightarrow \mu(x_1) \mid !x_1$	$\tau(x_1 \mid x_2) \Rightarrow \lambda(x_1) \mid \bar{\lambda}(x_2)$

Table 7.12: The stretched versions of the tiles for CCS with replicator.

COROLLARY 7.5.2
For any CCS processes P and Q, and action μ, $P \xrightarrow{\mu} Q \iff \widehat{\mathcal{R}}_{CCS} \vdash \mu(P) \Rightarrow Q$.

Thus, a typical query (at the meta-level) is `rewWith(`$\overline{\mu(P)}$`,depthBT(`$\overline{\texttt{tile}}$`))`, where $\overline{\mu(P)}$ is the meta-representation of the test $\mu(P)$ that can be used to see if the CCS process P can perform a transition labelled by μ. Then, the system tries to rewrite $\overline{\mu(P)}$ in all possible ways, until a solution of type `VH` is found (if it exists).

EXAMPLE 7.5.2 *Let us consider the CCS process $(\alpha.nil + \beta.nil)\backslash\alpha$. If the rules are applied without any meta-control, then a possible computation for the test $\beta((\alpha.nil + \beta.nil)\backslash\alpha)$ is:*

$$\beta((\alpha.nil + \beta.nil)\backslash\alpha) \Rightarrow \beta(\alpha.nil + \beta.nil)\backslash\alpha \Rightarrow \beta(\alpha.nil)\backslash\alpha$$

Such computation ends in a state that is not a solution (in fact it is the composition of the horizontal arrow $\alpha.nil$, followed by the vertical arrow $\beta(x_1)$, followed by the horizontal arrow $x_1\backslash\alpha$) and that cannot be further rewritten. Such computation is clearly discarded in the meta-controlled computation, and the only possible result `rewWith(Q,idle)`*, where* `Q` *is the meta-representation of $nil\backslash\alpha$, is returned.*

7.5.3 Reversing the Tiles for Finite CCS

We have already pointed out that the translation of tiles into rules allows for the *testing* of executable actions, but not for the generation of the possible moves of an agent. However, we noticed that if the vertical signature consists of *unary* actions, we can reverse the vertical arrow in the tile system and then rotate clockwise by 90 degrees the tiles when implementing the system, as illustrated in Figure 7.8 for the tile associated to the action prefix operator.

If we examine the 2-cell translations, then the motivation for this kind of swapping of arrows is clear:

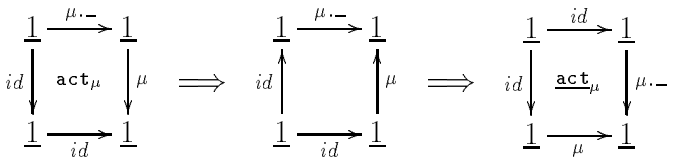


Figure 7.8: Reversing and rotating the tile for action prefix.

$$\frac{P \xrightarrow{\alpha} Q}{P[\alpha/\beta] \xrightarrow{\beta} Q[\alpha/\beta]} \qquad \frac{P \xrightarrow{\bar{\alpha}} Q}{P[\alpha/\beta] \xrightarrow{\bar{\beta}} Q[\alpha/\beta]} \qquad \frac{P \xrightarrow{\mu} Q}{P[\alpha/\beta] \xrightarrow{\mu} Q[\alpha/\beta]} \; (\mu \notin \{\alpha, \bar{\alpha}\})$$

Table 7.13: The rules for the relabelling operator.

The cell on the left states that if we try to force the process $\mu.P$ to perform a μ action, it succeeds. The other cell states that the process $\mu.P$ may perform the action μ. Consequently, an implementation using the first kind of rules can only be used to *test* CCS processes, whereas an implementation based on the second kind of rules may generate all the possible evolutions of the system.

Unfortunately, pursuing this approach we cannot deal with the replicator, because its transformed tile is not in term tile format (the vertical duplicator becomes a *coduplicator*). Therefore we restrict to consider *finite CCS*, where a *finite CCS process* is any term generated by the following grammar:

$$P ::= nil \mid \mu.P \mid P\backslash\alpha \mid P[\alpha/\beta] \mid P + P \mid P|P.$$

Notice that we have introduced the relabelling operator: the process $P[\alpha/\beta]$ behaves like P with actions α and $\bar{\alpha}$ relabelled by β and $\bar{\beta}$. The only difference w.r.t. the classical CCS operator is that we adopt a finitary approach, thus allowing a much simpler representation in the Maude language.

EXAMPLE 7.5.3 *Assuming* $\Delta = \{a_i \mid i \in \mathbb{N}\}$, *then the operator* $\_[\Phi]$ *with* $\Phi = \{[a_i/a_{i+1}] \mid i \in \mathbb{N}\}$ *cannot be defined by a finite application of single relabellings* $\_[\alpha/\beta]$. *However, since a finite process can only perform finitely many actions, then for each finite CCS process* P, *it is possible to "simulate"* $P[\Phi]$. *This is not the case of the full calculus (e.g., consider* $P = \mathtt{rec}\ x.a_0.x[\Phi]$*).*

The inference rules for the operational semantics of finite CCS processes are the same given in Table 7.10 for full CCS (but we do not have the replicator) plus the three rules in Table 7.13 for relabelling.

We define the following "reversed" tile system for finite CCS.

$$\underline{\texttt{act}}_{\mu} : 1 \triangleright\ x_1 \xrightarrow[\mu.x_1]{x_1} \mu(x_1) \qquad \underline{\texttt{res}}_{\mu,\alpha} : 1 \triangleright\ \mu(x_1) \xrightarrow[x_1 \setminus \alpha]{x_1 \setminus \alpha} \mu(x_1)\ \ (\text{if } \mu \notin \{\alpha, \bar{\alpha}\})$$

$$\underline{\texttt{rel}}_{\mu,\alpha,\beta} : 1 \triangleright\ \mu(x_1) \xrightarrow[x_1[\alpha/\beta]]{x_1[\alpha/\beta]} t \ \text{ where } t = \begin{cases} \beta(x_1) & \text{if } \mu = \alpha \\ \bar{\beta}(x_1) & \text{if } \mu = \bar{\alpha} \\ \mu(x_1) & \text{otherwise} \end{cases}$$

$$\underline{\texttt{lsum}}_{\mu} : 2 \triangleright\ \mu(x_1), x_2 \xrightarrow[x_1 + x_2]{x_1} \mu(x_1) \qquad \underline{\texttt{rsum}}_{\mu} : 2 \triangleright\ x_1, \mu(x_2) \xrightarrow[x_1 + x_2]{x_2} \mu(x_1)$$

$$\underline{\texttt{lpar}}_{\mu} : 2 \triangleright\ \mu(x_1), x_2 \xrightarrow[x_1 | x_2]{x_1 | x_2} \mu(x_1) \qquad \underline{\texttt{rpar}}_{\mu} : 2 \triangleright\ x_1, \mu(x_2) \xrightarrow[x_1 | x_2]{x_1 | x_2} \mu(x_1)$$

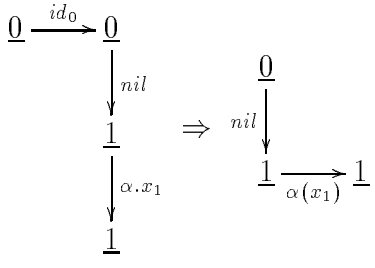
$$\underline{\texttt{syn}}_{\lambda} : 2 \triangleright\ \lambda(x_1), \bar{\lambda}(x_2) \xrightarrow[x_1 | x_2]{x_1 | x_2} \tau(x_1)$$

Table 7.14: The *reversed* tiles for finite CCS.

DEFINITION 7.5.3 (TERM TILE SYSTEM FOR FINITE CCS)
The TTS *associated to finite CCS is the tuple* $\mathcal{R}_{fCCS} = \langle \Sigma_A, \Sigma_P, N, R \rangle$, *where* $\Sigma_A = \{\mu : 1 \longrightarrow 1 \mid \mu \in Act\}$, Σ_P *is the signature of finite CCS processes, and the basic tiles are in Table 7.14.*

Here, the vertical dimension is associated to process descriptions, whereas the horizontal dimension represents the (*opposite* of the) dynamic evolution of the system.[2] For the reader already acquainted with the classical structure of tile systems, the previous definition may appear somewhat odd, because the two dimensions are reversed in a counterintuitive way. As already stressed, the transformation is important because the direct translation of the reversed system into a Maude module allows collecting the possible evolutions of a process, whereas the ordinary definition would allow only testing executable actions.

EXAMPLE 7.5.4 *Let us consider the simple process* $\alpha.nil$ *and let us suppose that we have an executable module both for the ordinary and for the reversed tile systems for CCS. In the reversed case the query is uniquely determined, and the system can give only one answer,*



[2] We say *opposite*, because the direction of the arrows representing the actions performed by the system is the opposite of their computationally intuitive direction.

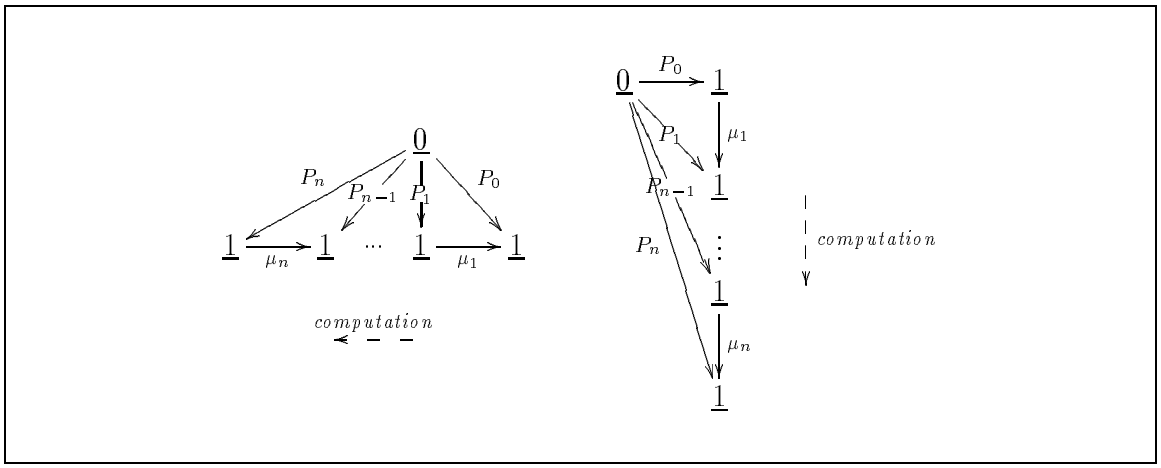


Figure 7.9: The computational comparison of the reversed and the classical tile systems for finite CCS.

which corresponds to the proof sequent $1_{nil} \cdot act_\alpha$. *In the other case, there are as many possible queries as actions in Act, but only one of them enables the rewriting.*

$$\underline{0} \xrightarrow{nil} \underline{1} \xrightarrow{\alpha.x_1} \underline{1} \Big\downarrow{\scriptstyle \beta(x_1)} \underline{1} \quad \Rightarrow \quad \underline{0} \Big\downarrow{\scriptstyle id_0} \underline{0} \xrightarrow[nil]{} \underline{1} \qquad \textit{iff} \;\; \alpha = \beta$$

Given a process P_0 (i.e., an arrow from 0 to 1 in the process dimension) the comparison between the two models can be graphically extended to computations $P_0 \xrightarrow{\mu_1} P_1 \ldots P_{n-1} \xrightarrow{\mu_n} P_n$ (see Figure 7.9, where dashed arrows represent the direction of computational evolutions of processes).

Analogously to [69], the following result holds, establishing the correspondence from the set-theoretic view of the traditional SOS semantics for CCS, and the sequents entailed by term tile logic.

PROPOSITION 7.5.3
The TTS $\mathcal{R}_{fCCS}$ *is uniform, and for any finite CCS agents* P *and* Q, *and action* μ*:*

$$P \xrightarrow{\mu} Q \in T \iff \mathcal{R}_{fCCS} \vdash_{ft} 0 \triangleright \xrightarrow[P]{Q} \mu(x_1) \,.$$

Proof. The proof is analogous to that of Proposition 7.5.1. □

From Proposition 7.5.3 it follows immediately that a suitable implementation of $\mathcal{R}_{fCCS}$ can be obtained by taking the rewriting system $\hat{\mathcal{R}}_{fCCS}$ defined in Section 7.4.2.2 (this time we illustrate the "stronger approach" to the modelling of uniform term tile systems), and considering a success predicate defined by means of the assertion:

```
ceq ok(top(t)) = true if t : VH .
```

Then, the meta-strategies defined in Section 7.3.2 for collecting the correct rewritings can be applied. We give the complete description of the resulting module CCS.

```
mod CCS is protecting MACHINE-INT .
sorts W H V HV VH U QH QV QHV QVH .
subsorts W < H V < HV VH < U .
         W < QH QV < QHV QVH < U .

sorts Channel Act .
subsort Channel < Act .
op a : MachineInt -> Channel .
op bar : Channel -> Channel .
op tau : -> Act .

op nil : -> U .
op pre : Act U -> U .
op res : U Channel -> U .
op rel : U Channel Channel -> U .
op plus : U U -> U .
op par : U U -> U .
op exec : U Act -> U .

op qnil : -> U .
op qpre : Act U -> U .
op qres : U Channel -> U .
op qrel : U Channel Channel -> U .
op qplus : U U -> U .
op qpar : U U -> U .
op qexec : U Act -> U .

vars P Q : U .
     A : Act .
     C D : Channel .

eq bar(bar(C)) = C .

mb nil : H .
cmb pre(A,P) : H if P : H .
cmb res(P,C) : H if P : H .
cmb rel(P,C,D) : H if P : H .
cmb plus(P,Q) : H if P : H and Q : H .
cmb par(P,Q) : H if P : H and Q : H .
cmb exec(P,A) : V if P : V .
```

```
mb nil : VH .
cmb pre(A,P) : VH if P : VH .
cmb res(P,C) : VH if P : VH .
cmb rel(P,C,D) : VH if P : VH .
cmb plus(P,Q) : VH if P : VH and Q : VH .
cmb par(P,Q) : VH if P : VH and Q : VH .
cmb exec(P,A) : HV if P : HV .

mb qnil : QH .
cmb qpre(A,P) : QH if P : QH .
cmb qres(P,C) : QH if P : QH .
cmb qrel(P,C,D) : QH if P : QH .
cmb qplus(P,Q) : QH if P : QH and Q : QH .
cmb qpar(P,Q) : QH if P : QH and Q : QH .
cmb qexec(P,A) : QV if P : QV .
mb qnil : QVH .
cmb qpre(A,P) : QVH if P : QVH .
cmb qres(P,C) : QVH if P : QVH .
cmb qrel(P,C,D) : QVH if P : QVH .
cmb qplus(P,Q) : QVH if P : QVH and Q : QVH .
cmb qpar(P,Q) : QVH if P : QVH and Q : QVH .
cmb qexec(P,A) : QHV if P : QHV .

op quote : U -> U .
cmb quote(P) : QH if P : H .
cmb quote(P) : QV if P : V .
cmb quote(P) : QHV if P : HV .
cmb quote(P) : QVH if P : VH .
cmb quote(P) : W if P : W .
eq quote(nil) = qnil .
eq quote(pre(A,P)) = qpre(A,quote(P)) .
eq quote(res(P,C)) = qres(quote(P),C) .
eq quote(rel(P,C,D)) = qrel(quote(P),C,D) .
eq quote(plus(P,Q)) = qplus(quote(P),quote(Q)) .
eq quote(par(P,Q)) = qpar(quote(P),quote(Q)) .
eq quote(exec(P,A)) = qexec(quote(P),A) .
ceq quote(P) = P if P : W .

op unquote : U -> U .
cmb unquote(P) : H if P : QH .
cmb unquote(P) : V if P : QV .
cmb unquote(P) : HV if P : QHV .
cmb unquote(P) : VH if P : QVH .
cmb unquote(P) : W if P : W .
```

```
eq unquote(qnil) = nil .
eq unquote(qpre(A,P)) = pre(A,unquote(P)) .
eq unquote(qres(P,C)) = res(unquote(P),C) .
eq unquote(qrel(P,C,D)) = rel(unquote(P),C,D) .
eq unquote(qplus(P,Q)) = plus(unquote(P),unquote(Q)) .
eq unquote(qpar(P,Q)) = par(unquote(P),unquote(Q)) .
eq unquote(qexec(P,A)) = exec(unquote(P),A) .
ceq unquote(P) = P if P : W .

op top : U -> U .
crl [qr] : top(P) => top(quote(P)) if P : VH .

rl  [qr] : qpre(A,P) => qexec(P,A) .
crl [qr] : qres(qexec(P,A),C) => qexec(qres(P,C),A)
    if A =/= C and A =/= bar(C) .
crl [qr] : qrel(qexec(P,A),C,D) => qexec(qrel(P,C,D),A)
    if A =/= C and A =/= bar(C) .
crl [qr] : qrel(qexec(P,A),C,D) => qexec(qrel(P,C,D),D)
    if A == C .

crl [qr] : qrel(qexec(P,A),C,D) => qexec(qrel(P,C,D),bar(D))
    if A == bar(C) .
rl  [qr] : qplus(qexec(P,A),Q) => qexec(P,A) .
rl  [qr] : qplus(Q,qexec(P,A)) => qexec(P,A) .
rl  [qr] : qpar(qexec(P,A),Q) => qexec(qpar(P,Q),A) .
rl  [qr] : qpar(Q,qexec(P,A)) => qexec(qpar(Q,P),A) .
crl [qr] : qpar(qexec(P,C),qexec(Q,D)) => qexec(qpar(P,Q),tau)
    if C == bar(D) .

crl [qr] : top(P) => top(unquote(P)) if P : QHV .

op ok : U -> Bool .
ceq ok(top(P)) = true if P : HV .
endm
```

The code exactly corresponds to the translation illustrated in section 7.4.2, but we use a more verbose syntax for the operators of the TTS. In particular:

- the denumerable set of basic actions is $\{\mathtt{a}(i) \mid i \in \mathbb{N}\}$,
- the special action τ is denoted by `tau`,
- the inactive process nil is denoted by `nil`,
- the action prefix $\mu.P$ is denoted by `pre(`μ`,`P`)`,

- the restriction $P\backslash\alpha$ is denoted by `res(`P`,`α`)`,
- the relabelling $P[\alpha/\beta]$ is denoted by `rel(`P`,`α`,`β`)`,
- the nondeterministic sum $P + Q$ is denoted by `plus(`P`,`Q`)`,
- the parallel composition $P|Q$ is denoted by `par(`P`,`Q`)`, and
- the dynamic evolution $\mu(P)$ is denoted by `exec(`P`,`μ`)`.

The sort `W` is necessary for executing partially specified queries (in this case the process variables that are used must be declared as constants having sort `W`).

EXAMPLE 7.5.5 *We show the result of a computation in* `ND-SEM[TREE[CCS]]`*, collecting the successful states reachable from the process* $(a_1.nil+a_2.nil)|\bar{a}_1.nil$*. We use a meta-meta-query to collect all the interesting solutions. The meta-meta-notation could require some acquaintance with meta-translations, but some tools will soon be available to perform automatic translations. For the moment, we hope that the indentation will suffice to make the reading easier.*

```
Maude> rew allRew('rewWith[('_[_]['']top,
('_[_]['']par, ('_,_[
              ('_[_]['']plus, ('_,_[
                              ('_[_]['']pre, ('_,_[('_[_]['']a, ''1]), ''nil])]),
                              ('_[_]['']pre, ('_,_[('_[_]['']a, ''2]), ''nil])])
                             ])]),
              ('_[_]['']pre, ('_,_[('_[_]['']bar, ('_[_]['']a, ''1])]), ''nil])])
             ])])]),
('nondet['']qr])], 'aux) .

rewrites: 26822 in 719ms cpu (729ms real) (37252 rewrites/second)
result TermSequence:
seq(
*** a1(nil | ā1.nil)
'rewWith[('_[_]['']top, ('_[_][
''exec, ('_,_[
        ('_[_]['']par, ('_,_[
                       ''nil,
                       ('_[_]['']pre, ('_,_[
                                      ('_[_]['']bar, ('_[_]['']a, ''1])]),
                                      ''nil])])
                      ])]),
        ('_[_]['']a, ''1])])])]), 'idle],

*** a1(ā1(nil | nil))
'rewWith[('_[_]['']top, ('_[_][
''exec, ('_,_[
        ('_[_]['']exec, ('_,_[
                        ('_[_]['']par, ('_,_[''nil, ''nil])]),
                        ('_[_]['']bar, ('_[_]['']a, ''1])])])]),
        ('_[_]['']a, ''1])])])]), 'idle],
```

```
*** ā1(a1(nil | nil))
'rewWith[('_[_]['']top, ('_[_][
''exec, ('_,_[
        ('_[_]['']exec, ('_,_[
                        ('_[_]['']par, ('_,_[''nil, ''nil])]),
                        ('_[_]['']a, ''1])])]),
        ('_[_]['']bar, ('_[_]['']a, ''1])])])])]), 'idle],

*** τ(nil | nil)
'rewWith[('_[_]['']top, ('_[_][
''exec, ('_,_[
        ('_[_]['']par, ('_,_[''nil, ''nil])]),
        ''tau])])]), 'idle],

*** a2(nil |ā1.nil)
'rewWith[('_[_]['']top, ('_[_][
''exec, ('_,_[
        ('_[_]['']par, ('_,_[
                        ''nil,
                        ('_[_]['']pre, ('_,_[
                                        ('_[_]['']bar, ('_[_]['']a, ''1])]),
                                        ''nil])])])]),
        ('_[_]['']a, ''2])])])]), 'idle],

*** a2(ā1(nil | nil))
'rewWith[('_[_]['']top, ('_[_][
''exec, ('_,_[
        ('_[_]['']exec, ('_,_[
                        ('_[_]['']par, ('_,_[''nil, ''nil])]),
                        ('_[_]['']bar, ('_[_]['']a, ''1])])])]),
        ('_[_]['']a, ''2])])])]), 'idle],

*** ā1(a2(nil | nil))
'rewWith[('_[_]['']top, ('_[_][
''exec, ('_,_[
        ('_[_]['']exec, ('_,_[
                        ('_[_]['']par, ('_,_[''nil, ''nil])]),
                        ('_[_]['']a, ''2])])]),
        ('_[_]['']bar, ('_[_]['']a, ''1])])])])]), 'idle],

*** ā1((a1.nil + a2.nil) | nil)
'rewWith[('_[_]['']top, ('_[_][
''exec, ('_,_[
        ('_[_]['']par, ('_,_[
                        ('_[_]['']plus, ('_,_[
                                        ('_[_]['']pre, ('_,_[
                                                        ('_[_]['']a, ''1]),
                                                        ''nil])]),
                                        ('_[_]['']pre, ('_,_[
                                                        ('_[_]['']a, ''2]),
                                                        ''nil])])])]),
                        ''nil])]),
        ('_[_]['']bar, ('_[_]['']a, ''1])])])])]), 'idle],
```

```
*** (a1.nil + a2.nil) | ā1.nil
'rewWith[('_[_]['']top,
('_[_]['']par, ('_,_[
              ('_[_]['']plus, ('_,_[
                              ('_[_]['']pre, ('_,_[('_[_]['']a, ''1]), ''nil])]),
                              ('_[_]['']pre, ('_,_[('_[_]['']a, ''2]), ''nil])])
                              ])]),
              ('_[_]['']pre, ('_,_[('_[_]['']bar, ('_[_]['']a, ''1])]), ''nil])])
              ])])]), 'idle])
```

Notice that all the possible interleaving computations of the initial process are collected (the last answer corresponds to the idle computation).

7.6 Process Tile Logic: Located CCS

In the spirit of *true concurrent semantics*, the notion of *concurrency* cannot be reduced to nondeterminism via interleaving as it happens for the implementations that we have considered in the last section. To overcome this problem, one possibility is to define a relation representing those pairs of events that can occur in any order, i.e., the *commuting diamonds* of the transition system [14, 58]. From an operational point of view, this corresponds to defining a concrete *concurrent machine* implementing the calculus. As an alternative, it could be possible to define a model where the notion of *observation* captures causal dependencies between events or between the places where they occur (e.g., their *locations*).

In [60], a uniform treatment for both the operational and the abstract concurrent semantics of a CCS-like process calculus is provided by means of tile logic. In particular, gs-graphs are used to model the structure of configurations and observations of the resulting (flat) tile systems. Gs-graphs are similar to term graphs, but can deal with hypersignatures. As discussed in Section 2.2, the term graph structure is essentially a weak cartesian category where two naturality axioms are missing, thus, from another perspective, defining a suitable symmetric strict monoidal category. Hence, their systems should be placed somewhere in between process tile logic and term tile logic. In this section we propose a graph-like presentation of those systems which can be implemented in Maude. Although we are convinced that it is possible to generalize this procedure, here we deal only with our case study.

Definition 7.6.1 (A Simple Process Calculus: Located CCS)
Let Δ, Λ, and Act be as in Section 7.5.1. Let Loc be a totally ordered (by $<$) denumerable set of locations*, ranged over by l. Then a* located process *P is a term generated by the following grammar:*

$$G ::= nil \mid \mu.G \mid G + G \mid G|G \qquad P ::= G \mid l :: P \mid P|P$$

where, for the sake of simplicity, we distinguish the ground processes *(i.e., processes without locations), ranged over by G, G', and so on.*

$$\frac{}{[\mu, l, G\rangle : \mu.G \xrightarrow[l]{\mu} l :: G} \qquad \frac{t : P \xrightarrow[k]{\mu} P',\ l \notin loc(k)}{l :: t : l :: P \xrightarrow[lk]{\mu} l :: P'}$$

$$\frac{t : G_1 \xrightarrow[k]{\mu} G_2}{t\langle +G : G_1 + G \xrightarrow[k]{\mu} G_2} \qquad \frac{t : G_1 \xrightarrow[k]{\mu} G_2}{G+\rangle t : G + G_1 \xrightarrow[k]{\mu} G_2}$$

$$\frac{t : P_1 \xrightarrow[k]{\mu} P_2,\ loc(k) \cap loc(P) = \emptyset}{t \lfloor P : P_1 | P \xrightarrow[k]{\mu} P_2 | P} \qquad \frac{t : P_1 \xrightarrow[k]{\mu} P_2,\ loc(k) \cap loc(P) = \emptyset}{P \rfloor t : P | P_1 \xrightarrow[k]{\mu} P | P_2}$$

$$\frac{t_1 : P_1 \xrightarrow[u_1]{\lambda} P_1',\ t_2 : P_2 \xrightarrow[u_2]{\bar{\lambda}} P_2',\ loc(u_1) \cap loc(P_2') = \emptyset = loc(u_2) \cap loc(P_1')}{t_1 \| t_2 : P_1 | P_2 \xrightarrow[u_1, u_2]{\tau} P_1' | P_2'}$$

Table 7.15: The operational semantics of located CCS.

Locations [15] are introduced to allow the external observer to see an action together with the location where it takes place. As an example, this approach distinguishes process $\alpha.\beta.nil + \beta.\alpha.nil$ from $\alpha.nil|\beta.nil$, because the second process can perform α and β separately in different places, while the first process cannot. The operational semantics is defined by a transition system whose labels consist of actions together with *strings of locations*, denoted by u. In a synchronization, the strings associated to the synchronizing actions are paired (in the strong version) or erased (in the weak case). We call a generic label of the transition system a *denotation*, and denote by Den the set of denotations (ranged over by k). In lk, the location l is concatenated with each string in k. As a matter of notation, we use $loc(P)$ and $loc(k)$ to indicate the set of location names occurring in process P or in denotation k. It follows that $loc(G) = \emptyset$ for any ground process G.

Definition 7.6.2 (Operational Semantics)
The inference rules in Table 7.15 define the transition algebra *TA of located CCS.*

To define the concurrent operational semantics, a *concurrency relation* χ is defined on the algebra of transitions and computations to identify the commuting diamonds of the system, following the approach proposed in [58].

Definition 7.6.3 (Concurrency Relation)
Let ($\_$ *then* $\_$ χ $\_$ *then* $\_$) *be a quaternary relation on transition proof terms, defined as the least commutative*[3] *relation defined by the structural rules in Table 7.16, where* $t_i : P_i \xrightarrow[k_i]{\mu_i} Q_i$, $t_i' : P_i' \xrightarrow[k_i']{\mu_i'} Q_i'$, *and* $t : P \xrightarrow[k]{\mu} Q$.

[3]Namely, (t_1 then t_2 χ t_3 then t_4) if (t_3 then t_4 χ t_1 then t_2).

$$\frac{}{(t_1 \lfloor P_2 \text{ then } Q_1 \rfloor t_2 \ \chi \ P_1 \rfloor t_2 \text{ then } t_1 \lfloor Q_2)} \qquad \frac{(t_1 \text{ then } t_2 \ \chi \ t_3 \text{ then } t_4)}{(l :: t_1 \text{ then } l :: t_2 \ \chi \ l :: t_3 \text{ then } l :: t_4)}$$

$$\frac{(t_1 \text{ then } t_2 \ \chi \ t_3 \text{ then } t_4)}{(t_1 \langle + G \text{ then } t_2 \ \chi \ t_3 \langle + G \text{ then } t_4)} \qquad \frac{(t_1 \text{ then } t_2 \ \chi \ t_3 \text{ then } t_4)}{(G + \rangle t_1 \text{ then } t_2 \ \chi \ G + \rangle t_3 \text{ then } t_4)}$$

$$\frac{(t_1 \text{ then } t_2 \ \chi \ t_3 \text{ then } t_4)}{(t_1 \lfloor P \text{ then } t_2 \lfloor P \ \chi \ t_3 \lfloor P \text{ then } t_4 \lfloor P)} \qquad \frac{(t_1 \text{ then } t_2 \ \chi \ t_3 \text{ then } t_4)}{(P \rfloor t_1 \text{ then } P \rfloor t_2 \ \chi \ P \rfloor t_3 \text{ then } P \rfloor t_4)}$$

$$\frac{(t_1 \text{ then } t_2 \ \chi \ t_3 \text{ then } t_4)}{(t_1 \| t \text{ then } t_2 \lfloor Q \ \chi \ t_3 \lfloor P \text{ then } t_4 \| t)} \qquad \frac{(t_1 \text{ then } t_2 \ \chi \ t_3 \text{ then } t_4)}{(t \| t_1 \text{ then } Q \rfloor t_2 \ \chi \ P \rfloor t_3 \text{ then } t \| t_4)}$$

$$\frac{(t_1 \text{ then } t_2 \ \chi \ t_3 \text{ then } t_4),\ (t'_1 \text{ then } t'_2 \ \chi \ t'_3 \text{ then } t'_4)}{(t_1 \| t'_1 \text{ then } t_2 \| t'_2 \ \chi \ t_3 \| t'_3 \text{ then } t_4 \| t'_4)}$$

Table 7.16: The rules for the concurrency relation.

The axiom identifies the basic diamonds, consisting of two transitions performed by two processes composed in parallel. Then the inductive rules propagate the diamonds in all the possible contexts. In [60], Ferrari and Montanari propose a tile system such that a translation $\{\!|\_|\!\}$ from transitions in *TA* to (freely generated) tiles can be inductively defined with the property that any diamond $(t_1 \text{ then } t_2 \ \chi \ t_3 \text{ then } t_4)$ implies $\{\!|t_1|\!\} \cdot \{\!|t_2|\!\} = \{\!|t_3|\!\} \cdot \{\!|t_4|\!\}$. Here we give a graphical representation of their tile system, using *hypergraphs* to model configurations and observations.[4] The labels of horizontal hyperarcs are taken over the signature

$$\Sigma_S = \{+ : 2 \longrightarrow 1, ! : 0 \longrightarrow 1, \kappa : 1 \longrightarrow 0\} \cup \{\mu_h : 1 \longrightarrow 1 \mid \mu \in Act\},$$

and vertical hyperarcs are labelled over the signature

$$\Sigma_D = \{T : 2 \longrightarrow 2\} \cup \{\mu_v : 1 \longrightarrow 1 \mid \mu \in Act\}.$$

Notice that each hyperarc is labelled with an operator whose arity and multiplicity exactly matches the number of source and target nodes. Each node intuitively represents a place where actions may occur, i.e., a location.

We briefly comment on the rules[5] of the *strong tile system* for the simple process calculus of Definition 7.6.1.

[4] Each *hyperarc* is represented with a labelled box connected to its source and target nodes. Unlabelled arcs represent *sharing* (or *aliasing*).

[5] The only auxiliary tiles needed here are permutation tiles.

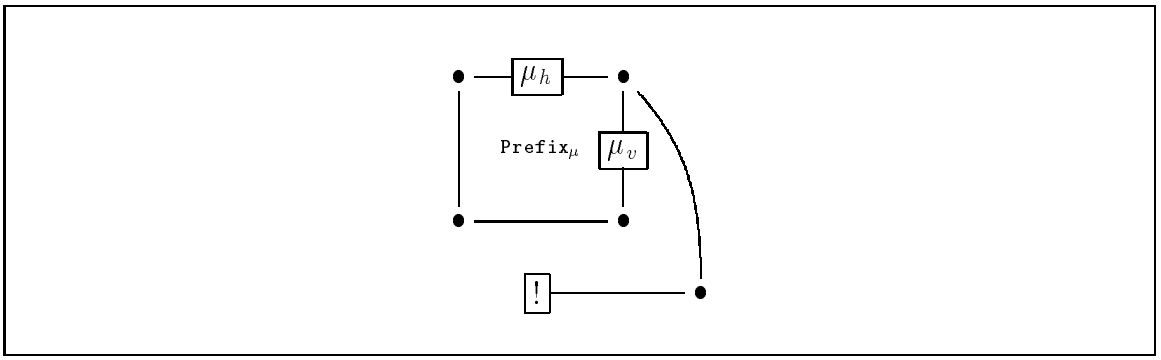


Figure 7.10: Tile for the action prefix in located CCS.

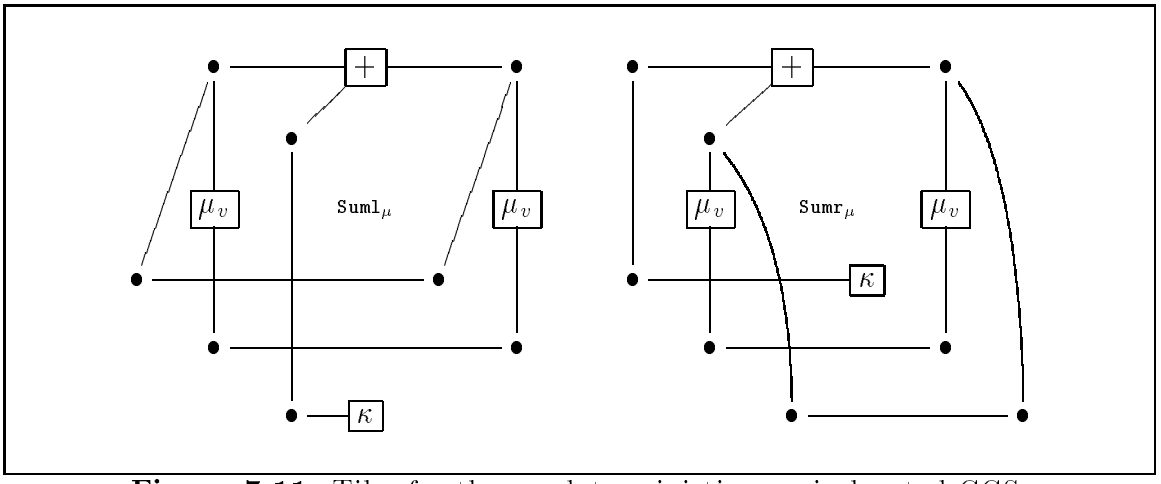


Figure 7.11: Tiles for the nondeterministic sum in located CCS.

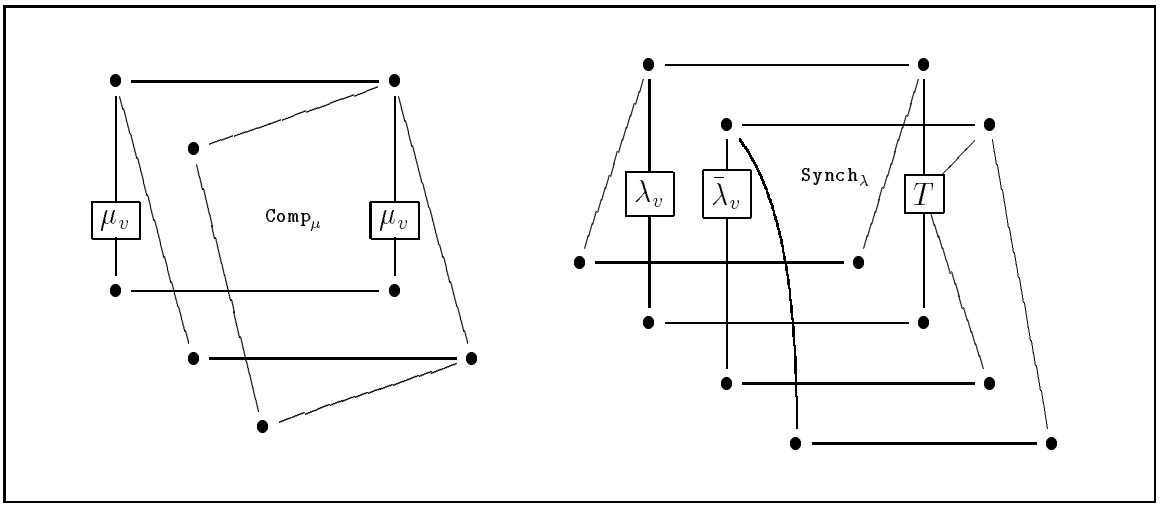


Figure 7.12: Asynchronous and synchronized communications in located CCS.

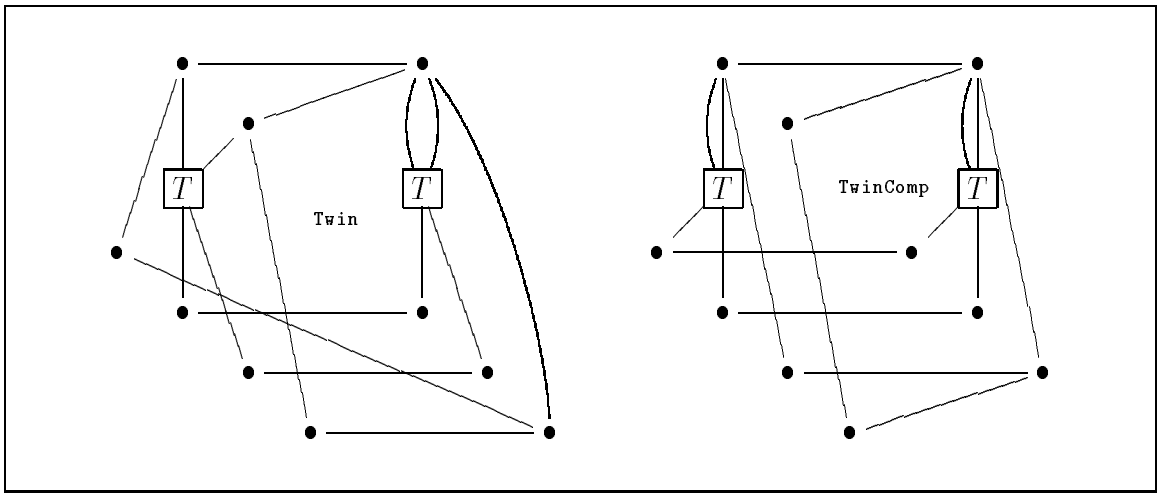


Figure 7.13: Propagation of synchronized communications in located CCS.

The first rule (Figure 7.10) simply states that a "prefix" hyperarc can execute the corresponding action evolving to a new location. The link to the old location is maintained, because other agents could be already attached there.

Two rules are needed to deal with the nondeterministic sum (see Figure 7.11). Whenever the "left" ("right") process makes a move, then the other process is eliminated via a nil binding κ. The locations promoted by the evolving process are propagated forward.

The rule $\texttt{Comp}_{\mu}$ is the most important (Figure 7.12). It states that if two processes share the same location and one of them is making a move, then its subprocesses will be allocated to a new location, whereas the parallel process will remain linked to the old location. We do not need to distinguish between a $\texttt{Compl}_{\mu}$ (left) and a $\texttt{Compr}_{\mu}$ (right), because they can be obtained one from the other using auxiliary permutation tiles.

The rule $\texttt{Synch}_{\lambda}$ allows synchronizing two parallel complementary moves. The resulting action T states that the two "new" locations will be both correlated to the two "old" locations where the complementary actions took place. Then two cases must be taken into account (see Figure 7.13). The two rules show how the T events propagate through the sharing. The `Twin` rules state that if the two synchronizing processes were allocated in the same place, then the synchronization is possible, but only one copy of that "old" location has to be maintained. Note that a rule such as `SynchComp` (Figure 7.14) is not needed here because the correct way to synchronize the processes yielding T would consist in using $\texttt{Comp}_{\lambda}$ and then $\texttt{Synch}_{\lambda}$.

As we have done in the example of finite CCS, also in the Maude implementation of this simple calculus, we "rotate" the rules moving observations to the horizontal dimension and configurations to the vertical one, then stretching the tiles to ordinary rewrite rules on mixed (vertical and horizontal) structure. Moreover, sharing is directly modelled with multiple pointers to the same node name (instead of using a special hyperarc ∇ and imposing the coherence axioms).

Permutations are avoided as well, through explicit name management. Let us take a standard denumerable set of nodes $\{\texttt{n}(i) \mid i \in \mathbb{N}\}$ (sort `Node`). The operator `ls` constructs lists of nodes (sort `NodeList`), where the empty list is denoted by `nil`. Similarly, the operator `set` constructs sets of nodes (sort `NodeSet`), the constant `empty` representing the empty set of nodes.

```
mod LOCCCS is protecting MACHINE-INT.
sorts Node NodeList NodeSet .
subsorts Node < NodeSet NodeList .
op n : MachineInt -> Node .
op nil : -> NodeList .
op ls : NodeList NodeList -> NodeList [assoc id: nil] .
op empty : -> NodeSet .
op set : NodeSet NodeSet -> NodeSet [assoc comm id: empty] .

var E : Node .
eq set(E,E) = E  .
```

A hyperarc (sort `Edge`) is then a triple consisting of a list of source nodes, a label (of sort `Label`) and a list of target nodes. The operator `edge` allows constructing generic hyperarcs. A hypergraph (sort `EdgeMSet`) is just a (multi)set collection of hyperarcs (constructor `ms` and neutral element `zero`). We make use of a top operator `top` to mark the whole actual configuration of the system (sort `State`).

```
sorts Label Edge EdgeMSet State .
subsort Edge < EdgeMSet .
op edge : NodeList Label NodeList -> Edge .
op zero : -> EdgeMSet .
op ms : EdgeMSet EdgeMSet -> EdgeMSet [assoc comm id: zero] .
op top : EdgeMSet -> State .
```

We define also several operators that allow extracting information of various kinds.

```
*** proj(NL) returns the set of nodes contained
*** in the node list NL
op proj : NodeList -> NodeSet .
*** sources(MS) returns the set of source nodes contained
*** in the hypergraph MS
op sources : EdgeMSet -> NodeSet .
*** targets(MS) returns the set of target nodes contained in MS
op targets : EdgeMSet -> NodeSet .
*** vertices(MS) returns the set of nodes contained in MS
op vertices : EdgeMSet -> NodeSet .
```

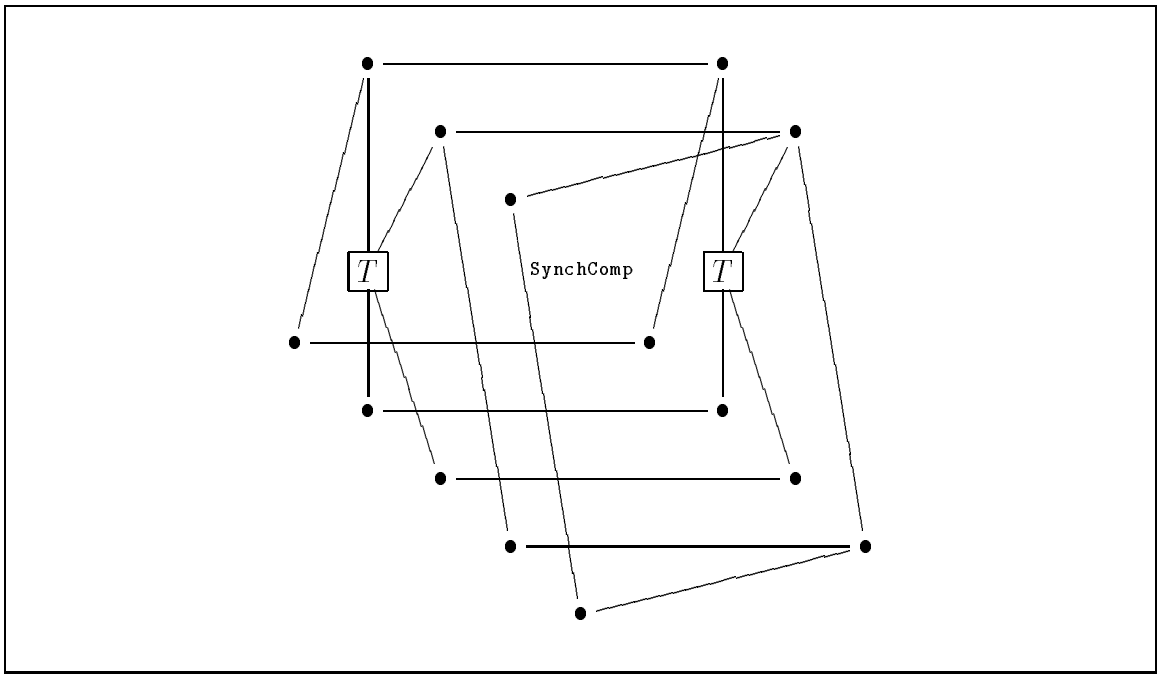


Figure 7.14: Unnecessary tile in located CCS.

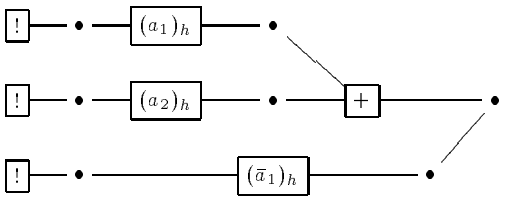
```
*** isIn(E,ES) = true iff the node E is in the set of nodes ES
op isIn : Node NodeSet -> Bool .
*** children(E,MS) returns the set of chidren of node E
*** in the hypergraph MS
op children : Node EdgeMSet -> NodeSet .
*** desc(E,MS) returns the set of descendants of node E
*** in the hypergraph MS
op desc : Node EdgeMSet -> NodeSet .
op descaux : NodeSet EdgeMSet NodeSet Node -> NodeSet .

vars E' : Node .
     ES ES' : NodeSet .
     EL EL' : NodeList .
     MS : EdgeMSet .
     F  : Label .

eq isIn(E,empty) = false .
eq isIn(E,E) = true .
ceq isIn(E,E') = false if E =/= E' .
eq isIn(E,set(E,ES)) = true .
ceq isIn(E,set(E',ES)) = isIn(E,ES) if E =/= E' .

eq proj(nil) = empty .
eq proj(E) = E .
eq proj(ls(E,EL)) = set(E,proj(EL)) .
```

```
eq sources(zero) = empty .
eq sources(edge(EL,F,EL')) = proj(EL) .
eq sources(ms(edge(EL,F,EL'),MS)) =
   set(proj(EL),sources(MS)) .

eq targets(zero) = empty .
eq targets(edge(EL,F,EL')) = proj(EL') .
eq targets(ms(edge(EL,F,EL'),MS)) =
   set(proj(EL'),targets(MS)) .

eq vertices(MS) = set(sources(MS),targets(MS)) .

eq children(E,zero) = empty .
ceq children(E,edge(EL,F,EL')) = empty
    if not(isIn(E,proj(EL))) .
ceq children(E,edge(EL,F,EL')) = proj(EL')
    if isIn(E,proj(EL)) .
ceq children(E,ms(edge(EL,F,EL'),MS)) = children(E,MS)
    if not(isIn(E,proj(EL))) .
ceq children(E,ms(edge(EL,F,EL'),MS)) =
    set(proj(EL'),children(E,MS))
    if isIn(E,proj(EL)) .

eq desc(E,MS) = descaux(children(E,MS),MS,empty,E) .
eq descaux(empty,MS,ES,E') = ES .
ceq descaux(E,MS,ES,E') = ES
    if isIn(E,set(E',ES)) .
ceq descaux(E,MS,ES,E') =
    descaux(children(E,MS),MS,set(E,ES),E')
    if not(isIn(E,set(E',ES))) .
ceq descaux(set(E,ES'),MS,ES,E') = descaux(ES',MS,ES,E')
    if isIn(E,set(E',ES)) .
ceq descaux(set(E,ES'),MS,ES,E') =
    descaux(set(children(E,MS),ES'),MS,set(E,ES),E')
    if not(isIn(E,set(E',ES))) .
```

The label of a hyperarc can be either of sort `HSign` (associated to Σ_S), or of sort `VSign` (associated to Σ_D). We fix a denumerable set of basic actions `a(`i`)` and their complementary actions `bar(a(`i`))` (sort `Channel`) for $i \in \mathbb{N}$, together with a special action `tau`. Given an action μ we denote the associated prefix operators μ_h in Σ_S and μ_v in Σ_D respectively by `h(`μ`)` and `v(`μ`)`. Regarding the other operators, $+$ is denoted by `plus`, $!$ is denoted by `dis`, κ is denoted by `codis` and T is denoted by `t`. A special horizontal operator `alias` is introduced to propagate possible renamings caused by rule $\texttt{Suml}_\mu$ and $\texttt{Sumr}_\mu$.

```
sorts Channel Act HSign VSign .
subsort Channel < Act .
subsorts HSign VSign < Label .

op a : MachineInt -> Channel .
op bar : Channel -> Channel .
op tau : -> Act .

op v : Act -> VSign .
op t : -> VSign .

op h : Act -> HSign .
op plus : -> HSign .
op dis : -> HSign .
op codis : -> HSign .
op alias : -> HSign .

vars BA : Channel .
     Ea : Node .
     EL'' : NodeList .

eq bar(bar(BA)) = BA .
eq ms(edge(EL,F,Ea),edge(Ea,alias,E)) =
   ms(edge(EL,F,E),edge(Ea,alias,E)) .
eq ms(edge(EL,F,ls(Ea,EL')),edge(Ea,alias,E)) =
   ms(edge(EL,F,ls(E,EL')),edge(Ea,alias,E)) .
eq ms(edge(EL,F,ls(EL',Ea,EL'')),edge(Ea,alias,E)) =
   ms(edge(EL,F,ls(EL',E,EL'')),edge(Ea,alias,E)) .
eq ms(edge(EL,F,ls(EL',Ea)),edge(Ea,alias,E)) =
   ms(edge(EL,F,ls(EL',E)),edge(Ea,alias,E)) .
eq ms(edge(nil,dis,E), edge(nil,dis,E)) = edge(nil,dis,E) .
eq ms(edge(E,codis,nil),edge(E,codis,nil)) = edge(E,codis,nil) .
ceq top(ms(MS,edge(Ea,alias,E))) = top(MS)
    if not(isIn(Ea,vertices(MS))) .
```

EXAMPLE 7.6.1 *As an example we show the hypergraph and its Maude representation (assuming a standard procedure assigning names to nodes, for which we only give an intuitive description) associated with the process* $(a_1.nil + a_2.nil)|\bar{a}_1.nil$*:*



```
ms(edge(nil,dis,n(1)), edge(n(1),h(a(1)),n(2)),
   edge(nil,dis,n(3)), edge(n(3),h(a(2)),n(4)), edge(ls(n(2),n(4)),plus,n(5)),
   edge(nil,dis,n(6)), edge(n(6),h(bar(a(1))),n(7)), edge(n(7),alias,n(5))) .
```

We are now ready to translate the tiles into rewrite rules. Since we are interested in applying the nondeterministic strategies shown in Section 7.3.2, we use the same label `step` for all the rules.

```
vars BA' : Channel .
     Mu : Act .
     Eb : Node .
*** Prefix(Mu)
rl [step] : top(ms(edge(E',h(Mu),E),MS)) =>
            top(ms(edge(E',v(Mu),E),
                   edge(nil,dis,E),MS)) .
*** Suml(Mu)
rl [step] : top(ms(edge(E',v(Mu),Ea),
                   edge(ls(Ea,Eb),plus,E),MS)) =>
            top(ms(edge(Eb,codis,nil),
                   edge(E',v(Mu),E),
                   edge(Ea,alias,E),MS)) .
*** Sumr(Mu)
rl [step] : top(ms(edge(E',v(Mu),Ea),
                   edge(ls(Eb,Ea),plus,E),MS)) =>
            top(ms(edge(Eb,codis,nil),
                   edge(E',v(Mu),E),
                   edge(Ea,alias,E),MS)) .
*** Synch(BA)
crl [step] : top(ms(edge(E,v(BA),Ea),
                    edge(E',v(BA'),Eb),MS)) =>
             top(ms(edge(ls(E,E'),t,ls(Ea,Eb)),MS))
    if bar(BA) == BA' .
```

In this representation, the other tiles become either trivial or special cases of the previous ones, and therefore are omitted. The problem is that not all the rewritings are correct: we have to filter computations. This can be done at the meta-level using the strategies for collecting rewritings. All that is needed is a good notion of *success*. In particular, we have to check if the actual state is acyclic and decomposable as an hypergraph with labels in `HSign` followed by an hypergraph with labels in `VSign`. We define the predicate `ok` as follows:

```
op ok : State -> Bool .
op okHV : EdgeMSet -> Bool .
op acyclic : EdgeMSet -> Bool .
```

```
op acycaux : NodeSet EdgeMSet -> Bool .
*** disjoint(ES,ES') = true iff ES ∩ ES' = ∅
op disjoint : NodeSet NodeSet -> Bool .
*** horiz(MS) = true iff all the hyperarcs of MS have
*** labels in HSign
op horiz : EdgeMSet -> Bool .

vars H : HSign .
     V : VSign .
     EL''' : NodeList .

eq horiz(zero) = true .
eq horiz(edge(EL,H,EL')) = true .
eq horiz(edge(EL,V,EL')) = false .
eq horiz(ms(MS,edge(EL,H,EL'))) = horiz(MS) .
eq horiz(ms(MS,edge(EL,V,EL'))) = false .
eq  disjoint(empty,ES') = true .
ceq disjoint(E,ES') = true
    if not(isIn(E,ES')) .
ceq disjoint(E,ES') = false
    if isIn(E,ES') .
ceq disjoint(set(E,ES),ES') = disjoint(ES,ES')
    if not(isIn(E,ES')) .
ceq disjoint(set(E,ES),ES') = false
    if isIn(E,ES') .
eq  acyclic(MS) = acycaux(sources(MS),MS) .
eq  acycaux(empty,MS) = true .
ceq acycaux(E,MS) = true
    if disjoint(E,desc(E,MS)) .
ceq acycaux(E,MS) = false
    if not(disjoint(E,desc(E,MS))) .
ceq acycaux(set(E,ES),MS) = acycaux(ES,MS)
    if disjoint(E,desc(E,MS)) .
ceq acycaux(set(E,ES),MS) = false
    if not(disjoint(E,desc(E,MS))) .
ceq okHV(ms(MS,edge(EL,V,EL'))) = okHV(MS)
    if disjoint(proj(EL'),sources(MS)) .
ceq okHV(MS) = true
    if horiz(MS) .
ceq okHV(ms(MS,edge(EL,V,EL'),edge(EL'',H,EL'''))) = false
    if not(disjoint(proj(EL'),proj(EL''))) .
ceq ok(top(MS)) = true
    if and(okHV(MS),acyclic(MS)) .
endm
```

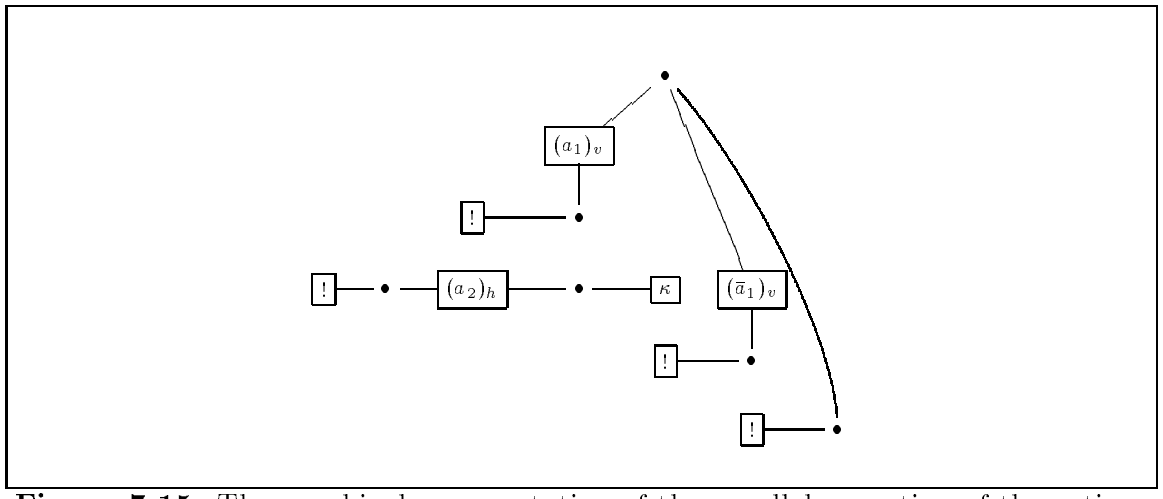


Figure 7.15: The graphical representation of the parallel execution of the actions a_1 and $\bar{a}_1$ in the process $(a_1.nil + a_2.nil)|\bar{a}_1.nil$.

It can be proved that successful states reachable in `LOCCCS` starting from the representation of a process P are exactly the behaviours of P in the tile system of [60].

EXAMPLE 7.6.2 *We show the result of a computation in* `ND-SEM[TREE[LOCCCS]]`*, collecting the successful states reachable from the hypergraph representation of the process* $(a_1.nil+a_2.nil)|\bar{a}_1.nil$ *illustrated in the Example 7.6.1. We use a meta-meta-query to collect all the possible (topmost) solutions.*

```
Maude> rew allRew('rewWith['_[_]['’top, '_[_]['’ms, '_,_[
          '_[_]['’edge, '_,_['’nil, ’’dis, '_[_]['’n,’’1]]],
          '_[_]['’edge, '_,_['’nil, ’’dis, '_[_]['’n,’’3]]],
          '_[_]['’edge, '_,_['’nil, ’’dis, '_[_]['’n,’’6]]],
          '_[_]['’edge, '_,_['_[_]['’n,’’1],
                            '_[_]['’h,'_[_]['’a, ’’1]],
                            '_[_]['’n,’’2]]],
          '_[_]['’edge, '_,_['_[_]['’n,’’3],
                            '_[_]['’h,'_[_]['’a, ’’2]],
                            '_[_]['’n,’’4]]],
          '_[_]['’edge, '_,_['_[_]['’n,’’6],
                            '_[_]['’h,'_[_]['’bar,'_[_]['’a, ’’1]]],
                            '_[_]['’n,’’7]]],
          '_[_]['’edge, '_,_['_[_]['’ls, '_,_['_[_]['’n,’’4], '_[_]['’n,’’2]]],
                            ’’plus,
                            '_[_]['’n,’’5]]],
          '_[_]['’edge, '_,_['_[_]['’n,’’7], ’’alias, '_[_]['’n,’’5]]]
          ]]], 'nondet['’step]], 'aux) .
```

The result is a sequence of meta-representations of terms in `TREE[LOCCCS]`*, each of the kind* `rewWith(..., idle)`*, where the actions that have been executed from the initial process can be easily detected looking at the edges containing the* `v` *operator.*

```
rewrites: 104970 in 2819ms cpu (2829ms real) (37223 rewrites/second)
result TermSequence:
seq('rewWith[('_[_]['']top, ('_[_]['']ms, ('_,_[
    ('_[_]['']edge, ('_,_['']nil, ''dis, ('_[_]['']n, ''1])])]),
    ('_[_]['']edge, ('_,_['']nil, ''dis, ('_[_]['']n, ''3])])]),
    ('_[_]['']edge, ('_,_['']nil, ''dis, ('_[_]['']n, ''5])])]),
    ('_[_]['']edge, ('_,_['']nil, ''dis, ('_[_]['']n, ''6])])]),
    ('_[_]['']edge, ('_,_[('_[_]['']n, ''1]),
                        ('_[_]['']v, ('_[_]['']a, ''1])]),
                        ('_[_]['']n, ''5])])]),
    ('_[_]['']edge, ('_,_[('_[_]['']n, ''3]),
                        ('_[_]['']h, ('_[_]['']a, ''2])]),
                        ('_[_]['']n, ''4])])]),
    ('_[_]['']edge, ('_,_[('_[_]['']n, ''4]), ''codis, ''nil])]),
    ('_[_]['']edge, ('_,_[('_[_]['']n, ''6]),
                        ('_[_]['']h, ('_[_]['']bar, ('_[_]['']a, ''1])])]),
                        ('_[_]['']n, ''5])])])
    ])])]), 'idle],

    'rewWith[('_[_]['']top, ('_[_]['']ms, ('_,_[
    ('_[_]['']edge, ('_,_['']nil, ''dis, ('_[_]['']n, ''1])])]),
    ('_[_]['']edge, ('_,_['']nil, ''dis, ('_[_]['']n, ''3])])]),
    ('_[_]['']edge, ('_,_['']nil, ''dis, ('_[_]['']n, ''5])])]),
    ('_[_]['']edge, ('_,_['']nil, ''dis, ('_[_]['']n, ''6])])]),
    ('_[_]['']edge, ('_,_[('_[_]['']n, ''1]),
                        ('_[_]['']h, ('_[_]['']a, ''1])]),
                        ('_[_]['']n, ''2])])]),
    ('_[_]['']edge, ('_,_[('_[_]['']n, ''2]), ''codis, ''nil])]),
    ('_[_]['']edge, ('_,_[('_[_]['']n, ''3]),
                        ('_[_]['']v, ('_[_]['']a, ''2])]),
                        ('_[_]['']n, ''5])])]),
    ('_[_]['']edge, ('_,_[('_[_]['']n, ''6]),
                        ('_[_]['']v, ('_[_]['']bar, ('_[_]['']a, ''1])])]),
                        ('_[_]['']n, ''5])])])
    ])])]), 'idle],

    'rewWith[('_[_]['']top, ('_[_]['']ms, ('_,_[
    ('_[_]['']edge, ('_,_['']nil, ''dis, ('_[_]['']n, ''1])])]),
    ('_[_]['']edge, ('_,_['']nil, ''dis, ('_[_]['']n, ''3])])]),
    ('_[_]['']edge, ('_,_['']nil, ''dis, ('_[_]['']n, ''5])])]),
    ('_[_]['']edge, ('_,_['']nil, ''dis, ('_[_]['']n, ''6])])]),
    ('_[_]['']edge, ('_,_[('_[_]['']n, ''1]),
                        ('_[_]['']h, ('_[_]['']a, ''1])]),
                        ('_[_]['']n, ''2])])]),
    ('_[_]['']edge, ('_,_[('_[_]['']n, ''2]), ''codis, ''nil])]),
    ('_[_]['']edge, ('_,_[('_[_]['']n, ''3]),
                        ('_[_]['']v, ('_[_]['']a, ''2])]),
                        ('_[_]['']n, ''5])])]),
    ('_[_]['']edge, ('_,_[('_[_]['']n, ''6]),
                        ('_[_]['']h, ('_[_]['']bar, ('_[_]['']a, ''1])])]),
                        ('_[_]['']n, ''5])])])
    ])])]), 'idle],
```

```
    'rewWith['_[_]''top, ('_[_]''ms, ('_,_[
    ('_[_]''edge, ('_,_[''nil, ''dis, ('_[_]''n, ''1])])]),
    ('_[_]''edge, ('_,_[''nil, ''dis, ('_[_]''n, ''3])])]),
    ('_[_]''edge, ('_,_[''nil, ''dis, ('_[_]''n, ''5])])]),
    ('_[_]''edge, ('_,_[''nil, ''dis, ('_[_]''n, ''6])])]),
    ('_[_]''edge, ('_,_[('_[_]''n, ''1]),
                        ('_[_]''h, ('_[_]''a, ''1])]),
                        ('_[_]''n, ''2])])]),
    ('_[_]''edge, ('_,_[('_[_]''n, ''3]),
                        ('_[_]''h, ('_[_]''a, ''2])]),
                        ('_[_]''n, ''4])])]),
    ('_[_]''edge, ('_,_[('_[_]''n, ''6]),
                        ('_[_]''v, ('_[_]''bar, ('_[_]''a, ''1])])]),
                        ('_[_]''n, ''5])])]),
    ('_[_]''edge, ('_,_[('_[_]''ls, ('_,_[('_[_]''n, ''4]),
                                          ('_[_]''n, ''2])])]),
                        ''plus,
                        ('_[_]''n, ''5])])])
    ])])]), 'idle],

    'rewWith['_[_]''top, ('_[_]''ms, ('_,_[
    ('_[_]''edge, ('_,_[''nil, ''dis, ('_[_]''n, ''1])])]),
    ('_[_]''edge, ('_,_[''nil, ''dis, ('_[_]''n, ''3])])]),
    ('_[_]''edge, ('_,_[''nil, ''dis, ('_[_]''n, ''5])])]),
    ('_[_]''edge, ('_,_[''nil, ''dis, ('_[_]''n, ''6])])]),
    ('_[_]''edge, ('_,_[('_[_]''n, ''1]),
                        ('_[_]''v, ('_[_]''a, ''1])]),
                        ('_[_]''n, ''5])])]),
    ('_[_]''edge, ('_,_[('_[_]''n, ''3]),
                        ('_[_]''h, ('_[_]''a, ''2])]),
                        ('_[_]''n, ''4])])]),
    ('_[_]''edge, ('_,_[('_[_]''n, ''4]), ''codis, ''nil])]),
    ('_[_]''edge, ('_,_[('_[_]''n, ''6]),
                        ('_[_]''v, ('_[_]''bar, ('_[_]''a, ''1])])]),
                        ('_[_]''n, ''5])])])
    ])])]), 'idle])
```

The last solution is represented in Figure 7.15.

Six possible combinations have been found, only the synchronization between a_1 and $\bar{a}_1$ (represented in Figure 7.16) is missing, because it can only be reached after having visited a successful state, i.e., it is not a topmost solution.

The query below (starting from the last one of the previous solutions) leads to the detection of the synchronization (note the label `t` *in the answer).*

```
Maude> rew allRew('rewWith[('_[_]''top, ('_[_]''ms, ('_,_[
          ('_[_]''edge, ('_,_[''nil, ''dis, ('_[_]''n, ''1])])]),
          ('_[_]''edge, ('_,_[''nil, ''dis, ('_[_]''n, ''3])])]),
          ('_[_]''edge, ('_,_[''nil, ''dis, ('_[_]''n, ''5])])]),
          ('_[_]''edge, ('_,_[''nil, ''dis, ('_[_]''n, ''6])])]),
          ('_[_]''edge, ('_,_[('_[_]''n, ''1]),
                               ('_[_]''v, ('_[_]''a, ''1])]),
```

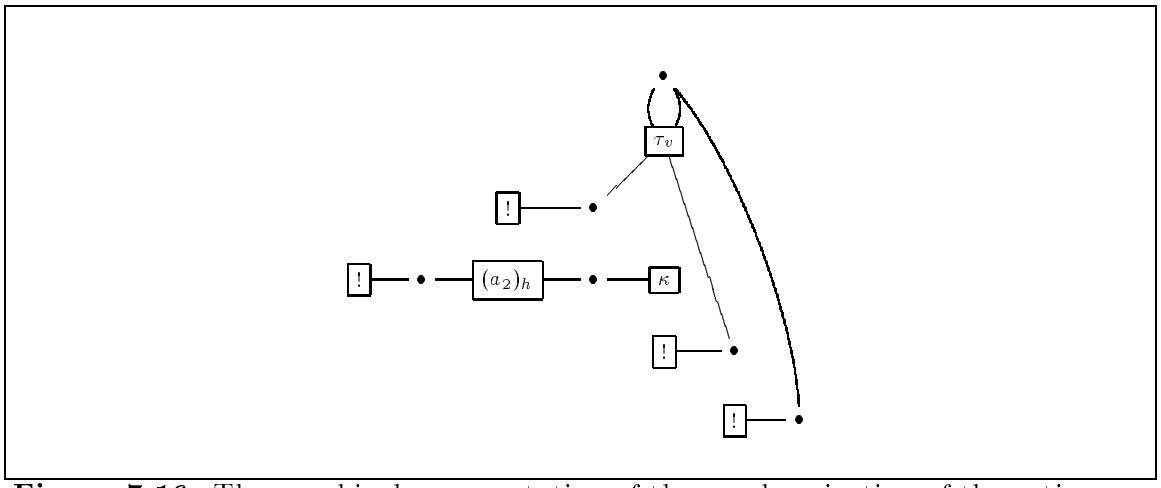


Figure 7.16: The graphical representation of the synchronization of the actions a_1 and $\bar{a}_1$ in the process $(a_1.nil + a_2.nil)|\bar{a}_1.nil$.

```
                        ('_[_]['']n, ''5])])]),
          ('_[_]['']edge, ('_,_[('_[_]['']n, ''3]),
                        ('_[_]['']h, ('_[_]['']a, ''2])]),
                        ('_[_]['']n, ''4])])]),
          ('_[_]['']edge, ('_,_[('_[_]['']n, ''4]), ''codis, ''nil])]),
          ('_[_]['']edge, ('_,_[('_[_]['']n, ''6]),
                        ('_[_]['']v, ('_[_]['']bar, ('_[_]['']a, ''1])])]),
                        ('_[_]['']n, ''5])])])
          ])])]), ('nondet['']step])], 'aux) .

rewrites: 4496 in 129ms cpu (139ms real) (34584 rewrites/second)
result Term:
'rewWith[('_[_]['']top, ('_[_]['']ms, ('_,_[
         ('_[_]['']edge, ('_,_['']nil, ''dis, ('_[_]['']n, ''1])])]),
         ('_[_]['']edge, ('_,_['']nil, ''dis, ('_[_]['']n, ''3])])]),
         ('_[_]['']edge, ('_,_['']nil, ''dis, ('_[_]['']n, ''5])])]),
         ('_[_]['']edge, ('_,_['']nil, ''dis, ('_[_]['']n, ''6])])]),
         ('_[_]['']edge, ('_,_[('_[_]['']n, ''3]),
                        ('_[_]['']h, ('_[_]['']a, ''2])]),
                        ('_[_]['']n, ''4])])]),
         ('_[_]['']edge, ('_,_[('_[_]['']n, ''4]), ''codis, ''nil])]),
         ('_[_]['']edge, ('_,_[('_[_]['']ls, ('_,_[('_[_]['']n, ''1]),
                                        ('_[_]['']n, ''6])])]),
                        ''t,
                        ('_[_]['']ls, ('_,_[('_[_]['']n, ''5]),
                                        ('_[_]['']n, ''5])])])])])
         ])])]), 'idle]
```

In comparison with the successful states reached starting from the same process in the module `ND-SEM[TREE[CCS]]` *(see Example 7.5.5), here the number of solutions is smaller than in the other case, because concurrent (interleaving) computations are identified.*

7.7 Summary

The definition of internal strategies to control nondeterministic rewritings in the tile system translations constitutes the main contribution of this chapter. The importance of similar mechanisms is well-known, and other languages (e.g., ELAN) have built-in constructs to deal with general forms of nondeterminism. Nevertheless, our approach is rather general (it is parametric w.r.t. a user-definable success predicate) and allows the application of several visiting policies, different from the depth-first (with backtracking) algorithms that are usually preferred in a built-in implementation for efficiency reasons, but that could diverge also in the presence of solutions.

Exploiting the relationships between tile logic and rewriting logic, we have defined general meta-strategies for simulating tile system specifications on a rewriting machinery equipped with reflective capabilities. We have implemented such strategies in Maude, and have experimented their application to several case studies.

Our general methodology for modelling tile systems can be summarized by the following steps: (1) translate the tiles into rewrite rules; (2) define, if necessary, a notion of successful computation (if the system is uniform, this can be done just looking at the actual term reached); (3) compute at the meta-level, using the internal strategies that discard wrong computations, until a successful answer is reached. This procedure has been fully illustrated for process tile logic using the example of located CCS, and for term tile logic by considering the examples of full CCS and finite CCS. For each model, we have tested the computations of simple processes. Our experiments are encouraging, because Maude seems to offer a good trade-off between rewriting kernel efficiency and layer-swapping management (from terms to their meta-representations and vice versa).

Chapter 8

Conclusions

We have investigated the foundational aspects of several tile models of computation, giving the original formal basis for their analysis and illustrating their application to many examples at different levels of complexity. The distinguishing feature of the tile models we have discussed is that configurations and observations have the same structure.

In the simpler case, where both configurations and observations are Petri net markings, this has generated an interesting net model, called zero-safe nets, which is naturally equipped with compositionality features (e.g., transition synchronization) that are missing in the standard treatment of Petri nets. This investigation has also provided some new insights in the theory of net abstraction/refinement, since we have shown that the abstract models of zero-safe nets are just ordinary Petri nets, whose construction yields a coreflection.

The other tile models that we have considered are based on richer structures, where suitable auxiliary tiles provide all the consistent rearrangements of the underlying structure shared between the two dimensions.

We have investigated the theoretical and implementation aspects of a general methodology for the modular specification of reactive systems using two such formats, called process and term tile logic. In particular, we have characterized their categorical models (respectively symmetric strict monoidal double categories and cartesian double categories) using generalized natural transformations as an alternative approach to the internal construction of [69].

This has given the basis for establishing a correspondence between tile logic and rewriting logic by relating their categorical models. As a consequence, the mapping of tile logic into rewriting logic becomes effective provided that we have the possibility to (meta) control the rewritings.

Indeed, using the internal strategies of Maude, we have defined a general meta-layer that can help the user to discard wrong computations and to analyze the successful deductions in tile logic.

These results yield a methodology for the executable specification of reactive systems in tile logic that consists of the following steps:

1. Define a tile model of computation of the given system, employing adequate mathematical structures to represent configurations and observations in such a way that the intrinsic modularity and the coordination mechanisms of tiles are fully exploited. Usually, this requires:

   (a) The identification of the basic interfaces of the system, which become the sorts of both the configuration and observation signatures.

   (b) The identification of the basic constructors of configurations. They have to reflect the basic functionalities provided by the system. In the case of process calculi the constructors are essentially the operators of the process signature.

   (c) The identification of the basic observables. Their choice often depends on the relevant aspects of computation that one is interested in. Moreover, the observation constructors must reflect the interoperability protocols of state components.

   (d) The choice of the proper algebraic structure where to interpret the basic configurations and observations. Such structures can range from terms to term graphs, to acyclic hypergraphs, to monoids, just to name a few.

   (e) The specification of the tiles that define the reactive behaviour of configurations, using a suitable tile model that is able to deal with the algebraic structures of configurations and observations (we remind that auxiliary tiles are freely generated).

   Often it is necessary to iterate such procedure several times to make sure that all the choices are consistent.

2. Translate tiles into rewrite rules.

3. Define, if necessary, a notion of successful computation. Such notion could depend on the whole computation proof, or just on a suitable abstraction. The optimal choice is possible if and only if the tile system is *uniform* and consists in the verification of simple constraints over the target of the rewriting computation.

4. Compute at the meta-level, using the internal strategies that discard wrong computations, until a successful answer (if any) is obtained.

This procedure has been fully illustrated for process tile logic by the example of located CCS, and for term tile logic by the examples of full and finite CCS. For each model we have tested simple processes. We believe that the possibility to have some running tool to experiment with is very important for the development and testing of tile specifications. Our experiments are encouraging because Maude seems to offer a good trade-off between rewriting kernel efficiency and layer-swapping management (from terms to their meta-representations and vice versa).

The original and innovative aspects of this thesis can be summarized by the following *key sentences*:

- Synchronization mechanism between net transitions.
- Behavioural abstraction/refinement of nets as universal constructions.
- Tile logic for net processes.
- Term tile logic.
- Symmetric strict monoidal double categories.
- Cartesian double categories with consistently chosen products.
- 2EVH-categories.
- Internal strategies for nondeterministic rewriting systems and their application to tile systems.
- Methodology for the executable specification of reactive systems.

Future Work

In this thesis, we have focused on tile models where both configurations and observations share the same underlying structure: commutative monoids for zero-safe nets, symmetric strict monoidal categories for process tile logic, and cartesian categories for term tile logic. Several lines of future research are still open. For example, it has still to be clarified how to deal with tile systems that require different non-trivial structures on both dimensions (e.g., what are the auxiliary tiles?). This is particularly important if one is not interested in the flat model, but instead requires an algebraic setting for the proofs.

Another interesting issue consists of the analysis of tile systems where both configurations and observations have the same (auxiliary) structure, but not all the tiles relating that structure are introduced in the systems.

An even more interesting issue is the development of a class of tile models equipped with *higher-order features*. This could be done by considering the analogous of cartesian closed categories in the setting of double categories. We aim at exploiting this framework for a powerful type paradigm for reactive calculi, where name extrusion is handled in an elegant way. For example, tile systems for reduction in λ-calculus, logic programming, and calculi with mobility seem to require either non-homogeneous or higher-order structures on configurations and observations.

Another open problem consists in the development of a sort of meta-theory of tile formats, able to ensure that certain properties are verified if the specification respects suitable format constraints. Also the relationship with other formalisms as e.g., *control structures* [112], deserves further researches.

Vous ne l'avez même pas lu, n'est-ce pas?
J'avais mis les pages 36, 123 et 247 à l'envers,
elles y sont toujours.

— DANIEL PENNAC, La Petite Marchande de Prose

Bibliography

[1] S. Abramsky, S. Gay, and R. Nagarajan, Interaction categories and the foundations of typed concurrent programming, in: M. Broy, Ed., *Deductive Program Design: Proceedings of the 1994 Marktoberdorf Summer School*, Nato ASI Series, 403–442, Springer-Verlag (1994).

[2] S.O. Anderson and J. Power, A representable approach to finite nondeterminism, *Theoretical Computer Science* **177**, 3–25 (1997).

[3] M. Barr and C. Wells, *Category Theory for Computing Science*, Prentice Hall Series in Computer Science, 2nd ed., London (1995). 1st ed., New York (1990).

[4] A. Bastiani and C. Ehresmann, Multiple functors I: Limits relative to double categories, *Cahiers de Topologie ed Géométrie Différentielle* **15**(3), 545–621 (1974).

[5] J.A. Bergstra and J.W. Klop, Process algebra for synchronous comunication, *Information and Control* **60**, 109–137 (1984).

[6] J.A. Bergstra and J.W. Klop, Algebra of communicating processes with abstraction, *Theoretical Computer Science* **37**(1), 77–121 (1985).

[7] K.L. Bernstein, A congruence theorem for structured operational semantics of higher-order languages, in *Proceedings 13th LICS Symposium*, IEEE Computer Society Press (1998).

[8] E. Best and R. Devillers, Sequential and concurrent behaviour in Petri net theory, *Theoretical Computer Science* **55**(1), 87–136 (1987).

[9] E. Best, R. Devillers, and J. Esparza, General refinement and recursion for the Petri box calculus, in: P. Enjalbert, A. Finkel, and K.W. Wagner, Eds., *Proceedings STACS'93*, *LNCS* **665**, 130–140, Springer-Verlag (1993).

[10] E. Best, R. Devillers, and J. Hall, The box calculus: A new causal algebra with multi-label communication, in: G. Rozenberg, Ed., *Advances in Petri Nets 1992*, *LNCS* **609**, 21–69, Springer-Verlag (1992).

[11] B. Bloom, S. Istrail, and A.R. Meyer, Bisimulation can't be traced, *Journal of the ACM* **42**(1), 232–268 (January 1995).

[12] P. Borovanský, C. Kirchner, and H. Kirchner, Controlling rewriting by rewriting, in: J. Meseguer, Ed., *Proceedings 1st WRLA'96*, *ENTCS* **4**, 21 pages, Elsevier (1996).

[13] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek, ELAN: A logical framework based on computational systems, in: J. Meseguer, Ed., *Proceedings 1st WRLA'96*, *ENTCS* **4**, 16 pages, Elsevier (1996).

[14] G. Boudol and I. Castellani, Flow models of distributed computations: Three equivalent semantics for CCS, *Information and Computation* **114**(2), 247–314 (1994).

[15] G. Boudol, I. Castellani, M. Hennessy, and A. Kiehn, Observing localities, *Theoretical Computer Science* **114**, 31–61 (1993).

[16] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer, Specification and proof in membership equational logic, in: M. Bidoit and M. Dauchet, Eds., *Proceedings of TAPSOFT'97*, *LNCS* **1214**, 67–92, Springer-Verlag (1997).

[17] W. Brauer, R. Gold, and W. Vogler, A survey of behaviour and Equivalence preserving refinements of Petri nets, in: G. Rozenberg, Ed., *Advances in Petri Nets 1990*, *LNCS* **483**, 1–46, Springer-Verlag (1991).

[18] C. Brown and D. Gurr, A categorical linear framework for Petri nets, in *Proceedings 5th LICS Symposium*, 208–218, IEEE Computer Society Press (1990).

[19] R. Bruni, A logic for modular descriptions of asynchronous and synchronized concurrent systems (abstract), in: C. Kirchner and H. Kirchner, Eds., *Proceedings 2nd WRLA'98*, *ENTCS* **15**, 10 pages, Elsevier (1998).

[20] R. Bruni, F. Gadducci, and U. Montanari, Normal forms for partitions and relations, in: J.L. Fiadeiro, Ed., *Proceedings 13th WADT'98*, *LNCS*, Springer-Verlag, to appear.

[21] R. Bruni, J. Meseguer, and U. Montanari, Process and term tile logic, Technical Report SRI-CSL-98-06, SRI International (1998). Also Technical Report TR-98-09, Department of Computer Science, University of Pisa (1998).

[22] R. Bruni, J. Meseguer, and U. Montanari, Internal strategies in a rewriting implementation of tile systems, in: C. Kirchner and H. Kirchner, Eds., *Proceedings 2nd WRLA'98*, *ENTCS* **15**, 20 pages, Elsevier (1998).

[23] R. Bruni, J. Meseguer, and U. Montanari, Implementing tile systems: Some examples from process calculi, in: P. Degano, U. Vaccaro, and G. Pirillo, Eds., *Proceedings 6th ICTCS'98*, World Scientific (1998).

[24] R. Bruni, J. Meseguer, and U. Montanari, Executable tile specifications for process calculi, in: J.-P. Finance, Ed., *Proceedings FASE'99*, *LNCS*, Springer-Verlag, to appear.

[25] R. Bruni, J. Meseguer, and U. Montanari, Symmetric and cartesian double categories as a semantic framework for tile logic, *MSCS*, to appear.

[26] R. Bruni, J. Meseguer, U. Montanari, and V. Sassone, A comparison of Petri net semantics under the collective token philosophy, in: J. Hsiang and A. Ohori, Eds., *Proceedings 4th ASIAN'98*, *LNCS* **1538**, 225–244, Springer-Verlag (1998).

[27] R. Bruni, J. Meseguer, U. Montanari, and V. Sassone, Functorial semantics for Petri nets under the individual token philosophy, submitted for publication (1999).

[28] R. Bruni and U. Montanari, Zero-safe nets, or transition synchronization made simple, in: C. Palamidessi and J. Parrow, Eds., *Proceedings EXPRESS'97*, *ENTCS* **7**, 19 pages, Elsevier (1997).

[29] R. Bruni and U. Montanari, Zero-safe nets: The individual token approach, in: F. Parisi-Presicce, Ed., *Proceedings 12th WADT'97*, *LNCS* **1376**, 122–140, Springer-Verlag (1998).

[30] R. Bruni and U. Montanari, Zero-safe nets: Comparing the collective and the individual token approaches, *Information and Computation*, to appear.

[31] A. Carboni and R.F.C. Walters, Cartesian bicategories I, *Annals of Pure and Applied Algebra* **49**, 11–32 (1987).

[32] V.E. Cazanescu and G. Stefanescu, Towards a new algebraic foundation of flowchart scheme theory, *Fundamenta Informaticae* **13**, 171–210 (1990).

[33] M. Clavel, *Reflection in General Logics and in Rewriting Logic with Applications to the Maude Language*, PhD Thesis. Universidad de Navarra (1998).

[34] M.G. Clavel, F. Duran, S. Eker, P. Lincoln, and J. Meseguer, *An Introduction to Maude (Beta Version)*, SRI International (March 1998).

[35] M.G. Clavel, S. Eker, P. Lincoln, and J. Meseguer, Principles of Maude, in: J. Meseguer, Ed., *Proceedings 1st WRLA'96*, *ENTCS* **4**, 25 pages, Elsevier (1996).

[36] M.G. Clavel, S. Eker, and J. Meseguer, Current design and implementation of the Cafe prover and Knuth-Bendix checker tools, Manuscript, SRI International, (October 1997).

[37] M. Clavel and J. Meseguer, Reflection and strategies in rewriting logic, in: J. Meseguer, Ed., *Proceedings 1st WRLA'96*, *ENTCS* **4**, 24 pages, Elsevier (1996).

[38] M. Clavel and J. Meseguer, Axiomatizing reflective logics and languages, in: G. Kiczales, Ed., *Proceedings Reflection'96*, San Francisco, USA, 263–288 (April 1996).

[39] M. Clavel and J. Meseguer, Internal strategies in a reflective logic, in: B. Gramlich and H. Kirchner, Eds., *Proceedings of the CADE-14 Workshop on Strategies in Automated Deduction*, Townsville, Australia, 1–12 (1997).

[40] F. Corradini and R. De Nicola, Distribution and locality of concurrent systems, in: S. Abiteboul and E. Shamir, Eds., *Proceedings ICALP'94*, *LNCS* **820**, 154–165, Springer-Verlag (1994).

[41] A. Corradini and F. Gadducci, A 2-categorical presentation of term graph rewriting, in: E. Moggi and G. Rosolini, Eds., *Proceedings CTCS'97*, *LNCS* **1290**, 87–105, Springer-Verlag (1997).

[42] A. Corradini and F. Gadducci, An algebraic presentation of term graphs, via gs-monoidal categories, *Applied Categorical Structures*, to appear.

[43] A. Corradini and F. Gadducci, Functorial semantics for multi-algebras, in: J.L. Fiadeiro, Ed., *Proceedings WADT'98*, *LNCS*, Springer-Verlag, to appear.

[44] A. Corradini, F. Gadducci, and U. Montanari, Relating two categorical models of concurrency, in: J. Hsiang, Ed., *Proceedings RTA'95. LNCS* **914**, 225–240, Springer-Verlag (1995).

[45] A. Corradini and U. Montanari, An algebraic semantics for structured transition systems and its application to logic programs, *Theoretical Computer Science* **103**, 51–106 (1992).

[46] P. Darondeau and P. Degano, Causal trees, in: G. Ausiello, M. Dezani-Ciancaglini, and S. Ronchi Della Rocca, Eds., *Proceedings ICALP'89*, *LNCS* **372**, 234–248, Springer-Verlag (1989).

[47] R. Dawson, A forbidden-suborder characterization of binarily-composable diagrams in double categories, *Theory and Applications of Categories* **1**(7), 146–155 (1995).

[48] R. Dawson and R. Paré, General associativity and general composition for double categories, *Cahiers de Topologie ed Géométrie Différentielle Catégoriques* **34**, 57–79 (1993).

[49] P. Degano, J. Meseguer, and U. Montanari, Axiomatizing net computations and processes, in *Proceedings 4th LICS Symposium*, 175-185, IEEE Computer Society Press (1989).

[50] P. Degano, J. Meseguer, and U. Montanari, Axiomatizing the algebra of net computations and processes, *Acta Informatica* **33**(7), 641–667 (October 1996).

[51] P. Degano, R. De Nicola, and U. Montanari, A distributed operational semantics for CCS based on condition/event systems, *Acta Informatica* **26**(1-2), 59–91 (1988).

[52] P. Degano, R. De Nicola, and U. Montanari, Partial ordering descriptions and observations of nondeterministic concurrent processes, in: J.W. de Bakker, W.P. de Roever, and G. Rozenberg, Eds., *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, *LNCS* **354**, 438–466, Springer-Verlag (1989).

[53] G. Denker, J. Meseguer, and C. Talcott, Protocol specification and analysis in Maude, in: N. Heintze and J. Wing, Eds., *Proceedings of Workshop on Formal Methods and Security Protocols*, Indianapolis, Indiana (1998).

[54] J. Desel, On abstraction of nets, in: G. Rozenberg, W. Reisig, and H. Plunnecke, Eds., *Advances in Petri Nets 1991*, *LNCS* **524**, 78–92, Springer-Verlag (1991).

[55] M.C. Eekele, M.J. Plasmeijer, and M.R. Sleep, Eds., *Term Graph Rewriting: Theory and Practice*, Wiley, London (1993).

[56] A. Ehresmann and C. Ehresmann, Multiple functors II: The monoidal closed category of multiple categories, *Cahiers de Topologie ed Géométrie Différentielle* **19**(3), 295–333 (1978).

[57] C. Ehresmann, Catégories structurées: I and II, *Ann. Éc. Norm. Sup.* **80**, 349–426, Paris (1963); III, *Topo. et Géo. Diff.* **5**, Paris (1963).

[58] G.L. Ferrari and U. Montanari, Towards the unification of models for concurrency, in: A. Arnold, Ed., *Proceeding CAAP'90*, *LNCS* **431**, 162–176, Springer-Verlag (1990).

[59] G.L. Ferrari and U. Montanari. A tile-based coordination view of asynchronous π-Calculus, in: I. Prívara and P. Ruzicka, Eds., *Proceedings MFCS'97*, *LNCS* **1295**, 52–70, Springer-Verlag (1997).

[60] G.L. Ferrari and U. Montanari, Tiles for concurrent and located calculi, in: C. Palamidessi and J. Parrow, Eds., *Proceedings of EXPRESS'97*, *ENTCS* **7**, 26 pages, Elsevier (1997).

[61] G.L. Ferrari and U. Montanari, Tile formats for located and mobile systems, *Information and Computation*, to appear.

[62] P. Freyd, Algebra valued functors in general and tensor products in particular, *Coll. Math.* **14**, 89–106 (1966).

[63] P. Freyd and A. Scedrov, *Categories, Allegories*, North-Holland Mathematical Library **39**, North-Holland (1990).

[64] K. Futatsugi and T. Sawada, Cafe as an extensible specification environment. in *Proceedings of the Kunming International CASE Symposium*, Kunming, China (November 1994).

[65] P. Gabriel and F. Ulmer, Lokal präsentierbare kategorien, *Lecture Notes in Mathematics* **221**, Springer-Verlag (1971).

[66] F. Gadducci, *On the Algebraic Approach to Concurrent Term Rewriting*, PhD Thesis TD-96-02, Department of Computer Science, University of Pisa (1996).

[67] F. Gadducci and U. Montanari, Enriched categories as models of computations, in: A. De Santis, Ed., *Proceedings 5th ICTCS'95*, 1–24, World Scientific (1996).

[68] F. Gadducci and U. Montanari, Rewriting rules and CCS, in: J. Meseguer, Ed., *Proceedings 1st WRLA'96*, *ENTCS* **4**, 19 pages, Elsevier (1996).

[69] F. Gadducci and U. Montanari, The tile model, in: G. Plotkin, C. Stirling, and M. Tofte, Eds., *Proof, Language and Interaction: Essays in Honour of Robin Milner*, MIT Press, to appear. Also Technical Report TR-96-27, Department of Computer Science, University of Pisa (1996).

[70] R.J. Van Glabbeek and U. Goltz, Refinement of actions and equivalence notions for concurrent systems, Hildesheimer Informatik Bericht 6/98, Institut fuer Informatik, Universitaet Hildesheim (1998).

[71] R.J. Van Glabbeek and G.D. Plotkin, Configuration structures, in: D. Kozen, Ed., *Proceedings 10th LICS Symposium*, 199–209, IEEE Computer Society Press (1995).

[72] R.J. Van Glabbeek and F. Vaandrager, Petri net models for algebraic theories of concurrency, in: J.W. de Bakker, A.J. Nijman, and P.C. Treleaven, Eds., *Proceedings of PARLE*, *LNCS* **259**, 224–242, Springer-Verlag (1987).

[73] J.A. Goguen, A categorical manifesto, *Mathematical Structures in Computer Science* **1**, 49–67 (1991).

[74] U. Goltz and W. Reisig, The non-sequential behaviour of Petri nets, *Information and Computation* **57**, 125–147 (1983).

[75] R. Gorrieri and U. Montanari, On the implementation of concurrent calculi into net calculi: Two case studies, *Theoretical Computer Science* **141**(1-2), 195–252 (1995).

[76] M. Grandis and R. Paré, Limits in double categories, Dipartimento di Matematica, Universitá di Genova, Italy, Preprint **359** (1997).

[77] J.F. Groote and F. Vaandrager, Structured operational semantics and bisimulation as a congruence, *Information and Computation* **100**, 202–260 (1992).

[78] C.A.R. Hoare, *Communication Sequential Processes*, Prentice-Hall (1985).

[79] A. Jeffrey, Premonoidal categories and a graphical view of programs, Technical report, COGS, University of Sussex (1997).

[80] P. Katis, N. Sabadini, and R.F.C. Walters, Bicategories of processes, *Journal of Pure and Applied Algebra* **115**, 141–178 (1997).

[81] P. Katis, N. Sabadini, and R.F.C. Walters, SPAN(Graph): A categorical algebra of transition systems, in: M. Johnson, Ed., *Proceedings AMAST'97*, *LNCS* **1349**, 307–321, Springer-Verlag (1997).

[82] G.M. Kelly and R.H. Street, Review of the elements of 2-categories, *Lecture Notes in Mathematics* **420**, 75–103, Springer-Verlag (1974).

[83] A. Kiehn, *A Structuring Mechanism for Petri Nets*, PhD Thesis, TUM-I8902, Institut für Informatik, Technische Universität München (1989).

[84] A. Kiehn, Petri net systems and their closure properties, in: G. Rozenberg, Ed., *Advances in Petri Nets 1989*, *LNCS* **424**, 306–328, Springer-Verlag (1990).

[85] A. Kiehn, Local and global causes, *Acta Informatica* **31**, 697–718 (1994).

[86] C. Kirchner and H. Kirchner, Eds., *Proceedings Second International Workshop on Rewriting Logic and Applications*, Pont-à-Mousson, France, *ENTCS* **15**, Elsevier (1998).

[87] A. Kock and G.E. Reyes, Doctrines in categorical logic, in: John Barwise, Ed., *Handbook of Mathematical Logic*, North Holland, 283–313 (1977).

[88] D.E. Knuth and P.B. Bendix, Simple word problem in universal algebra, in *Computational Problems in Abstract Algebra*, 263–297, Pergamon Press, Oxford (1970).

[89] C. Lair, Etude générale de la catégorie des esquisses, *Esquisses Math.* **24** (1974).

[90] K.G. Larsen and L. Xinxin, Compositionality through an operational semantics of contexts, in: M.S. Paterson, Ed., *Proceedings ICALP'90*, *LNCS* **443**, 526–539, Springer-Verlag (1990).

[91] F.W. Lawvere, Functorial semantics of algebraic theories, *Proceedings National Academy of Science of the United States of America* **50**(1), 869–872 (1963).

[92] F.W. Lawvere, Some algebraic problems in the context of functorial semantics of algebraic theories, in: S. MacLane, Ed., *Proceedings 2th Midwest Category Seminar*, *Lecture Notes in Mathematics* **61**, 41–61, Springer-Verlag (1968).

[93] S. MacLane, *Categories for the Working Mathematician*, Springer-Verlag (1971). 2nd ed. (1998).

[94] N. Martí-Oliet and J. Meseguer, Rewriting logic as a logical and semantic framework, SRI Technical Report, CSL-93-05, August 1993. To appear in: D. Gabbay, Ed., *Handbook of Philosophical Logic*, 2nd ed., Kluwer Academic Publishers.

[95] N. Martí-Oliet and J. Meseguer, General logics and logical frameworks, in: D. Gabbay, Ed., *What is a logical system?*, 355–392, Oxford University Press (1994).

[96] N. Martí-Oliet and J. Meseguer, Rewriting logic as a logical and semantic framework, in: J. Meseguer, Ed., *Proceedings 1st WRLA'96*, *ENTCS* **4**, 36 pages, Elsevier (1996).

[97] A. Mazurkiewicz, Trace theory, in: W. Brauer, W. Reisig, and G. Rozenberg, Eds., *Petri Nets: Applications and Relationships to Other Models of Concurrency, Advances in Petri Nets 1986*, *LNCS* **255**, 279–324, Springer-Verlag (1987).

[98] J. Meseguer, Rewriting as a unified model of concurrency, SRI Technical Report, CSL-90-02R (February 1990). Revised (June 1990). See the appendix on *Functorial Semantics of Rewrite Systems.*

[99] J. Meseguer, Conditional rewriting logic as a unified model of concurrency, *Theoretical Computer Science* **96**, 73–155 (1992).

[100] J. Meseguer, A logical theory of concurrent objects and its realization in the Maude language, in: G. Agha, P. Wegner, and A. Yonezawa, Eds., *Research Directions in Concurrent Object-Oriented Programming*, 314–390, MIT Press (1993).

[101] J. Meseguer, Rewriting logic as a semantic framework for concurrency: A progress report, in: U. Montanari and V. Sassone, Eds., *Proceedings CONCUR'96*, *LNCS* **1119**, 331-372, Springer-Verlag (1996).

[102] J. Meseguer, Membership algebra, Lecture and abstract at the *Dagstuhl Seminar on Specification and Semantics* (July 9, 1996).

[103] J. Meseguer, Ed. *Proceedings First International Workshop on Rewriting Logic and Applications*, Asilomar, California, *ENTCS* **4**, Elsevier (1996).

[104] J. Meseguer, Membership equational logic as a logical framework for equational specification, in: F. Parisi-Presicce, Ed., *Proceedings 12th WADT'97*, *LNCS* **1376**, 18–61, Springer-Verlag (1998).

[105] J. Meseguer and U. Montanari, Petri nets are monoids: A new algebraic foundation for net theory, in *Proceedings 3rd LICS Symposium*, 155–164, IEEE Computer Society Press (1988).

[106] J. Meseguer and U. Montanari, Petri nets are monoids, *Information and Computation* **88**, 105–154 (1990).

[107] J. Meseguer and U. Montanari, Mapping tile logic into rewriting logic, in: F. Parisi-Presicce, Ed., *Proceedings 12th WADT'97*, *LNCS* **1376**, 62–91, Springer-Verlag (1998).

[108] J. Meseguer, U. Montanari, and V. Sassone, Process versus unfolding semantics for place/transition Petri nets, *Theoretical Computer Science* **153**(1-2), 171–210 (1996).

[109] J. Meseguer, U. Montanari, and V. Sassone, Representation Theorems for Petri Nets, in: W. Brauer, C. Freksa, M. Jantzen, and R. Valk, Eds., *Foundations of Computer Science*, *LNCS* **1337**, 239–249, Springer-Verlag (1997).

[110] J. Meseguer and C. Talcott, Using rewriting logic to interoperate architectural description languages (I and II), Lectures at the *Santa Fe and Seattle DARPA-EDCS Workshops* (1997).

[111] R. Milner, *Communication and Concurrency*, Prentice-Hall (1989).

[112] R. Milner, Calculi for interaction, *Acta Informatica* **33**, 707–737 (1996).

[113] R. Milner, J. Parrow, and D. Walker, A calculus of mobile processes (parts I and II), *Information and Computation* **100**, 1–77 (1992).

[114] U. Montanari, M. Pistore, and D. Yankelevich, Efficient minimization up to location equivalence, in: H.R. Nielson, Ed., *Proceedings ESOP'96*, *LNCS* **1058**, Springer-Verlag (1996).

[115] U. Montanari and F. Rossi, Graph rewriting, constraint solving and tiles for coordinating distributed systems. *Applied Categorical Structures*, to appear.

[116] U. Montanari and C. Talcott, Can actors and π-agents live together ?, in: A.D. Gordon, A.M. Pitts, and C. Talcott, Eds., *Proceedings 2nd Workshop on Higher Order Operational Techniques in Semantics*, *ENTCS* **10**, 8 pages, Elsevier (1998).

[117] U. Montanari and D. Yankelevich, Location equivalence in a parametric setting, *Theoretical Computer Science* **149**(2), 299–332 (October 1995).

[118] H. Miyoshi, Modelling conditional rewriting logic in structured categories, in: J. Meseguer, Ed., *1st WRLA'96*, *ENTCS* **4**, 15 pages, Elsevier (1996).

[119] E.R. Olderog, Operational Petri net semantics for CCSP, in: G. Rozenberg, Ed., *Advances in Petri Nets 1987*, *LNCS* **266**, 196–223, Springer-Verlag (1987).

[120] C.A. Petri, *Kommunikation mit Automaten*, PhD thesis, Institut für Instrumentelle Mathematik, Bonn, Germany (1962).

[121] G. Plotkin, A structural approach to operational semantics, Technical Report DAIMI FN-19, Computer Science Department, Aarhus University (1981).

[122] J. Power and E. Robinson, Premonoidal categories and notions of computation, *Mathematical Structures in Computer Science* **7**, 453–468 (1998).

[123] W. Reisig, *Petri Nets*, Springer-Verlag (1985).

[124] G. Ristori, *Modelling Systems with Shared Resources via Petri Nets*, PhD thesis TD-94-05, Department of Computer Science, University of Pisa (1994).

[125] E. Robinson and G. Rosolini, Categories of partial maps, *Information and Computation* **79**, 95–130 (1988).

[126] V. Sassone, An axiomatization of the algebra of Petri net concatenable processes, *Theoretical Computer Science* **170**(1-2), 277–296 (1996).

[127] R. de Simone, Higher-level synchronising devices in Meije-SCCS, *Theoretical Computer Science* **37**, 245–267 (1985).

[128] G. Stefanescu, Algebra of flownomials, Technical Report SFB-Bericht 342/16/94 A, Technical University of München, Institut für Informatik (1994).

[129] R. Street, Higher categories, strings, cubes and simplex equations, *Applied Categorical Structures* **3**, 29–77 (1995).

[130] R. Street, Categorical structures, in: M. Hazewinkel, Ed., *Handbook of Algebra*, 529–577, Elsevier (1996).

[131] I. Suzuki and T. Murata, A method for stepwise refinement and abstraction of Petri nets, *Journal of Computer and System Sciences* **27**, 51–76 (1983).

[132] R. Valette, Analysis of Petri nets by stepwise refinement, *Journal of Computer and System Sciences* **18**, 35–46 (1979).

[133] P. Viry, Rewriting modulo a rewrite system, Technical Report TR-95-20, Department of Computer Science, University of Pisa (1995).

[134] W. Vogler, Behaviour preserving refinements of Petri nets, in: G. Tinhofer and G. Schmidt, Eds., *Proceedings 12th International Workshop on Graph-Theoretic Concepts in Computer Science*, *LNCS* **246**, 82–93, Springer-Verlag (1987).

[135] W. Vogler, Failures semantics of Petri nets and the refinement of places and transitions, Technical Report TUM-I9003, Institut für Informatik, Technische Universität München (1990).

[136] G. Winskel, Event structure semantics of CCS and related languages, in: M. Nielsen and E. Meineche Schmidt, Eds., *Proceedings ICALP'82*, *LNCS* **140**, 561–567, Springer-Verlag (1982).

[137] G. Winskel, Petri nets, algebras, morphisms and compositionality, *Information and Computation* **72**, 197–238 (1987).

Appendix A

The Axioms of Process Tile Logic

Let $\mathcal{R} = \langle \Sigma_H, \Sigma_V, N, R \rangle$ be a process tile system, and let S be the common underlying set of sorts of Σ_H and Σ_V. The class $A_p(\mathcal{R})$ of *abstract process sequents* informally described in Definition 5.3.4 has as elements the equivalence classes of $P_p(\mathcal{R})$ modulo the following set of axioms on proof terms:
Associativity Axioms for $\_ \otimes \_$, $\_ * \_$, and $\_ \cdot \_$ (whenever the sequents α, β and γ can be correctly composed):

$$\alpha \otimes (\beta \otimes \gamma) = (\alpha \otimes \beta) \otimes \gamma \qquad \alpha * (\beta * \gamma) = (\alpha * \beta) * \gamma \qquad \alpha \cdot (\beta \cdot \gamma) = (\alpha \cdot \beta) \cdot \gamma$$

Identity Axioms (for any $\alpha : h \xrightarrow[v]{u} g \in P_p(\mathcal{R})$):

$$1_v * \alpha = \alpha = \alpha * 1_u \qquad 1^h \cdot \alpha = \alpha = \alpha \cdot 1^g$$

Monoidality Axioms (for any $h, g \in \mathbf{S}(\Sigma_H)$, $\alpha \in P_p(\mathcal{R})$, and $v, u \in \mathbf{S}(\Sigma_V)$):

$$1^{h \otimes g} = 1^h \otimes 1^g \qquad 1_{id_0} \otimes \alpha = \alpha = \alpha \otimes 1_{id_0} \qquad 1_{v \otimes u} = 1^v \otimes 1^u$$

Functoriality Axioms:
Identities (for any $a \in S$, and composable arrows $h, g \in \mathbf{S}(\Sigma_H)$, and $v, u \in \mathbf{S}(\Sigma_V)$):

$$1_{v;u} = 1_v \cdot 1_u \qquad 1_{id_a} = 1^{id_a} \qquad 1^{h;g} = 1^h * 1^g$$

Compositions (whenever both sides are defined):

$$(\alpha \otimes \beta) \cdot (\gamma \otimes \delta) = (\alpha \cdot \gamma) \otimes (\beta \cdot \delta) \qquad (\alpha \otimes \beta) * (\gamma \otimes \delta) = (\alpha * \gamma) \otimes (\beta * \delta)$$

$$(\alpha * \beta) \cdot (\gamma * \delta) = (\alpha \cdot \gamma) * (\beta \cdot \delta)$$

Auxiliary Operators
(for any $a, b \in S$, and composable arrows $v, v' \in \mathbf{S}(\Sigma_V)$, and $u, u' \in \mathbf{S}(\Sigma_V)$):

$$\gamma_{v;v',u;u'} = \gamma_{v,u} \cdot \gamma_{v',u'} \qquad \gamma_{id_a, id_b} = 1^{\gamma_{a,b}}$$

(for any $a, b \in S$, and composable arrows $h, h' \in \mathbf{S}(\Sigma_H)$, and $g, g' \in \mathbf{S}(\Sigma_H)$):

$$\rho_{h;h',g;g'} = \rho_{h,g} * \rho_{h',g'} \qquad \rho_{id_a,id_b} = 1_{\gamma_{a,b}}$$

Naturality Axioms for *derived operators* γ and ρ
(for any sequents $\alpha : h \xrightarrow[v]{u} g,\ \alpha' : h' \xrightarrow[v']{u'} g' \in P_p(\mathcal{R})$):

$$(\alpha \otimes \alpha') * \gamma_{u,u'} = \gamma_{v,v'} * (\alpha' \otimes \alpha) \qquad (\alpha \otimes \alpha') \cdot \rho_{g,g'} = \rho_{h,h'} \cdot (\alpha' \otimes \alpha)$$

Uniqueness Axioms:
Naturality axioms for σ
(for any $v : a \longrightarrow c,\ v' : a' \longrightarrow c' \in \mathbf{S}(\Sigma_V)$, and $h : a \longrightarrow b,\ h' : a' \longrightarrow b' \in \mathbf{S}(\Sigma_H)$):

$$\gamma_{v,v'} \cdot \sigma_{c,c'} = \sigma_{a,a'} \cdot 1_{v' \otimes v} \qquad \rho_{h,h'} * \sigma_{b,b'} = \sigma_{a,a'} * 1^{h' \otimes h}$$

Naturality axioms for σ'
(for any $v : a \longrightarrow c,\ v' : a' \longrightarrow c' \in \mathbf{S}(\Sigma_V)$, and $h : a \longrightarrow b,\ h' : a' \longrightarrow b' \in \mathbf{S}(\Sigma_H)$):

$$1_{v \otimes v'} \cdot \sigma'_{c,c'} = \sigma'_{a,a'} \cdot \gamma_{v,v'} \qquad 1^{h \otimes h'} * \sigma'_{b,b'} = \sigma'_{a,a'} \cdot \gamma_{h,h'}$$

Coherence Axioms for γ (for any $u, w, v \in \mathbf{S}(\Sigma_V)$):

$$\gamma_{u \otimes w,v} = (1_u \otimes \gamma_{w,v}) * (\gamma_{u,v} \otimes 1_w) \qquad \gamma_{v,u} * \gamma_{u,v} = 1_{v \otimes u} \qquad \gamma_{id_0,id_0} = 1_{id_0}$$

Coherence Axioms for ρ (for any $h, f, g \in \mathbf{S}(\Sigma_H)$):

$$\rho_{h \otimes f,g} = (1^h \otimes \rho_{f,g}) * (\rho_{h,g} \otimes 1^f) \qquad \rho_{h,g} \cdot \rho_{g,h} = 1^{h \otimes g} \qquad \rho_{id_0,id_0} = 1_{id_0}$$

Coherence Axioms for σ (for any $a, b, c \in S$):

$$\sigma_{a \otimes b,c} = ((1_{id_a} \otimes \sigma_{b,c}) * 1^{\gamma_{a,c} \otimes id_b}) \cdot (\sigma_{a,c} \otimes 1_{id_b}) \qquad (\sigma_{a,b} * 1^{\gamma_{b,a}}) \cdot \sigma_{b,a} = 1_{id_{a \otimes b}} \qquad \sigma_{0,0} = 1_{id_0}$$

Coherence Axioms for σ' (for any $a, b, c \in S$):

$$\sigma'_{a \otimes b,c} = (1_{id_a} \otimes \sigma'_{b,c}) \cdot (1_{id_a \otimes \gamma_{b,c}} * (\sigma'_{a,c} \otimes 1_{id_b})) \qquad \sigma'_{a,b} \cdot (1^{\gamma_{a,b}} * \sigma'_{b,a}) = 1_{id_{a \otimes b}} \qquad \sigma_{0,0} = 1_{id_0}$$

Double Coherence Axioms (for any $a, b \in S$):

$$\sigma'_{a,b} * \sigma_{a,b} = \gamma_{id_a,id_b} \qquad \sigma'_{a,b} \cdot \sigma_{a,b} = \rho_{id_a,id_b}$$

Appendix B

The Axioms of Term Tile Logic

Let $\mathcal{R} = \langle \Sigma_H, \Sigma_V, N, R \rangle$ be a (one-sorted) term tile system. We say that $\mathcal{R}$ *entails* the class $A_t(\mathcal{R})$ of *abstract term sequents*, whose elements are equivalence classes of $P_t(\mathcal{R})$ modulo the following set of axioms on proof terms:[1]

Associativity Axioms for $\_\otimes\_$, $\_*\_$, and $\_\cdot\_$ (whenever the sequents α, β and γ can be correctly composed):

$$\alpha \otimes (\beta \otimes \gamma) = (\alpha \otimes \beta) \otimes \gamma \qquad \alpha * (\beta * \gamma) = (\alpha * \beta) * \gamma \qquad \alpha \cdot (\beta \cdot \gamma) = (\alpha \cdot \beta) \cdot \gamma$$

Identity Axioms (for any $\alpha : h \xrightarrow[v]{u} g \in P_t(\mathcal{R})$):

$$1_v * \alpha = \alpha = \alpha * 1_u \qquad 1^h \cdot \alpha = \alpha = \alpha \cdot 1^g$$

Monoidality Axioms (for any $h, g \in \mathbf{A}(\Sigma_H)$, $\alpha \in P_t(\mathcal{R})$, and $v, u \in \mathbf{A}(\Sigma_V)$):

$$1^{h \otimes g} = 1^h \otimes 1^g \qquad 1_{id_0} \otimes \alpha = \alpha = \alpha \otimes 1_{id_0} \qquad 1_{v \otimes u} = 1^v \otimes 1^u$$

Functoriality Axioms:

Identities (for any $n \in \mathbb{N}$, and composable arrows $h, g \in \mathbf{A}(\Sigma_H)$, and $v, u \in \mathbf{A}(\Sigma_V)$):

$$1_{v;u} = 1_v \cdot 1_u \qquad 1_{id_n} = 1^{id_n} \qquad 1^{h;g} = 1^h * 1^g$$

Compositions (whenever both sides are defined):

$$(\alpha \otimes \beta) \cdot (\gamma \otimes \delta) = (\alpha \cdot \gamma) \otimes (\beta \cdot \delta) \qquad (\alpha \otimes \beta) * (\gamma \otimes \delta) = (\alpha * \gamma) \otimes (\beta * \delta)$$

$$(\alpha * \beta) \cdot (\gamma * \delta) = (\alpha \cdot \gamma) * (\beta \cdot \delta)$$

Auxiliary Operators: Symmetries
(for any $n, m \in \mathbb{N}$, and composable arrows $v, v' \in \mathbf{A}(\Sigma_V)$, and $u, u' \in \mathbf{A}(\Sigma_V)$):

$$\gamma_{v;v',u;u'} = \gamma_{v,u} \cdot \gamma_{v',u'} \qquad \gamma_{id_n, id_m} = 1^{\gamma_{n,m}}$$

[1] Notice that we adopt the notation of algebraic theories (Section 2.2) instead of the equivalent "terms and substitutions" notation used in Definitions 5.3.6 and 5.3.7.

(for any $n \in \mathbb{N}$, and composable arrows $h, h' \in \mathbf{A}(\Sigma_H)$, and $g, g' \in \mathbf{A}(\Sigma_H)$):

$$\rho_{h;h',g;g'} = \rho_{h,g} * \rho_{h',g'} \qquad \rho_{id_n,id_m} = 1_{\gamma_{n,m}}$$

Auxiliary Operators: Duplicators
(for any $n \in \mathbb{N}$, and composable arrows $v, v' \in \mathbf{A}(\Sigma_V)$):

$$\nabla_{v;v'} = \nabla_v \cdot \nabla_{v'} \qquad \nabla_{id_n} = 1^{\nabla_n}$$

(for any $n \in \mathbb{N}$, and composable arrows $h, h' \in \mathbf{A}(\Sigma_H)$):

$$\delta_{h;h'} = \delta_h * \delta_{h'} \qquad \delta_{id_n} = 1_{\nabla_n}$$

Auxiliary Operators: Dischargers
(for any $n \in \mathbb{N}$, and composable arrows $v, v' \in \mathbf{A}(\Sigma_H)$):

$$!_{v;v'} = !_v \cdot !_{v'} \qquad !_{id_n} = 1^{!_n}$$

(for any $n \in \mathbb{N}$, and composable arrows $h, h' \in \mathbf{A}(\Sigma_H)$):

$$\dagger_{id_n} = 1_{!_n} \qquad \dagger_{h;h'} = \dagger_h * \dagger_{h'}$$

Naturality Axioms (symmetries) for *derived operators* γ and ρ
(for any sequents $\alpha : h \xrightarrow[v]{u} g,\ \alpha' : h' \xrightarrow[v']{u'} g' \in P_t(\mathcal{R})$):

$$(\alpha \otimes \alpha') * \gamma_{u,u'} = \gamma_{v,v'} * (\alpha' \otimes \alpha) \qquad (\alpha \otimes \alpha') \cdot \rho_{g,g'} = \rho_{h,h'} \cdot (\alpha' \otimes \alpha)$$

Naturality Axioms (duplicators) for *derived operators* ∇ and δ
(for any sequents $\alpha : h \xrightarrow[v]{u} g \in P_t(\mathcal{R})$):

$$\alpha * \nabla_u = \nabla_v * (\alpha \otimes \alpha) \qquad \alpha \cdot \delta_g = \delta_h \cdot (\alpha \otimes \alpha)$$

Naturality Axioms (dischargers) for *derived operators* ! and †
(for any sequents $\alpha : h \xrightarrow[v]{u} g \in P_t(\mathcal{R})$):

$$\alpha * !_u = !_v \qquad \alpha \cdot \dagger_g = \dagger_h$$

Uniqueness Axioms:
Naturality axioms for σ
(for any $v : n \longrightarrow k$, $v' : n' \longrightarrow k'$ in $\mathbf{A}(\Sigma_V)$, and $h : n \longrightarrow m$, $h' : n' \longrightarrow m'$ in $\mathbf{A}(\Sigma_H)$):

$$\gamma_{v,v'} \cdot \sigma_{k,k'} = \sigma_{n,n'} \cdot 1_{v' \otimes v} \qquad \rho_{h,h'} * \sigma_{m,m'} = \sigma_{n,n'} * 1^{h' \otimes h}$$

Naturality axioms for σ'
(for any $v : n \longrightarrow k$, $v' : n' \longrightarrow k'$ in $\mathbf{A}(\Sigma_V)$, and $h : n \longrightarrow m$, $h' : n' \longrightarrow m'$ in $\mathbf{A}(\Sigma_H)$):

$$1_{v \otimes v'} \cdot \sigma'_{k,k'} = \sigma'_{n,n'} \cdot \gamma_{v,v'} \qquad 1^{h \otimes h'} * \sigma'_{m,m'} = \sigma'_{n,n'} \cdot \gamma_{h,h'}$$

Naturality axioms for π (for any $v : n \longrightarrow k \in \mathbf{A}(\Sigma_V)$, and $h : n \longrightarrow m \in \mathbf{A}(\Sigma_H)$):

$$\nabla_v \cdot \pi_k = \pi_n \cdot 1_{v \otimes v} \qquad \delta_h * \pi_m = \pi_n * 1^{h \otimes h}$$

Naturality axioms for τ (for any $v : n \longrightarrow k \in \mathbf{A}(\Sigma_V)$, and $h : n \longrightarrow m \in \mathbf{A}(\Sigma_H)$):

$$1_v \cdot \tau_k = \tau_n \cdot \nabla_v \qquad 1^h * \tau_m = \tau_n * \delta_h$$

Naturality axioms for ϕ (for any $v : n \longrightarrow k \in \mathbf{A}(\Sigma_V)$, and $h : n \longrightarrow m \in \mathbf{A}(\Sigma_H)$):

$$!_v \cdot \phi_k = \phi_n \qquad \dagger_h * \phi_m = \phi_n$$

Naturality axioms for ψ (for any $v : n \longrightarrow k \in \mathbf{A}(\Sigma_V)$, and $h : n \longrightarrow m \in \mathbf{A}(\Sigma_H)$):

$$1_v \cdot \psi_k = \psi_n \qquad 1^h * \psi_m = \psi_n$$

Coherence Axioms for γ, ∇, and $!$ (for any $u, w, v \in \mathbf{A}(\Sigma_V)$):

$$\begin{array}{ll}
\gamma_{u \otimes w, v} = (1_u \otimes \gamma_{w,v}) * (\gamma_{u,v} \otimes 1_w) & \nabla_v * \gamma_{v,v} = \nabla_v \\
\nabla_{v \otimes u} = (\nabla_v \otimes \nabla_u) * (1_v \otimes \gamma_{v,u} \otimes 1_u) & \nabla_v * (1_v \otimes \nabla_v) = \nabla_v * (\nabla_v \otimes 1_v) \\
!_{v \otimes u} = !_v \otimes !_u & \nabla_v * (1_v \otimes !_v) = 1_v \\
\gamma_{id_0, id_0} = 1_{id_0} = \nabla_{id_0} !_{id_0} & \gamma_{v,u} * \gamma_{u,v} = 1_{v \otimes u}
\end{array}$$

Coherence Axioms for ρ, δ, and $\dagger$ (for any $h, f, g \in \mathbf{A}(\Sigma_H)$):

$$\begin{array}{ll}
\rho_{h \otimes f, g} = (1^h \otimes \rho_{f,g}) * (\rho_{h,g} \otimes 1^f) & \delta_h \cdot \rho_{h,h} = \delta_h \\
\delta_{h \otimes g} = (\delta_h \otimes \delta_g) \cdot (1^h \otimes \rho_{h,g} \otimes 1^g) & \delta_h \cdot (1^h \otimes \delta_h) = \delta_h \cdot (\delta_h \otimes 1^h) \\
\dagger_{h \otimes g} = \dagger_h \otimes \dagger_g & \delta_h \cdot (1^h \otimes \dagger_h) = 1^h \\
\rho_{id_0, id_0} = 1_{id_0} = \delta_{id_0} = \dagger_{id_0} & \rho_{h,g} \cdot \rho_{g,h} = 1^{h \otimes g}
\end{array}$$

Coherence Axioms for σ, π, and ϕ (for any $n, m, k \in \mathbb{N}$):

$$\begin{array}{l}
\sigma_{n \otimes m, k} = ((1_{id_n} \otimes \sigma_{m,k}) * 1^{\gamma_{n,k} \otimes id_m}) \cdot (\sigma_{n,k} \otimes 1_{id_m}) \\
\pi_{n \otimes m} = ((\pi_n \otimes \pi_m) * 1^{id_n \otimes \gamma_{n,m} \otimes id_m}) \cdot (1_{id_n} \otimes \sigma_{n,m} \otimes 1_{id_m}) \\
\phi_{n \otimes m} = \phi_n \otimes \phi_m \\
(\pi_n * 1^{\gamma_{n,n}}) \cdot \sigma_{n,n} = \pi_n \\
(\pi_n * 1^{\nabla_n \otimes id_n}) \cdot (\pi_n \otimes 1_{id_n}) = (\pi_n * 1^{id_n \otimes \nabla_n}) \cdot (1_{id_n} \otimes \pi_n) \\
(\pi_n * 1^{id_n \otimes !_n}) \cdot (1_{id_n} \otimes \phi_n) = 1_{id_n} \\
(\sigma_{n,m} * 1^{\gamma_{m,n}}) \cdot \sigma_{m,n} = 1_{id_{n \otimes m}} \\
\sigma_{0,0} = \pi_0 = \phi_0 = 1_{id_0}
\end{array}$$

Coherence Axioms for σ', τ, and ψ (for any $n, m, k \in \mathbb{N}$):

$$\begin{array}{l}
\sigma'_{n \otimes m, k} = (1_{id_n} \otimes \sigma'_{m,k}) \cdot (1_{id_n \otimes \gamma_{m,k}} * (\sigma'_{n,k} \otimes 1_{id_m})) \\
\tau_{n \otimes m} = (\tau_n \otimes \tau_m) \cdot (1^{\nabla_n \otimes \nabla_m} * (1_{id_n} \otimes \sigma'_{n,m} \otimes 1_{id_m})) \\
\psi_{n \otimes m} = \psi_n \otimes \psi_m \\
\tau_n \cdot (1^{\nabla_n} * \sigma'_{n,n}) = \tau_n \\
\tau_n \cdot (1^{\nabla_n} * (\tau_n \otimes 1_{id_n})) = \tau_n \cdot (1^{\nabla_n} * (1_{id_n} \otimes \tau_n)) \\
\tau_n \cdot (1^{\nabla_n} * (1_{id_n} \otimes \psi_n)) = 1_{id_n} \\
\sigma'_{n,m} \cdot (1^{\gamma_{n,m}} * \sigma'_{m,n}) = 1_{id_{n \otimes m}} \\
\sigma'_{0,0} = \tau_0 = \psi_0 = 1_{id_0}
\end{array}$$

Double Coherence Axioms (for any $n, m \in \mathbb{N}$):

$$\sigma'_{n,m} * \sigma_{n,m} = \gamma_{id_n,id_m} \qquad \sigma'_{n,m} \cdot \sigma_{n,m} = \rho_{id_n,id_m}$$
$$\tau_n * \pi_n = \nabla_{id_n} \qquad \tau_n \cdot \pi_n = \delta_{id_n}$$
$$\psi_n * \phi_n = !_{id_n} \qquad \psi_n \cdot \phi_n = \dagger_{id_n}$$

Appendix C

Hypertransformations

C.1 Multiple Categories

As explained in Section 5.2.3, since double categories have two different notions of composition, it is not clear which of them should be used for a general notion of double natural transformation. Ehresmann [57] noticed that natural transformations can be expressed in terms of functors toward higher fold categories. The key point is that a natural transformation is in some sense *a functorial collection of commuting squares* (also called *quartets*). Therefore, to obtain a suitable notion of natural transformation for n-fold categories we have to consider quartets in all the n dimensions.

REMARK C.1.1 *The notation used in this appendix has been introduced in Section 5.2.*

DEFINITION C.1.1 ($\mathcal{D}$-WISE TRANSFORMATION)
Let $\mathcal{A}$ be a category, and let $\mathcal{D}$ be a double category. We denote by $T(\mathcal{D},\mathcal{A})$ the category of $\mathcal{D}$-wise transformations *from $\mathcal{A}$ to $\mathcal{D}$, whose objects are functors from $\mathcal{A}$ to the vertical 1-category $\mathcal{V}$ of $\mathcal{D}$ and whose composition law is deduced from that of the category $\mathcal{D}^*$, i.e., there is an arrow α from $P:\mathcal{A}\longrightarrow\mathcal{V}$ to $Q:\mathcal{A}\longrightarrow\mathcal{V}$ if and only if there is a functor $\alpha:\mathcal{A}\longrightarrow\mathcal{D}^{\cdot}$ such that $s^*\circ\alpha=P$ and $t^*\circ\alpha=Q$, and given two arrows α and α' such that $t^*\circ\alpha=s^*\circ\alpha'$ their composition is equal to the functor mapping each arrow v of $\mathcal{A}$ onto the cell $\alpha(v)*\alpha'(v)$ (see Figure C.1).*

Moreover, T extends to a functor as foolows: given a double functor $F:\mathcal{D}\longrightarrow\mathcal{E}$ and a functor $G:\mathcal{B}\longrightarrow\mathcal{A}$ it suffices to define $T(F,G)=F_1\circ\alpha\circ G$, where the functor $F_1:\mathcal{D}^{\cdot}\longrightarrow\mathcal{E}^{\cdot}$ is the component of F relative to the vertical structure.

EXAMPLE C.1.2 *Let $\mathbf{2}$ be the category with two objects 0 and 1, and one arrow $z:0\longrightarrow 1$ plus the identities. Then, for any double category $\mathcal{D}$, we have that $T(\mathcal{D},\mathbf{2})\simeq\mathcal{D}^*$.*

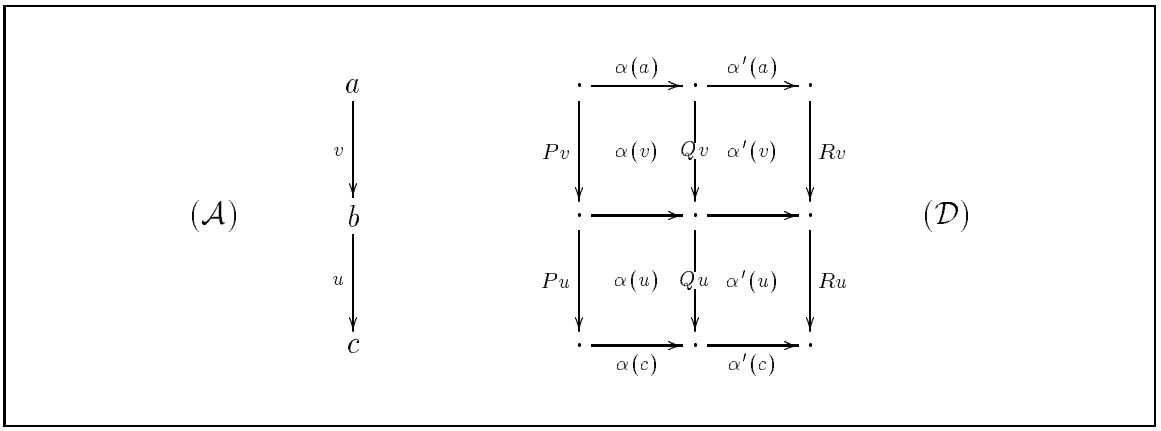


Figure C.1: $\mathcal{D}$-wise transformations α and α'

PROPOSITION C.1.1
Let $\mathcal{A}$ and $\mathcal{C}$ be two categories. The category $\mathcal{C}^{\mathcal{A}}$ whose arrows are the natural transformations between functors from $\mathcal{A}$ to $\mathcal{C}$ (see page 30) is isomorphic to the category $T(\square\mathcal{C}, \mathcal{A})$, where $\square\mathcal{C}$ is the double category of quartets of $\mathcal{C}$ (see Example 5.2.2). In fact, a functor from $\mathcal{A}$ to the vertical category of $\square\mathcal{C}$ (usually denoted by $\boxminus\mathcal{C}$) identifies a natural transformation between functors from $\mathcal{A}$ to $\mathcal{C}$ and vice versa.

This notion can be generalized to n-fold categories by constructing the $2n$-fold category of quartets of quartets... (n times). Since double categories are 2-fold categories, this means that we need to define the 4-fold category of horizontal quartets of vertical quartets. Once a notion of *multiple functors* among categories of different folds has been given, then the notion of hypertransformation arises naturally as a multiple functor between the source n-fold category and the $2n$-fold category of quartets of quartets ... (n times) generated by the target n-fold category. This means that a hypertransformation involving two n-fold categories $\vec{\mathcal{C}}$ and $\vec{\mathcal{D}}$ relates 2^n n-fold functors from $\vec{\mathcal{C}}$ to $\vec{\mathcal{D}}$.

DEFINITION C.1.2 (n-FOLD CATEGORY)
Let n be a positive integer. An n-fold category $\vec{\mathcal{C}}$ *is a tuple of n categories $(\mathcal{C}^1, \ldots, \mathcal{C}^n)$ with the same set of morphisms $\mathcal{C}$ satisfying the* permutability axiom*:*

$$(\mathcal{C}^i, \mathcal{C}^j) \text{ is a double category for each pair } (i,j) \text{ of integers } 1 \leq i < j \leq n.$$

An element of $\mathcal{C}$ is called a block*. The category $\mathcal{C}^i$ is called the* i-th category of $\vec{\mathcal{C}}$*, and its composition is denoted by $\_;_i\_$. Notice that the set of objects of $\mathcal{C}^i$ defines a sub-category of $\mathcal{C}^j$ for each $i \neq j$.*

If $\vec{\mathcal{C}}$ and $\vec{\mathcal{D}}$ are n-fold categories, a n-fold functor *$F : \vec{\mathcal{C}} \longrightarrow \vec{\mathcal{D}}$ is a map F from the set of blocks $\mathcal{C}$ to $\mathcal{D}$ that defines a functor $F : \mathcal{C}^i \longrightarrow \mathcal{D}^i$ for each i. We denote by $\mathbf{Cat}_n$ the category whose objects are the small n-fold categories and whose morphisms are the n-fold functors between them.*

By convention, a 0-fold category is just a set, and a 1-fold category is a category. Thus $\mathbf{Cat}_0$ is the category **Set**, $\mathbf{Cat}_1$ is **Cat**, and $\mathbf{Cat}_2$ is **DCat**.

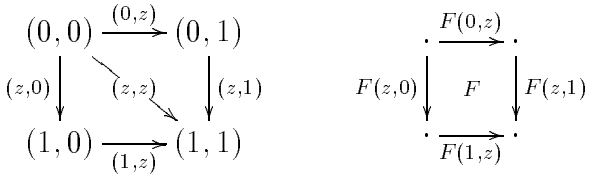


Figure C.2: Quartets as functors.

If γ is a permutation over the set $\{1,\ldots,n\}$ then $(\mathcal{C}^{\gamma(1)},\ldots,\mathcal{C}^{\gamma(n)})$ is also an n-fold category and it is denoted by $\vec{\mathcal{C}}^{\gamma}$. If $i_1,\ldots,i_m$ is a sequence of m distinct elements of $\{1,\ldots,n\}$, we denote by $\mathcal{C}^{i_1,\ldots,i_m}$ the m-fold category $(\mathcal{C}^{i_1},\ldots,\mathcal{C}^{i_m})$.

Definition C.1.3 (Multiple Category)
The category **MCat** *of* multiple categories *is defined as follows. Its objects are the small n-fold categories, for every integer n. Let $\vec{\mathcal{C}}$ be an n-fold category and let $\vec{\mathcal{D}}$ be an m-fold category with $n \leq m$, then the morphisms $F : \vec{\mathcal{C}} \longrightarrow \vec{\mathcal{D}}$ in* **MCat**, *called* multiple functors, *are the n-fold functors F from $\vec{\mathcal{C}}$ to the n-fold category $(\mathcal{D}^1,\ldots,\mathcal{D}^n)$ (if $n > m$, then there is no morphism from $\vec{\mathcal{C}}$ to $\vec{\mathcal{D}}$).*

Definition C.1.4 (Internal Hom of **MCat**)
Let $\vec{\mathcal{C}}$ be an n-fold category and $\vec{\mathcal{D}}$ an m-fold category. We denote by $Hom(\vec{\mathcal{C}},\vec{\mathcal{D}})$ the multiple category of multiple functors *from $\vec{\mathcal{C}}$ to $\vec{\mathcal{D}}$, which is defined as follows:*

- *if $n > m$, then it is the empty set,*
- *otherwise (i.e., $n \leq m$), it is the $(m-n)$-fold category, on the set of multiple functors $F : \vec{\mathcal{C}} \longrightarrow \vec{\mathcal{D}}$, whose i-th composition, for $1 \leq i \leq m-n$, is defined as $F(A);_{i+n} F'(A)$ iff the composite exists in $\mathcal{D}^{i+n}$ for each block A of $\mathcal{C}$.*

Remark C.1.3 *For each pair (i,j) with $1 \leq i \leq m-n$ and $1 \leq j \leq n$, the category $Hom(\vec{\mathcal{C}},\vec{\mathcal{D}})^i$ is a subcategory of the category of $(\mathcal{D}^j,\mathcal{D}^{i+n})$-wise transformations from $\mathcal{C}^j$ to $(\mathcal{D}^j,\mathcal{D}^{i+n})$. The permutability axiom is satisfied by $Hom(\vec{\mathcal{C}},\vec{\mathcal{D}})$, since it is satisfied by $\vec{\mathcal{D}}$ and compositions are defined pointwise from that of $\vec{\mathcal{D}}$.*

Example C.1.4 *If $\mathcal{A}$ and $\mathcal{C}$ are categories, then $Hom(\mathcal{A},\Box\mathcal{C})$ is the category $\mathcal{C}^{\mathcal{A}}$ of natural transformations between functors from $\mathcal{A}$ to $\mathcal{C}$.*

Now, consider the category $\mathbf{2}\times\mathbf{2}$, and let $\mathcal{C}$ be a category. A functor $F : \mathbf{2}\times\mathbf{2} \longrightarrow \mathcal{C}$ is entirely determined by the (commutative) square F of $\mathcal{C}$ (see Figure C.2), because $F(z,z)$ is just the diagonal of this square. Moreover, every quartet of $\mathcal{C}$ can be obtained in this way. Thus, we can identify the set of functors $Hom(\mathbf{2}\times\mathbf{2},\mathcal{C})$ with the set of quartets of $\mathcal{C}$ (which is also the set of blocks of $\Box\mathcal{C}$).

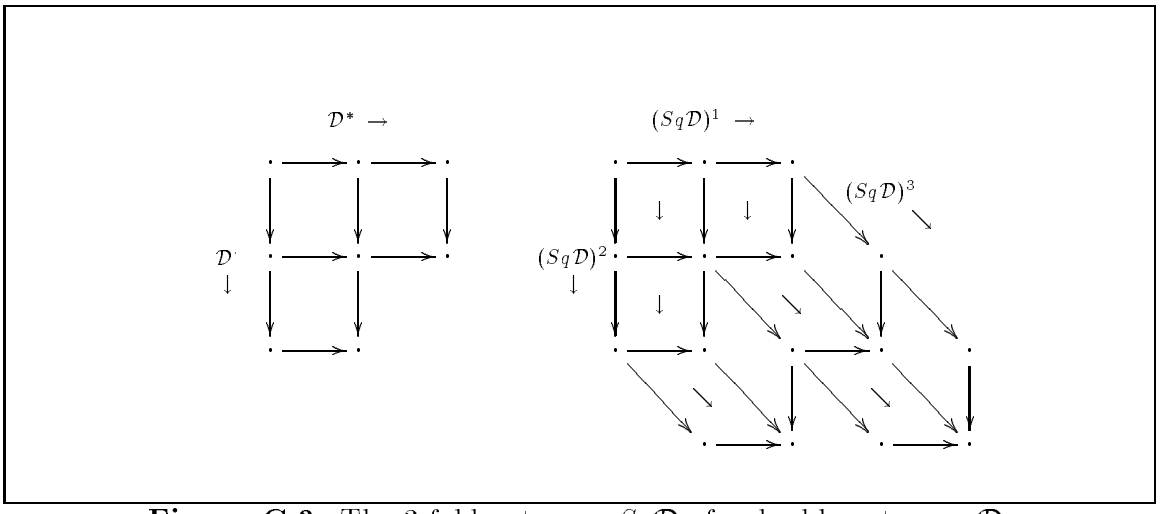


Figure C.3: The 3-fold category $Sq\mathcal{D}$ of a double category $\mathcal{D}$.

Let $\vec{\mathcal{C}}$ be an n-fold category ($n > 1$). Consider the functors from $\mathbf{2} \times \mathbf{2}$ to its first category $\mathcal{C}^1$. It follows that, over the set of quartets of $\mathcal{C}^1$ we have not only the double category $\square\mathcal{C}^1$, but also the $(n-1)$-fold category $Hom(\mathbf{2} \times \mathbf{2}, \vec{\mathcal{C}})$ whose i-th composition is deduced pointwise from that of $\mathcal{C}^{i+1}$, i.e., given two quartets $Q = (A, B, C, D)$ and $Q' = (A', B', C', D')$ of $\mathcal{C}^1$ (this means that $A ;_1 C = B ;_1 D$ and $A' ;_1 C' = B' ;_1 D'$) then

$$Q ;_i Q' = ((A ;_{i+1} A'), (B ;_{i+1} B'), (C ;_{i+1} C'), (D ;_{i+1} D'))$$

(if the four compositions are defined in $\mathcal{C}^{i+1}$).

Definition C.1.5 (Multiple Category of Quartets)
The multiple category of quartets *of $\vec{\mathcal{C}}$, denoted by $Sq\mathcal{C}$, is the $(n+1)$-fold category on the set of commuting squares of $\mathcal{C}^1$ such that: $(Sq\mathcal{C})^{1,\ldots,n-1} = Hom(\mathbf{2} \times \mathbf{2}, \vec{\mathcal{C}})$, $(Sq\mathcal{C})^n = \overset{\square}{\square}\mathcal{C}^1$ and $(Sq\mathcal{C})^{n+1} = \square\square\mathcal{C}^{n+1}$.*

The previous construction induces a functor from $\mathbf{Cat}_n$ to $\mathbf{Cat}_{n+1}$ called *square* and denoted by $Sq_{n,n+1}$, mapping an n-fold functor $F : \vec{\mathcal{C}} \longrightarrow \vec{\mathcal{D}}$ to the $(n+1)$-fold functor $SqF : Sq\mathcal{C} \longrightarrow Sq\mathcal{D}$ such that

$$SqF(A, B, C, D) = (FA, FB, FC, FD).$$

C.2 The 3-fold category $Sq\mathcal{D}$

Given a double category $\mathcal{D} = (\mathcal{D}_\cdot, \mathcal{D}^*)$, the 3-fold category $Sq\mathcal{D}$ is defined as follows: its 2-nd and 3-rd categories are the vertical and horizontal categories $\overset{\square}{\square}\mathcal{D}_\cdot$ and $\square\square\mathcal{D}_\cdot$ of quartets of the first category $\mathcal{D}_\cdot$ of $\mathcal{D}$, and the first composition is deduced pointwise from that of $\mathcal{D}^*$ (see Figure C.3).

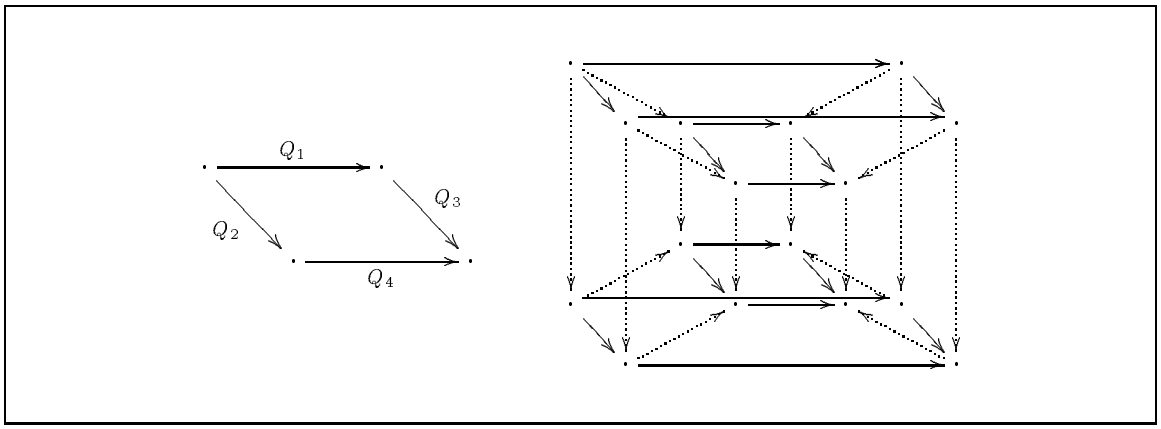


Figure C.4: A block in the 4-fold category $SqSq\mathcal{D}$ of a double category $\mathcal{D}$.

C.3 The 4-fold category $SqSq\mathcal{D}$

The 4-fold category $SqSq\mathcal{D}$ is constructed as follows:

- the set of its blocks is $\square((Sq\mathcal{D})^1)$, i.e., each block (Q_1, Q_2, Q_3, Q_4) is a quartet of $(Sq\mathcal{D})^1$, where $Q_i = (A_i, B_i, C_i, D_i)$ is a quartet of $\mathcal{D}$ (i.e., A_i, B_i, C_i and D_i are cells of $\mathcal{D}$ and $A_i \cdot C_i = B_i \cdot D_i$) for $i = 1, \ldots, 4$. Thus, we can picture a block of $SqSq\mathcal{D}$ as the *frame* in Figure C.4.

- The 1-st and 2-nd compositions are deduced pointwise from those of $\boxminus(\mathcal{D})$ and $\square\square(\mathcal{D})$, thus they consist in putting one frame *below* the other and one frame *inside* the other.

- The 3-rd and 4-th categories of $SqSq\mathcal{D}$ are the categories $\boxminus((Sq\mathcal{D})^1)$ and $\square\square((Sq\mathcal{D})^1)$, whose compositions are induced from that of $\mathcal{D}^*$ and consist in putting one frame *behind* the other (i.e., the block in Figure C.4 has source Q_1 and target Q_4) and one frame *beside* the other (source Q_2 and target Q_3).

C.4 Hypertransformation

We conclude by relating the definition of generalized natural transformation as given in Section 5.2.3 to the more general notion of *hypertransformation*.

DEFINITION C.4.1 (HYPERTRANSFORMATION)
Let $\vec{\mathcal{C}}$ and $\vec{\mathcal{D}}$ be n-fold categories. Then consider the category $\square_n\vec{\mathcal{D}}$ which is obtained by applying n times the Sq construction and then swapping the order of the $2n$ categories in such a way that, for each $i = 1, ..., n$ the i-th category of $\square_n\vec{\mathcal{D}}$ is derived from $\boxminus(\mathcal{D}^i)$, and the $(i+n)$-th category of $\square_n\vec{\mathcal{D}}$ is derived from $\square\square(\mathcal{D}^i)$. Then $Hom(\vec{\mathcal{C}}, \square_n\vec{\mathcal{D}})$ is the category of hypertransformations *from $\vec{\mathcal{C}}$ to $\vec{\mathcal{D}}$.*

Let $\mathcal{D}$ and $\mathcal{E}$ be double categories. We denote by $\Box_2\mathcal{E}$ the 4-fold category which is obtained by permuting the 2-nd and 3-rd compositions in $SqSq\mathcal{E}$. It follows that a multiple functor η from $\mathcal{D}$ to $\Box_2\mathcal{E}$ maps objects of $\mathcal{D}$ into cells of $\mathcal{E}$, horizontal arrows of $\mathcal{D}$ into horizontal commuting squares (of cells) of $\mathcal{E}$, vertical arrows of $\mathcal{D}$ into vertical commuting squares (of cells) of $\mathcal{E}$, and cells of $\mathcal{D}$ into commuting hypercubes of $\mathcal{E}$. The functoriality of η means that if two cells $A, B \in \mathcal{D}$ can be vertically (horizontally) composed, then the hypercube ηB can be composed inside (beside) ηA. Therefore, the collection of hypercubes $\{\eta A\}_{A\in\mathcal{D}}$ corresponds to the collection of commuting hypercubes of a generalized transformation, as illustrated in Section 5.2.3. In fact, the double category $Hom(\mathcal{D}, \Box_2\mathcal{E})$ can be described as follows:

- its objects (i.e., vertices) are double functors from $\mathcal{D}$ to $\mathcal{E}$;
- its horizontal arrows (i.e., objects for the 1-st category of $Hom(\mathcal{D}, \Box_2\mathcal{E})$) are natural $*$-transformations;
- its vertical arrows (i.e., objects for the 2-nd category of $Hom(\mathcal{D}, \Box_2\mathcal{E})$) are natural $\cdot$-transformations;
- its cells are generalized natural transformations between four double functors.

Finally, notice that compositions are deduced pointwise from the compositions of one frame *inside* the other and one frame *beside* the other (we remind that hypertransformations behave functorially w.r.t. the two compositions of one frame *below* the other and one frame *behind* the other).

Appendix D

Maude

In this appendix we summarize some interesting features of Maude and explain some major differences between the Maude-like notation that we have employed and the syntax of the Maude implementation.

D.1 Basic Syntax

Functional modules define data types and functions on them by means of equational theories. The equational logic on which Maude functional modules are based is membership equational logic: it supports sorts, subsorts, overloading of function symbols, and also membership axioms, where a term is asserted to have a certain sort if a condition consisting of a conjunction of equations and of membership assertions is satisfied.

We can illustrate some of these ideas using the module `FRAG-CCS` in Table D.1 that corresponds to a fragment of CCS (see Section 7.3). The module is introduced with the functional module syntax

`fmod` *name* `is ... endfm`

and has a name `FRAG-CCS`. The declaration `protecting MACHINE-INT` imports a built-in module of machine integers.

The sorts and subsorts of this module are introduced by the declarations

```
sorts Channel Act Process .
subsort Channel < Act .
```

respectively (the latter declares that the set of channels is contained in the set of actions, which also includes tha action `tau`).

The operators are declared using the syntax:

`op` f `:` s_1 `...` s_n `->` s `.`

```
fmod FRAG-CCS is
 protecting MACHINE-INT .
 sorts Channel Act Process .
 subsort Channel < Act .

 op a : MachineInt -> Channel .
 op tau : -> Act .
 op bar : Channel -> Channel .

 op nil : -> Process .
 op pre : Act Process -> Process .
 op plus : Process Process -> Process [assoc comm id: nil] .
 op par : Process Process -> Process [assoc] .

 var A : Channel .

 eq bar(bar(A)) = A .
endfm
```

Table D.1: A fragment of CCS in Maude.

where $n \geq 0$ (if $n = 0$ as for `nil` and `tau`, then f is a constant of sort s).

The attribute `[assoc]` states that the parallel composition `par` is associative. This information is used by the Maude engine that matches the equations in the module regardless of how parentheses are left- or right-associated. Moreover the simpler syntax `par(`P_1`,`P_2`,...,`P_n`)` can be used for any $n \in \mathbb{N}$, exploiting the associativity of `par`.

Similarly, the attribute `comm` declares the commutativity of `plus`, and the attribute `id: nil` says that `nil` is the identity for `plus`.

In general, the Maude engine can rewrite modulo different combinations of associativity, commutativity, identity (left-, right-, or two sided), and idempotency [34]. Therefore, data structures as lists, sets, and multisets can be naturally represented in Maude.

The equations are defined using the syntax:

`eq` t `=` t' `.`

where the terms t and t' can also involve typed variables that must be declared using the following syntax:

`var` x_1 `:` s_1 `.`
⋮
`var` x_n `:` s_n `.`

or (for k variables of the same sort s):

`vars` x_1 ... x_k `:` s `.`

Also conditional equations can be declared. In this case the syntax is

`ceq` t `=` t' `if` ϕ_1 `and` $\cdots$ `and` ϕ_m `.`

where each ϕ_i is either an equation t_i `=` t'_i or a membership assertion t_i `:` s_i.

REMARK D.1.1 *Actually for equations in the conditional part of the sentences we should use the syntax* t_i `==` t'_i*, but we believe that this little abuse of notation will not create any confusion.*

In a similar way, unconditional and conditional membership assertions can be stated using the syntax:

`mb` t `:` s `.`
`cmb` t `:` s `if` ϕ_1 `and` $\cdots$ `and` ϕ_m `.`

As an example, we can define a sort of sequential processes (i.e., processes not containing parallel composition) for our fragment of CCS, using the following assertions:

```
sort SeqProcess .
subsort SeqProcess < Process .
var M : Act .
vars P Q : Process .
mb nil : SeqProcess .
cmb pre(M,P) : SeqProcess
    if P : SeqProcess .
cmb plus(P,Q) : SeqProcess
    if P : SeqProcess and Q : SeqProcess .
```

The kind of rewriting typical of functional modules consists of replacement of equals by equals (until the equivalent, fully evaluated value is found). In general, however, a set of rewrite rules is neither terminating nor Church-Rosser. The most general Maude modules are system modules (`mod` *name* `is` ... `endm`) that specify initial models of a rewrite theory. They extend functional modules by a set of labelled rewrite rules that can also be conditional. The syntax for this kind of rules is the following:

`rl` `[`*lab*`]` `:` t `=>` t' `.`
`crl` `[`*lab*`]` `:` t `=>` t' `if` ϕ_1 `and` $\cdots$ `and` ϕ_m `.`

Notice that the conditions of a conditional rewrite rule cannot contain rewrite tests. For example the following sentence, expressing the usual dynamic evolution of CCS processes, is unparsable:

`crl plus(`P`,`Q`) =>` P' `if` P `=>` P' `.`

D.2 Shorthands

To shorten the notation, we have used many (intuitive) shorthands throughout the paper. For completeness, we try to summarize here most of them.

D.2.1 Variable Declarations

We write

```
vars x⃗1 : s1 .
     x⃗2 : s2 .
      ⋮
     x⃗n : sn .
```

as a shorthand for

```
vars x⃗1 : s1 .
vars x⃗2 : s2 .
⋮
vars x⃗n : sn .
```

D.2.2 Subsort Declarations

We write

```
subsorts s1 ... sk < s'1 ... s'n .
```

as a shorthand for

```
subsort s1 < s'1 .
           ⋮
subsort s1 < s'n .
⋮
subsort sk < s'1 .
           ⋮
subsort sk < s'n .
```

We also write

```
subsorts s⃗1 < s⃗2 < ... <  s⃗n .
```

as a shorthand for

```
subsorts s⃗1 < s⃗2 .
subsorts s⃗2 < s⃗3 .
⋮
subsorts s⃗n−1 < s⃗n .
```

Furthermore, we sometimes write

```
subsorts Sl1 .
         Sl2 .
          ⋮
         Sln .
```

where each Sl_i is a list of subsort expressions, as a shorthand for

```
subsorts Sl1 .
subsorts Sl2 .
⋮
subsorts Sln .
```

D.2.3 Membership Assertions

We write

```
mbs t1 ... tk : s .
```

as a shorthand for

```
mb t1 : s .
⋮
mb tk : s .
```

D.2.4 Using `iff` in a Conditional Sentence

We write

```
cmb ψ1 ··· ψn iff φ1 and ··· and φm .
```

where each ψ_i is a membership assertion, as a shorthand for

```
cmb ψ1 if φ1 and ··· and φm .
        ⋮
cmb ψn if φ1 and ··· and φm .
cmb/ceq φ1 if ψ1 and ··· and ψn .
        ⋮
cmb/ceq φm if ψ1 and ··· and ψn .
```

where the use of the symbol `ceq`, rather than `cmb`, in the last m sentences depends on the kind of each sentence ϕ_i (equation or membership assertion).

Similarly for

`ceq` ψ_1 $\cdots$ ψ_n `iff` ϕ_1 `and` $\cdots$ `and` ϕ_m .

where each ψ_i is an equation.

D.3 Built-ins

D.3.1 Booleans

In the present version of Maude, the sort `Bool` with constants `true` and `false` is added implicitly to any module. However, no boolean functions are added, so that `and` must be defined explicitly. For simplicity, we have assumed that the built-in module comes equipped with the usual operations of `and`, `or`, and `not` defined as follows (as we have done to run our examples):

```
op _and_ : Bool Bool -> Bool [assoc comm] .
op _or_ : Bool Bool -> Bool [assoc comm] .
op not : Bool -> Bool .

var TV : Bool .

eq true and TV = TV .
eq false and TV = false .

eq true or TV = true .
eq false or TV = TV .

eq not(true) = false .
eq not(false) = true .
```

D.3.2 Machine Integers

We have used the functional module `MACHINE-INT` in Section 7.3. It provides a fast arithmetic data type for general purpose programming. The idea is that the (infinite) constants of sort `MachineInt` represent the C++ data type *int*, and the various operations defined in the module represent their C++ counterparts. We refer the interested reader to [34] for its description.

D.3.3 Quoted Identifiers

The module `QID` plays an important role for meta-programming in Maude. It defines an infinite set of constants of sort `Qid` with names such as `'a`, `'aa`, `''1a2b3c`, etc. that are used to reify the names of sorts, operators and variables in the meta-level.

D.4 The Meta-Level

For efficiency reasons, the Maude implementation provides key features of the universal (finitely presented) rewrite theory U for rewriting logic in a built-in module called `META-LEVEL`. In particular, `META-LEVEL` provides sorts `Term` and `Module`, so that the meta-representations of terms and modules belong respectively to the sort `Term` and to the sort `Module`.

It also provides three functions `meta-reduce`, `meta-apply`, and `meta-rewrite` that return respectively the representation of the reduced form of a term t using the equations in a module T, the representation of the result of applying a rule labelled l in the module T to a term t *at the top*, and the representation of the result of rewriting a term t with the equations and the rules of a module T using Maude's default interpreter (this last feature has not been used in this paper).

We refer the interested reader to [34] for the extensive definition of the module `META-LEVEL` in the current version of Maude. For example, the representation of module `FRAG-CCS` in `META-LEVEL` is the following term of sort `Module`:

```
fmod('FRAG-CCS,
     protecting('MACHINE-INT),
     sort(qidSet('Channel, 'Act, 'Process)),
     subsort('Channel, 'Act),
     opDeclList(
       opDecl('a, 'MachineInt, 'Channel, emptyAttrSet),
       opDecl('tau, nilQidList, 'Act, emptyAttrSet),
       opDecl('bar, 'Channel, 'Channel, emptyAttrSet),
       opDecl('nil, nilQidList, 'Process, emptyAttrSet),
       opDecl('pre, qidList('Act, 'Process), 'Process,
              emptyAttrSet),
       opDecl('plus, qidList('Process, 'Process), 'Process,
              attrSet(assoc, comm, id('nil))),
       opDecl('par, qidList('Process, 'Process), 'Process,
              attrSet(assoc))),
     varDecl('A, 'Channel),
     emptyMembAxSet,
     eq('bar['bar['A]], 'A)
    )
```

Note that names of sorts, operators and variables are represented as quoted identifiers in the module `META-LEVEL`. For example, the operator `plus` is reified as `'plus`. Terms are reified as elements of the data type `Term` (complex terms are represented using the constructors `_[_]` and `_,_`). For example, the process

```
pre(a(1),nil)
```

is meta-represented as

```
'pre['a['1],'nil]
```

If an operator has infix syntax, then its meta-representation includes underscores for its argument places. For example the meta-representation of the term

```
true == false
```

is

```
'_==_['true, 'false]
```

Meta-representation can be iterated. For example, the meta-representation of the meta-term

```
'pre['a['1],'nil]
```

is

```
'_[_]['']pre, '_,_['_[_]['']a, ''1], ''nil]]
```

The declaration

```
protecting META-LEVEL[T] .
```

imports the module `META-LEVEL`, declares a new constant `T` of sort `Module`, and adds an equation making `T` equal to the representation of T in `META-LEVEL`. Therefore, we can regard `META-LEVEL` as a module-transforming operation that maps a module T to another module `META-LEVEL[`T`]` that is a definitional extension of U.

We have assumed a simplified (but consistent) version of `META-LEVEL`, whose relevant sorts and operators are listed in section 7.3.2. The main difference is that we have defined a parametric version of `META-LEVEL` (with a generic module T as the only parameter), assuming that `meta-reduce` and `meta-apply` apply reductions and rewriting only in the module T, which is passed as parameter. Furthermore, for the operation `meta-apply` we are not interested in the argument that can be used to apply a substitution to the variables in the rules of the module before testing their matching with the term to be rewritten. Therefore, the domain and codomain of the functions that we are using becomes

```
op meta-reduce : Term -> Term .
op meta-apply : Term Label Nat -> ResultPair .
```

instead of

```
op meta-reduce : Module Term -> Term .
op meta-apply : Module Term Qid Substitution MachineInt ->
                ResultPair .
```

(we have used the more intuitive names `Label` and `Nat` for the sorts `Qid` and `MachineInt` to facilitate the notation to readers not acquainted with the Maude syntax).

Our changes aimed at a more compact and readable Maude-like notation, abstracting from the details of the actual implementation. However, we have tested and experimented the corresponding version of each module defined in the paper using the beta version of Maude.

D.5 Rewriting Commands

D.5.1 Reduce

`reduce` t .

Causes the specified term t to be reduced using the equations and membership axioms in the current module. It admits the abbreviated form

`red` t .

D.5.2 Rewrite

`rewrite` t .

Causes the specified term t to be rewritten using the rules, equations and membership axioms in the current module. It admits the abbreviated form

`rew` t .

It is also possible to give a bound for the number of rule applications using the form

`rew [`n`]` t .

D.6 Parameterized Modules and Infix Operators

Functional modules can be unparameterized, or they can be parameterized with functional theories as their parameters. However, the present beta version only implements unparameterized modules. Furthermore, in the current version module hierarchies — except for protecting importation of built-in modules — are not supported either. Finally, although we have extensively used infix syntax for many operators, only prefix syntax is allowed in the current version, except for the syntax of built-in operators. Of course, all these restrictions will be removed in the future full version.

Index

Symbols

A

C

D

The screen suddenly filled up with the word:
LOADOVERLOADOVERLOADOVERLOADOVERLOADOVERLOADOVERLOAD
There was another pop, and then an explosion from the
CPU. Flames belched out of the cabinet and then died away.

— STEPHEN KING, Word Processor of the Gods