Università di Pisa

Dipartimento di Informatica

Technical Report : TR-98-09

Process and Term Tile Logic

R. Bruni, J. Meseguer, U. Montanari

July 22, 1998

ADDR: Corso Italia 40,56125 Pisa,Italy. TEL: +39-50-887111. FAX: +39-50-887226

Process and Term Tile Logic*

Roberto Bruni† bruni@di.unipi.it

José Meseguer‡ meseguer@csl.sri.com

Ugo Montanari§ ugo@di.unipi.it

Abstract

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 two of the authors, focusing on some interesting cases where the mathematical structures representing *configurations* (i.e., states) and *effects* (i.e., observable actions) 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 descriptions of two such cases where (net) *process-like* and usual *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 a recently developed framework, called *partial membership equational logic*, particularly suitable to deal with categorical structures. 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. This implementation uses a meta-layer to control the rewritings, so that only tile proofs are accepted. Making use of the *reflective* capabilities of the *Maude* language, some (general) *internal strategies* are then defined to implement the mapping from tile systems into rewriting systems, and some interesting applications related to the implementation of *concurrent process calculi* are presented.

*Research supported by Office of Naval Research Contracts N00014-95-C-0225 and N00014-96-C-0114, by National Science Foundation Grant CCR-9633363, and by the Information Technology Promotion Agency, Japan, as part of the Industrial Science and Technology Frontier Program "New Models for Software Architechture" sponsored by NEDO (New Energy and Industrial Technology Development Organization). Also research supported in part by U.S. Army contract DABT63-96-C-0096 (DARPA); CNR Integrated Project *Metodi e Strumenti per la Progettazione e la Verifica di Sistemi Eterogenei Connessi mediante Reti di Comunicazione*; and Esprit Working Groups *CONFER2* and *COORDINA*. Research carried out in part while the first and the third authors were visiting at Computer Science Laboratory, SRI International, and the third author was visiting scholar at Stanford University.

†Dipartimento di Informatica, Università di Pisa. Corso Italia n.40, 56125 Pisa, Italia.

‡Computer Science Laboratory, SRI International. 333 Ravenswood Av., Menlo Park CA 94025-3493, U.S.A.

§Dipartimento di Informatica, Università di Pisa. Corso Italia n.40, 56125 Pisa, Italia.

Contents

1 Introduction

The *tile model* [32, 35] 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 basic modules, which may interact through their interfaces. Roughly speaking, we consider a module to be just an *open* (e.g., partially specified) configuration of the system. Then, the behaviour of a whole system is defined as a coordinated evolution of its submodules. The name "tile" is due to the graphic representation of such rules. Graphically, a tile has the form

$$\begin{tikzcd} \circ \arrow[r, "s"] \arrow[d, "a"'] & \circ \arrow[d, "b"] \\ \circ \arrow[r, "s'"'] & \circ \end{tikzcd}$$

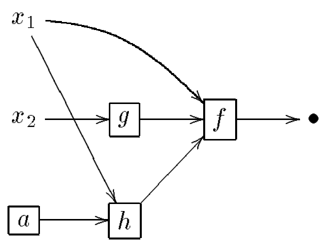
and textually it is written $s \xrightarrow[b]{a} s'$, stating that the *initial configuration* s of the system evolves to the *final configuration* s' producing an *effect* b, which can be observed by the rest of the system. However, such a step is allowed if and only if the subcomponents of s (which is in general an *open* configuration) evolve to the subcomponents of s', producing the *trigger* a. The vertices $\circ$ of the tile are called *interfaces*. Tiles can be composed horizontally (through side effects), vertically (computational evolutions of a certain component), and in parallel (concurrent steps) to generate larger steps. It is evident that the tile model extends rewriting logic [50] (in the non-conditional case), taking into account rewriting with side effects and rewriting synchronization, and can be naturally equipped with observational equivalences and congruences based on effects. In fact, in (non-conditional) rewriting systems, both triggers and effects are just identities; therefore rewriting steps may be applied freely, i.e., without interacting with the rest of the system. Thus, unconditional rewriting logic is obviously embedded in the tile formalism as a special case. The main goal of this paper is to investigate this connection in the opposite direction extending the results of [58] to the case in which configurations and effects rely on common auxiliary structures (e.g., for tupling, projecting or permuting interfaces). This is useful because there exist several languages based on rewriting logic, and the implementation of a conservative mapping of tiles into rewriting logic supports the execution of tile specifications. The nature of such structures will be more evident after a brief survey of the motivation for the introduction of tile systems, and of the techniques and tools employed in their semantical characterization.

The rich compositional nature of 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 widely used formalisms, intuitively arising from the operational understanding of a computational system. 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, representing the possible evolutions of the system. A set of *actions* (or *labels*) A is sometimes introduced to take into account also observational aspects: T becomes a ternary relation $T \subseteq S \times A \times S$, and an external observer may have discriminating capabilities over different evolutions between the same pair of states. In many cases, taking advantage of a possible compositional structure over the states, the relation T can be inductively defined according to that structure. As an example, the states of a *Petri net* [66] are multisets of *places*, an elementary evolution is a *transition* t that rewrites a multiset u_t to a multiset v_t, and 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. Thus, evolutions of a multiset are defined in terms of its subsets, and disjoint subsets may *concurrently* evolve. Another significant paradigm is given by *term rewriting systems* [50], where the states are *terms* of an algebra, and elementary evolutions are *rewriting steps* obtained (by closure under *substitution* and *contextualization*) from a set of *rewriting rules* (with free variables). Also the well-known *structural operational semantics* approach (*SOS*) [65] is a relevant generalization of this kind of methodology. We are especially interested in SOS specifications for *process description algebras* [2, 39, 59], where states are *terms* of a free algebra – whose operators reflect the basic composition aspects of the system – and a set of *inference rules* (guided by the structure of the states) inductively defines the transition relation. In recent years, the expressiveness and properties of a variety of SOS rule formats have been investigated and compared [67, 5, 37, 4]. *Context systems* [43], and *structured transition systems* [22, 26] are two interesting developments of the SOS approach. In the former, the transition relation is extended to *contexts* (that is, terms where free variables may occur) instead of *closed terms*, thus characterizing the behaviour of partially specified components of a system. In the latter, also transitions are equipped with an algebraic structure, usually by lifting the structure defined on the states in such a way that *computationally equivalent* evolutions are identified in the algebra of transitions. A similar methodology is also at the basis of *rewriting logic* [51, 53]: a logic theory is associated to a term 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. As an important result, equivalent computations correspond to the same sequent, and therefore deduction becomes equivalent to concurrent computing.

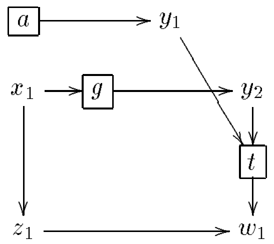
The *tile model* [32, 35] allows expressing rewrite rules with side effects, extending both the SOS approach and also context systems to a framework where the rules have a very general format, and, as already noticed, trigger and effects extends also rewriting systems with a mechanism of rewriting synchronization. This aspect is very important when modelling process algebras via a rewrite system, because the behaviour of most process algebras depends on the interaction between agents and "the rest of the world". By analogy with rewriting logic, the tile model also comes equipped with a purely logical presentation [35], where tiles are just considered as special (proof) sequents subject to certain inference rules. Since rewriting logic can be considered as a semantic framework for the study of concurrent systems with state changes, *tile logic* can be thought of as a logic of concurrent systems with conditional state changes and synchronization. Given a tile system, the associated tile logic is obtained by adding some auxiliary tiles and then freely composing in all possible ways (i.e., horizontally, vertically and in parallel) both auxiliary and basic tiles. Auxiliary tiles may be necessary to represent consistent rearrangements of the interfaces due to the topological structure of the actual configuration. To give a formal definition of auxiliary structure we assume the existence of the *categories* of configurations and effects (e.g., states in S and actions in A of the associated transition systems are arrows of categories). The advantages of using category theory in computer science are well summarized in [36]. We just remark here the following aspects: (a) suitable classes of (structure-preserving) *functors* between categories (representing transition systems) offer an immediate definition of *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*, *(co)reflections*, etc.) may be used to define a notion of *optimal* model; (e) *(co)limits* often summarize useful compositions also from a model theoretic viewpoint.

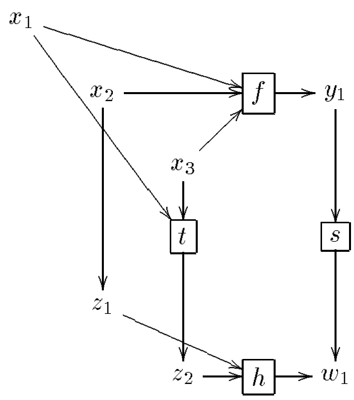
Moreover, categories generalize transition systems in an obvious way: states are *objects* and transitions are *arrows* equipped with a partial composition operator $\_;\_$ (associative and with *identities*), corresponding to the intuitive sequential composition of transitions for expressing computations (identities represent idle components of the system). As an example, monoidal categories can effectively model Petri net behaviours [57]; in particular, for each Petri net N, there exists a freely generated strictly symmetric strict monoidal category $\mathcal{T}[N]$ such that the monoidal operation $\_\otimes\_$ defines parallel composition of Best-Devillers processes, and the functoriality axiom (of tensor product $\_\otimes\_$) expresses a basic fact about the *true concurrency* of the model. A second example, showing that the use of categories offer a general and convenient characterization also of configurations, is given by *Lawvere theories*. An *algebraic theory* [44, 45, 40] is just a cartesian category having natural numbers as objects. The *free* algebraic theory associated to a (one-sorted) signature Σ is called the Lawvere theory for Σ, and is denoted with $\mathbf{Th}[\Sigma]$ (also $\mathcal{L}_\Sigma$): the arrows from m to

n are in a 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. From this point of view, the use of a *wires and boxes* notation turns out to be very useful for a visual and intuitive understanding of the role played by auxiliary structure: variables are represented by wires (we assume an implicit total order of the variables involved) and the operator of the signature are denoted by boxes labelled with the name of the operator. For instance, the term $f(x_1, g(x_2), h(x_1, a))$ over the signature $\Sigma = \{a : 0 \longrightarrow 1, g : 1 \longrightarrow 1, h : 2 \longrightarrow 1, f : 3 \longrightarrow 1\}$ and variables $x_1 < x_2$ admits the following graphical representation:



It should be obvious that wire duplications (e.g., of x_1) and crossing of wires (e.g., of x_2 and a copy of x_1) are auxiliary, in the sense that they belong to any wires and boxes model, independently from the underlying signature. It follows that, if we use the wires and boxes notation for configurations and effects, 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. To illustrate this point, let us consider a simple tile system where the above signature Σ is the signature of configurations, and $\Sigma' = \{s : 1 \longrightarrow 1, t : 2 \longrightarrow 1\}$ is the signature of effects, having the following basic tiles:





Then, it should be clear that the configuration $f(a, x_1, g(x_2))$ should be able to evolve to $h(x_1, x_2)$, producing an effect s (as a result of the horizontal composition, or synchronization, of the two tiles). However, we cannot compose the tiles in the obvious way without rearranging the interfaces, because the arguments of trigger t are separated by a variable in the initial (input) interface of the second tile (notice the crossing of wires), while the first tile applies only to adjacent arguments (notice that it is always possible to put an idle component in parallel with the first tile to model the second argument of f). Thus we have the following *naïve* characterization of auxiliary tiles:

> Auxiliary tiles coincide with the consistent rearrangements of interfaces in both dimensions, where consistency means that the composition of the wire transformations induced by the initial configuration and the effect of the tile is equivalent to the composition of the wire transformations due to the trigger and the final configuration.

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.

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, in the field of term rewriting, the states are terms over a certain signature (i.e., arrows of the associated Lawvere theory), and rewriting steps are transitions between two terms (with variables). It has been shown in [50], that a rewriting theory $\mathcal{R}$ yields a cartesian *2-category*[1] $\mathcal{L}_{\mathcal{R}}$, 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 [33], that if also side-effects are to be taken into consideration during the rewriting process, then *double categories* [25, 1, 41] 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 effect propagations, and the latter describing state evolutions. 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 tiles freely generate a (monoidal) double category which constitutes the natural operational characterization[2] in the spirit of initial model semantics.

In this paper we consider two main interesting cases of shared auxiliary structures. In particular the notions of *Process Tile Logic* and *Term Tile Logic* are introduced:

- *Flat* (e.g., any two sequents having the same "border" are identified, thus no emphasis is given upon the axiomatization of logic proofs) versions of process tile logic have been shown to be especially useful for defining compositional models of computation of mobile calculi, and causal and located concurrent systems [27, 28]. The auxiliary tiles of process tile logic express consistent permutations of interfaces along the horizontal and vertical structures.
- Term tile logic should represent the obvious extension of term rewriting logic. Connections between the two logics are particularly interesting because in both logics the underlying cartesian category structure manifests itself at the level of syntax, allowing the use of the standard term notation with term substitution as composition. 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.

The natural semantics of process and term tile logics 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 categories* and *Cartesian double categories* ("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. We believe that this should not be the case, both conceptually and for the kind of applications we have in mind; therefore we propose a broader notion of double cartesianity by developing an alternative approach, following the idea of *hyper-transformations* [25] for many-fold categories, and exploiting the results for double categories. In particular, we define the notion of *generalized transformations*, which act in both dimensions, and assert 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

[1] A 2-category [41, 46] is a category $\mathcal{C}$ such that, for any two objects a, and b, the class $\mathcal{C}[a,b]$ of arrows from a to b in $\mathcal{C}$, forms a (vertical) category. The arrows of these hom-categories are called *cells* and satisfy particular composition properties. As an example, the category $\mathbf{Cat}$ of categories and functors is a 2-category. Actually, $\mathbf{Cat}[\mathcal{C},\mathcal{C}']$ is the category having the functors from $\mathcal{C}$ to $\mathcal{C}'$ as objects, and the natural transformations between such functors as arrows.

[2] The tiles are cells, the contexts are arrows of the 1-horizontal category, the side-effects are the arrows of the vertical 1-category, and 0-objects model connections between the somehow syntactic horizontal category and the dynamic vertical evolution.

happens for the 1-dimensional case. Moreover, by doing that, we give evidence for the usefulness of axiomatizing the resulting double categories, thus allowing for the definition of more significant models than the flat ones. Actually such models could also take into account the structure of proofs. 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 a recently developed, more general framework, called *partial membership equational logic* [54, 56, 10]. In doing so, we extend the result of [58], by defining an extended version of 2-categories, called 2EVH-categories, providing a systematic connection between models of tile logic and of rewriting logic. The idea is to "stretch" double cells into ordinary 2-cells as pictured below, mantaining the capability to distinguish between configurations and effects, whereas the auxiliary structure becomes shared, i.e., it belongs to both classes.

$$\begin{tikzcd} \circ \arrow[r, "s"] \arrow[d, "a"'] & \circ \arrow[d, "b"] \\ \circ \arrow[r, "s'"'] & \circ \end{tikzcd} \qquad \begin{tikzcd} & \circ \arrow[dr, bend left, "b"] & \\ \circ \arrow[ur, bend left, "s"] \arrow[dr, bend right, "a"'] & \Downarrow & \circ \\ & \circ \arrow[ur, bend right, "s'"'] & \end{tikzcd}$$

Doing this, 2EVH-categories are able to *simulate* – in the sense that the algebraic structure of the original 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 flattening process 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. From the facts 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 (in the sense that the composition of s with b is equivalent to the composition of a with s'), 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 different definitions of auxiliary tiles that we have sketched in this introduction coincide.

Partial membership equational logic is particularly suitable for the modelling and the embedding of categorical structures, firstly because the sequential composition of arrows is a partial operation (e.g., it is defined if and only if the target of the first argument is equal to the source of the second argument), and secondly because membership predicates over a poset of sorts allow modelling the objects as a subset of the arrows and arrows as a subset of cells (as it is usually done in category theory). Moreover, the *tensor product construction* illustrated in [58] can be easily formulated in partial membership equational logic and this allows for a convenient definition of monoidal double categories as the tensor product of the theory of categories (twice) with the theory of monoids.

Though the results are very satisfying from a theoretical perspective, they cannot be applied directly to rewriting implementations of tile systems, because we are interested only in *correct* computations. Indeed, we need suitable meta-strategies to control the possible nondeterminism contained in a tile specification and in its translation. This could be summarized by saying that "the rewriting engine must be able to filter rewriting computations". To overcome this difficulty, we make use of the *reflective* capabilities [17, 18] of the rewriting logic language Maude [15] to define suitable *internal strategies* [19], which help the user control the computation and 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 rewriting systems in general. We have experimented with Maude some executable tile specifications of interesting CCS-like process calculi, and have successfully developed and applied general internal strategies to filter and collect tile computations.

The structure of the paper is as follows. In Section 2 we recall some basic facts about *algebraic theories*, *rewriting logic*, and *tile logic* (Section 2.1), and then we introduce the new tile models based on process-like and term structures of configurations and effects. Each model is presented in its flat version first, then is equipped with an algebra of proofs, and then naturally equivalent proof terms are equated to characterize the natural semantic framework of the logic.

In Section 3, we introduce suitable categorical models for process and term tile logic, developing the notion of *generalized transformation* and *diagonal categories* to deal with symmetries, duplicators and dischargers. As a result, we propose a precise characterization of *symmetric strict monoidal double categories* and *cartesian double categories with chosen products*.

In Section 4 and 5 we present the full comparison between tile logic and rewriting logic through partial membership equational logic, then showing how to map tiles into ordinary rewrite rules. As a result of this comparison, we obtain a correct rewriting implementation of tile logic, in which different tile sequents having the same "border" cannot always be distinguished. This implementation requires a meta-layer to control the rewritings, so that only tile proofs are accepted. In Section 6 we present some general meta-strategies (written in the Maude language) fulfilling this last requirement.

In Section 7 we apply the previous results to show how Maude – thanks to its reflective capabilities and, in particular, to the possibility of defining internal strategy languages – can in fact be used to prototype and execute tile rewriting systems. In particular, we define executable implementations of some CCS-like process calculi (namely, finite CCS and located CCS), preserving their original semantics.

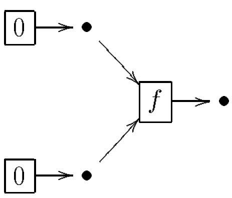
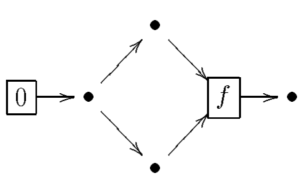
2 Tile Logic

Tiles are rewrite rules with side-effects, extending the SOS approach to *open systems* and also to *heterogeneous systems*. A generic tile has the form $s \xrightarrow[b]{a} s'$, stating that the partially specified configuration s may evolve to s' producing an observable effect b, but this rewriting step is allowed if and only if the subcomponents of s evolve to the subcomponents of s' producing the observation a, which is the trigger of the rule. 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. In fact, tiles can be combined by means of three composition operators, extending those defined on their border: parallel ($\_ \otimes \_$), horizontal ($\_ * \_$), and vertical ($\_ \cdot \_$) composition. Parallel composition intuitively corresponds to the concurrent rewriting of disjoint components of the system. Vertical composition models successive rewriting, i.e., computations. Horizontal composition synchronizes evolutions of a configuration and its subcomponents.

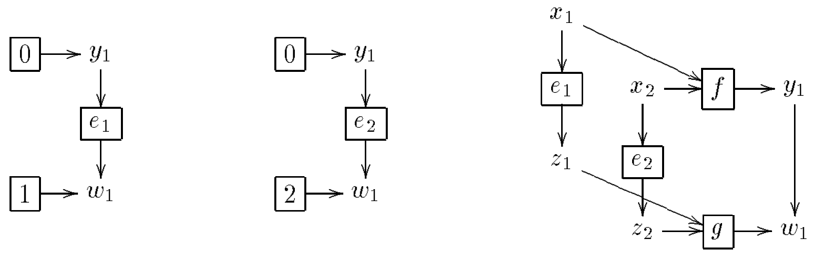
Although tile systems are essentially monoidal double categories [25], the tile model allows for a purely logical presentation, where tiles are considered as *sequents* (subject to certain inference rules and normalization axioms), in the style of *rewriting logic*. Then, *deduction* in the tile logic exactly corresponds to *computing* in the tile model (i.e., applying composition rules in all possible ways, starting from a set of *basic tiles*), and the axioms of tile logic identify equivalent *proofs* of a sequent entailed by the logic.

The simplest possible interpretation of structured configurations and observations is considered in [11, 12], consisting of P/T net markings. As an important result, horizontal composition in the tile model yields a notion of *transition synchronization*, an important feature for compositionality, missing in ordinary nets (where only *token* synchronization is provided), and usually achieved through complex constructions. As an another example, tile models for most process algebras [35] have process terms as configurations, and elements of the free monoid on observable actions (which are unary symbols) as observations. However, when either causality aspects or bound names are taken into account, it is possible to consider more general horizontal and vertical structures, dealing with (local and global) *names*.

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 process algebra with restriction (or name hiding mechanisms) 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. In general, names can be freely α-converted, because the only important information they offer is *sharing*. The wires and boxes notation presented in the introduction can give an intuitive understanding of a name sharing mechanism. Let us consider a certain signature Σ with constants 0, 1 and 2, and binary operators f and g. Then the configurations c_1 and c_2 in the picture below can model quite different systems.

$c_1 \equiv$  $c_2 \equiv$ 

In a *value-oriented* interpretation, both c_1 and c_2 yield the same term $f(0,0)$. Instead, in a *reference-oriented* interpretation, c_1 and c_2 define different situations: in the former the two subcomponents of the f box are uncorrelated, while in the latter they point to the same *shared* location. The difference becomes even more clear, if we assume a tile system in which the configuration 0 may be rewritten either to 1, producing an effect e_1, or to 2, with effect $e_2 \neq e_1$, and the configuration $f(x_1, x_2)$ may be rewritten to $g(z_1, z_2)$ only if x_1 yields e_1 and x_2 yields e_2 as triggers, becoming z_1 and z_2, i.e., the basic tiles of the system are as follows:



Then, c_1 may be rewritten, while c_2 cannot; in fact, if we try to rewrite 0 with the first tile, the same effect e_1 is propagated to both arguments of f, and the configuration is stuck, because we cannot apply the third tile, and similarly if we try to rewrite 0 with the second tile.

Term graphs [24] 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[3]. The distinction is made very precisely by the axiomatization of *algebraic theories*: terms and term graphs differ by two axioms, representing, in a categorical setting, the *naturality* of transformations for copying and discharging arguments [20]. Term graphs have been shown useful in [27] to define a tile model for the (asynchronous) π-calculus [60] (one of the most studied mobile calculi), and in [28] to represent both the operational and the abstract semantics of CCS [59] with locations [9] within the tile model. In both cases, *flat* versions of the tile model are used, and the general notion of *tile bisimilarity* [35] is employed to quotient out configurations, thus recovering the ordinary abstract semantics.

In this section we introduce two versions of tile logic, called *Process Tile Logic*, and *Term Tile Logic*. They model two specific situations in which the structure of configurations and observations are quite similar, and a set of *auxiliary tiles* seems to capture precisely their similarity.

Configuration and observation in process tile logic 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 [23]. 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 and cooperative evolutions of distributed agents. Models proposed in [27, 28] are essentially flat process tile logic modelsa equipped with "ad-hoc" notions of sharing and garbage collection. Auxiliary tiles for process tile logic are essentially tiles for consistent permutations of interfaces.

Term tile logic is the natural generalization of term rewriting logic. Here, both configurations and observations are term algebras. Thanks to the work of Lawvere relating *algebraic theories* and cartesian categories, and to classical results on cartesianity (with chosen products) as enriched monoidality, the auxiliary structure which allows the generation of the term algebra starting from a signature is characterized by three natural transformations called *symmetries*, *duplicators*, and *dischargers*. Similarly, auxiliary tiles of term tile logic are the consistent generalization of such transformations w.r.t. the two dimensions of tile systems.

Intuitively, in process and term tile logic, configurations and observations have in common the auxiliary structure, i.e., the possibility of re-arranging the interfaces as explained in the introduction. Moreover, auxiliary tiles model exactly the consistent re-arrangements, 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.

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

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 3. However, for the sake of an easier presentation, and to afford a better intuitive understanding of the main ideas with the minimum machinery possible, we have chosen to reverse the "mathematically natural" order of the two formalizations.

2.1 Background

2.1.1 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 2.1 [(Hyper)Signatures] A *many-sorted hyper-signature* Σ over a set S of sorts is a family $\{\Sigma_{w,w'}\}_{w,w' \in S^*}$ of sets of operators. A *many-sorted signature* is just a hyper-signature such that $\Sigma_{w,w'} \neq \emptyset \Longrightarrow w' \in S$, i.e., a family $\{\Sigma_{w,s}\}_{w \in S^*, s \in S}$. If S is a singleton, we denote the hyper-signature (signature)Σ is called *one-sorted* and is simply denoted by the family $\{\Sigma_{n,m}\}_{n,m \in \mathbb{N}}$ ($\{\Sigma_n\}_{n \in \mathbb{N}}$). $\square$

Definition 2.2 [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*. We use the standard notation $f : a \longrightarrow b$ to denote an arrow f with source a and target b. A graph G is *reflexive* if there exists an *identity* function $id : O_G \longrightarrow A_G$ such that $\forall a \in O_G$, $\partial_0(id(a)) = a = \partial_1(id(a))$; 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). $\square$

It is immediate that a many-sorted hyper-signature Σ over S may 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 of the signature (i.e., $f : w \longrightarrow w' \in A_{G_\Sigma}$ iff $f \in \Sigma_{w,w'}$).

For simplicity, throughout the paper we will consider one-sorted hyper-signature only, but the results extend immediately to the many-sorted case.

Definition 2.3 [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 natural numbers (i.e., 0 is the neutral element, and the monoidal operation $\_\otimes\_$ is defined as $n \otimes m = n+m$), and arrows those generated by the following inference rules:

$$(generators)\ \frac{f \in \Sigma_{n,m}}{f : n \longrightarrow m \in \mathbf{G}(\Sigma)} \qquad (pairing)\ \frac{t : n \longrightarrow m,\ t' : n' \longrightarrow m' \in \mathbf{G}(\Sigma)}{t \otimes t' : n \otimes n' \longrightarrow m \otimes m' \in \mathbf{G}(\Sigma)}$$

$$(identities)\ \frac{n \in \mathbb{N}}{id_n : n \longrightarrow n \in \mathbf{G}(\Sigma)}$$

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 $n, m \in \mathbb{N}$. $\square$

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. For the sake of simplicity, we treat here one-sorted signatures only, but the extension to the many-sorted case should follow immediately.

Definition 2.4 [Monoidal Theories, Symmetric Theories] Given a (hyper)signature Σ, the associated *monoidal theory* $\mathbf{M}(\Sigma)$ is the monoidal graph with objects the elements of the additive monoid of natural numbers (i.e., 0 is the neutral element, and the monoidal operation $\_ \otimes \_$ is defined as $n \otimes m = n + m$), and arrows those generated by the following inference rules:

$$(generators)\ \frac{f \in \Sigma_{n,m}}{f : n \longrightarrow m \in \mathbf{M}(\Sigma)} \qquad (pairing)\ \frac{t : n \longrightarrow m,\ t' : n' \longrightarrow m' \in \mathbf{M}(\Sigma)}{t \otimes t' : n \otimes n' \longrightarrow m \otimes m' \in \mathbf{M}(\Sigma)}$$

$$(identities)\ \frac{n \in \mathbb{N}}{id_n : n \longrightarrow n \in \mathbf{M}(\Sigma)} \qquad (composition)\ \frac{t : n \longrightarrow m,\ t' : m \longrightarrow k \in \mathbf{M}(\Sigma)}{t; t' : n \longrightarrow k \in \mathbf{M}(\Sigma)}$$

Moreover, $\_\otimes\_$ is associative on arrows with identity id_0, the composition operator $\_;\_$ is associative, and the arrows of $\mathbf{M}(\Sigma)$ satisfy the *identity axiom* ($\forall t : n \longrightarrow m$), $id_n; t = t = t; id_m$, and the *functoriality axiom* $(s \otimes t);(s' \otimes t') = (s; s') \otimes (t; t')$ (whenever compositions $s; s'$ and $t; t'$ are defined).

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:

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

Moreover, the arrows of $\mathbf{S}(\Sigma)$ satisfy the *naturality axiom* ($\forall t : n \longrightarrow m,\ t' : n' \longrightarrow m'$),

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

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

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

□

Actually, a (symmetric) monoidal theory is just a particular (symmetric) strict monoidal category [46], namely the free such category generated by the signature Σ.

Definition 2.5 [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:

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

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

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

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

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

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

□

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, Def. 2.5 can be proved equivalent to the classical Lawvere theory construction $\mathbf{Th}[\Sigma]$, dating back to the early work of Lawvere [44]. A classical result states the equivalence of these theories with the usual term algebra.

Definition 2.6 [Σ-Algebra] Given a signature $\Sigma = \{\Sigma_n\}_{n\in\mathbb{N}}$, a *$\Sigma$-algebra* is a set A, together with an assignment of a function $A_f : A^n \longrightarrow A$ for each $f \in \Sigma_n$. $\square$

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 2.7 *Let Σ be a signature. Then, for all $n, m \in \mathbb{N}$, there exists a one-to-one correspondence between the set $\mathbf{A}(\Sigma)[n, m]$ of arrows from n to 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 the constructive definition of algebraic theories separates very nicely the auxiliary structure from the Σ-structure (better than the ordinary description involving the meta-operation of substitution). Moreover, the naturality axioms of ∇ and $!$ allow a controlled form of *duplication* and *discharging* of information.

2.1.2 Rewriting Logic

Rewriting logic [50, 51, 53] 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 [48, 49]. A workshop [55] has been recently dedicated to a great miscellany of different aspects of rewriting logic, relating many different subjects (object-oriented programming, reflection, external and internal strategies, different categorical interpretations of rewriting logic, semantic basis for language implementations, actor systems). Here we just sketch an introductory description of the subject and the original 2-algebraic semantics as proposed by Meseguer in [50]. A short summary of the reflective capabilities of rewriting logic will be given in Section 6.2.

Let Σ be a signature. Given a set E of Σ-equations (i.e., sentences of the form $t = t'$ with $t, t' \in T_\Sigma(X)$), $T_{\Sigma,E}$ (resp. $T_{\Sigma,E}(X)$) denotes the Σ-algebra of equivalence classes of ground Σ-terms modulo the equations in E (the Σ-algebra of equivalence classes of Σ-terms with variables in X modulo the equations in E). We denote the congruence modulo E by $\_ =_E \_$, and the E-equivalence class of a Σ-term t by $[t]_E$, or just $[t]$.

Definition 2.8 [Rewrite Theory] 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 $r : [t] \Rightarrow [t']$. $\square$

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

Definition 2.9 [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 following *rules of deduction*.
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

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

where $t(\vec{w}/\vec{x})$ denotes the *simultaneous* substitution of w_i for x_i in t. □

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 Def. 2.9 by decorating the sequents with appropriate proof terms.

Definition 2.10 [Proof Terms of Rewrite 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 following *decorated* rules of deduction.
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

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

Each of the rules presented above 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). □

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 2.11 [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:

$$\forall \alpha, \beta, \gamma, \quad \alpha \circ (\beta \circ \gamma) = (\alpha \circ \beta) \circ \gamma$$

$$
\begin{tikzcd}
\mathcal{T}_{\mathcal{R}}(X)^n & \langle [w_1], \ldots, [w_n] \rangle \arrow[d, "\langle \alpha_1, \ldots, \alpha_n \rangle"] & {[t(\vec{w})]} \arrow[r, "r(\vec{w})"] \arrow[d, "{[t(\vec{\alpha})]}"'] \arrow[dr, dotted, "r(\vec{\alpha})"] & {[t'(\vec{w})]} \arrow[d, "{[t'(\vec{\alpha})]}"] & \mathcal{T}_{\mathcal{R}}(X) \\
 & \langle [w'_1], \ldots, [w'_n] \rangle & {[t(\vec{w}')]} \arrow[r, "r(\vec{w}')"'] & {[t'(\vec{w}')]} &
\end{tikzcd}
$$

Figure 1: Graphical representation of the **Exchange** law as a natural transformation.

Identities:

$$\forall \alpha : [t] \Rightarrow [t'], \quad \alpha \circ [t'] = \alpha = [t] \circ \alpha$$

Functoriality of the Σ-algebraic structure ($\forall f \in \Sigma_n$)
Preservation of composition:

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

Preservation of identities:

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

E-axioms ($\forall t(x_1, \ldots, x_n) = t'(x_1, \ldots, x_n) \in E$)

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

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

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

Note that the set X of variables is actually a parameter of these constructions, and we need not assume X to be fixed and countable. In particular, for $X = \emptyset$ we adopt the notation $\mathcal{T}_{\mathcal{R}}$. $\square$

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 Fig. 1.

Lemma 2.12 ([51]) *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 $[t'(x_1, \ldots, x_n)] : \mathcal{T}_{\mathcal{R}}(X)^n \longrightarrow \mathcal{T}_{\mathcal{R}}(X)$.*

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 2.13 [$\mathcal{R}$-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 $\mathcal{R}$. $\square$

The following theorem [51] characterizes the relevance of the models $\mathcal{T}_\mathcal{R}$ and $\mathcal{T}_\mathcal{R}(X)$.

Theorem 2.14 (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 natural numbers as objects, and where an equivalence class $[t(x_1, \ldots, x_n)]$ is viewed as an arrow $[t(x_1, \ldots, x_n)] : n \longrightarrow 1$ (from n placeholders for the n ordered variables of t to the placeholder for the result), arrow composition being substitution (that is, given n arrows $[u_i(y_1, \ldots, y_m)] : m \longrightarrow 1$, for $i = 1, \ldots, n$, and an arrow $[t(x_1, \ldots, x_n)] : n \longrightarrow 1$, the composition between $\langle [u_1], \ldots, [u_n] \rangle : m \longrightarrow n$ and $[t(x_1, \ldots, x_n)]$

$$m \xrightarrow{\langle [u_1], \ldots, [u_n] \rangle} n \xrightarrow{[t]} 1$$

yields $[t(\vec{u}/\vec{x})] : m \longrightarrow 1$ as a result). In particular, for $T = (\Sigma, \emptyset)$, we have $\mathcal{L}_T \simeq \mathbf{A}(\Sigma)$, thanks to Proposition 2.7.

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 natural numbers as objects, E-equivalence classes of terms $[t(x_1, \ldots, x_n)]$ as arrows $[t(x_1, \ldots, x_n)] : n \longrightarrow 1$, and equivalence classes of proof terms $[\alpha] : [t(x_1, \ldots, x_n)] \Rightarrow [t'(x_1, \ldots, x_n)]$ as cells, with vertical composition given by $[\alpha] \circ [\gamma] = [\alpha \circ \gamma]$, and horizontal composition $\alpha ; \beta$ 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 Fig. 2, the exchange law states the coherence between $\_;\_$ and $\_ \circ \_$.

As a matter of fact, given an $\mathcal{R}$-systems $\mathcal{S}$, 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 **Cat**. This result can be summarized by saying that $\mathcal{L}_\mathcal{R}$ does for $\mathcal{R}$-systems what in the **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 [21] it is noticed that when the rules of $\mathcal{R}$ are not right linear – that is, there is a repeated occurrence of a variable

$$
\begin{array}{c}
\begin{tikzcd} m \arrow[r, "\langle w_1,\ldots,w_n\rangle"] & n \arrow[r, bend left, "t"] \arrow[r, bend right, "t'"'] \arrow[r, phantom, "\Downarrow r"] & 1 \end{tikzcd} \\
\circ \\
\begin{tikzcd} m \arrow[r, bend left, "\langle w_1,\ldots,w_n\rangle"] \arrow[r, bend right, "\langle w'_1,\ldots,w'_n\rangle"'] \arrow[r, phantom, "\Downarrow\vec{\alpha}"] & n \arrow[r, "t'"] & 1 \end{tikzcd}
\end{array}
= \begin{tikzcd} m \arrow[r, bend left, "\langle w_1,\ldots,w_n\rangle"] \arrow[r, bend right, "\langle w'_1,\ldots,w'_n\rangle"'] \arrow[r, phantom, "\Downarrow\vec{\alpha}"] & n \arrow[r, bend left, "t"] \arrow[r, bend right, "t'"'] \arrow[r, phantom, "\Downarrow r"] & 1 \end{tikzcd} =
\begin{array}{c}
\begin{tikzcd} m \arrow[r, bend left, "\langle w_1,\ldots,w_n\rangle"] \arrow[r, bend right, "\langle w'_1,\ldots,w'_n\rangle"'] \arrow[r, phantom, "\Downarrow\vec{\alpha}"] & n \arrow[r, "t"] & 1 \end{tikzcd} \\
\circ \\
\begin{tikzcd} m \arrow[r, "\langle w'_1,\ldots,w'_n\rangle"] & n \arrow[r, bend left, "t"] \arrow[r, bend right, "t'"'] \arrow[r, phantom, "\Downarrow r"] & 1 \end{tikzcd}
\end{array}
$$

$$[r(\vec{w}/\vec{x}) \circ t'(\vec{\alpha}/\vec{x})] = [\vec{\alpha}; r](= [r(\vec{\alpha}/\vec{x})]) = [t(\vec{\alpha}/\vec{x}) \circ r(\vec{w'}/\vec{x})]$$

Figure 2: Graphical representation of the **Exchange** law in $\mathcal{L}_{\mathcal{R}}$.

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 model. In this way, the relationship between rewriting logic models and event structures is clarified, and useful connections with other concurrency models are provided. In [64] the treatment of conditional rules in the functorial model for conditional rewriting logic of [51] is generalized and reformulated in terms of *weighted limits* rather than 2-limits, i.e., using *inserters* instead of subequalizers (which however coincide in **Cat**). For simplicity, in what follows we restrict ourselves to the original semantics proposed by Meseguer [50], and to rewrite theories with an empty set of equations (i.e., $E = \emptyset$).

2.1.3 Algebraic Tile Logic

The rule format of *algebraic rewriting systems* [35] extends the one of rewriting systems to deal with side-effects 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 2.15 [Algebraic Rewriting System] An *algebraic rewriting system* (*ARS* for short) $\mathcal{R}$ is a quadruple $\langle \Sigma_H, \Sigma_V, N, R\rangle$, where Σ_H and Σ_V are 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 $s : n \longrightarrow m$, $t : k \longrightarrow l$, $a : n \longrightarrow k$, and $b : m \longrightarrow l$ for some natural numbers n, m, k, and l. □

As usual, we will write such a rule d as a sequent $d : s \xrightarrow[b]{a} t$ or, graphically, as the tile

$$\begin{tikzcd} n \arrow[r, "s"] \arrow[d, "a"'] \arrow[dr, phantom, "d"] & m \arrow[d, "b"] \\ k \arrow[r, "t"'] & l \end{tikzcd}$$

thus making explicit the source and target of each operator. It follows that a term rewriting system is just an ARS with $\Sigma_V = \emptyset$, and a context system [43] is an ARS 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 ARS) 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 of 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 2.16 [Algebraic Tile Logic] Let $\mathcal{R} = \langle \Sigma_H, \Sigma_V, N, R \rangle$ be an ARS. 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 following *deduction rules*[4]:

Basic Proof Sequents

Generators:

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

Identities:

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

Auxiliary Sequents

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

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

Composition Rules

Parallel composition:

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

Sequential compositions:

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

Moreover, we say that $\mathcal{R}$ *entails* the class $S_a(\mathcal{R})$ of *flat algebraic sequents* $s \xrightarrow[b]{a} t$ (written $\mathcal{R} \vdash_{fa} s \xrightarrow[b]{a} t$) iff there exists a decorated algebraic sequent $\alpha \in P_a(\mathcal{R})$ such that $\alpha : s \xrightarrow[b]{a} t$. $\square$

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.

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

Definition 2.17 [Abstract Algebraic Tile Logic] Let $\mathcal{R} = \langle \Sigma_H, \Sigma_V, N, R\rangle$ be an ARS. We say that $\mathcal{R}$ *entails* the class $A_a(\mathcal{R})$ of *abstract algebraic sequents*, whose elements are equivalence classes of proof terms in $P_a(\mathcal{R})$ modulo the following set of axioms on proof terms:
Associativity Axioms for $\_ \otimes \_$, $\_ * \_$, and $\_ \cdot \_$.
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 $c, d \in \mathbf{M}(\Sigma_V)$):

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

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 $a, b, c \in \mathbf{M}(\Sigma_V)$):

$$\begin{array}{ll} \gamma_{a\otimes b,c} = (1_a \otimes \gamma_{b,c}) * (\gamma_{a,c} \otimes 1_b) & \nabla_a * \gamma_{a,a} = \nabla_a \\ \nabla_{a\otimes b} = (\nabla_a \otimes \nabla_b) * (1_a \otimes \gamma_{a,b} \otimes 1_b) & \nabla_a * (1_a \otimes \nabla_a) = \nabla_a * (\nabla_a \otimes 1_a) \\ !_{a\otimes b} = !_a \otimes !_b & \nabla_a * (1_a \otimes !_a) = 1_a \\ \gamma_{id_0,id_0} = 1_{id_0} = \nabla_{id_0} = !_{id_0} & \gamma_{a,b} * \gamma_{b,a} = 1_{a\otimes b} \end{array}$$

□

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

Definition 2.18 [Tile Bisimulation] Let $\mathcal{R} = \langle \Sigma_H, \Sigma_V, N, R\rangle$ be an ARS. A symmetric 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 [35]. An algebraic theory for CCS recovering the ordinary strong congruence via tile bisimulation has been defined in [32, 35]. In Section 7.1 we will adapt such a model to give an executable specification in term tile logic.

2.2 Naïve Process Tile Logic

The main limitation of algebraic tile logic is that configurations and effects can share only objects, even when their structure is very similar. As shown in the introduction, using the wires and boxes notation, it would be desirable to have a model where consistent re-arrangements of interfaces are allowed by default. In this section we define *process tile logic*, where all the consistent overlappings of wires are always introduced for free.

Flat versions of tile logic based on (richer) symmetric monoidal categories on both dimensions have been extensively used in [27, 28] They offer a very general specification framework. We formalize here this kind of situations, allowing also for the definition of more general versions than just flat ones (e.g., 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). As already said, for simplicity, we will consider only one-sorted hyper-signatures.

Definition 2.19 [Process Tile Rewrite System] A *process tile rewrite system* (*pTRS* for short) $\mathcal{R}$ is a quadruple $\langle \Sigma_H, \Sigma_V, N, R\rangle$, where $\Sigma_H = \bigcup_{i,j\in\mathbb{N}} \Sigma_{H,i,j}$ and $\Sigma_V = \bigcup_{i,j\in\mathbb{N}} \Sigma_{V,i,j}$ are hyper-signatures, 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 $\forall d \in N$, if $R(d) = \langle h, u, v, g\rangle$, 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). □

In the following we will write a generic rule r such that $R(r) = \langle h, v, u, g\rangle$ either as the tile

$$\begin{tikzcd} n \arrow[r, "h"] \arrow[d, "v"'] \arrow[dr, phantom, "r"] & m \arrow[d, "u"] \\ k \arrow[r, "g"'] & l \end{tikzcd}$$

(for appropriate natural numbers n, m, k and l) or as the sequent $r : h \xrightarrow[u]{v} g$. As an example, it is now possible to define a pTRS for the system presented in the introduction using the wires and boxes notation. In fact, let us consider a pTRS where $\Sigma_H = \{a : 0 \longrightarrow 1, g : 1 \longrightarrow 1, h : 2 \longrightarrow 1, f : 3 \longrightarrow 1\}$, $\Sigma_V = \{s : 1 \longrightarrow 1, t : 2 \longrightarrow 1\}$, and $N = \{r_1, r_2\}$, with $r_1 : a \otimes g \xrightarrow[t]{id_1} id_1$, and $r_2 : f \xrightarrow[s]{(\gamma_{1,1}\otimes id_1);(id_1\otimes t)} h$. Notice that a symmetry appears in the trigger of r_2. Now let us suppose that an auxiliary tile $\sigma : \gamma_{1,1} \xrightarrow[\gamma_{1,1}]{id_2} id_2$ is also introduced. Then, the three tiles can be composed to obtain a new tile for rewriting the configuration $(a \otimes id_1 \otimes g); f : 2 \longrightarrow 1$ (intuitively corresponding to $f(a, x_1, g(x_2))$), into $h : 2 \longrightarrow 1$ (i.e., $h(x_1, x_2)$) without triggers and yielding an effect $s : 1 \longrightarrow 1$. The scheme of composition is the following:

$$\begin{tikzcd} \bullet \arrow[r] \arrow[dd] \arrow[ddr, phantom, "r_1"] & \bullet \arrow[r] \arrow[d] \arrow[dr, phantom, "\sigma\otimes 1_{id_1}"] & \bullet \arrow[r] \arrow[d] \arrow[ddr, phantom, "r_2"] & \bullet \arrow[dd] \\ & \bullet \arrow[r] \arrow[d] \arrow[dr, phantom, "1_{id_1\otimes t}"] & \bullet \arrow[d] & \\ \bullet \arrow[r] & \bullet \arrow[r] & \bullet \arrow[r] & \bullet \end{tikzcd}$$

2.2.1 The Inference Rules for Process Tile Logic

At this point, it should be clear that rules in $\mathcal{R}$ can be interpreted as labelled sequents in an adequate *logic of tiles*. Starting from a pTRS, it should be 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 algebraic tile logic. In a certain sense, 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 tile systems. Moreover, some auxiliary tiles should be added in order to allow for consistent reorderings of configurations and side-effects. In what follows, we will 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 $Sym[n,n]$ of arrows from n to n in Sym. Since Sym does not depend on the signature Σ, we assume that Sym is a subcategory of both $\mathbf{S}(\Sigma_H)$ and $\mathbf{S}(\Sigma_V)$.

Definition 2.20 [Process Tile Sequents] Let $\mathcal{R} = \langle \Sigma_H, \Sigma_V, N, R\rangle$ be a pTRS. We say that $\mathcal{R}$ *entails* the class $S_p(\mathcal{R})$ of *flat process sequents* obtained by a finite number of applications of the following *inference rules*:

Basic Sequents

Generators and Identities:

$$(gen)\ \frac{r : h \xrightarrow[u]{v} g \in R(N)}{h \xrightarrow[u]{v} g \in S_p(\mathcal{R})} \qquad (v\text{-}ref)\ \frac{v : n \longrightarrow k \in \mathbf{S}(\Sigma_V)}{id_n \xrightarrow[v]{v} id_k \in S_p(\mathcal{R})} \qquad (h\text{-}ref)\ \frac{h : n \longrightarrow m \in \mathbf{S}(\Sigma_H)}{h \xrightarrow[id_m]{id_n} h \ \in S_p(\mathcal{R})}$$

Auxiliary Sequents

Symmetries:

$$(swap)\ \frac{n \in \mathbb{N},\ s_1, s_2, s_3, s_4 \in Sym_n,\ s_1; s_2 = s_3; s_4}{s_1 \xrightarrow[s_3]{s_2} s_4 \in S_p(\mathcal{R})}$$

Composition Rules

Parallel composition:

$$(par)\ \frac{h \xrightarrow[u]{v} g \in S_p(\mathcal{R}),\ h' \xrightarrow[u']{v'} g' \in S_p(\mathcal{R})}{h \otimes h' \xrightarrow[u \otimes u']{v \otimes v'} g \otimes g' \ \in S_p(\mathcal{R})}$$

Sequential compositions:

$$(hor)\ \frac{h \xrightarrow[u]{v} g \in S_p(\mathcal{R}),\ h' \xrightarrow[u']{u} g' \in S_p(\mathcal{R})}{h; h' \xrightarrow[u']{v} g; g' \in S_p(\mathcal{R})} \qquad (vert)\ \frac{h \xrightarrow[u]{v} g \in S_p(\mathcal{R}),\ g \xrightarrow[u']{v'} g' \in S_p(\mathcal{R})}{h \xrightarrow[u;u']{v;v'} g' \ \in S_p(\mathcal{R})}$$

For any sequent $h \xrightarrow[u]{v} g \in S_p(\mathcal{R})$ we write $\mathcal{R} \vdash_{fp} h \xrightarrow[u]{v} g$, to be read as "$\mathcal{R}$ entails the (flat process sequent) $h \xrightarrow[u]{v} g$". $\square$

The auxiliary inference rule $(swap)$ extensively adds all the sequents needed for simulating any interface re-arrangement. It is possible to obtain the same result starting from a reduced set of auxiliary sequents and using the composition rules. As an example, we could replace rule $(swap)$ by two simpler rules:

$$(swap)\ \frac{n \in \mathbb{N},\ s \in Sym_n}{s \xrightarrow[id_n]{s} id_n \in S_p(\mathcal{R})} \qquad (swap')\ \frac{n \in \mathbb{N},\ s \in Sym_n}{id_n \xrightarrow[s]{id_n} s \in S_p(\mathcal{R})}$$

In fact, 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 $Sym_n \subseteq \mathbf{S}(\Sigma_H)$, we can apply the rule $(h\text{-}ref)$ to entail the sequents $s_1 \xrightarrow[id_n]{id_n} s_1$, $s_4 \xrightarrow[id_n]{id_n} s_4$, and

$s_1;s_2 \xrightarrow[id_n]{id_n} s_1;s_2 = s_1;s_2 \xrightarrow[id_n]{id_n} s_3;s_4$. Next, the two new rules (*swap*) and (*swap*') yield respectively the sequents $s_3 \xrightarrow[id_n]{s_3} id_n$ and $id_n \xrightarrow[s_2]{id_n} s_2$. Finally we can compose the entailed sequents via rules (*hor*) and (*vert*) to let $\mathcal{R}$ entail the sequent $s_1 \xrightarrow[s_2]{s_3} s_4$ (see the composition scheme in the picture below).

$$\begin{tikzcd}
\bullet \arrow[r, "s_1"] \arrow[d] & \bullet \arrow[r] \arrow[d] & \bullet \arrow[d, "s_2"] \\
\bullet \arrow[r, "s_1"] \arrow[d] & \bullet \arrow[r, "s_2"] & \bullet \arrow[d] \\
\bullet \arrow[r, "s_3"] \arrow[d, "s_3"'] & \bullet \arrow[r, "s_4"] \arrow[d] & \bullet \arrow[d] \\
\bullet \arrow[r] & \bullet \arrow[r, "s_4"'] & \bullet
\end{tikzcd}$$

But it is possible to do much better: 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:

$$(swap)\ \frac{}{\gamma_{1,1} \xrightarrow[id_2]{\gamma_{1,1}} id_2 \in S_p(\mathcal{R})} \qquad (swap')\ \frac{}{id_2 \xrightarrow[\gamma_{1,1}]{id_2} \gamma_{1,1} \in S_p(\mathcal{R})}$$

Another interesting result is that not only the horizontal swapping of effects (as the one in algebraic tile logic) can be easily recovered, but also the vertical swapping of configurations is now entailed, i.e., the following proposition can be easily proved.

Proposition 2.21 *Let $\mathcal{R} = \langle \Sigma_H, \Sigma_V, N, R\rangle$ be a pTRS. Then, for any two effects $v : n \longrightarrow k, u : m \longrightarrow l \in \mathbf{S}(\Sigma_V)$, and for any two configurations $h : n \longrightarrow m, g : k \longrightarrow l \in \mathbf{S}(\Sigma_H)$, the sequents $\gamma_{n,m} \xrightarrow[u\otimes v]{v\otimes u} \gamma_{k,l}$ and $h \otimes g \xrightarrow[\gamma_{m,l}]{\gamma_{n,k}} g \otimes h$ are both entailed by $\mathcal{R}$.*

We remark that the class of sequents $S_p(\mathcal{R})$ is flat, in the sense that we are not able to distinguish how a certain sequent has been entailed.

2.2.2 Proof Terms for Process Tile Logic

A more concrete version of process tile logic can be defined if we decorate the sequents with *proof terms*. Then, proof terms can be axiomatized in order to capture equivalent proofs according to the intuitive double 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 2.23), the only difference is that proof terms are now made explicit.

Definition 2.22 [Process Tile Logic] Let $\mathcal{R} = \langle \Sigma_H, \Sigma_V, N, R\rangle$ be a pTRS. 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 following inference rules:

Basic Proof Sequents

Generators and Identities:

$$\frac{r : h \xrightarrow[u]{v} g \in R(N)}{r : h \xrightarrow[u]{v} g \in P_p(\mathcal{R})} \qquad \frac{v : n \longrightarrow k \in \mathbf{S}(\Sigma_V)}{1_v : id_n \xrightarrow[v]{v} id_k \in P_p(\mathcal{R})} \qquad \frac{h : n \longrightarrow m \in \mathbf{S}(\Sigma_H)}{1^h : h \xrightarrow[id_m]{id_n} h \in P_p(\mathcal{R})}$$

Auxiliary Proof Sequents

Symmetries:

$$\frac{}{\sigma_{1,1}:\ \gamma_{1,1} \xrightarrow[id_2]{\gamma_{1,1}} id_2 \ \in P_p(\mathcal{R})} \qquad \frac{}{\sigma'_{1,1}:\ id_2 \xrightarrow[\gamma_{1,1}]{id_2} \gamma_{1,1} \ \in P_p(\mathcal{R})}$$

Composition Rules

Parallel composition:

$$\frac{\alpha : h \xrightarrow[u]{v} g \in P_p(\mathcal{R}),\ \alpha' : h' \xrightarrow[u']{v'} g' \in P_p(\mathcal{R})}{\alpha \otimes \alpha' :\ h \otimes h' \xrightarrow[u \otimes u']{v \otimes v'} g \otimes g' \ \in P_p(\mathcal{R})}$$

Sequential compositions:

$$\frac{\alpha : h \xrightarrow[u]{v} g \in P_p(\mathcal{R}),\ \beta : h' \xrightarrow[u']{u} g' \in P_p(\mathcal{R})}{\alpha * \beta : h; h' \xrightarrow[u']{v} g; g' \in P_p(\mathcal{R})} \qquad \frac{\alpha : h \xrightarrow[u]{v} g \in P_p(\mathcal{R}),\ \beta : g \xrightarrow[u']{v'} g' \in P_p(\mathcal{R})}{\alpha \cdot \beta :\ h \xrightarrow[u;u']{v;v'} g' \ \in P_p(\mathcal{R})}$$

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. As an example, consider the following recursive definition:

$$\begin{array}{rcl}
\sigma_{0,k} & = & 1_{id_k} \\
\sigma_{1,k+1} & = & ((\sigma_{1,1} \otimes 1_{id_1}) * 1^{id_1 \otimes \gamma_{1,k}}) \cdot \sigma_{1,k} \\
\sigma_{n+1,k} & = & ((1_{id_n} \otimes \sigma_{1,k}) * 1^{\gamma_{n,k} \otimes id_1}) \cdot (\sigma_{n,k} \otimes 1_{id_1})
\end{array}$$

It follows that for any $n, m \in \mathbb{N}$, then $\sigma_{n,m} :\ \gamma_{n,m} \xrightarrow[id_{m \otimes n}]{\gamma_{n,m}} id_{m \otimes n} \ \in P_p(\mathcal{R})$. An analogous construction yields the sequents $\sigma'_{n,m} :\ id_{n \otimes m} \xrightarrow[\gamma_{n,m}]{id_{n \otimes m}} \gamma_{n,m} \ \in P_p(\mathcal{R})$, for any $n, m \in \mathbb{N}$. Furthermore, the swappings of configuration and effects can be easily constructed as follows:

- for any two effects $v : n \longrightarrow k,\ u : m \longrightarrow l \in \mathbf{S}(\Sigma_V)$ then

$$\gamma_{v,u} = \sigma_{n,m} \cdot 1_{\gamma_{m,n}} \cdot 1_{v \otimes u} \cdot \sigma'_{k,l} :\ \gamma_{n,m} \xrightarrow[u \otimes v]{v \otimes u} \gamma_{k,l} \ \in P_p(\mathcal{R});$$

- for any two configurations $h : n \longrightarrow m,\ g : k \longrightarrow l \in \mathbf{S}(\Sigma_H)$ then

$$\rho_{h,g} = \sigma_{n,k} * 1^{\gamma_{k,n}} * 1^{h \otimes g} * \sigma'_{m,l} :\ h \otimes g \xrightarrow[\gamma_{m,l}]{\gamma_{n,k}} g \otimes h \ \in P_p(\mathcal{R}).$$

We remark that, in general the cells $1^{\gamma_{n,m}}$ and $1_{\gamma_{n,m}}$ are different.

$$\begin{tikzcd}
n \otimes m \arrow[r, "\gamma_{n,m}"] \arrow[d, "id_{n \otimes m}"'] \arrow[dr, phantom, "1^{\gamma_{n,m}}"] & m \otimes n \arrow[d, "id_{m \otimes n}"] \\
n \otimes m \arrow[r, "\gamma_{n,m}"'] & m \otimes n
\end{tikzcd} \neq \begin{tikzcd}
n \otimes m \arrow[r, "id_{n \otimes m}"] \arrow[d, "\gamma_{n,m}"'] \arrow[dr, phantom, "1_{\gamma_{n,m}}"] & n \otimes m \arrow[d, "\gamma_{n,m}"] \\
m \otimes n \arrow[r, "id_{m \otimes n}"'] & m \otimes n
\end{tikzcd}$$

Proposition 2.23 *Given a pTRS $\mathcal{R}$, then $\mathcal{R} \vdash_{fp} h \xrightarrow[u]{v} g \iff \exists \alpha : h \xrightarrow[u]{v} g \in P_p(\mathcal{R})$.*

2.2.3 Axiomatizing Process Tile Logic

The axiomatization we propose aims at the identification of intuitively equivalent tile computation 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 2.24 [Abstract Process Tile Logic] Let $\mathcal{R} = \langle \Sigma_H, \Sigma_V, N, R \rangle$ be a pTRS. 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 Def. 2.17.
Identity Axioms as in Def. 2.17.
Monoidality Axioms as in Def. 2.17.
Functoriality Axioms for *identities* and *composition* as in Def. 2.17.
Functoriality Axioms for *derived operators* γ and ρ, stating that the swapping of two effects (configurations) respects identities and sequential composition.

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 stating that any two compositions of basic auxiliary sequents ($\sigma_{1,1}$ and $\sigma'_{1,1}$) and identity sequents of configuration and effects (1^h and 1_v) yielding the same flat sequent are identified. In Appendix A it is shown that these axioms can be partitioned in two main subclasses: naturality axioms for σ and σ', and coherence axioms for γ, ρ, σ, and σ'. $\square$

Abusing the notation, we will write $\mathcal{R} \vdash_p \alpha$ to denote the entailment of the abstract process sequent of α and not just the decorated sequent α.

2.3 Naïve Term Tile Logic

In this section we aim at defining a tile format where configurations and side-effects are just (tuples of) terms over two distinct signatures, and composition becomes just substitution. Since we want to use a compact notation, to fix the correspondence on the vertices of the tiles, we should decorate the arrows with *assignments* rather than with terms, as the following example illustrates.

Example 2.25 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 cells below should both represent different ways of computing $(n+1) \times m$, for a generic input pair (n, m), also preserving the input m, i.e., the result should be the pair $\langle (n+1) \times m, m \rangle$:

$$\begin{tikzcd}
2 \arrow[r, "\langle x+1,y \rangle"] \arrow[d, "\langle x\times y,y \rangle"'] \arrow[dr, phantom, "A"] & 2 \arrow[d, "\langle z\times w,w \rangle"] \\
2 \arrow[r, "\langle z+w,w \rangle"'] & 2
\end{tikzcd}
\qquad
\begin{tikzcd}
(x,y) \arrow[r, "\langle z:=x+1,w:=y \rangle"] \arrow[d, "\langle z:=x\times y,w:=y \rangle"'] \arrow[dr, phantom, "B"] & 2 \arrow[d, "\langle x:=z\times w,y:=w \rangle"] \\
2 \arrow[r, "\langle x:=z+w,y:=w \rangle"'] & 2
\end{tikzcd}$$

Tile A is ambiguous, because it does not give any information about the correspondence between the variables z and w and the terms over x and y. For instance, a horizontal-vertical computation could compute $\langle z \times w, w \rangle [x+1/w, y/z] = \langle (x+1) \times y, y+1 \rangle$, while a vertical-horizontal computation could give $\langle z+w, w \rangle [x \times y/w, y/z] = \langle y + x \times y, x \times y \rangle$, and the results would not match. However, the horizontal-vertical computation $\langle z \times w, w\rangle[x + 1/z, y/w] = \langle (x + 1) \times y, y\rangle$ and the vertical-horizontal computation $\langle z + w, w\rangle[x \times y/z, y/w] = \langle x \times y + y, y\rangle$ are compatible, but only if we assume that the term sequences labelling the arrows are ordered. On the other hand, tile B gives the correct correspondence by making explicit the substitutions, but it becomes quite verbose. Moreover, in the latter case we should add α-conversion, to match variable names when composing the tiles. We prefer to introduce a more compact, easy to understand, standard notation. Thus we impose a total order on the variable names, and assume a standard order (i.e., from left to right) on term sequences decorating the arrows. Furthermore, since names do not matter, we assume that for sequences of terms involving n (ordered) variables, a canonical (ordered) set of names $X_n = \{x_1 < \cdots < x_n\}$ is used. In the previous example the canonical, non-ambiguous tile is

$$\begin{tikzcd} 2 \arrow[r, "\langle x_1+1, x_2\rangle"] \arrow[d, "\langle x_1 \times x_2, x_2\rangle"'] & 2 \arrow[d, "\langle x_1 \times x_2, x_2\rangle"] \\ 2 \arrow[r, "\langle x_1+x_2, x_2\rangle"'] & 2 \end{tikzcd}$$

□

As we have done for process tile logic, in what follows we will consider the simpler case of one-sorted signatures. The extension to the many-sorted case should not present any particular difficulty apart from a more complex notation.

Definition 2.26 [Term Tile Rewrite System] A *term (tile) rewrite system* (*tTRS* for short) $\mathcal{R}$ is a quadruple $\langle \Sigma_H, \Sigma_V, N, R\rangle$, where $\Sigma_H = \bigcup_{i\in\mathbb{N}} \Sigma_{H,i}$ and $\Sigma_V = \bigcup_{i\in\mathbb{N}} \Sigma_{V,i}$ are signatures (each $\Sigma_{\_,i}$ containing the function symbols of arity i), 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. □

To shorten the notation we will write a generic rule r such that $R(r) = \langle \vec{h}, \vec{v}, u, g\rangle$ with $\vec{h} \in T_{\Sigma_H}(X_n)^m$, and $\vec{v} \in T_{\Sigma_V}(X_n)^k$ either as the tile

$$\begin{tikzcd} n \arrow[r, "\vec{h}"] \arrow[d, "\vec{v}"'] \arrow[dr, phantom, "r"] & m \arrow[d, "u"] \\ k \arrow[r, "g"'] & 1 \end{tikzcd}$$

or as the sequent $r : n \triangleleft \langle \vec{h}\rangle \xrightarrow[\langle u\rangle]{\langle \vec{v}\rangle} \langle g\rangle$, thus making explicit the number of variables in the "north-western" corner of the tile (the values m and k can be easily recovered from the lengths of the term vectors decorating the tile). Abusing the notation, we denote by λ the empty vector of terms over $T_{\Sigma_H}(X_n)$ and $T_{\Sigma_V}(X_n)$ for each $n \in \mathbb{N}$.

Rules in $\mathcal{R}$ can be interpreted as labelled sequents in a *logic of tiles*. Starting from a tTRS, it should be possible to derive all the tiles obtained by (finite) application of some *inference rules*, that 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 should be added in order to allow for consistent reorderings, duplications, and discharging of the variables, state components, and side-effects.

Example 2.27 Consider the standard tile defined in Ex. 2.25, and suppose that one wants to compute $(n+1)\times n$, which has only one argument ($n=m$). We do not want to redefine an instantiation of our rule for that particular case, because this would undermine the modularity of the specification. The simplest solution consists in allowing the user to duplicate its input. But this must happen consistently in both the horizontal and vertical dimensions. This can be done by introducing a simple rule $1 \triangleleft \langle x_1, x_1\rangle \xrightarrow[\langle x_1,x_2\rangle]{\langle x_1,x_1\rangle} \langle x_1, x_2\rangle$. Informally, both its initial configuration and trigger (term vector $\langle x_1, x_1\rangle$) duplicate a variable, and both its final configuration and effect consider the resulting copies as distinct variables (term vector $\langle x_1, x_2\rangle$). To obtain the expected sequent we may then consider the *horizontal identity* $2 \triangleleft \langle x_1, x_2\rangle \xrightarrow[\langle x_1\times x_2,x_2\rangle]{\langle x_1\times x_2,x_2\rangle} \langle x_1, x_2\rangle$ relative to the horizontal source $\langle x_1\times x_2, x_2\rangle$ of the basic sequent $2 \triangleleft \langle x_1+1, x_2\rangle \xrightarrow[\langle x_1\times x_2,x_2\rangle]{\langle x_1\times x_2,x_2\rangle} \langle x_1+x_2, x_2\rangle$. We can now vertically compose the two auxiliary sequents, thus obtaining $1 \triangleleft \langle x_1, x_1\rangle \xrightarrow[\langle x_1\times x_2,x_2\rangle]{\langle x_1\times x_1,x_1\rangle} \langle x_1, x_2\rangle$, which may be horizontally composed with the original sequent. The result is the desired sequent $1 \triangleleft \langle x_1+1, x_1\rangle \xrightarrow[\langle x_1\times x_2,x_2\rangle]{\langle x_1\times x_1,x_1\rangle} \langle x_1+x_2, x_2\rangle$. □

2.3.1 The Inference Rules for Term Tile Logic

Definition 2.28 [Term Tile Sequents] Let $\mathcal{R} = \langle \Sigma_H, \Sigma_V, N, R\rangle$ be a tTRS. 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 following *deduction rules*.

Basic Sequents

Generators:

$$(gen)\ \frac{r : n \triangleleft \langle \vec{h}\rangle \xrightarrow[\langle u\rangle]{\langle \vec{v}\rangle} \langle g\rangle \in R(N)}{n \triangleleft \langle \vec{h}\rangle \xrightarrow[\langle u\rangle]{\langle \vec{v}\rangle} \langle g\rangle \in S_t(\mathcal{R})}$$

Identities:

$$(v\text{-}ref)\ \frac{\vec{v} \in (T_{\Sigma_V}(X_n))^k}{n \triangleleft \langle x_1, ..., x_n\rangle \xrightarrow[\langle \vec{v}\rangle]{\langle \vec{v}\rangle} \langle x_1, ..., x_k\rangle \in S_t(\mathcal{R})} \qquad (h\text{-}ref)\ \frac{\vec{h} \in (T_{\Sigma_H}(X_n))^m}{n \triangleleft \langle \vec{h}\rangle \xrightarrow[\langle x_1,\ldots,x_m\rangle]{\langle x_1,\ldots,x_n\rangle} \langle \vec{h}\rangle \in S_t(\mathcal{R})}$$

Auxiliary Sequents

Symmetries:

$$(swap)\ \frac{}{2 \triangleleft \langle x_2, x_1\rangle \xrightarrow[\langle x_1,x_2\rangle]{\langle x_2,x_1\rangle} \langle x_1, x_2\rangle \in S_t(\mathcal{R})} \qquad (swap')\ \frac{}{2 \triangleleft \langle x_1, x_2\rangle \xrightarrow[\langle x_2,x_1\rangle]{\langle x_1,x_2\rangle} \langle x_2, x_1\rangle \in S_t(\mathcal{R})}$$

Duplicators:

$$(dup)\ \frac{}{1 \triangleleft \langle x_1, x_1\rangle \xrightarrow[\langle x_1,x_2\rangle]{\langle x_1,x_1\rangle} \langle x_1, x_2\rangle \in S_t(\mathcal{R})} \qquad (dup')\ \frac{}{1 \triangleleft \langle x_1\rangle \xrightarrow[\langle x_1,x_1\rangle]{\langle x_1\rangle} \langle x_1, x_1\rangle \in S_t(\mathcal{R})}$$

Dischargers:

$$(dis)\ \frac{}{1 \triangleleft \langle \lambda\rangle \xrightarrow[\langle \lambda\rangle]{\langle \lambda\rangle} \langle \lambda\rangle \in S_t(\mathcal{R})} \qquad (dis')\ \frac{}{1 \triangleleft \langle x_1\rangle \xrightarrow[\langle \lambda\rangle]{\langle x_1\rangle} \langle \lambda\rangle \in S_t(\mathcal{R})}$$

Composition Rules

Parallel composition:

$$(par)\ \frac{n \triangleleft \langle \vec{h} \rangle \xrightarrow[\langle \vec{u} \rangle]{\langle \vec{v} \rangle} \langle \vec{g} \rangle \ \in S_t(\mathcal{R}),\ n' \triangleleft \langle \vec{h'} \rangle \xrightarrow[\langle \vec{u'} \rangle]{\langle \vec{v'} \rangle} \langle \vec{g'} \rangle \ \in S_t(\mathcal{R})}{(n+n') \triangleleft \langle \vec{h}, \vec{h'}[x_{i+n}/x_i]_{i=1}^{n'} \rangle \xrightarrow[\langle \vec{u}, \vec{u'}[x_{i+m}/x_i]_{i=1}^{m'} \rangle]{\langle \vec{v}, \vec{v'}[x_{i+n}/x_i]_{i=1}^{n'} \rangle} \langle \vec{g}, \vec{g'}[x_{i+k}/x_i]_{i=1}^{k'} \rangle \ \in S_t(\mathcal{R})}$$

(where $\left|\vec{h}\right| = m$, $|\vec{v}| = k$, $\left|\vec{h'}\right| = m'$, and $\left|\vec{v'}\right| = k'$)

Sequential compositions:

$$(hor)\ \frac{n \triangleleft \langle \vec{h} \rangle \xrightarrow[\langle \vec{u} \rangle]{\langle \vec{v} \rangle} \langle \vec{g} \rangle \ \in S_t(\mathcal{R}),\ m \triangleleft \langle \vec{h'} \rangle \xrightarrow[\langle \vec{u'} \rangle]{\langle \vec{u} \rangle} \langle \vec{g'} \rangle \ \in S_t(\mathcal{R}),\ \left|\vec{h}\right| = m,\ |\vec{u}| = k}{n \triangleleft \langle \vec{h'}[h_i/x_i]_{i=1}^{m} \rangle \xrightarrow[\langle \vec{u'} \rangle]{\langle \vec{v} \rangle} \langle \vec{g'}[g_i/x_i]_{i=1}^{k} \rangle \ \in S_t(\mathcal{R})}$$

$$(vert)\ \frac{n \triangleleft \langle \vec{h} \rangle \xrightarrow[\langle \vec{u} \rangle]{\langle \vec{v} \rangle} \langle \vec{g} \rangle \ \in S_t(\mathcal{R}),\ k \triangleleft \langle \vec{g} \rangle \xrightarrow[\langle \vec{u'} \rangle]{\langle \vec{v'} \rangle} \langle \vec{g'} \rangle \ \in S_t(\mathcal{R}),\ |\vec{v}| = k,\ |\vec{u}| = l}{n \triangleleft \langle \vec{h} \rangle \xrightarrow[\langle \vec{u'}[u_i/x_i]_{i=1}^{l} \rangle]{\langle \vec{v'}[v_i/x_i]_{i=1}^{k} \rangle} \langle \vec{g'} \rangle \ \in S_t(\mathcal{R})}$$

where the notation $\vec{s}[t_i/x_i]_{i=1}^{n}$ denotes the simultaneous substitution of the variables $x_1, \ldots, x_n$ with the corresponding terms $t_1, \ldots, t_n$ in all the terms of the tuple $\vec{s}$. For any sequent $n \triangleleft \langle \vec{h} \rangle \xrightarrow[\langle \vec{u} \rangle]{\langle \vec{v} \rangle} \langle \vec{g} \rangle \ \in S_t(\mathcal{R})$ we write $\mathcal{R} \vdash_{ft} n \triangleleft \langle \vec{h} \rangle \xrightarrow[\langle \vec{u} \rangle]{\langle \vec{v} \rangle} \langle \vec{g} \rangle$, to be read as "$\mathcal{R}$ entails the (flat) sequent $n \triangleleft \langle \vec{h} \rangle \xrightarrow[\langle \vec{u} \rangle]{\langle \vec{v} \rangle} \langle \vec{g} \rangle$ ". □

We briefly comment on the above inference rules, showing also some examples of deduction in the style of Ex. 2.27. 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 $S(\mathcal{R})$ w.r.t. all the permutations, tuplings and projections of configurations and effects, and are independent from the particular tTRS. We can divide the auxiliary rules in three subclasses: 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 shall be simulated in the other dimension via an analogous swapping. The term-tile framework allows using a term-like notation instead of symmetries γ as in Section 2.2, but the rules (*swap*) and (*swap*') define exactly the same basic sequents $\gamma_{1,1} \xrightarrow[id_2]{\gamma_{1,1}} id_2 \ \in S_p(\mathcal{R})$, and $id_2 \xrightarrow[\gamma_{1,1}]{id_2} \gamma_{1,1} \ \in S_p(\mathcal{R})$ of process tile logic. Analogously to process tile logic, more complex swappings can be entailed by $\mathcal{R}$. As an example, for any $n, m \in \mathbb{N}$, the sequent

$$(n+m) \triangleleft \langle x_{n+1}, ..., x_{n+m}, x_1, ..., x_n \rangle \xrightarrow[\langle x_1, \ldots, x_{n+m} \rangle]{\langle x_{n+1}, \ldots, x_{n+m}, x_1, \ldots, x_n \rangle} \langle x_1, ..., x_{n+m} \rangle$$

(swapping the first n variables of the interface with the successive m variables), is in $S_t(\mathcal{R})$. Also the swapping of either side-effects or configurations can be handled as in process tile logic.

As an example, this means that, for any two effects $v \in T_{\Sigma_V}(X_n)$, and $u \in T_{\Sigma_V}(X_m)$ then the sequent

$$(n+m) \triangleleft \langle x_{n+1}, ..., x_{n+m}, x_1, ..., x_n \rangle \xrightarrow[\langle u, u[x_{m+i}/x_i]_{i=1}^{n} \rangle]{\langle v, u[x_{n+i}/x_i]_{i=1}^{m} \rangle} \langle x_2, x_1 \rangle$$

is in $S_t(\mathcal{R})$.

The second class of auxiliary rules contains the rules for "making consistent copies". Rules (*dup*) and (*dup*') duplicate a variable of the interface in both horizontal and vertical dimension. A particular application of rule (*dup*) has been shown in Ex. 2.27. Using the notation of algebraic theories, rules (*dup*) and (*dup*') should be rewritten as

$$\frac{}{\nabla_1 \xrightarrow[id_2]{\nabla_1} id_2 \in S_t(\mathcal{R})} \qquad \frac{}{id_1 \xrightarrow[\nabla_1]{id_1} \nabla_1 \in S_t(\mathcal{R})}$$

For instance, suitably composing the basic auxiliary tiles for duplication and swapping, together with identities, we can conclude that for any $n \in \mathbb{N}$ the tTRS $\mathcal{R}$ entails the sequent

$$n \triangleleft \langle x_1, \ldots, x_n, x_1 \ldots, x_n \rangle \xrightarrow[\langle x_1, \ldots, x_{2n} \rangle]{\langle x_1, \ldots, x_n, x_1 \ldots, x_n \rangle} \langle x_1, \ldots, x_{2n} \rangle$$

duplicating n variables. Moreover, $S_t(\mathcal{R})$ contains also the sequents for the duplication of any effect and any configuration. As an example, we illustrate this result for a generic configuration $h \in T_{\Sigma_H}(X_n)$. From rule (*h-ref*) we get the sequent $n \triangleleft \langle h \rangle \xrightarrow[\langle x_1 \rangle]{\langle x_1, \ldots, x_n \rangle} \langle h \rangle$, which can be horizontally composed with the basic sequent of (*dup*') to obtain $n \triangleleft \langle h \rangle \xrightarrow[\langle x_1, x_1 \rangle]{\langle x_1, \ldots, x_n \rangle} \langle h, h \rangle$. Similarly, we can (horizontally) compose the basic sequent of rule (*swap*) together with the parallel composition of the identity sequents for h (with itself). The resulting sequent is $n \triangleleft \langle h, h \rangle \xrightarrow[\langle x_1, x_2 \rangle]{\langle x_1, \ldots, x_n, x_1, \ldots, x_n \rangle} \langle h, h[x_{i+n}/x_i]_{i=1}^{n} \rangle$. The two results can now be vertically composed, yielding the sequent $n \triangleleft \langle h \rangle \xrightarrow[\langle x_1, x_1 \rangle]{\langle x_1, \ldots, x_n, x_1, \ldots, x_n \rangle} \langle h, h[x_{i+n}/x_i]_{i=1}^{n} \rangle$.

Example 2.29 Consider a vertical signature with a unary operator $v(\_)$. A similar construction to the one illustrated above yields the sequent $1 \triangleleft \langle x_1, x_1 \rangle \xrightarrow[\langle v(x_1), v(x_2) \rangle]{\langle v(x_1) \rangle} \langle x_1, x_2 \rangle$. By vertical composition with the basic sequent $1 \triangleleft \langle x_1, x_1 \rangle \xrightarrow[\langle x_1, x_2 \rangle]{\langle x_1, x_1 \rangle} \langle x_1, x_2 \rangle$ we derive also the sequent $1 \triangleleft \langle x_1, x_1 \rangle \xrightarrow[\langle v(x_1), v(x_2) \rangle]{\langle v(x_1), v(x_1) \rangle} \langle x_1, x_2 \rangle$. Making a backward interpretation, this 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 introduce dischargers. Rules (*dis*) and (*dis*') consistently discharge variables in both dimensions. Analogously to symmetries and duplicators, they can be composed to get more complex rules for projecting a configuration, etc.

The composition rules define a general scheme for combining sequents which are "already" entailed in order to get new ones. 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 $2 \triangleleft \langle x_1 \rangle \xrightarrow[\langle x_1 \rangle]{\langle x_1 \rangle} \langle x_1 \rangle$ with itself, we obtain the sequent $2 \triangleleft \langle x_1, x_2 \rangle \xrightarrow[\langle x_1,x_2 \rangle]{\langle x_1,x_2 \rangle} \langle x_1, x_2 \rangle$. Horizontal sequential composition, described by the rule *(hor)*, is partial. It applies exclusively if the side-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 same trigger of the first sequent, the same side-effects of the second sequent and consistently substituting the variables involved in the components of the second sequents with the corresponding components of the first sequent. Similarly for the vertical composition. These operations have been extensively used in the previous examples.

2.3.2 Proof Terms for Term Tile Logic

As for $S_p(\mathcal{R})$, the resulting class of sequents $S_t(\mathcal{R})$ is flat. We provide a more concrete inference system by decorating the entailed 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). We remark that the resulting *logic* is the same as before (see Proposition 2.31).

Definition 2.30 [Term Tile Logic] Let $\mathcal{R} = \langle \Sigma_H, \Sigma_V, N, R \rangle$ be a tTRS. 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 following inference rules:

Basic Proof Sequents

$$\frac{r : n \triangleleft \langle \vec{h} \rangle \xrightarrow[\langle u \rangle]{\langle \vec{v} \rangle} \langle g \rangle \ \in R(N)}{r : n \triangleleft \langle \vec{h} \rangle \xrightarrow[\langle u \rangle]{\langle \vec{v} \rangle} \langle g \rangle \ \in P_t(\mathcal{R})}$$

$$\frac{\vec{v} \in (T_{\Sigma_V}(X_n))^k}{1_{\vec{v}} : n \triangleleft \langle x_1, \ldots, x_n \rangle \xrightarrow[\langle \vec{v} \rangle]{\langle \vec{v} \rangle} \langle x_1, \ldots, x_k \rangle \ \in P_t(\mathcal{R})} \qquad \frac{\vec{h} \in (T_{\Sigma_H}(X_n))^m}{1^{\vec{h}} : n \triangleleft \langle \vec{h} \rangle \xrightarrow[\langle x_1, \ldots, x_m \rangle]{\langle x_1, \ldots, x_n \rangle} \langle \vec{h} \rangle \ \in P_t(\mathcal{R})}$$

Auxiliary Proof Sequents

Symmetries:

$$\frac{}{\sigma_{1,1} : 2 \triangleleft \langle x_2, x_1 \rangle \xrightarrow[\langle x_1,x_2 \rangle]{\langle x_2,x_1 \rangle} \langle x_1, x_2 \rangle \ \in P_t(\mathcal{R})} \qquad \frac{}{\sigma'_{1,1} : 2 \triangleleft \langle x_1, x_2 \rangle \xrightarrow[\langle x_2,x_1 \rangle]{\langle x_1,x_2 \rangle} \langle x_2, x_1 \rangle \ \in P_t(\mathcal{R})}$$

Duplicators:

$$\frac{}{\pi_1 : 1 \triangleleft \langle x_1, x_1 \rangle \xrightarrow[\langle x_1,x_2 \rangle]{\langle x_1,x_1 \rangle} \langle x_1, x_2 \rangle \ \in P_t(\mathcal{R})} \qquad \frac{}{\tau_1 : 1 \triangleleft \langle x_1 \rangle \xrightarrow[\langle x_1,x_1 \rangle]{\langle x_1 \rangle} \langle x_1, x_1 \rangle \ \in P_t(\mathcal{R})}$$

Dischargers:

$$\frac{}{\phi_1 : 1 \triangleleft \langle \lambda \rangle \xrightarrow[\langle \lambda \rangle]{\langle \lambda \rangle} \langle \lambda \rangle \ \in P_t(\mathcal{R})} \qquad \frac{}{\psi_1 : 1 \triangleleft \langle x_1 \rangle \xrightarrow[\langle \lambda \rangle]{\langle x_1 \rangle} \langle \lambda \rangle \ \in P_t(\mathcal{R})}$$

Composition Rules

Parallel composition:

$$\frac{\alpha : n \triangleleft \langle \vec{h} \rangle \xrightarrow[\langle \vec{u} \rangle]{\langle \vec{v} \rangle} \langle \vec{g} \rangle \ \in P_t(\mathcal{R}),\ \ \alpha' : n' \triangleleft \langle \vec{h'} \rangle \xrightarrow[\langle \vec{u'} \rangle]{\langle \vec{v'} \rangle} \langle \vec{g'} \rangle \ \in P_t(\mathcal{R})}{\alpha \otimes \alpha' : (n+n') \triangleleft \langle \vec{h}, \vec{h'}[x_{i+n}/x_i]_{i=1}^{n'} \rangle \xrightarrow[\langle \vec{u}, \vec{u'}[x_{i+m}/x_i]_{i=1}^{m'} \rangle]{\langle \vec{v}, \vec{v'}[x_{i+n}/x_i]_{i=1}^{n'} \rangle} \langle \vec{g}, \vec{g'}[x_{i+k}/x_i]_{i=1}^{k'} \rangle \ \in P_t(\mathcal{R})}$$

(where $\left|\vec{h}\right| = m$, $|\vec{v}| = k$, $\left|\vec{h'}\right| = m'$, and $\left|\vec{v'}\right| = k'$)

Sequential compositions:

$$\frac{\alpha : n \triangleleft \langle \vec{h} \rangle \xrightarrow[\langle \vec{u} \rangle]{\langle \vec{v} \rangle} \langle \vec{g} \rangle \ \in P_t(\mathcal{R}),\ \ \alpha' : m \triangleleft \langle \vec{h'} \rangle \xrightarrow[\langle \vec{u'} \rangle]{\langle \vec{u} \rangle} \langle \vec{g'} \rangle \ \in P_t(\mathcal{R}),\ \ \left|\vec{h}\right| = m,\ |\vec{u}| = k}{\alpha * \alpha' : n \triangleleft \langle \vec{h'}[h_i/x_i]_{i=1}^{m} \rangle \xrightarrow[\langle \vec{u'} \rangle]{\langle \vec{v} \rangle} \langle \vec{g'}[g_i/x_i]_{i=1}^{k} \rangle \ \in P_t(\mathcal{R})}$$

$$\frac{\alpha : n \triangleleft \langle \vec{h} \rangle \xrightarrow[\langle \vec{u} \rangle]{\langle \vec{v} \rangle} \langle \vec{g} \rangle \ \in P_t(\mathcal{R}),\ \ \alpha' : k \triangleleft \langle \vec{g} \rangle \xrightarrow[\langle \vec{u'} \rangle]{\langle \vec{v'} \rangle} \langle \vec{g'} \rangle \ \in P_t(\mathcal{R}),\ \ |\vec{v}| = k,\ |\vec{u}| = l}{\alpha \cdot \alpha' : n \triangleleft \langle \vec{h} \rangle \xrightarrow[\langle \vec{u'}[u_i/x_i]_{i=1}^{l} \rangle]{\langle \vec{v'}[v_i/x_i]_{i=1}^{k} \rangle} \langle \vec{g'} \rangle \ \in P_t(\mathcal{R})}$$

For any sequent $\alpha : n \triangleleft \langle \vec{h} \rangle \xrightarrow[\langle \vec{u} \rangle]{\langle \vec{v} \rangle} \langle \vec{g} \rangle \ \in P_t(\mathcal{R})$ we say that $\mathcal{R}$ entails α, written either $\mathcal{R} \vdash_t \alpha$ or more verbosely, $\mathcal{R} \vdash_t \alpha : n \triangleleft \langle \vec{h} \rangle \xrightarrow[\langle \vec{u} \rangle]{\langle \vec{v} \rangle} \langle \vec{g} \rangle$. □

The inference rules are the same as those in Def. 2.28 (flat version). The only difference is that now each sequent is decorated with a proof term uniquely describing the deduction process which led to that particular sequent.

Proposition 2.31 *Given a tTRS $\mathcal{R}$, then*

$$\mathcal{R} \vdash_{ft} n \triangleleft \langle \vec{h} \rangle \xrightarrow[\langle \vec{u} \rangle]{\langle \vec{v} \rangle} \langle \vec{g} \rangle \iff \exists \alpha : n \triangleleft \langle \vec{h} \rangle \xrightarrow[\langle \vec{u} \rangle]{\langle \vec{v} \rangle} \langle \vec{g} \rangle \ \in P_t(\mathcal{R}).$$

Proof terms allow 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 24). Similarly, the sequents

$$\pi_n : n \triangleleft \langle x_1, \ldots, x_n, x_1 \ldots, x_n \rangle \xrightarrow[\langle x_1, \ldots, x_{2n} \rangle]{\langle x_1, \ldots, x_n, x_1 \ldots, x_n \rangle} \langle x_1, \ldots, x_{2n} \rangle$$

and

$$\tau_n : n \triangleleft \langle x_1, \ldots, x_n \rangle \xrightarrow[\langle x_1, \ldots, x_n, x_1 \ldots, x_n \rangle]{\langle x_1, \ldots, x_n \rangle} \langle x_1, \ldots, x_n, x_1 \ldots, x_n \rangle$$

for the duplication of interfaces can be defined as follows:

$$\begin{aligned}
\pi_0 &= 1_{id_0} \\
\pi_{n+1} &= ((\pi_n \otimes \pi_1) * (1_{id_n \otimes \gamma_{n,1} \otimes id_1})) \cdot (1_{id_n} \otimes \sigma_{n,1} \otimes 1_{id_1}) \\
\tau_0 &= 1_{id_0} \\
\tau_{n+1} &= ((\tau_n \otimes \tau_1) * (1_{\nabla_1 \otimes \nabla_n})) \cdot (1_{id_n} \otimes \sigma_{n,1} \otimes 1_{id_1})
\end{aligned}$$

Furthermore, the sequents for duplicating configurations and effects can be constructed through the following expressions:

- for any effect $\vec{v} \in (T_{\Sigma_V}(X_n))^k$, then

$$\nabla_{\vec{v}} = (1_{\vec{v}} \cdot \tau_k) * (\pi_n \cdot 1_{\vec{v}\otimes\vec{v}}) : n \triangleleft \langle x_1, ..., x_n, x_1, ..., x_n \rangle \xrightarrow[\langle \vec{v}, \vec{v}[x_{n+i}/x_i]_{i=1}^n \rangle]{\langle \vec{v} \rangle} \langle x_1, , ..., x_k, x_1, ..., x_k \rangle$$

- for any configuration $\vec{h} \in (T_{\Sigma_H}(X_n))^m$, then

$$\delta_{\vec{h}} = (1^{\vec{h}} * \tau_m) \cdot (\pi_n \cdot 1^{\vec{h}\otimes\vec{h}}) : n \triangleleft \langle \vec{h} \rangle \xrightarrow[\langle x_1, \ldots, x_m, x_1, \ldots, x_m \rangle]{\langle x_1, \ldots, x_n, x_1, \ldots, x_n \rangle} \langle \vec{h}, \vec{h}[x_{n+i}/x_i]_{i=1}^n \rangle$$

Dischargers also admit a similar generalization, yielding the sequents:

$$\phi_n : n \triangleleft \langle \lambda \rangle \xrightarrow[\langle \lambda \rangle]{\langle \lambda \rangle} \langle \lambda \rangle \qquad \psi_n : n \triangleleft \langle x_1, \ldots, x_n \rangle \xrightarrow[\langle \lambda \rangle]{\langle x_1, \ldots, x_n \rangle} \langle \lambda \rangle$$

$$!_{\vec{v}} = (1_{\vec{V}} \cdot \psi_k) * \phi_n : n \triangleleft \langle \lambda \rangle \xrightarrow[\langle \lambda \rangle]{\langle \vec{v} \rangle} \langle \lambda \rangle \qquad \dagger_{\vec{h}} = (1^{\vec{h}} * \psi_m) \cdot \phi_n : n \triangleleft \langle \vec{h} \rangle \xrightarrow[\langle \lambda \rangle]{\langle \lambda \rangle} \langle \lambda \rangle$$

where

$$\begin{aligned} \phi_0 &= 1_{id_0} \\ \phi_{n+1} &= \phi_n \otimes \phi_1 \\ \psi_0 &= 1_{id_0} \\ \psi_{n+1} &= \psi_n \otimes \psi_1 \end{aligned}$$

2.3.3 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. As in the case of process tile logic, 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 a briefer and informal description of the more interesting properties. We refer the interested reader to Appendix B for the complete axiomatization.

Definition 2.32 [Abstract Term Tile Logic] Let $\mathcal{R} = \langle \Sigma_H, \Sigma_V, N, R \rangle$ be a tTRS. 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 Def. 2.17.
Identity Axioms as in Def. 2.17.
Monoidality Axioms as in Def. 2.17.
Functoriality Axioms for *identities* and *composition* as in Def. 2.17.
Functoriality Axioms for *derived operators* γ and ρ, stating that the swapping of two effects (configurations) respects identities and sequential composition.
Functoriality Axioms for *derived operators* ∇ and δ, stating that the duplication of effects (configurations) respects identities and sequential composition.
Functoriality Axioms for *derived operators* ! and †, stating that the discharging of effects (configurations) respects identities and sequential composition.

Naturality Axioms for *derived operators* γ and ρ (for any sequents $\alpha : h \xrightarrow[v]{u} g$, $\alpha' : h' \xrightarrow[v']{u'} g' \in P_t(\mathcal{R})$):

$$(\alpha \otimes \alpha') * \gamma_{u,u'} = \gamma_{v,v'} * (\alpha' \otimes \alpha) \qquad (\alpha \otimes \alpha') \cdot \rho_{g,g'} = \rho_{h,h'} \cdot (\alpha' \otimes \alpha)$$

Naturality Axioms for *derived operators* ∇ and δ (for any 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 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, 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 effects (1^h and 1_v) yielding the same flat sequent are identified. In Appendix B it is shown that these axioms can be partitioned in 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 the decorated sequent α.

The comparison with rewriting logic suggested us to 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 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* could 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 3) and its relationship to cartesian 2-categories (Section 4). After that we will eventually show how it is possible to derive the double category $\mathcal{L}_\mathcal{D}(\mathcal{R})$ via a free construction (Def. 5.8).

3 Double Categories

As far as we know, all previous attempts – based on internal constructions – for the definition of symmetric, and, more generally, of cartesian double categories have led to an asymmetric model, where the cartesian structure is fully exploited only in one dimension. We believe that this should not be the case, both conceptually and for the kind of applications to models of concurrency that we are developing; therefore, in this section, we propose a broader notion of symmetric and cartesian double category that behaves the same in the horizontal and vertical dimension and fits very well our concurrency applications.

For the 1-dimensional case there is a suitable equational characterization of cartesian categories (with chosen products) in terms of monoidal categories equipped with three natural transformations, called *symmetry*, *duplicator* and *discharger*. In the same way, we look for corresponding notions in the double case.

The first problem is that the internal definition of natural transformation could be 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 [25] proposed another way of expressing natural transformations 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* in the target category, also called *quartets*. 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, then 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 is defined upon 2^n n-fold functors.

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 hypertransformations are defined upon $2^2 = 4$ double functors. This yields a definition of transformations which act in both dimensions, asserting the correctness of the two ways of transforming the structure (first horizontally and then vertically, or vice versa). However, we propose the equivalent definition of *generalized natural transformation* as a more concrete rephrasing of the hyper-approach for the 2-fold case. A generalized transformation involves four double functors, two "horizontal" transformations and two "vertical" transformations.

We can then instantiate this notion of generalized transformation to deal with symmetries, duplicators, and dischargers. This study involves the complete case analysis of the possible combinations. As an interesting result, we find out that each transformation generates two different notions. 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 the generalized transformations in order to state their "coherence" (in the sense that all the horizontal, vertical and parallel composition of the basic components and possibly some identity cells, yielding cells with the same border, must be identified) is a more subtle problem. The solution that we propose relies on the characterization of *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. These operations can always be defined in a double category, and allow expressing the coherence axioms of our generalized transformations as the intuitive rephrasing of the well-known Kelly-MacLane coherence axioms.

Hopefully this work could provide suitable notions of symmetric monoidal and cartesian double categories. Our main motivation is that their explicit axiomatizations can be used to enrich the expressive power of models based on rewrite rules with side effects and rewriting synchronization such as the *tile model* [32, 35]. Although in Section 2 we equipped the tile model with a purely logical presentation, where the tiles are just considered as special sequents subject to certain inference rules, tile systems can be more generally seen as monoidal double categories where the tiles are just cells, the configurations are arrows of the 1-horizontal category, and the side-effects are arrows of the vertical 1-category, objects being just variables which are used to connect the somehow syntactic horizontal category with the dynamic vertical evolution. Moreover, the axiomatization of symmetric monoidal and cartesian double categories that we propose allows the definition of more significative models than the flat ones. Actually such models could take into account the structure of the proof.

In Section 3.1 we give some preliminary definitions regarding double categories, and in Sections 3.2 and 3.3 we introduce the concepts of *generalized inverse* of a cell and of *diagonal categories*. Section 3.4 formalizes and explains the notion of *generalized transformation*, which is essential in the rest of the paper. In Sections 3.5 and 3.6 we incrementally enrich the basic monoidal structure of cells, first with *generalized symmetries* and then with *generalized dischargers* and *generalized duplicators*, also presenting their complete axiomatization.

3.1 Notation

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 [41]).

Definition 3.1 [Double Category] A *double category* $\mathcal{D}$ consists of a collection $a, b, c, \ldots$ of *objects* (also called *0-cells*), a collection $h, g, f, \ldots$ of *horizontal arrows* (also called *horizontal 1-cells*), a collection $v, u, w, \ldots$ of *vertical arrows* (also called *vertical 1-cells*) and a collection $A, B, C, \ldots$ of *double cells* (also called *cells*). Objects and horizontal arrows form the *horizontal 1-category* $\mathcal{H}$, with identity id^a for each object a, and composition $\_ * \_$.

$$a \xrightarrow{h} b \; * \; b \xrightarrow{g} c \; = \; a \xrightarrow{h*g} c \qquad\qquad a \xrightarrow{id^a} a$$

Objects and vertical arrows form also a category, the *vertical 1-category* $\mathcal{V}$, with identity id_a for each object a, and composition $\_ \cdot \_$. To shorten the notation, and because, when we consider lower-dimensional objects to be included in higher dimensional ones the notions indeed coincide, sometimes we will refer to both id^a and id_a either with the object name a or with id_a.

$$\begin{array}{c} a \\ \Big\downarrow v \\ b \\ \cdot \\ b \\ \Big\downarrow u \\ c \end{array} \; = \; \begin{array}{c} a \\ \Big\downarrow v\cdot u \\ c \end{array} \qquad\qquad \begin{array}{c} a \\ \Big\downarrow id_a \\ a \end{array}$$

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, if h is the vertical source, g is the vertical target, v is the horizontal source, and u is the horizontal target, then h and v have the same source object, g and u have the same target object, the target of h is equal to the source of u, and the target of v is equal to the source of g. Graphically these constraints can be represented by the diagram:

$$\begin{tikzcd} a \arrow[r, "h"] \arrow[d, "v"'] \arrow[dr, phantom, "A"] & b \arrow[d, "u"] \\ c \arrow[r, "g"'] & d \end{tikzcd}$$

To shorten the notation we simply write $A : h \xrightarrow[u]{v} g$. In addition, cells can be composed both horizontally ($\_ * \_$) and vertically ($\_ \cdot \_$) as follows: given $B : f \xrightarrow[w]{u} k$, 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 the following pastings of diagrams:

$$\begin{tikzcd} a \arrow[r, "h"] \arrow[d, "v"'] \arrow[dr, phantom, "A"] & b \arrow[r, "f"] \arrow[d, "u"] \arrow[dr, phantom, "B"] & a' \arrow[d, "w"] \\ c \arrow[r, "g"'] & d \arrow[r, "k"'] & c' \end{tikzcd} = \begin{tikzcd} a \arrow[r, "h*f"] \arrow[d, "v"'] \arrow[dr, phantom, "A*B"] & a' \arrow[d, "w"] \\ c \arrow[r, "g*k"'] & c' \end{tikzcd} \qquad \begin{tikzcd} a \arrow[r, "h"] \arrow[d, "v"'] \arrow[dr, phantom, "A"] & b \arrow[d, "u"] \\ c \arrow[r, "g"] \arrow[d, "z"'] \arrow[dr, phantom, "C"] & d \arrow[d, "s"] \\ a' \arrow[r, "h'"'] & b' \end{tikzcd} = \begin{tikzcd} a \arrow[r, "h"] \arrow[d, "v \cdot z"'] \arrow[dr, phantom, "A \cdot C"] & b \arrow[d, "u \cdot s"] \\ a' \arrow[r, "h'"'] & b' \end{tikzcd}$$

Moreover, given a fourth cell $D : k \xrightarrow[t]{s} f'$ the *exchange law* holds:

$$(A \cdot C) * (B \cdot D) = (A * B) \cdot (C * D)$$

$$\begin{tikzcd} \circ \arrow[r] \arrow[d] \arrow[dr, phantom, "A"] & \circ \arrow[d] \\ \circ \arrow[r] \arrow[d] \arrow[dr, phantom, "C"] & \circ \arrow[d] \\ \circ \arrow[r] & \circ \end{tikzcd} * \begin{tikzcd} \circ \arrow[r] \arrow[d] \arrow[dr, phantom, "B"] & \circ \arrow[d] \\ \circ \arrow[r] \arrow[d] \arrow[dr, phantom, "D"] & \circ \arrow[d] \\ \circ \arrow[r] & \circ \end{tikzcd} = \begin{array}{c} \begin{tikzcd} \circ \arrow[r] \arrow[d] \arrow[dr, phantom, "A"] & \circ \arrow[r] \arrow[d] \arrow[dr, phantom, "B"] & \circ \arrow[d] \\ \circ \arrow[r] & \circ \arrow[r] & \circ \end{tikzcd} \\ \cdot \\ \begin{tikzcd} \circ \arrow[r] \arrow[d] \arrow[dr, phantom, "C"] & \circ \arrow[r] \arrow[d] \arrow[dr, phantom, "D"] & \circ \arrow[d] \\ \circ \arrow[r] & \circ \arrow[r] & \circ \end{tikzcd} \end{array}$$

Under these rules, cells form both a horizontal category $\mathcal{D}^*$ and a vertical category $\mathcal{D}^{\cdot}$, with identities $1_v : a \xrightarrow[v]{v} c$ and $1^h : h \xrightarrow[b]{a} h$, respectively. 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).

$$\begin{tikzcd} a \arrow[r, "h"] \arrow[d, "a"'] \arrow[dr, phantom, "1^h"] & b \arrow[r, "g"] \arrow[d, "b"] \arrow[dr, phantom, "1^g"] & c \arrow[d, "c"] \\ a \arrow[r, "h"'] & b \arrow[r, "g"'] & c \end{tikzcd} = \begin{tikzcd} a \arrow[r, "h*g"] \arrow[d, "a"'] \arrow[dr, phantom, "1^{h*g}"] & c \arrow[d, "c"] \\ a \arrow[r, "h*g"'] & c \end{tikzcd} \qquad \begin{tikzcd} a \arrow[r, "a"] \arrow[d, "v"'] \arrow[dr, phantom, "1_v"] & a \arrow[d, "v"] \\ b \arrow[r, "b"] \arrow[d, "u"'] \arrow[dr, phantom, "1_u"] & b \arrow[d, "u"] \\ c \arrow[r, "c"'] & c \end{tikzcd} = \begin{tikzcd} a \arrow[r, "a"] \arrow[d, "v \cdot u"'] \arrow[dr, phantom, "1_{v \cdot u}"] & a \arrow[d, "v \cdot u"] \\ c \arrow[r, "c"'] & c \end{tikzcd}$$

Furthermore, horizontal and vertical identities of identities coincide, i.e. $1_{id_a} = 1^{id^a}$ and are simply denoted by 1_a. $\square$

A double category $\mathcal{D}$ has two possible interpretations as an internal category in **Cat**. That is, due to the symmetric role played by the horizontal and vertical dimensions of a double category, it is possible to adopt a *transposed* approach in the internal construction:

1. As the internal category $(\mathcal{V}, \mathcal{D}^{\cdot}, s^*, t^*, \_ * \_, i^*)$, where[5] the functors s^* and t^* map each arrow onto its corresponding horizontal source and target respectively, functor $\_ * \_$ defines horizontal composition of cells, and functor i^* maps each (vertical) arrow of $\mathcal{V}$ onto its

[5] Remember that category $\mathcal{V}$ has objects in $\mathcal{O}$ and arrows in $\mathcal{V}$ equipped with composition $\_ \cdot \_$, and that the objects of category $\mathcal{D}^{\cdot}$ are horizontal arrows, while its morphisms are cells equipped with vertical composition $\_ \cdot \_$.

(horizontal) identity cell. This corresponds to picture a generic cell $A : h \longrightarrow g$ of $\mathcal{D}^{\cdot}$ as below

$$
\text{(in } \mathcal{D}^{\cdot}\text{)} \quad
\begin{tikzcd}
h \arrow[d, Rightarrow, "A"'] \\
g
\end{tikzcd}
\qquad
\begin{tikzcd}
s^*(h) \arrow[r, "h"] \arrow[d, dotted, "s^*(A)"'] \arrow[dr, phantom, "\overset{A}{\Rightarrow}"] & t^*(h) \arrow[d, dotted, "t^*(A)"] \\
s^*(g) \arrow[r, "g"'] & t^*(g)
\end{tikzcd}
\quad \text{(in } \mathcal{D}^*\text{)}
$$

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 $h \in \mathcal{H}$.

Given two double categories $\mathcal{D}$ and $\mathcal{E}$, a *double functor* $F : \mathcal{D} \longrightarrow \mathcal{E}$ is a 4-tuple of functions[6] mapping objects to objects, horizontal and vertical arrows to horizontal and vertical arrows, and cells to cells, preserving identities and compositions of all kinds. We denote by **DCat** the category of double categories and double functors.

3.2 Inverse

Since double categories have two operators of composition, the definition of the inverse of a cell is not straightforward. We propose the following:

Definition 3.2 [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 iff 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 the 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 the inverses of the vertical arrows on the border of A).

Then, A has a *generalized inverse* iff A has both a $*$-inverse and a $\cdot$-inverse, and there exists a cell A^{-1} such that[7], $A^{-1} \cdot A^* = 1^{g^{-1}}$, $A^* \cdot A^{-1} = 1^{h^{-1}}$ (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 $A^{\cdot}$). □

$$
\begin{tikzcd}
a \arrow[r, "h"] \arrow[d, "v"'] \arrow[dr, phantom, "A"] & b \arrow[d, "u"] \\
c \arrow[r, "g"'] & d
\end{tikzcd}
\qquad
\begin{tikzcd}
b \arrow[r, "h^{-1}"] \arrow[d, "u"'] \arrow[dr, phantom, "A^*"] & a \arrow[d, "v"] \\
d \arrow[r, "g^{-1}"'] & c
\end{tikzcd}
$$

$$
\begin{tikzcd}
c \arrow[r, "g"] \arrow[d, "v^{-1}"'] \arrow[dr, phantom, "A^{\cdot}"] & d \arrow[d, "u^{-1}"] \\
a \arrow[r, "h"'] & b
\end{tikzcd}
\qquad
\begin{tikzcd}
d \arrow[r, "g^{-1}"] \arrow[d, "u^{-1}"'] \arrow[dr, phantom, "A^{-1}"] & c \arrow[d, "v^{-1}"] \\
b \arrow[r, "h^{-1}"'] & a
\end{tikzcd}
$$

For instance, it follows that $(A \cdot A^{\cdot}) * (A^* \cdot A^{-1}) = (A * A^*) \cdot (A^{\cdot} * A^{-1}) = 1_a$, and that $(A^{-1})^{-1} = A$.

[6] 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.

[7] This definition can be summarized by saying that the $*$-inverse of the $\cdot$-inverse $(A^{\cdot})^*$ of cell A is equal to the $\cdot$-inverse of the $*$-inverse $(A^*)^{\cdot}$ of A and it is denoted by A^{-1}.

3.3 Diagonal Categories

Sometimes, due to the particular kind of cells involved, 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* subcategories.

In fact, those cells having identities as both horizontal and vertical target are the arrows of a *diagonal* category whose composition $\_ \triangleleft \_$ is defined uniquely as pictured below.

$$
\begin{tikzcd}
a \arrow[r, "h"] \arrow[d, "v"'] \arrow[dr, phantom, "A"] & b \arrow[d, "b"] \\
b \arrow[r, "b"'] & b
\end{tikzcd}
\triangleleft
\begin{tikzcd}
b \arrow[r, "g"] \arrow[d, "u"'] \arrow[dr, phantom, "B"] & c \arrow[d, "c"] \\
c \arrow[r, "c"'] & c
\end{tikzcd}
=
\begin{tikzcd}
a \arrow[r] \arrow[d] \arrow[dr, phantom, "A"] & b \arrow[r] \arrow[d] \arrow[dr, phantom, "1^g"] & c \arrow[d] \\
b \arrow[r] \arrow[d] \arrow[dr, phantom, "1_u"] & b \arrow[r] \arrow[d] \arrow[dr, phantom, "B"] & c \arrow[d] \\
c \arrow[r] & c \arrow[r] & c
\end{tikzcd}
= (A * 1^g) \cdot (1_u * B)
$$

In a similar way, we could also define a diagonal composition $\_ \triangleright \_$ for those cells having identities as both horizontal and vertical source:

$$
\begin{tikzcd}
a \arrow[r, "a"] \arrow[d, "a"'] \arrow[dr, phantom, "A"] & a \arrow[d, "v"] \\
a \arrow[r, "h"'] & b
\end{tikzcd}
\triangleright
\begin{tikzcd}
b \arrow[r, "b"] \arrow[d, "b"'] \arrow[dr, phantom, "B"] & b \arrow[d, "u"] \\
b \arrow[r, "g"'] & c
\end{tikzcd}
=
\begin{tikzcd}
a \arrow[r] \arrow[d] \arrow[dr, phantom, "A"] & a \arrow[r] \arrow[d] \arrow[dr, phantom, "1_v"] & a \arrow[d] \\
a \arrow[r] \arrow[d] \arrow[dr, phantom, "1^h"] & b \arrow[r] \arrow[d] \arrow[dr, phantom, "B"] & b \arrow[d] \\
b \arrow[r] & b \arrow[r] & c
\end{tikzcd}
= (A * 1_v) \cdot (1^h * B)
$$

3.4 Transformations between Double Functors

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 [25] the notion of *hypertransformation* is proposed as the generalization of natural transformations to the n-fold case. We propose the following definition of *generalized natural transformation* as a more concrete rephrasing of the hyper-view[8] for double categories. 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 expressing relationships between natural $*$-transformations and $\cdot$-transformations (which are the two possible notions of internal transformations suggested by the internal category viewpoint).

Definition 3.3 [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

$$
\begin{tikzcd}
F_{00} \arrow[r, "\alpha_{0\_}"] \arrow[d, "\beta_{\_0}"'] \arrow[dr, phantom, "\eta"] & F_{01} \arrow[d, "\beta_{\_1}"] \\
F_{10} \arrow[r, "\alpha_{1\_}"'] & F_{11}
\end{tikzcd}
$$

where:

[8]In the original approach, emphasis was given to showing that the category **MCat** of multiple functors is cartesian closed.

- 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 (that is, the objects of $\mathcal{D}^*$) 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 (that is, the objects of $\mathcal{D}^{\cdot}$) of $\mathcal{D}$ to the category $\mathcal{E}^*$, and
- the symbol η defines both a natural transformation from $\alpha_{0\_}$ to $\alpha_{1\_}$ (seen as functors from $\mathcal{V}$ to $\mathcal{E}^{\cdot}$) and also from $\beta_{\_0}$ to $\beta_{\_1}$ (seen as functors from $\mathcal{H}$ to $\mathcal{E}^*$).

□

To shorten the notation, we will denote the generalized transformation just by η, using a figure to represents also its components.

We now explain more in 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 of $\mathcal{D}$ we have the naturality law $F_{00}A * \alpha_{0\_,u} = \alpha_{0\_,v} * F_{01}A$.

$$\begin{tikzcd} a \arrow[r, "h"] \arrow[d, "v"'] \arrow[dr, phantom, "A"] & b \arrow[d, "u"] \\ c \arrow[r, "g"'] & d \end{tikzcd} \qquad \begin{tikzcd} F_{00}a \arrow[r, "F_{00}h"] \arrow[d, "F_{00}v"'] \arrow[dr, phantom, "F_{00}A"] & F_{00}b \arrow[r, "\alpha_{0\_,b}"] \arrow[d, "F_{00}u"] \arrow[dr, phantom, "\alpha_{0\_,u}"] & F_{01}b \arrow[d, "F_{01}u"] \\ F_{00}c \arrow[r, "F_{00}g"'] & F_{00}d \arrow[r, "\alpha_{0\_,d}"'] & F_{01}d \end{tikzcd} = \begin{tikzcd} F_{00}a \arrow[r, "\alpha_{0\_,a}"] \arrow[d, "F_{00}v"'] \arrow[dr, phantom, "\alpha_{0\_,v}"] & F_{01}a \arrow[r, "F_{01}h"] \arrow[d, "F_{01}v"] \arrow[dr, phantom, "F_{01}A"] & F_{01}b \arrow[d, "F_{01}u"] \\ F_{00}c \arrow[r, "\alpha_{0\_,c}"'] & F_{01}c \arrow[r, "F_{01}g"'] & F_{01}d \end{tikzcd}$$

It also follows that, for each horizontal arrow $h : a \longrightarrow b$ in $\mathcal{D}$, then $F_{00}h * \alpha_{0\_,b} = \alpha_{0\_,a} * F_{01}h$, e.g., 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 two more 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 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 $\mathcal{V}$.

$$\begin{tikzcd} a \arrow[d, "v"'] \\ c \end{tikzcd} \qquad \begin{tikzcd} F_{00}a \arrow[r, "\alpha_{0\_,a}"] \arrow[d, "F_{00}v"'] \arrow[dr, phantom, "\alpha_{0\_,v}"] & F_{01}a \arrow[d, "F_{01}v"] \\ F_{00}c \arrow[r, dashed, "\alpha_{0\_,c}"] \arrow[d, "s^*(\eta_c)"'] \arrow[dr, phantom, "\eta_c"] & F_{01}c \arrow[d, "t^*(\eta_c)"] \\ F_{10}c \arrow[r, "\alpha_{1\_,c}"'] & F_{11}c \end{tikzcd} = \begin{tikzcd} F_{00}a \arrow[r, "\alpha_{0\_,a}"] \arrow[d, "s^*(\eta_a)"'] \arrow[dr, phantom, "\eta_a"] & F_{01}a \arrow[d, "t^*(\eta_a)"] \\ F_{10}a \arrow[r, dashed, "\alpha_{1\_,a}"] \arrow[d, "F_{10}v"'] \arrow[dr, phantom, "\alpha_{1\_,v}"] & F_{11}a \arrow[d, "F_{11}v"] \\ F_{10}c \arrow[r, "\alpha_{1\_,c}"'] & F_{11}c \end{tikzcd}$$

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 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 $\mathcal{E}^*$ satisfying $F_{00}A \cdot \beta_{\_0,g} = \beta_{\_0,h} \cdot F_{10}A$ (see picture below), and $F_{01}A \cdot \beta_{\_1,g} = \beta_{\_1,h} \cdot F_{11}A$ for each cell A of $\mathcal{D}$.

$$
\begin{tikzcd}
a \arrow[r, "h"] \arrow[d, "v"'] \arrow[dr, phantom, "A"] & b \arrow[d, "u"] \\
c \arrow[r, "g"'] & d
\end{tikzcd}
\qquad
\begin{tikzcd}
F_{00}a \arrow[r, "F_{00}h"] \arrow[d, "F_{00}v"'] \arrow[dr, phantom, "F_{00}A"] & F_{00}b \arrow[d, "F_{00}u"] \\
F_{00}c \arrow[r, "F_{00}g"] \arrow[d, "\beta_{\_0,c}"'] \arrow[dr, phantom, "\beta_{\_0,g}"] & F_{00}d \arrow[d, "\beta_{\_0,d}"] \\
F_{10}c \arrow[r, "F_{10}g"'] & F_{10}d
\end{tikzcd}
=
\begin{tikzcd}
F_{00}a \arrow[r, "F_{00}h"] \arrow[d, "\beta_{\_0,a}"'] \arrow[dr, phantom, "\beta_{\_0,h}"] & F_{00}b \arrow[d, "\beta_{\_0,b}"] \\
F_{10}a \arrow[r, "F_{10}h"] \arrow[d, "F_{10}v"'] \arrow[dr, phantom, "F_{10}A"] & F_{10}b \arrow[d, "F_{10}u"] \\
F_{10}c \arrow[r, "F_{10}g"'] & F_{10}d
\end{tikzcd}
$$

The generalized 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:

$$
a \xrightarrow{h} b
\qquad
\begin{tikzcd}
F_{00}a \arrow[r, "F_{00}h"] \arrow[d, "\beta_{\_0,a}"'] \arrow[dr, phantom, "\beta_{\_0,h}"] & F_{00}b \arrow[r, "s^{\cdot}(\eta_b)"] \arrow[d, "\beta_{\_0,b}"] \arrow[dr, phantom, "\eta_b"] & F_{01}b \arrow[d, "\beta_{\_1,b}"] \\
F_{10}a \arrow[r, "F_{10}h"'] & F_{10}b \arrow[r, "t^{\cdot}(\eta_b)"'] & F_{11}b
\end{tikzcd}
=
\begin{tikzcd}
F_{00}a \arrow[r, "s^{\cdot}(\eta_a)"] \arrow[d, "\beta_{\_0,a}"'] \arrow[dr, phantom, "\eta_a"] & F_{01}a \arrow[r, "F_{01}h"] \arrow[d, "\beta_{\_1,a}"] \arrow[dr, phantom, "\beta_{\_1,h}"] & F_{01}b \arrow[d, "\beta_{\_1,b}"] \\
F_{10}a \arrow[r, "t^{\cdot}(\eta_a)"'] & F_{11}a \arrow[r, "F_{11}h"'] & F_{11}b
\end{tikzcd}
$$

It follows that, for each object $a \in \mathcal{O}$, the shape of the cell η_a is

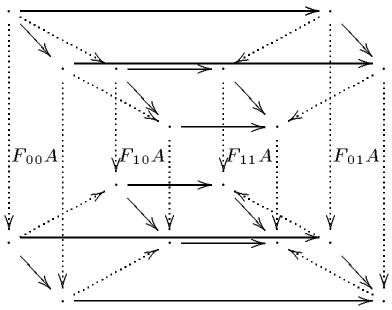
$$
\begin{tikzcd}
F_{00}a \arrow[r, "\alpha_{0\_,a}"] \arrow[d, "\beta_{\_0,a}"'] \arrow[dr, phantom, "\eta_a"] & F_{01}a \arrow[d, "\beta_{\_1,a}"] \\
F_{10}a \arrow[r, "\alpha_{1\_,a}"'] & F_{11}a
\end{tikzcd}
$$

As an example, it follows directly from the definition that, given a cell A, all the cell pastings on the right of the picture yield identical results.

$$
\begin{tikzcd}
\cdot \arrow[r, "h"] \arrow[d, "v"'] \arrow[dr, phantom, "A"] & \cdot \arrow[d, "u"] \\
\cdot \arrow[r, "g"'] & \cdot
\end{tikzcd}
\qquad
\begin{array}{ccc}
\begin{tikzcd}
\cdot \arrow[r] \arrow[d] \arrow[dr, phantom, "F_{00}A"] & \cdot \arrow[r] \arrow[d] \arrow[dr, phantom, "\alpha_{0\_,u}"] & \cdot \arrow[d] \\
\cdot \arrow[r] \arrow[d] \arrow[dr, phantom, "\beta_{\_0,g}"] & \cdot \arrow[r] \arrow[d] \arrow[dr, phantom, "\eta_d"] & \cdot \arrow[d] \\
\cdot \arrow[r] & \cdot \arrow[r] & \cdot
\end{tikzcd}
& = &
\begin{tikzcd}
\cdot \arrow[r] \arrow[d] \arrow[dr, phantom, "\alpha_{0\_,v}"] & \cdot \arrow[r] \arrow[d] \arrow[dr, phantom, "F_{01}A"] & \cdot \arrow[d] \\
\cdot \arrow[r] \arrow[d] \arrow[dr, phantom, "\eta_c"] & \cdot \arrow[r] \arrow[d] \arrow[dr, phantom, "\beta_{\_1,g}"] & \cdot \arrow[d] \\
\cdot \arrow[r] & \cdot \arrow[r] & \cdot
\end{tikzcd}
\\
= & & = \\
\begin{tikzcd}
\cdot \arrow[r] \arrow[d] \arrow[dr, phantom, "\beta_{\_0,h}"] & \cdot \arrow[r] \arrow[d] \arrow[dr, phantom, "\eta_b"] & \cdot \arrow[d] \\
\cdot \arrow[r] \arrow[d] \arrow[dr, phantom, "F_{10}A"] & \cdot \arrow[r] \arrow[d] \arrow[dr, phantom, "\alpha_{1\_,u}"] & \cdot \arrow[d] \\
\cdot \arrow[r] & \cdot \arrow[r] & \cdot
\end{tikzcd}
& = &
\begin{tikzcd}
\cdot \arrow[r] \arrow[d] \arrow[dr, phantom, "\eta_a"] & \cdot \arrow[r] \arrow[d] \arrow[dr, phantom, "\beta_{\_1,h}"] & \cdot \arrow[d] \\
\cdot \arrow[r] \arrow[d] \arrow[dr, phantom, "\alpha_{1\_,v}"] & \cdot \arrow[r] \arrow[d] \arrow[dr, phantom, "F_{11}A"] & \cdot \arrow[d] \\
\cdot \arrow[r] & \cdot \arrow[r] & \cdot
\end{tikzcd}
\end{array}
$$

All the naturality equations are faithfully represented by the commuting hypercube pictured below (to ease the interpretation, we draw vertical arrows as dotted lines):

$$
\begin{tikzcd}
a \arrow[dr, "h"] \arrow[dd, dotted, "v"'] & \\
& b \arrow[dd, dotted, "u"] \\
c \arrow[dr, "g"'] \arrow[ur, phantom, "A"] & \\
& d
\end{tikzcd}
$$



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). 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 give 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.

Remark 3.4 The notion of generalized transformation generalizes $*$- and $\cdot$-transformations. Indeed, a $*$-transformation $\alpha : F \Rightarrow G : \mathcal{D} \longrightarrow \mathcal{E}$ yields a corresponding generalized transformation 1^α. Similarly, a $\cdot$-transformation $\beta : F \Rightarrow H : \mathcal{D} \longrightarrow \mathcal{E}$ yields a corresponding generalized transformation 1_β:

$$
\begin{tikzcd}
F \arrow[r, "\alpha"] \arrow[d, "1_F"'] \arrow[dr, phantom, "1^\alpha"] & G \arrow[d, "1_G"] \\
F \arrow[r, "\alpha"'] & G
\end{tikzcd}
\qquad
\begin{tikzcd}
F \arrow[r, "1_F"] \arrow[d, "\beta"'] \arrow[dr, phantom, "1_\beta"] & F \arrow[d, "\beta"] \\
H \arrow[r, "1_H"'] & H
\end{tikzcd}
$$

Remark 3.5 $*$- and $\cdot$-transformations are instances of a more general pattern. Notice that, if we restrict ourselves to 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:

$$
\begin{tikzcd}
F \arrow[r] \arrow[d] \arrow[dr, phantom, "1"] & F \arrow[d] \\
F \arrow[r] & F
\end{tikzcd}
\quad
\begin{tikzcd}
F \arrow[r] \arrow[d] \arrow[dr, phantom, "2"] & F \arrow[d] \\
F \arrow[r] & G
\end{tikzcd}
\quad
\begin{tikzcd}
F \arrow[r] \arrow[d] \arrow[dr, phantom, "3"] & F \arrow[d] \\
G \arrow[r] & G
\end{tikzcd}
\quad
\begin{tikzcd}
F \arrow[r] \arrow[d] \arrow[dr, phantom, "4"] & G \arrow[d] \\
F \arrow[r] & G
\end{tikzcd}
\quad
\begin{tikzcd}
F \arrow[r] \arrow[d] \arrow[dr, phantom, "5"] & G \arrow[d] \\
G \arrow[r] & G
\end{tikzcd}
\quad
\begin{tikzcd}
G \arrow[r] \arrow[d] \arrow[dr, phantom, "6"] & G \arrow[d] \\
G \arrow[r] & G
\end{tikzcd}
$$

3.5 Symmetric Monoidal Double Categories

As a matter of notation, in what follows we favour the horizontal dimension, by using the common symbols associate to ordinary symmetries, duplicators and dischargers to denote $*$-transformations rather than $\cdot$-transformations.

Definition 3.6 [(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:

$$
\begin{tikzcd}
\mathcal{D}\times\mathcal{D}\times\mathcal{D} \arrow[r, "\otimes\times 1"] \arrow[d, "1\times\otimes"'] & \mathcal{D}\times\mathcal{D} \arrow[d, "\otimes"] \\
\mathcal{D}\times\mathcal{D} \arrow[r, "\otimes"'] & \mathcal{D}
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathcal{D} \arrow[r, "\langle 1,e\rangle"] \arrow[dr, "1"'] & \mathcal{D}\times\mathcal{D} \arrow[d, "\otimes"] & \mathcal{D} \arrow[l, "\langle e,1\rangle"'] \arrow[dl, "1"] \\
& \mathcal{D} &
\end{tikzcd}
$$

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. □

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 $\otimes \circ X$) which verifies some additional coherence axioms [46]. A *double symmetry* is a generalized natural transformation, with a generalized inverse, and it verifies some similar axioms.

Definition 3.7 [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.

$$
\begin{tikzcd}
\otimes \arrow[r, "\gamma"] \arrow[d, "\rho"'] \arrow[dr, phantom, "\sigma"] & \otimes\circ X \arrow[d, "1"] \\
\otimes\circ X \arrow[r, "1"'] & \otimes\circ X
\end{tikzcd}
$$

This means that all the following equations have to be satisfied:

- *Naturality* of γ and ρ:

  For any pair of cells A, A' in $\mathcal{D}$,

$$
\begin{tikzcd}
a \arrow[r, "h"] \arrow[d, "v"'] \arrow[dr, phantom, "A"] & b \arrow[d, "u"] \\
c \arrow[r, "g"'] & d
\end{tikzcd}
\qquad
\begin{tikzcd}
a' \arrow[r, "h'"] \arrow[d, "v'"'] \arrow[dr, phantom, "A'"] & b' \arrow[d, "u'"] \\
c' \arrow[r, "g'"'] & d'
\end{tikzcd}
$$

$$(A \otimes A') * \gamma_{u,u'} = \gamma_{v,v'} * (A' \otimes A) \qquad (A \otimes A') \cdot \rho_{g,g'} = \rho_{h,h'} \cdot (A' \otimes A).$$

- *Functoriality* of γ and ρ:
  For any vertical arrows $v : a \longrightarrow c$, $v' : a' \longrightarrow c'$, $w : c \longrightarrow d$, and $w' : c' \longrightarrow d'$ in $\mathcal{D}$,

$$\gamma_{v \cdot w, v' \cdot w'} = \gamma_{v,v'} \cdot \gamma_{w,w'}.$$

  For any horizontal arrows $h : a \longrightarrow b$, $h' : a' \longrightarrow b'$, $f : b \longrightarrow c$, and $f' : b' \longrightarrow c'$ in $\mathcal{D}$,

$$\rho_{h * f, h' * f'} = \rho_{h,h'} * \rho_{f,f'}.$$

  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 $\mathcal{D}$,

$$\rho_{h,g} * \sigma_{b,d} = \sigma_{a,c} * 1^{g \otimes h}.$$

- Kelly-MacLane *coherence axioms*[9] for γ, ρ and σ:
  For any vertical arrows $v : a \longrightarrow c$, $u : b \longrightarrow d$, and $w : a' \longrightarrow c'$ in $\mathcal{D}$,

$$\gamma_{u \otimes w, v} = (1_u \otimes \gamma_{w,v}) * (\gamma_{u,v} \otimes 1_w),$$

$$\gamma_{v,u} * \gamma_{u,v} = 1_{v \otimes u}.$$

  For any horizontal arrows $h : a \longrightarrow b$, $g : c \longrightarrow d$, and $f : a' \longrightarrow b'$ in $\mathcal{D}$,

$$\rho_{f \otimes g, h} = (1^f \otimes \rho_{g,h}) \cdot (\rho_{f,h} \otimes 1^g),$$

$$\rho_{h,g} \cdot \rho_{g,h} = 1^{h \otimes g}.$$

  For any objects a, b, and c in $\mathcal{D}$,

$$\sigma_{a \otimes b, c} = (1_a \otimes \sigma_{b,c}) \triangleleft (\sigma_{a,c} \otimes 1_b),$$

$$\sigma_{a,b} \triangleleft \sigma_{b,a} = 1_{a \otimes b}.$$

$\square$

[9] For instance, considering γ and translating the more general definition of symmetric monoidal category into the special case of symmetric *strict* monoidal category it could seem that also the axiom $\gamma_{id_e, v} = 1_v$ 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 $\gamma_{id_e, v} = \gamma_{id_e \otimes id_e, v} = (1_e \otimes \gamma_{id_e, v}) * (\gamma_{id_e, v} \otimes 1_e) = \gamma_{id_e, v} * \gamma_{id_e, v}$. Thus, composing with γ_{v, id_e}, we obtain $1_{id_e \otimes v} = \gamma_{id_e, v}$. Finally, recalling that $id_e \otimes v = v$ we can conclude that $\gamma_{id_e, v} = 1_v$.

The generalized inverse of $\sigma_{a,b}$ can be easily defined in terms of σ, γ and ρ as follows:

- 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 3.8 The above notation could be somehow misleading, because the generalized inverse $\sigma^{-1}_{a,b}$ 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 3.9 [Category **SsMDCat**] We denote by **SsMDCat** the category of SsMD's and monoidal double functors preserving all the symmetries. □

Proposition 3.10 *The forgetful functor from* **SsMDCat** *to* **Set***, mapping a SsMD into its set of objects has a left adjoint which 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 3.11 *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.*

It is easy to show that the axiomatization of proof sequents for the process tile logic as given in Def. 2.24 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.

3.6 Cartesian Double Categories (with consistently chosen products)

A fairly general notion of double products should require the products to exist according to all the four possible compositions that we have seen: horizontal ($\_ * \_$), vertical ($\_ \cdot \_$), and diagonal ($\_ \triangleleft \_$, and $\_ \triangleright \_$).

We recall the ordinary definition of cartesian category.

Definition 3.12 [Terminal Object, Products, Cartesian Category] 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 from c to t.

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 $g = q; \Pi_b$. 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.

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

Pursuing the analogy with the 1-dimensional case, we propose the following definitions

Definition 3.13 [Double Product] Given a Double category $\mathcal{D}$, we say that $\mathcal{D}$ has all *double (binary) products* if the categories $\mathcal{D}^{*}$, $\mathcal{D}^{\cdot}$, $\mathcal{D}^{\triangleleft}$, and $\mathcal{D}^{\triangleright}$ all have (binary) products.

We say that $\mathcal{D}$ has a *double terminal object* if the categories $\mathcal{D}^{*}$, $\mathcal{D}^{\cdot}$, $\mathcal{D}^{\triangleleft}$, and $\mathcal{D}^{\triangleright}$ all have a terminal object. □

Definition 3.14 [Cartesian Double Category] A double category $\mathcal{D}$ is called a *cartesian double category* 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 the different dimensions, but simply states their existence. Thus we adopt the convention that not only the products are *chosen* in all the four dimensions, but that they are also *consistently chosen*. For simplicity, from now on, we will consider only this kind of cartesian double categories, thus avoiding to always specify that they have "consistently chosen products".

Remark 3.15 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 will discuss could be also called *with most chosen products*.

Let $\Delta : \mathcal{D} \longrightarrow \mathcal{D} \times \mathcal{D}$ be the double functor which makes a copy of the argument: i.e., such that for each $A \in \mathcal{D}$, $\Delta(A) = (A, A)$. 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 duplicators* and *double dischargers* are generalized natural transformations verifying similar coherence axioms.

Definition 3.16 [Cartesian Double Categories with Consistently Chosen Products] A *cartesian double category* 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.

$$\begin{tikzcd} 1_{\mathcal{D}} \arrow[r, "\nabla"] \arrow[d, "\delta"'] \arrow[dr, phantom, "\pi"] & \otimes \circ \Delta \arrow[d, "1"] \\ \otimes \circ \Delta \arrow[r, "1"'] & \otimes \circ \Delta \end{tikzcd} \qquad \begin{tikzcd} 1_{\mathcal{D}} \arrow[r, "1"] \arrow[d, "1"'] \arrow[dr, phantom, "\tau"] & 1_{\mathcal{D}} \arrow[d, "\delta"] \\ 1_{\mathcal{D}} \arrow[r, "\nabla"'] & \otimes \circ \Delta \end{tikzcd} \qquad \begin{tikzcd} 1_{\mathcal{D}} \arrow[r, "!"] \arrow[d, "\dagger"'] \arrow[dr, phantom, "\phi"] & e \arrow[d, "1"] \\ e \arrow[r, "1"'] & e \end{tikzcd} \qquad \begin{tikzcd} 1_{\mathcal{D}} \arrow[r, "1"] \arrow[d, "1"'] \arrow[dr, phantom, "\psi"] & 1_{\mathcal{D}} \arrow[d, "\dagger"] \\ 1_{\mathcal{D}} \arrow[r, "!"'] & e \end{tikzcd}$$

This means that all the following equations have to be satisfied:

- *Naturality* of ∇, $!$, $\dagger$, and δ:
  For any cell A in $\mathcal{D}$,

$$\begin{tikzcd} a \arrow[r, "h"] \arrow[d, "v"'] \arrow[dr, phantom, "A"] & b \arrow[d, "u"] \\ c \arrow[r, "g"'] & d \end{tikzcd}$$

$$A * \nabla_u = \nabla_v * (A \otimes A), \qquad A * !_u = !_v, \qquad A \cdot \dagger_g = \dagger_h, \qquad A \cdot \delta_g = \delta_h \cdot (A \otimes A).$$

- *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 $\mathcal{D}$,

  $$\begin{array}{ll}
  \pi_{a\otimes b} = (\pi_a \otimes \pi_b) \lhd (1_a \otimes \sigma_{a,b} \otimes 1_b), & \tau_{a\otimes b} = (\tau_a \otimes \tau_b) \rhd (1_a \otimes \sigma^{-1}_{a,b} \otimes 1_b), \\
  \phi_{a\otimes b} = \phi_a \otimes \phi_b, & \psi_{a\otimes b} = \psi_a \otimes \psi_b, \\
  \pi_e = 1_e = \phi_e, & \tau_e = 1_e = \psi_e, \\
  \pi_a \lhd (\pi_a \otimes 1_a) = \pi_a \lhd (1_a \otimes \pi_a), & \tau_a \rhd (\tau_a \otimes 1_a) = \tau_a \rhd (1_a \otimes \tau_a), \\
  \pi_a \lhd \sigma_{a,a} = \pi_a, & \tau_a \rhd \sigma^{-1}_{a,a} = \tau_a, \\
  \pi_a \lhd (1_a \otimes \phi_a) = 1_a, & \tau_a \rhd (1_a \otimes \psi_a) = 1_a.
  \end{array}$$

- *double coherence axioms* for π, ϕ, τ, and ψ:

  For any objects a in $\mathcal{D}$,

  $$\begin{array}{ll}
  \tau_a * \pi_a = \nabla_a, & \tau_a \cdot \pi_a = \delta_a, \\
  \psi_a * \phi_a = !_a, & \psi_a \cdot \phi_a = \dagger_a.
  \end{array}$$

  □

Since most of the axioms are either just a rephrasing of those for the 1-dimensional case or are induced by the definition of generalized transformations, we assume 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). When we considered the symmetric generalization, we should have also required similar axioms for σ and σ^{-1}. 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 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 transformation

$$\begin{tikzcd}
1_{\mathcal{D}} \arrow[r, "1"] \arrow[d, "\delta"'] \arrow[dr, phantom, "\tau\cdot\pi"] & 1_{\mathcal{D}} \arrow[d, "\delta"] \\
\otimes \circ \Delta \arrow[r, "1"'] & \otimes \circ \Delta
\end{tikzcd}$$

that is already present as the identity of δ.

Definition 3.17 [Category **CDCat**] We call **CDCat** the category of cartesian double categories and monoidal double functors preserving all the symmetries, duplicators and dischargers. □

Proposition 3.18 *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 3.19 *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.*

Similarly to the case of process tile logic, it is possible to show that the axiomatization of proof sequents for term tile logic stated in Def. 2.32 makes $A_t(\mathcal{R})$ into a cartesian double category, defining a suitable initial model for term tile logic.

4 Relating Double Categories with Extended 2-Categories

The definition of cartesian double categories adopted in Section 3 imposes strong restrictions on the way in which 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.

Since 2-categories are one of the most natural frameworks to interpret the semantics of a rewriting system, we believe that a *consistent* mapping from cartesian double categories to 2-categories could offer a good theoretical support for implementing term tile rewrite systems. In fact, rewriting logic is the semantic basis of several language implementation efforts [7, 30, 15]. In particular, the language Maude [15, 52], developed at SRI International, is based on rewriting logic and is efficiently implemented. Maude comes also equipped with the important feature of user-definable internal execution strategies [19] which allow the meta-control of rewriting computations. This is very important because a correct embedding of double categories into 2-categories heavily depends on the execution of suitable (internal) rewriting strategies.

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* [56]. The features of partial membership equational logic (partiality, poset of sorts, membership assertions) offer a natural framework for the specification of categorical structures. In [58] two of the authors presented an extended version of 2-categories, called 2VH-categories, able to include appropriately the structure of double categories. The theory of 2VH-categories can be easily expressed as a theory in partial membership equational logic, as well as the theories of 2-categories and double categories. Moreover, in [58] a notion of *tensor product* of theories is explicitly defined using the formulae of partial membership equational logic. The tensor product construction allows a very elegant definition of double categories (the tensor product of the theory of categories with itself) and their monoidal version (the tensor product of the theory of double categories with the theory of monoids).

In this section we introduce a richer version of 2VH-categories, able to include in an appropriate sense the structure of symmetric strict monoidal double categories and cartesian double categories (and more generally, double categories with shared structure) by putting in evidence what the auxiliary common structure is (thus avoiding the explicit definition of the numerous axioms presented in Section 3).

4.1 Partial Membership Equational Logic

This section defines the basic notions of *partial membership equational logic* (*PMEqtl*) [56, 10, 54]. 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, in which the poset of sorts has a single connected component. A more detailed exposition for the many-kinded case can be found in [56].

4.1.1 Partial Algebras and Membership Equational Theories

Definition 4.1 [Partially Ordered Signature] A *partially ordered signature* (*po-signature*) is a triple $\Omega = (S, \leq, \Sigma)$, with $(S, \leq)$ a poset with a top element $\top$, and $\Sigma = \{\Sigma_k\}_{k \in \mathbb{N}}$ a family of sets of operators, indexed by natural numbers. $(S, \leq)$ is called the poset of *sorts* of Ω. □

Definition 4.2 [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 $A_f : A_\top^k \rightharpoonup A_\top$.

Given two partial Ω-algebras A and B, an *Ω-homomorphism* from A to B is a function $h : A_\top \to B_\top$ 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 $h|_s : A_s \to B_s$);
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 4.3 [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

$$t : s$$

where $t \in T_\Sigma(X)$, and $s \in S$.

General Ω-sentences are then Horn clauses of the form

$$\forall \tilde{X}\ t = t' \Leftarrow u_1 = v_1 \ \wedge \ \ldots \ \wedge \ u_n = v_n \ \wedge \ w_1 : s_1 \ \wedge \ \ldots \ \wedge \ w_m : s_m$$

or of the form

$$\forall \tilde{X}\ t : s \Leftarrow u_1 = v_1 \ \wedge \ \ldots \ \wedge \ u_n = v_n \ \wedge \ w_1 : s_1 \ \wedge \ \ldots \ \wedge \ w_m : s_m$$

where the t, t', u_i, v_i and w_j are all terms in $T_\Sigma(X)$. □

Definition 4.4 [Theory, Model] Given a partial Ω-algebra A and a variable declaration $\tilde{X}$, we can define assignments $a : \tilde{X} \to A$ 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 then 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 Γ 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 Γ. In other words, the pair $T = (\Omega, \Gamma)$ 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 [58]. The theory is presented in a self-explanatory Maude-like notation [15], which will be used extensively in the rest of the paper. In the following, we will denote theories either by their Maude name (e.g., `CAT`), or by their indexed notation (e.g., T_{CAT}).

Example 4.5 [*Categories*] The theory of categories T_{CAT} is a theory in PMEqtl. 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 binary composition operator is defined iff the codomain of the first argument is equal to the domain of the second argument. As usual in many presentation, here objects are identified with the corresponding identity arrows. Moreover, to shorten the notation, we write

$\texttt{cmb}\ \psi_1\ \cdots\ \psi_n\ \texttt{iff}\ \phi_1\ \texttt{and}\ \cdots\ \texttt{and}\ \phi_m\ \ .$

where each ψ_i is a membership assertion, as a shorthand for

$$
\begin{array}{l}
\texttt{cmb}\ \psi_1\ \texttt{if}\ \phi_1\ \texttt{and}\ \cdots\ \texttt{and}\ \phi_m\ \ . \\
\qquad\quad\vdots \\
\texttt{cmb}\ \psi_n\ \texttt{if}\ \phi_1\ \texttt{and}\ \cdots\ \texttt{and}\ \phi_m\ \ . \\
\texttt{cmb/ceq}\ \phi_1\ \texttt{if}\ \psi_1\ \texttt{and}\ \cdots\ \texttt{and}\ \psi_n\ \ . \\
\qquad\quad\vdots \\
\texttt{cmb/ceq}\ \phi_m\ \texttt{if}\ \psi_1\ \texttt{and}\ \cdots\ \texttt{and}\ \psi_n\ \ .
\end{array}
$$

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). This and most of the other shorthands that we use are also summarized in Appendix D. Then the theory of categories can be defined as follows.

```
fth CAT is
  sorts Object Arrow .
  subsort Object < Arrow .
  ops d(_) c(_) _;_ .
  vars f g h : Arrow .
       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
```

It is easy to check that a model of `CAT` is exactly a category, and that a `CAT`-homomorphism is exactly a functor. □

Definition 4.6 [Signature and Theory Morphism] Given two po-signatures $\Omega = (S, \leq, \Sigma)$ and $\Omega' = (S', \leq', \Sigma')$, a *signature morphism* $H : \Omega \to \Omega'$ is given by:

1. a monotonic function $H : (S, \leq) \to (S', \leq')$, and
2. an $\mathbb{N}$-indexed family of functions $\{H_k : \Sigma_k \to \Sigma'_k\}_{k \in \mathbb{N}}$.

Such a signature morphism induces a forgetful functor $U_H : \mathbf{PAlg}_{\Omega'} \to \mathbf{PAlg}_{\Omega}$, 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[10] $f \in \Sigma_k$, $U_H(A')_f = A'_{H(f)} \cap (\underbrace{A'_{H(\top)} \times \ldots \times A'_{H(\top)}}_{k} \times A'_{H(\top)})$;

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.

Given theories (Ω, Γ) and (Ω', Γ'), a *theory morphism* $H : (\Omega, \Gamma) \to (\Omega', \Gamma')$ is a signature morphism $H : \Omega \to \Omega'$ such that $U_H(\mathbf{PAlg}_{\Omega',\Gamma'}) \subseteq \mathbf{PAlg}_{\Omega,\Gamma}$, so that U_H restricts to a forgetful functor $U_H : \mathbf{PAlg}_{\Omega',\Gamma'} \to \mathbf{PAlg}_{\Omega,\Gamma}$. $\Box$

The reader is referred to [56] for proof-theoretical conditions on Γ and Γ' ensuring that a signature morphism $H : \Omega \to \Omega'$ is a theory morphism $H : (\Omega, \Gamma) \to (\Omega', \Gamma')$.

Proposition 4.7 (Free Construction Associated to a Theory Morphism [56]) *Given a theory morphism $H : (\Omega, \Gamma) \to (\Omega', \Gamma')$, its associated forgetful functor $U_H : \mathbf{PAlg}_{\Omega',\Gamma'} \to \mathbf{PAlg}_{\Omega,\Gamma}$ has a left adjoint $F_H : \mathbf{PAlg}_{\Omega,\Gamma} \to \mathbf{PAlg}_{\Omega',\Gamma'}$.*

Definition 4.8 [Conservative, Complete and Persistent Morphisms] A theory morphism $H : (\Omega, \Gamma) \to (\Omega', \Gamma')$ is *conservative* (resp. *complete, persistent*) *w.r.t. sort s* if, for each algebra $A \in \mathbf{PAlg}_{\Omega,\Gamma}$, the component $(\eta_A)_s : A_s \to (U_H(F_H(A))_s$ corresponding to s of the unit of the adjunction associated to H is injective (resp. surjective, bijective). The morphism H is *conservative* (resp. *complete, persistent*) if it is conservative (resp. complete, persistent) w.r.t. all $s \in S$. $\Box$

Definition 4.9 [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 itself an Ω-algebra, and that the inclusion function $B \subseteq A$ is an Ω-homomorphism. $\Box$

Lemma 4.10 *For any set Γ 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 4.11 [*Subcategories*] For the theory `CAT` of Example 4.5, the subalgebras of a category $\mathcal{C}$ are exactly its subcategories. $\Box$

Note that the notion of Ω-subalgebra is strictly stronger than that of Ω-monomorphism. It is easy to check that, in the category $\mathbf{PAlg}_\Omega$, $m : C \to A$ is a monomorphism iff the associated function $m : C_\top \to A_\top$ is injective. Of course, by taking the smallest image of m, any such monomorphism always factors through an isomorphism and an inclusion $C \xrightarrow{\sim} B \hookrightarrow A$, where B is a *weak* subalgebra, as defined below.

Definition 4.12 [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. $\Box$

In general, given a set Γ of Horn sentences in PMEqtl, and a partial algebra $A \in \mathbf{PAlg}_{\Omega,\Gamma}$, a weak subalgebra B of A need not satisfy the sentences Γ. 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 $(\Omega, \Gamma) =$ `CAT`, the following relationship happens to hold between subalgebras and weak subalgebras.

Example 4.13 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. $\Box$

[10]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 $\top$.

4.1.2 The Tensor Product Construction

Given a signature $\Omega = (S, \leq, \Sigma)$ and a category $\mathcal{C}$ with finite limits and a with suitable poset of canonical inclusions $\mathcal{I}$, in [58] it is shown how to define partial Ω-algebras in $\mathcal{C}$, denoted by $\mathbf{PAlg}_{\Omega}(\mathcal{C})$ and $\mathbf{PAlg}_{\Omega,\Gamma}(\mathcal{C})$. 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 $T = (\Omega, \Gamma)$ and $T' = (\Omega', \Gamma')$. Moreover, in a way analogous to algebraic theories [45, 29], to lim theories [31], and to sketches [42], in [58] the construction of a *tensor theory* $T \otimes T'$ in partial membership equational logic is given 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 favor instead the notion of tensor product associated to subalgebras. Nevertheless, in the special case $T' = \mathtt{CAT}$, because of the property mentioned in Example 4.13, the definition of $\mathbf{PAlg}_T(\mathbf{PAlg}_{T_{CAT}})$ is the same whether we choose subalgebras or instead weak subalgebras as canonical inclusions in $\mathbf{PAlg}_{T_{CAT}}$.

The explicit definition of $T \otimes T'$ is as follows.

Definition 4.14 [Tensor Product] Let $T = (\Omega, \Gamma)$ and $T' = (\Omega', \Gamma')$ 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[11] $f^l \in (\Sigma \otimes \Sigma')_n$ 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 in Γ

$$\alpha \;=\; \forall (x_1 : \overline{s_1}, \ldots, x_m : \overline{s_m})\; \varphi(\vec{x}) \Leftarrow c(\vec{x})$$

with $\overline{s_i} = \{s_{i1}, \ldots, s_{il_i}\}$, $1 \leq i \leq m$, we introduce an axiom

$$\alpha^l \;=\; \forall (x_1 : \overline{s_1^l}, \ldots, x_m : \overline{s_m^l})\; \varphi^l(\vec{x}) \Leftarrow c^l(\vec{x})$$

with $\overline{s_i^l} = \{(s_{i1}, \top'), \ldots, (s_{il_i}, \top')\}$, $1 \leq i \leq m$, 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 $\beta \in \Gamma'$ the axiom β^r and impose all these axioms.

[11] Here, superscripts l and r of operators stand respectively for *left* and *right*.

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:

$$\begin{array}{ll} \forall \vec{x} & f^l(g^r(\vec{x_{1\_}}), \ldots, g^r(\vec{x_{n\_}})) = g^r(f^l(\vec{x_{\_1}}), \ldots, f^l(\vec{x_{\_m}})) \Leftarrow \\ & \Leftarrow \bigwedge_{1 \leq i \leq n} g^r(\vec{x_{i\_}}) : (\top, \top') \;\;\wedge\;\; \bigwedge_{1 \leq j \leq m} f^l(\vec{x_{\_j}}) : (\top, \top'). \end{array}$$

where

$$\begin{array}{lll} \vec{x} & = \{x_{ij} : (\top, \top')\}^{1 \leq i \leq n}_{1 \leq j \leq m}, & \\ \vec{x_{i\_}} & = \{x_{ij} : (\top, \top')\}_{1 \leq j \leq m}, & 1 \leq i \leq n \\ \vec{x_{\_j}} & = \{x_{ij} : (\top, \top')\}_{1 \leq i \leq n}, & 1 \leq j \leq m. \end{array}$$

□

The essential property of $T \otimes T'$ is expressed in the following theorem, whose proof will be given elsewhere.

Theorem 4.15 (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 $H : T_1 \rightarrow T_2$ and $G : T'_1 \rightarrow T'_2$ are theory morphisms, we have an associated theory morphism:

$$H \otimes G : T_1 \otimes T'_1 \rightarrow T_2 \otimes T'_2.$$

It can be shown that the tensor product of theories is associative and commutative up to isomorphism, that is, that 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.

Example 4.16 [*Double categories*] A *double category* has been defined [25] as a category structure on **Cat**, that is, as an object of $\mathbf{PAlg}_{T_{CAT}}(\mathbf{PAlg}_{T_{CAT}}) = \mathbf{DCat}$. 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 refdcatposet):

$$
\begin{tikzcd}
 & Cell & \\
Harrow \arrow[ur] & & Varrow \arrow[ul] \\
 & Object \arrow[ul] \arrow[ur] &
\end{tikzcd}
$$

Figure 3: The poset of sorts for T_{DCAT}.

$$
\begin{array}{ll}
(\texttt{Object}, \texttt{Object}) = \texttt{Object}, & (\texttt{Arrow}, \texttt{Arrow}) = \texttt{Square}, \\
(\texttt{Arrow}, \texttt{Object}) = \texttt{Harrow}, & (\texttt{Object}, \texttt{Arrow}) = \texttt{Varrow}, \\
\texttt{Object} \leq \texttt{Harrow} \leq \texttt{Square}, & \texttt{Object} \leq \texttt{Varrow} \leq \texttt{Square}.
\end{array}
$$

For the operations in $\Omega \otimes \Omega'$ we adopt the intuitive North-East-West-South notation:

$$d^l = w,\ c^l = e,\ d^r = n,\ c^r = s,\ (\_;\_)^l = \_*\_,\ (\_;\_)^r = \_\cdot\_.$$

The presentation of double categories in Maude-like notation is thus as follows [58].

```
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
```

Notice that in the above axiomatization 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 $Harrow \cap Varrow = Object$. □

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
```

4.1.3 2-Categories and 2VH-Categories

2-Categories [41] are probably the best known kind of enriched category. In particular, they yield models of rewriting logic in a very natural way [50]. 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 `_∘_`. The explicit Maude-like definition of 2-categories [58] 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 _∘_ ) .
endfth
```

The extended version of a 2-category proposed in [58], called a 2VH-category, includes the double category structure and has the poset of sorts shown in Figure 4.

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

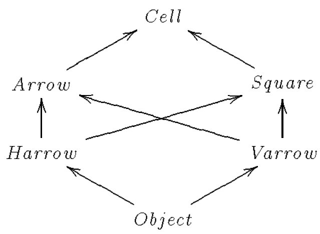


Figure 4: The poset of sorts for T_{2VHCAT}.

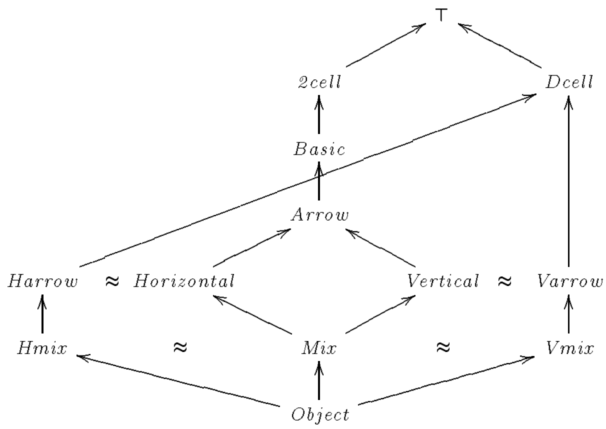


Figure 5: The poset of sorts for $T_{2EVHCAT}$.

of the 2-category. We omit the complete description of T_{2VHCAT} (see [58]), but 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, in which the problem is elegantly solved. Then, analogous results to those in [58] can be proved in the extended setting.

4.2 Extended 2VH-Categories

Basically, we could think of auxiliary constructors (i.e., symmetries, duplicators and dischargers) as shared structure 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 5. 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 5, 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 refcompsec). 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 in which the source and target arrows of the cell are decomposed into the composition of Horizontal and Vertical arrows. Moreover there are some cells that don't 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 follows from the definition of the more general mapping.

We present the Maude-like definition of the theory $T_{2EVHCAT}$, alternating the source code with some explanations and examples. We start by giving the formal translation of Figure 5 and fixing the variable notation for each subsort.

```
fth 2EVHCAT is
  including 2CAT renamed by (sort Cell to 2cell) .
  sorts Mix Horizontal Vertical Basic ⊤
        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 (resp. Vertical) arrows is also a Horizontal (resp. 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 which 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 the double cells starting from the Basic cells. Informally, we want to distinguish between all the possible double cells which are generated by different decomposition 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

$$\begin{tikzcd} & b \arrow[dr, dotted, "u"] & \\ a \arrow[ur, "h"] \arrow[dr, dotted, "v"'] \arrow[rr, phantom, "s\Downarrow"] & & d \\ & c \arrow[ur, "g"'] & \end{tikzcd}$$

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 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.

$$\begin{tikzcd} & b \arrow[dr, dotted, "b"] & \\ a \arrow[ur, "m"] \arrow[dr, dotted, "m"'] \arrow[rr, phantom, "m_1\Downarrow"] & & b \\ & b \arrow[ur, "b"'] & \end{tikzcd} \qquad \begin{tikzcd} & b \arrow[dr, dotted, "b"] & \\ a \arrow[ur, "m"] \arrow[dr, dotted, "a"'] \arrow[rr, phantom, "m_2\Downarrow"] & & b \\ & a \arrow[ur, "m"'] & \end{tikzcd} \qquad \begin{tikzcd} & a \arrow[dr, dotted, "m"] & \\ a \arrow[ur, "a"] \arrow[dr, dotted, "m"'] \arrow[rr, phantom, "m_3\Downarrow"] & & b \\ & b \arrow[ur, "b"'] & \end{tikzcd} \qquad \begin{tikzcd} & a \arrow[dr, dotted, "m"] & \\ a \arrow[ur, "a"] \arrow[dr, dotted, "a"'] \arrow[rr, phantom, "m_4\Downarrow"] & & b \\ & a \arrow[ur, "m"'] & \end{tikzcd}$$

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 = `mkh(`m`)` and m_3 = `mkv(`m`)`. The other two kinds of 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 3. 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 `πcell(_)`, 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 `mk(_:_,_,_,_)` operation. As usual, we will refer to the sources and targets of double cells with the self-explanatory "cardinal-point" notation `n(_)`, `s(_)`, `e(_)`, and `w(_)`. We begin by defining the sort of the four projections for each kind of arrows (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 `πcell(_)` and the sources and targets of the `2cells`. Notice that, once either the source `d(_)` or the target `c(_)` of a `2cell` has been defined, then the axioms of 2-categories give also the value of the source `l(_)` and target `r(_)` of the `2cell` under consideration.

```
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)) .
```

We now define horizontal ($\_ * \_$) and vertical ($\_ \cdot \_$) composition operations between composable `Dcells` in terms of the given compositions between underlying `2cells`. As before, 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 $T_{2EVHCAT}$.

4.3 Monoids and Symmetries

For the expressiveness of our approach it is essential to add a monoidal structure to the model presented above. In fact, the shared structures on which we focus 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 of [58] (see Def. 4.14). It suffices to introduce a theory `MON` of (strict) monoids 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
```

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 `f(`t_1`,`t_2`...,`t_n`)` can be used for any $n \in \mathbb{N}$. The membership assertion `mb` $i \otimes j$ `: Monoid` just specifies that the monoidal operation is total. Then, the theory of (strict) monoidal extended 2VH-categories can be expressed in a Maude-like notation as follows:

```
fth MON2EVHCAT is MON ⊗ 2EVHCAT renamed by (
  sorts (Monoid,Object) to Object . (Monoid,Mix) to Mix .
        (Monoid,Hmix) to Hmix . (Monoid,Vmix) to Hmix .
        (Monoid,Harrow) to Harrow . (Monoid,Varrow) to Harrow .
        (Monoid,Horizontal) to Horizontal . (Monoid,Vertical) to Vertical .
        (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(_:_,_,_,_) .  πcell(_) right to πcell(_) .
       mkh(_) right to mkh(_) . mkv(_) right to mkv(_) . ) .
endfth
```

In the same way, it is easy to define the monoidal theories of categories, 2-categories and double categories by very similar tensor product constructions between `MON` and `CAT`, `2CAT` and `DCAT`, called `MONCAT`, `MON2CAT` and `MONDCAT` respectively (also illustrated in [58]).

```
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 Harrow .
        (Monoid,Arrow) to Arrow . (Monoid,Square) to Square .
        (Monoid,Cell) to Cell .
        (Monoid,⊤) to ⊤ .
  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
```

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 `Objects` we introduce an arrow `sym(`a`,`b`)` of sort `Mix`, which plays the role of the symmetry for a and b. The naturality axiom schema is defined for each pair of `Arrows` t and t', and the coherence axioms are stated to equate all the different compositions of symmetries leading to the same final result.

```
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
```

A similar construction applies to the case of 2-categories. In this case, the Maude-like definition should include `MON2CAT` instead on `MONCAT`, the source and target functions `d(_)` and `c(_)` should be renamed by `l(_)` and `r(_)`, and the naturality axiom should involve 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 holds also for generic arrows. Thus we propose the following shorter notation for the definition of `SYM2CAT`:

```
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
```

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 emphasizing 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.

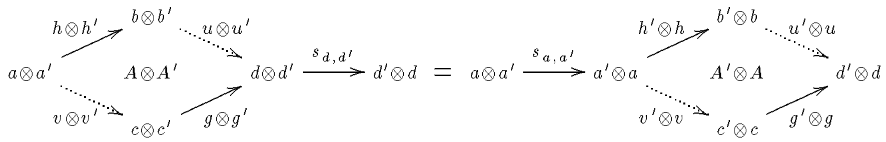
The case of symmetric monoidal double categories (the theory SYMDCAT) is more involved and requires all the axioms presented in Section 3.5 for the generalized symmetries, plus some additional axioms induced by the membership logic. We leave as an exercise for the really very interested reader to translate the description of symmetric (strict) monoidal double categories into Maude-like notation.

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 (respectively γ(_,_), ρ(_,_), and σ(_,_)) acting on Dcells.

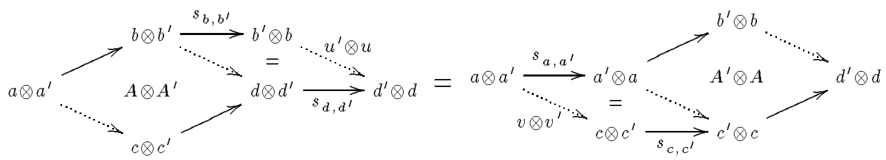
```
fth SYM2EVHCAT is
  including MON2EVHCAT .
  including SYM2CAT .
  ops γ(_,_) ρ(_,_) σ(_,_) .
  vars a a' b b' : Object .
       ha ha'   : Harrow .
       va va'   : Varrow .
  mb  sym(a,b) : Mix .
  ceq ρ(ha,ha') =  mk((πcell(ha)⊗πcell(ha'));sym(b,b'):
                      (πcell(ha)⊗πcell(ha')),sym(b,b'),
                      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
```

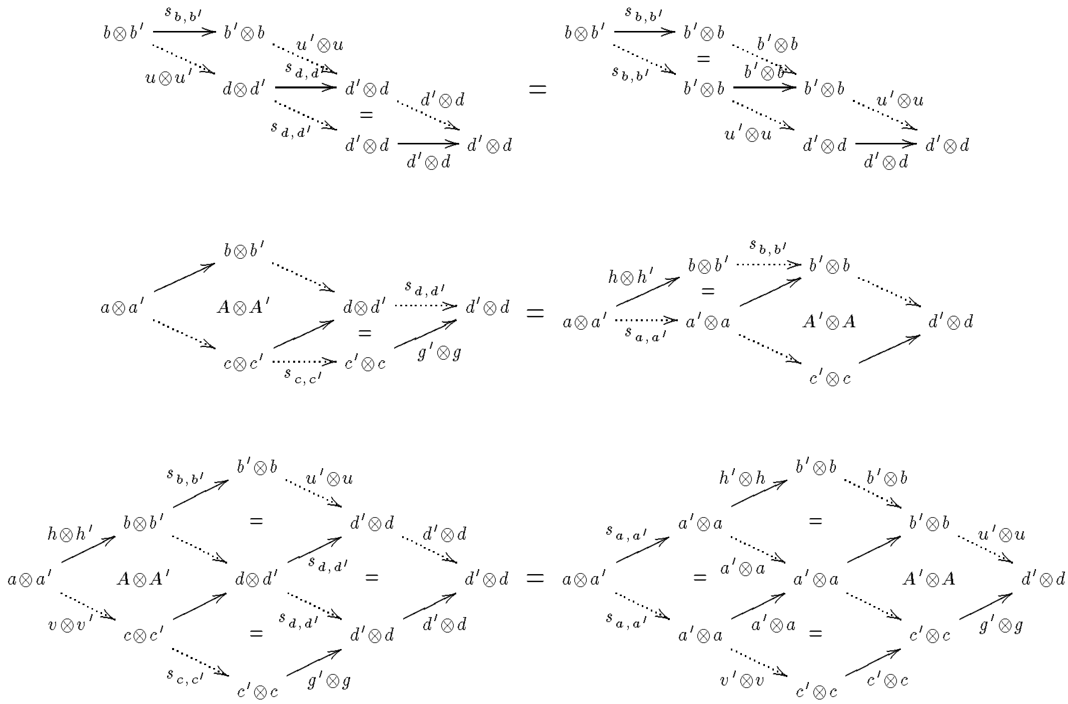
Theorem 4.17 *The obvious signature morphism from* SYMDCAT *to* SYM2EVHCAT *is a theory morphism.*

Proof (Sketch). We need to prove that the axioms of symmetric monoidal double categories can be derived from the axioms of SYM2EVHCAT. We graphically hint at the part of the naturality proof. From the axiom



the following equations of symmetric double categories can then be derived:





□

The theory morphism above can be specified in Maude-like notation as:

```
view SE from SYMDCAT to SYM2EVHCAT is
     sort Square to Dcell .
endview
```

Then, the following result holds.

Theorem 4.18 *The theory morphism* SE *from* SYMDCAT *to* SYM2EVHCAT *is persistent w.r.t. sorts* Objects, Harrow, *and* Varrow *and it is complete.*

This means that, in general, it is possible for some double cells which are distinguished in SYMDCAT to be identified as the same Dcell.

4.4 Cartesian Theories

The theory of cartesian categories is expressed as follows:

```
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
```

As for the symmetric 2-categories, the theory of cartesian 2-categories can be defined simply adding the naturalities on cells.

```
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
```

The definition of cartesian double categories is subject to the explicit axiomatization given in Section 3.6. As for the symmetric strict monoidal case it is possible to derive the needed axiomatization from the one of cartesian extended 2VH-categories given below. 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 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
```

Theorem 4.19 *The obvious signature morphism between* `CARTDCAT` *and* `CART2EVHCAT` *is a theory morphism.*

The theory morphism above can be specified in Maude-like notation as:

```
view CE from CARTDCAT to CART2EVHCAT is
     sort Square to Dcell .
endview
```

As in the case of symmetric double categories, an analogous result holds for the cartesian case.

Theorem 4.20 *The theory morphism* `CE` *from* `CARTDCAT` *to* `CART2EVHCAT` *is persistent w.r.t. sorts* `Objects`*,* `Harrow`*, and* `Varrow` *and it is complete.*

5 Computads

The notion of *computad* [68, 69] allows 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 mantaining a modular approach to the system description.

Definition 5.1 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* 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 (vertical) source and target of $F_d(s)$ are the images through[12] F_v (F_h) of the horizontal (vertical) source and target of s.

A computad is *symmetric* (*cartesian*) if both H and V are symmetric monoidal categories (cartesian categories) with symmetries $\gamma = \{\gamma_{a,b}\}_{a,b\in O}$ and $\rho = \{\rho_{a,b}\}_{a,b\in O}$, respectively (with duplicators $\nabla = \{\nabla_a\}_{a\in O}$, $\delta = \{\delta_a\}_{a\in O}$, and dischargers $! = \{!_a\}_{a\in O}$ and $\dagger = \{\dagger_a\}_{a\in O}$, respectively); c-morphisms then preserve the additional symmetric monoidal (cartesian) structure. □

The Maude-like definition of symmetric (cartesian) computad is obtained by replacing `MONCAT` with `SYMCAT` (`CARTCAT`) in the theory `CTD` defined in [58].

```
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 .
```

[12] 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.

```
  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
```

A very similar construction applies to the cartesian case.

```
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
```

A brief explanation is necessary. We import two separated symmetric (respectively, cartesian) structures for both horizontal and vertical arrows. At this level, only the 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 (respectively, 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 [58] for the monoidal case can then be extended to the symmetric and cartesian cases.

Proposition 5.2 *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
```

Thus, 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
```

Proposition 5.3 *The forgetful functor* $U_{\mathrm{SD}} : \mathbf{SymDCat} \longrightarrow \mathbf{SymCtd}$ *associated to the theory morphism* `SD` *has a left adjoint* $F_{\mathrm{SD}} : \mathbf{SymCtd} \longrightarrow \mathbf{SymDCat}$*. Similarly, the forgetful functor* $U_{\mathrm{SVH}} : \mathbf{Sym2EVHCat} \longrightarrow \mathbf{SymCtd}$ *has a left adjoint* $F_{\mathrm{SVH}} : \mathbf{SymCtd} \longrightarrow \mathbf{Sym2EVHCat}$*. Furthermore,* F_{SVH} *is given by the composition of the functor* F_{SD} *with the left adjoint* F_{SE} *to the forgetful functor* $U_{\mathrm{SE}} : \mathbf{Sym2EVHCat} \longrightarrow \mathbf{SymDCat}$*.*

Proposition 5.4 *The forgetful functor* $U_{\mathrm{CD}} : \mathbf{CartDCat} \longrightarrow \mathbf{CartCtd}$ *has a left adjoint* $F_{\mathrm{CD}} : \mathbf{CartCtd} \longrightarrow \mathbf{CartDCat}$*. Similarly, the forgetful functor* $U_{\mathrm{CVH}} : \mathbf{Cart2EVHCat} \longrightarrow \mathbf{CartCtd}$ *has a left adjoint* $F_{\mathrm{CVH}} : \mathbf{CartCtd} \longrightarrow \mathbf{Cart2EVHCat}$*. Furthermore,* F_{CVH} *is given by the composition of the functor* F_{CD} *with the left adjoint* F_{CE} *to the forgetful functor* $U_{\mathrm{CE}} : \mathbf{Cart2EVHCat} \longrightarrow \mathbf{CartDCat}$*.*

5.1 VH-computads

Taking advantage of the sort `Basic`, it is possible to follow an alternative construction, still obtaining an analogous result. The idea is to reduce each cartesian (respectively symmetric) computad to a suitable cartesian (respectively symmetric) 2-computad, called *VH-computad*, which can then be used to freely generate the associated cartesian (respectively symmetric) 2EVH-category.

Definition 5.5 A *VH-computad* is a quadruple $\langle A, H, V, D\rangle$, where H and V are lluf[13] subcategories of the category A (i.e., H, V, and A have exactly the same objects), and D is a set of cells.

[13] A *lluf* subcategory of a category $\mathcal{C}$ is just a subcategory of $\mathcal{C}$ having exactly the same objects as C.

Each cell has assigned a pair of compatible arrows in A as vertical source and target, respectively. 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 *cartesian* (respectively, *symmetric*) if both A, H, and V are cartesian (respectively, symmetric) categories. □

```
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
```

```
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
```

Notice that the theories SYMVHCTD and CARTVHCTD only differ in the the imported theories.

Proposition 5.6 *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.*

Analogously, 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 5.14 establishes the relevance of these alternative constructions.

5.2 Term Tile Rewriting 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 tTRS 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 rewrite 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).

Definition 5.7 Let $\mathcal{R} = \langle \Sigma_H, \Sigma_V, N, R\rangle$ be a tTRS. 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

$$\begin{tikzcd} n \arrow[r, "\vec{h}"] \arrow[d, "\vec{v}"'] \arrow[dr, phantom, "r"] & m \arrow[d, "u"] \\ k \arrow[r, "g"'] & 1 \end{tikzcd} \in T_R \qquad \Longleftrightarrow \qquad r : n \lhd \langle \vec{h}\rangle \xrightarrow[\langle u\rangle]{\langle \vec{v}\rangle} \langle g\rangle \ \in \mathcal{R}$$

□

Definition 5.8 Given a tTRS $\mathcal{R}$, the *cartesian tile logic* of $\mathcal{R}$ is the cartesian double category $\mathcal{L}_{\mathcal{D}}(\mathcal{R}) = F_{CE}(Ctd(\mathcal{R}))$ freely generated from the computad $Ctd(\mathcal{R})$ by the left adjoint functor F_{CE} described in Proposition 5.4. For $\alpha \in F_{CE}(Ctd(\mathcal{R}))_{Square}$ we also write $\mathcal{R} \vdash_c \alpha$ ($\mathcal{R} \vdash_{fc} \_$ for flat sequents). □

Theorem 5.9 *Given a tTRS* $\mathcal{R} = \langle \Sigma_H, \Sigma_V, N, R\rangle$*, then* $\mathcal{R} \vdash_c \alpha \Longleftrightarrow \mathcal{R} \vdash_t \alpha$.

Definition 5.10 Given a tTRS $\mathcal{R} = \langle \Sigma_H, \Sigma_V, N, R\rangle$ the *extended logic* of $\mathcal{R}$ is the cartesian 2EVH-category $F_{CVH}(Ctd(\mathcal{R})) = F_{CE}(F_{CD}(Ctd(\mathcal{R})))$ freely generated from $Ctd(\mathcal{R})$ by the left adjoint functor described in Proposition 5.4. For $\beta \in F_{CVH}(Ctd(\mathcal{R}))_{Dcell}$ we also write $\mathcal{R} \vdash_e \beta$ ($\mathcal{R} \vdash_{fe} \_$ for flat sequents). □

Corollary 5.11 *Given a tTRS* $\mathcal{R} = \langle \Sigma_H, \Sigma_V, N, R\rangle$*, 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 instead of from the double computad and the result does not change.

Next, we define how to construct a VH-Computad starting from a given tile-computad.

Definition 5.12 Let $\mathcal{R} = \langle \Sigma_H, \Sigma_V, N, R\rangle$ be a tTRS. The associated cartesian vh-computad $Cvh(\mathcal{R})$ is the quadruple $\langle T_{\Sigma_H \cup \Sigma_V}(X), T_{\Sigma_H}(X), T_{\Sigma_V}(X), D_R\rangle$, where the set of basic cells D_R is such that

$$D_R = \left\{ \hat{r} : \begin{tikzcd} n \arrow[r, bend left, "u(\vec{h})"] \arrow[r, bend right, "g(\vec{v})"'] \arrow[r, phantom, "\Downarrow"] & 1 \end{tikzcd} \ \middle|\ r : n \lhd \langle \vec{h}\rangle \xrightarrow[\langle u\rangle]{\langle \vec{v}\rangle} \langle g\rangle \ \in R \right\}$$

□

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 5.13 Given a tTRS $\mathcal{R}$, the *cartesian VH-logic* of $\mathcal{R}$ is the cartesian 2EVH-category $F_{C2VH}(Cvh(\mathcal{R}))$ freely generated from $Cvh(\mathcal{R})$ by the left adjoint functor associated to the theory morphism of proposition 5.6. For $\beta \in F_{C2VH}(Ctd(\mathcal{R}))_{Dcell}$ we also write $\mathcal{R} \vdash_{cVH} \beta$ ($\mathcal{R} \vdash_{fcVH}$ - for flat sequents). □

Theorem 5.14 *Given a tTRS* $\mathcal{R} = \langle \Sigma_H, \Sigma_V, N, R\rangle$, *then* $\mathcal{R} \vdash_{ft} h \xrightarrow[u]{v} g \Longleftrightarrow \mathcal{R} \vdash_{fcVH} h \xrightarrow[u]{v} g$.

Completely analogous results hold for process tile logic.

The constraint imposed on the extended logics that the rewrite proof 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 we have studied – the term tile rewrite system is *uniform*, then the internal strategy becomes very simple and may be inserted in a standard way directly in the specification layer (see Section 6.6).

Definition 5.15 [Uniform Systems] A cartesian (respectively symmetric strict monoidal) double category $\mathcal{D}$ is *uniform* if the 2EVH-category $F_{CE}(\mathcal{D})$ (respectively $F_{SE}(\mathcal{D})$) satisfies the following conditional membership axiom:
$(\forall A : \texttt{2cell},\ h, g : \texttt{Horizontal},\ v, u : \texttt{Vertical})$.

$$\texttt{mk}(A : h, u, v, g) : \texttt{Dcell} \Leftarrow d(A) = h; u \wedge c(A) = v; g$$

A tTRS (respectively pTRS) $\mathcal{R}$ is *uniform* if its associated cartesian (respectively symmetric strict monoidal) double category is uniform. □

6 Dealing with Nondeterminism

The theoretical results presented in the previous sections show that there is a strong relationship between the sequents which are derivable in the two (not so far apart) worlds of term tile logic and term rewriting logic. Unfortunately, these results cannot be applied directly in rewriting implementations of tile systems, because of the implicit nondeterminism in the specification. For instance, the fact that a sequent is entailed by the inference rules of rewriting logic does not imply that in an actual implementation of the system the particular rewriting computation leading to that sequent will be performed. Therefore, we are in need of a methodological approach which could drive the computation along the correct paths. 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 use Maude to define these strategies as a general layer to be placed on top of the specification layer.

6.1 Nondeterministic Rewriting Systems

In most cases, the behaviour of a process in a concurrent system is dependent upon the behaviours of the other processes cooperating in the same system. For instance, in some critical states, a process must have the opportunity of checking incoming communications from many sources, without, at least in principle, granting a privilege to some source or to a particular kind of input. Thus, a specification language for concurrent systems cannot leave out of consideration some mechanism for expressing (guarded) non-deterministic choices in the body of a process. Such a mechanism should allow dealing with the possible interactions between each process and the "rest of the world".

We can distinguish between three implementations – namely *conditional choice*, *don't know nondeterminism*, and *don't care nondeterminism* – of the mechanism described above, each corresponding to a different language construct[14]. Just to fix the notation, we introduce some abstract definitions, assuming that both the description of the *statements* of our language, and the notion of *state* of the system, are elsewhere defined and known "a priori" by the reader (i.e., a statement can involve some communication on a given channel, or some assignment to a shared variable, while the state can be thought of as either local, private to the process, or global, assuming the knowledge of a snapshot of the whole system at a given moment). Then, a *guard* is a predicate over the collection of *states*. The evaluation $Gd(st)$ of a guard Gd in a state st can give three possible results: *true*, *false*, or *undecided*. The evaluation of the empty guard ϵ returns *true* in any state st. If $Gd(st) = true$, then we say that the state st *satisfies* the guard Gd. If $Gd(st) = false$ then we say that the guard Gd *fails* in the state st. Otherwise, we say that state st *postpones* the guard Gd. Now we analyze in some detail the differences between the three approaches listed above. Each construct is a finite collection of *clauses*. The first component of a clause is a guard and the second component is a statement, which is called the *body* of the clause.

1. **Conditional choice**. This approach provides the user with a powerful control tool on the execution, because the programmer **knows** which clause will be chosen if more than one is satisfied. We denote this construct by

$$(Gd_1 \rightarrow Stat_1; Gd_2 \rightarrow Stat_2; \ldots; Gd_n \rightarrow Stat_n),$$

where the symbol "$\rightarrow$" is read as *then*. The behaviour of the conditional choice statement is as follows. Its guards Gd_1, Gd_2, ..., Gd_n are evaluated in the current state st. If a guard fails, then the corresponding clause is "deleted". If all clauses are deleted, then the choice statement *fails*. If the state st satisfies the **first remaining** guard, say Gd_i, then the conditional choice statement is replaced by the body $Stat_i$ of the corresponding clause. If there are no satisfied guards (i.e., all the remaining guards are postponed by state st and

[14]Although the syntax used for each construct has been conveniently adapted from a concurrent constraint language [38], the notions considered here are very general and widely used [6] in computer science.

there's at least one clause left), then the statement *suspends* until all the guards will fail or at least one will be satisfied.

2. **Don't care nondeterminism**. This construct is especially useful in concurrent programming, where processes should be able to *react* to incoming information arriving from different sources. If the conditional choice operator is used instead, then the sources are totally ordered w.r.t. the implicit *priority* given by the listing order of the clauses. At the semantic level this represents an unfair policy. The syntax is

$$(Gd_1!Stat_1; Gd_2!Stat_2; \ldots; Gd_n!Stat_n)$$

where the symbol "!" is read *commit*, and where the overall construct is sometimes referred to as a *committed choice*. The behaviour of committed choice is as follows. Its guards Gd_1, Gd_2,..., Gd_n are evaluated in the current state st. As in the case of conditional choice statement, if a guard fails, then the corresponding clause is "deleted", and if all clauses are deleted, then the choice statement *fails*. But if **any** (not necessarily the first) of the remaining guards, say Gd_i, is satisfied by state st, then the committed choice is replaced by the body $Stat_i$ of the corresponding clause. Here, the main assumption is that whatever will be the selected choice, the system will continue behaving in the expected way. For instance, this approach is well suited whenever the Church-Rosser property holds.

3. **Don't know nondeterminism**. Sometimes it is not enough to explore just one branch, because many problems (e.g., in Artificial Intelligence or in Operations Research) are currently solvable only by resorting to some sort of *search* or by collecting the successful computations. In this case, the nondeterminism expressed by the constructor leads to a **parallel** exploration of the enabled branches. However, performance considerations suggest alternative ways when exploring the nondeterministic tree of choices (i.e., depth first equipped with backtracking instead of breadth first). The relevant point is that, under some assumption (i.e., finiteness of the tree) the user may explore all the branches, and collect all the solutions. The syntax of this construct is

$$(Gd_1?Stat_1; Gd_2?Stat_2; \ldots; Gd_n?Stat_n)$$

where the symbol "?" is read as *wait*. The behaviour of the nondeterministic choice construct is as follows. Its guards Gd_1, Gd_2,..., Gd_n are evaluated in the current state st. As for the previous statements, if a guard fails, then the corresponding clause is "deleted", and if all clauses are deleted, then the whole statement *fails*. However, if **only one** clause remains and it is satisfied by the current store st, then the choice statement is said to be *determinate* and it is replaced with the body of the corresponding clause. Otherwise, if there are more clauses left, the statement is said to be *nondeterminate* and the alternative computation paths are explored **concurrently** (in this case, also suspended guards are carried on).

In some sense, all the above descriptions are partial specifications of different informal operational semantics of some system evolving through states. Now suppose that the system comes equipped with a general notion of success and failure, which is represented by a partial predicate $ok(\_)$, defined over the collection of states. We say that a state st is *final* iff $ok(st) \in \{true, false\}$. Moreover, a computation c of the system is *successful* iff c reaches a final state st such that $ok(st) = true$ and for every state st' visited by c, then $ok(st) \neq false$. A successful computation c is *minimal* iff for every state $st' \neq st$ visited by c, $ok(st) \notin \{true, false\}$. A computation c is *failing* if $ok(st) = false$ for some state st visited by c. Obviously, one would be interested in discharging all the failing computations. This means that as soon as a failure is detected the system should stop and a new run, where different choices are made, should be considered. "Don't know" nondeterminism is very important in this case, because it allows exploring the possible computation, thus capturing all the successful computations. In this sense, if a successful computation exists, then it is surely captured by "don't know" nondeterminism. To ensure that "don't care" nondeterminism also leads always to a successful computation it is necessary that all the enabled choices lead to success. If conditional choice is considered, then it is necessary that the first enabled choice (w.r.t. the listing order) leads to success.

In (conditional) rewriting logic, a similar problem arises whenever multiple (local) rewritings are enabled for the same term, and some of them may lead to undesired computations. As an example the rewriting rules

```
crl [choice] : t(X) => t1(X) if G1(X) .
        ⋮
crl [choice] : t(X) => tn(X) if Gn(X) .
```

describe a system in which the term `t(X)` defines the nondeterministic choice

$$(\mathtt{G}_1(\mathtt{X})\ \ \kappa\ \ \mathtt{t}_1(\mathtt{X}); \ldots ; \mathtt{G}_n(\mathtt{X})\ \ \kappa\ \ \mathtt{t}_n(\mathtt{X}))$$

where the value of $\kappa \in \{\rightarrow, !, ?\}$, depends on the rewriting engine.

Maude has a default interpreter for rewrite theories that applies rules in a top-down fashion and ensure fairness in the choice of rules to be applied. This amounts to a form of "don't care" nondeterminism in which the user has virtually no control on the application of the rules. But since Maude is a reflective language it is possible to give the user full control of the rewriting by importing the metalevel of some specification, and then guiding the computation with suitable (meta-programmed) strategies [19].

6.2 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 deduction 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 metacircular 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* [17, 13] 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 metalevel may be expressed within the logic.

Given a logic, we say that it is *reflective* [18, 19] 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 ($U \in \mathcal{C}$), representation can be iterated, so that we get a *reflective tower*

$$T \vdash \varphi \Longleftrightarrow U \vdash \overline{T \vdash \varphi} \Longleftrightarrow U \vdash \overline{U \vdash \overline{T \vdash \varphi}} \cdots .$$

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 metacircular 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 such that can simulate all other finitely presented rewrite theories, including itself), making rewriting logic reflective [17, 13].

A rewrite theory T consists of a signature Σ of operators, a set E of equations, and a set of labelled rewrite rules. 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 consider 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 theory. The representation function used in [17, 13] $\overline{(\_ \vdash \_)}$ 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

$$T \vdash t \Rightarrow t' \Longleftrightarrow U \vdash \langle \overline{T}, \overline{t} \rangle \Rightarrow \langle \overline{T}, \overline{t'} \rangle,$$

where the function $\overline{(\_)}$ recursively defines the representation of rules, terms, etc. as terms in U.

6.3 Collective Strategies in Maude

Maude 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, that sends T to a definitional extension of U that defines how rewriting in T is accomplished at the metalevel. For instance, a typical semantic definition that one wants to have in *Meta-Level(T)* is that of *meta-apply*$(\overline{t}, \overline{l})$, that simulates, at the metalevel, 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.

6.3.1 The Kernel

The Maude implementation supports meta-programming of strategies via a module `META-LEVEL` defined in [14, 19], 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 a module T have sorts $\overline{t}$ `: Term` and $\overline{T}$ `: Module`. 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 in Peano representation 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, with matches that fail to satisfy the condition of their rule discarded; (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 $\overline{t}'_r$ which is returned as a result, paired with the match used in the reduction (the operator `{_,_}` is used to construct the term).

To make easier the notation, 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.

`META-LEVEL` can be considered in our terminology as a kernel internal strategy language for rewriting logic. We describe below the part of the signature of the module `META-LEVEL` that is relevant for our presentation (omitting the semantic 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`.

```
mod META-LEVEL[T :: Module] is
   sorts Qid Term TermList Label Nat Assignment Substitution ResultPair.
   subsorts Qid < Term < TermList .
            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 .
   vdots
endm
```

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 [14] for an extensive introduction to the subject.

6.3.2 Collection of Rewritings

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 "don't know nondeterministic" specifications. In Maude it becomes a module-transforming operation `ND-SEM` which maps a module T to another module `ND-SEM[`T`]` that extends the strategy kernel `META-LEVEL`, previously defined. Basically 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, a label l, and a natural number n and it evaluates to the sequence[15] 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.

```
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 .
```

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, execept 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.

```
  op last : Term Label Nat -> TermSequence .
```

[15] Here we discuss sequences with repetitions. If one is interested (for efficiency reasons, or whatever else) to *sequences without repetitions*, then the simple axiom `eq seq(`t`, `TL`, `t`) = seq(`t`, `TL`)` should be added for any term t and any term sequence TL.

```
  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
```

Now it is easy to 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 mechanism for the nondeterministic rewriting trees in T.

A strategy expression [19] 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(_)`** as described in Section 6.1.

```
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 .
  op rewWithBF : TermSequence TermSet Label -> StrategyExpression .
  op breadth : Label -> Strategy .
  vars TL : TermSequence .
       l : Label .

  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 .
```

We briefly comment on the breadth-first algorithm. 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. 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)` and we are done.

```
 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 .
```

The implementation of the strategy `depth(l)` 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 intead 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 rewriting are calculated. To improve the efficiency of the depth-first visit, we propose the following variant: 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.

```
  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
```

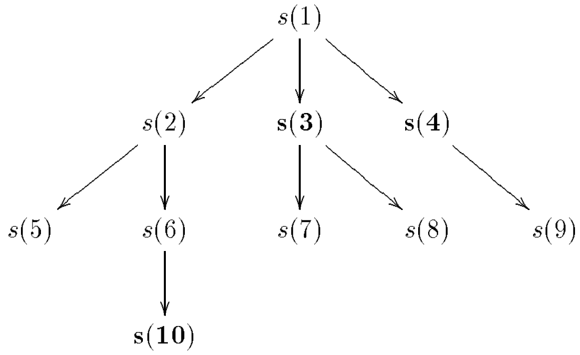
```
            then rewWith(t,idle)
            else (if meta-reduce('ok[t]) == 'false
                     then rewWithBT(PL,set(t,TS),l)
                     else (if meta-apply(t,l,n) == error*,empty
                              then rewWithBT(PL,set(t,TS),l)
                              else rewWithBT(seqPair(pair(
                                        extTerm(meta-apply(t,l,n)),0),
                                        pair(t,suc(n)),PL),set(t,TS),l)
                          fi)
                 fi)
        fi)
    fi .
```

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). Since we look for some control mechanism over nondeterministic computations, we could use a rewriting rule with label `aux` instead of an equation.

```
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 .
```

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 application 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 very simple notion of *meta-success* via a predicate `ok` at the meta-level. 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 iff 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 `rewInWithND(`LT`, `TS`, `l`)` where LT is a `TermList` such that all the terms in LT are successful (and LT is not empty).

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 $s(10)$.




```
mod EX is

  sort State .
  ops s : Nat -> State .
      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
```

Executing 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). Executing the meta-queries

```
Maude> rew rewWith('s['1], breadth('choice)) .
Maude> rew rewWith('s['1], depth('choice)) .
Maude> rew rewWith('s['1], nondet('choice)) .
```

leads respectively to the list of results

```
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 executions (in `TREE[TREE[EX]]`) of the meta-meta-queries

```
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])])
```

A better solution to the problem of collecting the "solutions" of the system can be given by analyizing the nature of the nondeterministic rule `aux`. It is possible to distinguish two cases depending on the selected term t. 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 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 a (conditional) equation and a (conditional) 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 recognise the sequences of solutions.

```
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 .
```

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])
```

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 $\mathtt{ND-SEM}[T]$ allows collecting and analyzing all the possible one-step rewritings of a term (modulo the equations of T);
- the module $\mathtt{TREE}[T]$ allows analyzing one solution among those reachable from a term, depending on the adopted strategy among the three proposed;
- the module $\mathtt{ND-SEM}[\mathtt{TREE}[T]]$ allows collecting and analyzing all the possible (subtree-topmost) solutions reachable from a term. Notice that each solution (if any) is reachable with only one step of rewriting. In Section 7 we will illustrate some application of this procedure to the executable implementation of two tile systems for CCS-like process calculi.

Furthermore, if we assume that we are interested only in solutions which differ from the "initial" term, then we could change the equations

```
eq rewWith(t,depth(l)) = rewWithDF(t,emptySet,l) .
eq rewWith(t,breadth(l)) = rewWithBF(t,emptySet,l) .
eq rewWith(t,nondet(l)) = rewWithND(t,emptySet,l) .
```

which are contained in the module $\mathtt{TREE}[T]$ by the following equations

```
eq rewWith(t,depth(l)) = rewWithDF(allRew(t,l),t,l) .
eq rewWith(t,breadth(l)) = rewWithBF(allRew(t,l),t,l) .
eq rewWith(t,nondet(l)) = rewWithND(allRew(t,l),t,l) .
```

Then, the module $\mathtt{TREE}[\mathtt{TREE}[T]]$ would allow us to collect and analyze all the possible solutions of the system. The result is an abstract view of the system T equipped with one-step nondeterministic rewritings from one solution to its reachable solutions.

6.4 Nondeterminism and Term Tile Systems

The general strategies that we have presented apply immediately to the translations of uniform 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 `VH` configurations as we will see in Section 6.6. In fact, a typical query in a tile system could be something like "derive all (some of) the tiles with a given horizontal source $\vec{h}$ and vertical target $\vec{u}$":

$$
\begin{tikzcd}
n \arrow[r, "\vec{h}"] & m \arrow[d, "\vec{u}"] \\
& n'
\end{tikzcd}
\quad \Rightarrow \quad
\begin{tikzcd}
n \arrow[d, "\vec{v_1}"'] & \\
m_1 \arrow[r, "\vec{g_1}"'] & n'
\end{tikzcd}
,
\begin{tikzcd}
n \arrow[d, "\vec{v_2}"'] & \\
m_2 \arrow[r, "\vec{g_2}"'] & n'
\end{tikzcd}
\quad \ldots \quad
\begin{tikzcd}
n \arrow[d, "\vec{v_i}"'] & \\
m_i \arrow[r, "\vec{g_i}"'] & n'
\end{tikzcd}
$$

But also more constrained queries are feasible like "derive all (some of) the tiles with a given vertical source $\vec{h}$ and horizontal target $\vec{u}$ which have the identity as horizontal source". In both cases, our strategies have been succesfully implemented and tested.

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 use of terms to define the vertical arrows. In this case this is the only correct procedure. However in some of the examples that we have considered when developing this approach, especially in the case of CCS-like process algebras [35], we realized that the vertical and horizontal dimension could be swapped in such a way that the intuitive queries are of the kind "derive all one-step transitions for a given agent P". This is possible because the vertical signature consists of *unary* actions. So we can reverse the vertical arrow in the tile rewrite system and then rotate clockwise by 90 degrees the tiles when implementing the system, as illustrated below for the tiles defining the action of a prefix component of the system:

$$
\begin{tikzcd}
1 \arrow[r, "\mu.\_"] \arrow[d, "id"'] \arrow[dr, phantom, "pr_\mu"] & 1 \arrow[d, "\mu"] \\
1 \arrow[r, "id"'] & 1
\end{tikzcd}
\Longrightarrow
\begin{tikzcd}
1 \arrow[r, "\mu.\_"] \arrow[dr, phantom, "pr_\mu"] & 1 \\
1 \arrow[r, "id"'] \arrow[u, "id"] & 1 \arrow[u, "\mu"']
\end{tikzcd}
\Longrightarrow
\begin{tikzcd}
1 \arrow[r, "id"] \arrow[d, "id"'] \arrow[dr, phantom, "pr_\mu"] & 1 \arrow[d, "\mu.\_"] \\
1 \arrow[r, "\mu"'] & 1
\end{tikzcd}
$$

If we examine the 2-cell translations, then the motivation for this kind of swapping of arrows is clear:

$$
\begin{tikzcd}
1 \arrow[r, bend left, "\mu(\mu.P)"] \arrow[r, bend right, "P"'] \arrow[r, phantom, "\Downarrow"] & 1
\end{tikzcd}
\Longrightarrow
\begin{tikzcd}
1 \arrow[r, bend left, "\mu.P"] \arrow[r, bend right, "\mu(P)"'] \arrow[r, phantom, "\Downarrow"] & 1
\end{tikzcd}
$$

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 process, whereas an implementation based on the second kind of rules may generate all the possible evolutions of the system.

6.5 Non Uniform Case

If the tTRS 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 which led to that state. This means that the structure of the meta-state would become very large very fast during the execution, and that the computations would be affected by becoming very slow. Since at present we don't have any meaningful examples of non-uniform systems, we are not really interested in having such an implementation.

6.6 Uniform Case

In this section we show how it is possible to make use of the membership assertions to directly model uniform cartesian theories.

Let $\mathcal{R} = \langle \Sigma_H, \Sigma_V, N, R\rangle$ be a generic tTRS, where Σ_H and Σ_V are two (one-sorted) disjoint signatures and $R(N)$ is a set of rules having the form

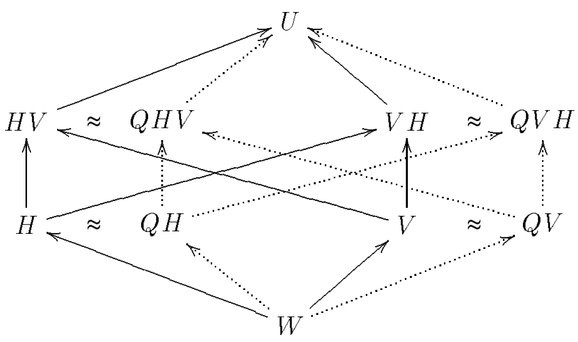


Figure 6: The poset of sorts for $\widehat{\mathcal{R}}$.

$$\begin{tikzcd} n \arrow[r, "\vec{h}"] \arrow[d, "\vec{v}"'] \arrow[dr, phantom, "r"] & m \arrow[d, "u"] \\ k \arrow[r, "g"'] & 1 \end{tikzcd}$$

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 cartesian 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 5.4) is uniform.

Then, we define a rewrite theory $\widehat{\mathcal{R}}$, where the poset of its sorts is the one illustrated in Fig. 6. 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 the signature Σ_H (similarly for `VH`); the sorts `QH`, `QV`, `QHV`, and `QVH` are *quoted versions* of the corresponding sorts described before (we will denote the quoted version of a signature Σ_S by $\Sigma_{S'}$, adopting the convention that all the operators of the latter are *syntactically* quoted version 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 X1 ... Xmax : U .

cmb h(X1,...,Xn) : H iff X1 ... Xn : H .
cmb v(X1,...,Xm) : V iff X1 ... Xn : V .
cmb h(X1,...,Xn) : VH iff X1 ... Xn : VH .
cmb v(X1,...,Xm) : HV iff X1 ... Xn : HV .

cmb qh(X1,...,Xn) : QH iff X1 ... Xn : QH .
cmb qv(X1,...,Xm) : QV iff X1 ... Xn : QV .
cmb qh(X1,...,Xn) : QVH iff X1 ... Xn : QVH .
cmb qv(X1,...,Xm) : QHV iff X1 ... Xn : 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(X1) : QH iff X1 : H .
cmb quote(X1) : QV iff X1 : V .
cmb quote(X1) : QHV iff X1 : HV .
cmb quote(X1) : QVH iff X1 : VH .
cmb quote(X1) : W iff X1 : W .

eq quote(h(X1,...,Xn)) = qh(quote(X1),...,quote(Xn)) .
eq quote(v(X1,...,Xm)) = qv(quote(X1),...,quote(Xm)) .
ceq quote(X1) = X1 if X1 : W .

cmb unquote(X1) : H iff X1 : QH .
cmb unquote(X1) : V iff X1 : QV .
cmb unquote(X1) : HV iff X1 : QHV .
cmb unquote(X1) : VH iff X1 : QVH .
cmb unquote(X1) : W iff X1 : W .

eq unquote(qh(X1,...,Xn)) = h(unquote(X1),...,unquote(Xn)) .
eq unquote(qv(X1,...,Xm)) = v(unquote(X1),...,unquote(Xm)) .
ceq unquote(X1) = X1 if X1 : 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 following result may be easily proved via a simple inspection of the rules in $\hat{\mathcal{R}}$.

Theorem 6.1 *Given a uniform cartesian tile rewrite system $\mathcal{R}$ then*

$$\mathcal{R} \vdash_{ft} \vec{h} \xrightarrow[u]{\vec{v}} g \Longleftrightarrow \hat{\mathcal{R}} \vdash \mathtt{top(u(}\vec{\mathtt{h}}\mathtt{))} \quad \Rightarrow \quad \mathtt{top(g(}\vec{\mathtt{v}}\mathtt{))}.$$

In Section 7 this translation is applied to the tTRS for *finite CCS*, and an example of execution is illustrated in detail.

7 Maude as a Semantic Framework

In this section we will show how Maude – thanks to its reflective capabilities and, in particular, thanks to the possibility of defining internal strategy languages – can in fact be used to prototype and execute tile rewriting systems.

7.1 Finite CCS

Milner's *Calculus for Communicating Systems* (*CCS*) [59] is among the best 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 [47, 70]. We present here an executable implementation of CCS arising from the translation in term tile logic of the tile rewriting system given in [35] for *finite* CCS.

Definition 7.1 [Calculus of Communicating Systems] Let Δ (ranged over by α) be the set of *basic actions*, and let $\bar{\Delta}$ be 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 *finite CCS process* is any term generated by the following grammar:

$$P ::= nil \ \mid\ \mu.P \ \mid\ P\backslash\alpha \ \mid\ P[\alpha/\beta] \ \mid\ P+P \ \mid\ P|P.$$

We let P, Q, R, ...range over the set $Proc$ of CCS processes. □

Assuming the reader reasonably familiar with the notation, we give an informal description of CCS algebra operators: the constant nil yields the *inactive* process (i.e., it cannot perform any action); the process $\mu.P$ is a process behaving like P, but only after the execution of the communication μ ($\mu.\_$ is called a *prefix* operator); the process $P\backslash\alpha$ is the process P with the α and $\bar{\alpha}$ actions blocked by the *restriction* operator $\_\backslash\alpha$; the process $P[\alpha/\beta]$ behaves like P with actions α and $\bar{\alpha}$ relabelled by β and $\bar{\beta}$; the process $P+Q$ is the *nondeterministic (guarded) sum* of processes P and Q; finally, the process $P|Q$ is the parallel composition of processes P and Q. Notice that the only difference w.r.t. the traditional CCS operators is given by the relabelling, for which we adopt a finitary approach, thus allowing a much simpler representation in the Maude language. However, the non finitary case could also be handled as well in our framework.

Example 7.2 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 relabellings in the process algebra of Def. 7.1. However, since a finite process can only perform finitely many actions, then for each finite CCS process P, it is possible to "simulate" $P[\Phi]$ in the process algebra of Def. 7.1. □

Given a process P, its dynamic behaviour is usually described by a transition system, presented in the SOS style, where the transition relation is freely generated from a set of inference rules. The rules for CCS are usually presented via inference *schemes* that are parametric w.r.t. either the action performed, or the operator involved, or the underlying processes (i.e., there are three families of operators indexed by actions, namely $\{\mu.\_ \mid \mu \in Act\}$, $\{\_\backslash\alpha \mid \alpha \in \Lambda\}$, and $\{\_[\alpha/\beta] \mid \alpha, \beta \in \Lambda\}$).

Definition 7.3 [Operational Semantics of CCS] The *CCS transition system* is the relation $T \subseteq Proc \times Act \times Proc$ inductively generated from the following set of axioms and inference rules

$$\frac{}{\mu.P \xrightarrow{\mu} P} \qquad \frac{P \xrightarrow{\mu} Q}{P\backslash\alpha \xrightarrow{\mu} Q\backslash\alpha}\ \mu \notin \{\alpha, \bar{\alpha}\} \qquad \frac{P \xrightarrow{\mu} Q}{P[\alpha/\beta] \xrightarrow{\mu[\alpha/\beta]} Q[\alpha/\beta]}$$

$$\frac{P \xrightarrow{\mu} Q}{P+R \xrightarrow{\mu} Q} \qquad \frac{P \xrightarrow{\mu} Q}{R+P \xrightarrow{\mu} Q}$$

$$\frac{P \xrightarrow{\mu} Q}{P|R \xrightarrow{\mu} Q|R} \qquad \frac{P \xrightarrow{\alpha} Q,\ P' \xrightarrow{\bar{\alpha}} Q'}{P|P' \xrightarrow{\tau} Q|Q'} \qquad \frac{P \xrightarrow{\mu} Q}{R|P \xrightarrow{\mu} R|Q}$$

where $P \xrightarrow{\mu} Q$ means that $(P, \mu, Q) \in T$, and the action $\mu[\alpha/\beta]$ is defined as follows:

$$\mu[\alpha/\beta] = \begin{cases} \beta & \text{if } \mu = \alpha \\ \bar{\beta} & \text{if } \mu = \bar{\alpha} \\ \mu & \text{otherwise} \end{cases} .$$

□

The operational meaning is that a CCS process P may perform an action μ becoming the process Q iff it is possible to inductively construct a sequence of rule applications to conclude that $P \xrightarrow{\mu} Q$. More generally, a CCS process P_0 may evolve to process P_n iff there exists a *computation* $P_0 \xrightarrow{\mu_1} P_1 \ldots P_{n-1} \xrightarrow{\mu_n} P_n$. In [35] it is shown how to associate an algebraic (tile) rewriting system to CCS. We adapt Gadducci and Montanari's definition to settle the following tTRS for CCS.

Definition 7.4 [Term Tile System for Finite CCS] The tTRS associated to CCS is the tuple $\mathcal{R}_{CCS} = \langle \Sigma_A, \Sigma_P, N, R \rangle$, where $\Sigma_A = \{\mu : 1 \longrightarrow 1 \mid \mu \in Act\}$, Σ_P is the signature of CCS processes, and the rules are the following:

$$act_\mu : 1 \lhd \langle x_1 \rangle \xrightarrow[\langle \mu.x_1 \rangle]{\langle x_1 \rangle} \langle \mu(x_1) \rangle \qquad res_{\mu,\alpha} : 1 \lhd \langle \mu(x_1) \rangle \xrightarrow[\langle x_1 \backslash \alpha \rangle]{\langle x_1 \backslash \alpha \rangle} \langle \mu(x_1) \rangle \ \ (\text{if } \mu \notin \{\alpha, \bar{\alpha}\})$$

$$rel_{\mu,\alpha,\beta} : 1 \lhd \langle \mu(x_1) \rangle \xrightarrow[\langle x_1[\alpha/\beta] \rangle]{\langle x_1[\alpha/\beta] \rangle} \langle t \rangle \ \text{ with } t = \begin{cases} \beta(x_1) & \text{if } \mu = \alpha \\ \bar{\beta}(x_1) & \text{if } \mu = \bar{\alpha} \\ \mu(x_1) & \text{otherwise} \end{cases}$$

$$\langle +_\mu : 2 \lhd \langle \mu(x_1), x_2 \rangle \xrightarrow[\langle x_1 + x_2 \rangle]{\langle x_1 \rangle} \langle \mu(x_1) \rangle \qquad +\rangle_\mu : 2 \lhd \langle x_1, \mu(x_2) \rangle \xrightarrow[\langle x_1 + x_2 \rangle]{\langle x_2 \rangle} \langle \mu(x_1) \rangle$$

$$\rfloor_\mu : 2 \lhd \langle \mu(x_1), x_2 \rangle \xrightarrow[\langle x_1 | x_2 \rangle]{\langle x_1 | x_2 \rangle} \langle \mu(x_1) \rangle \qquad \lfloor_\mu : 2 \lhd \langle x_1, \mu(x_2) \rangle \xrightarrow[\langle x_1 | x_2 \rangle]{\langle x_1 | x_2 \rangle} \langle \mu(x_1) \rangle$$

$$||_\lambda : 2 \lhd \langle \lambda(x_1), \bar{\lambda}(x_2) \rangle \xrightarrow[\langle x_1 | x_2 \rangle]{\langle x_1 | x_2 \rangle} \langle \tau(x_1) \rangle$$

□

Here, the vertical dimension is associated to process descriptions, whereas the horizontal dimension represents the (*opposite* of the) dynamic evolution of the system[16]. For the reader already acquainted with the tile system of Gadducci and Montanari, the previous definition may appear somewhat odd, because the two dimensions are reversed in a counterintuitive way. The reason is that the direct translation of our system in a Maude module allows collecting the possible evolutions of a process, whereas the ordinary definition would allow only testing executable actions. The following example should help making our motivation more clear.

[16] We say *opposite*, because the direction of the arrows representing the actions performed by the system is the opposite from that of their computationally intuitive direction.

Example 7.5 Let us consider the simple process $\alpha.nil.$ and let us suppose that we have an executable module both for our system and for the tile system defined in [35, 58]. In our case the query is uniquely determined, and the system can give only one answer:

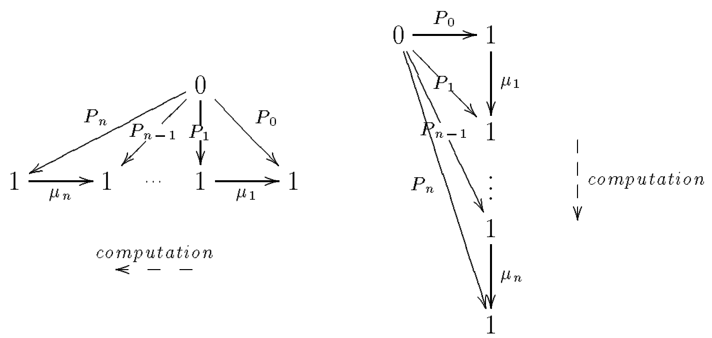
$$\begin{tikzcd} 0 \arrow[r, "id_0"] & 0 \arrow[d, "nil"] \\ & 1 \arrow[d, "\alpha.x_1"] \\ & 1 \end{tikzcd} \Rightarrow \begin{tikzcd} 0 \arrow[d, "nil"'] & \\ 1 \arrow[r, "\alpha(x_1)"'] & 1 \end{tikzcd}$$

which corresponds to the proof sequent $1_{nil} \cdot act_\alpha$. In the other case, there are as many possible queries as the actions in Act, but only one of them enables the rewriting.

$$\begin{tikzcd} 0 \arrow[r, "nil"] & 1 \arrow[r, "\alpha.x_1"] & 1 \arrow[d, "\beta(x_1)"] \\ & & 1 \end{tikzcd} \Rightarrow \begin{tikzcd} 0 \arrow[d, "id_0"'] & \\ 0 \arrow[r, "nil"'] & 1 \end{tikzcd} \qquad \text{iff} \;\; \alpha = \beta$$

□

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 $P_0 \xrightarrow{\mu_1} P_1 \ldots P_{n-1} \xrightarrow{\mu_n} P_n$:



where dashed arrows represent the direction of computational evolutions of processes.

Analogously to [35], 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 7.6 *For any CCS agents P and Q, and action μ:*

$$P \xrightarrow{\mu} Q \in T \iff \mathcal{R}_{CCS} \vdash_{ft} 0 \triangleleft \langle\rangle \xrightarrow[\langle Q \rangle]{\langle P \rangle} \langle \mu(x_1) \rangle \ .$$

Moreover, the tTRS $\mathcal{R}_{CCS}$ guarantees the following result:

Proposition 7.7 *The tTRS $\mathcal{R}_{CCS}$ is uniform.*

From Proposition 7.7 it follows immediately that a suitable implementation of $\mathcal{R}_{CCS}$ can be obtained by taking the rewriting system $\hat{\mathcal{R}}_{CCS}$ defined in Section 6.6, 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 6.3 for collecting the correct rewritings can be directly 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(A)) = A .

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, 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(Q,P),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 6.6, but we use a more verbose syntax for the operators of the tTRS. In particular, we assume that:

- 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).

Notice that the sort `W` is necessary for executing partially specified queries (in this case the process variable that are used must be declared as constants having sort `W`).

In order to apply the internal strategies for collecting the behaviours of a process, the definition of the function `allRew` in module `ND-SEM` must be slightly changed. The reason is that the function `meta-apply` apply the rewriting only on top of the term, whereas the rules for CCS can occur in a proper subterm. Therefore, we define a new `allRew` that evaluates all the possible rewritings of the term and also of every proper subterm, by a recursive exploration of each argument, accomplished by the auxuliary 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 function `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 .

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 .
```

Example 7.8 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 suffices to make easier the reading.

```
Maude> rew allRew('rewWith[('_[_][''top,
('_[_]['par, ('_,_[
              ('_[_]['plus, ('_,_[
                              ('_[_]['pre, ('_,_[('_[_]['a, ''1]), ''nil])]),
                              ('_[_]['pre, ('_,_[('_[_]['a, ''2]), ''nil])])
                             ])]),
              ('_[_]['pre, ('_,_[('_[_]['bar, ('_[_]['a, ''1])]), ''nil])])
              ])])]),
('convnondet[''qr])], 'caux) .

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). □

7.2 Concurrent and 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 implementation 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 [8, 26]. 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 [28], 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, term graphs are used to model the structure of configurations and effects of the resulting (flat) tile systems. As discussed in Section 2.1.1, the term graph structure is essentially a weak cartesian category where two naturality axioms are missing. From another perspective, a weak cartesian category is just a particular symmetric strict monoidal category. Thus, 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 easily formulated in term tile logic and hence 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 7.9 [Simple Process Calculus] Let Δ, Λ, and Act be as in Def. 7.1. 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:

$$G ::= nil \ \mid\ \mu.G \ \mid\ G+G \ \mid\ G|G$$

$$P ::= G \ \mid\ l :: P \ \mid\ P|P$$

where, for the sake of simplicity, we distinguish the *ground processes* (i.e., processes without locations), ranged over by G, G', and so on. □

Locations [9] 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 7.10 [Operational Semantics] The *transition algebra* TA of our simple process calculus is defined by the following axioms and inference rules:

$$\frac{}{[\mu, l, G\rangle : \mu.G \xrightarrow[l]{\mu} l :: G} \qquad \frac{t : P \xrightarrow[k]{\mu} P',\ l \notin loc(k)}{l :: t : l :: P \xrightarrow[lk]{\mu} l :: P'}$$

$$\frac{t : G_1 \xrightarrow[k]{\mu} G_2}{t\langle +G : G_1 + G \xrightarrow[k]{\mu} G_2} \qquad \frac{t : G_1 \xrightarrow[k]{\mu} G_2}{G+\rangle t : G + G_1 \xrightarrow[k]{\mu} G_2}$$

$$\frac{t : P_1 \xrightarrow[k]{\mu} P_2,\ loc(k) \cap loc(P) = \emptyset}{t\lfloor P : P_1|P \xrightarrow[k]{\mu} P_2|P} \qquad \frac{t : P_1 \xrightarrow[k]{\mu} P_2,\ loc(k) \cap loc(P) = \emptyset}{P\rfloor t : P|P_1 \xrightarrow[k]{\mu} P|P_2}$$

$$\frac{t_1 : P_1 \xrightarrow[u_1]{\lambda} P_1',\ t_2 : P_2 \xrightarrow[u_2]{\bar{\lambda}} P_2',\ loc(u_1) \cap loc(P_2') = \emptyset = loc(u_2) \cap loc(P_1')}{t_1 \| t_2 : P_1|P_2 \xrightarrow[u_1,u_2]{\tau} P_1'|P_2'}$$

□

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 [26].

Definition 7.11 [Concurrency Relation] Let $(\_\ \text{then}\ \_\ \chi\ \_\ \text{then}\ \_)$ be a quaternary relation on transition proof terms, defined as the least commutative[17] relation defined by the following structural rules:

$$\frac{}{(t_1\lfloor P_2\ \text{then}\ Q_1\rfloor t_2\ \chi\ P_1\rfloor t_2\ \text{then}\ t_1\lfloor Q_2)} \qquad \frac{(t_1\ \text{then}\ t_2\ \chi\ t_3\ \text{then}\ t_4)}{(l :: t_1\ \text{then}\ l :: t_2\ \chi\ l :: t_3\ \text{then}\ l :: t_4)}$$

$$\frac{(t_1\ \text{then}\ t_2\ \chi\ t_3\ \text{then}\ t_4)}{(t_1\langle +G\ \text{then}\ t_2\ \chi\ t_3\langle +G\ \text{then}\ t_4)} \qquad \frac{(t_1\ \text{then}\ t_2\ \chi\ t_3\ \text{then}\ t_4)}{(G+\rangle t_1\ \text{then}\ t_2\ \chi\ G+\rangle t_3\ \text{then}\ t_4)}$$

$$\frac{(t_1\ \text{then}\ t_2\ \chi\ t_3\ \text{then}\ t_4)}{(t_1\lfloor P\ \text{then}\ t_2\lfloor P\ \chi\ t_3\lfloor P\ \text{then}\ t_4\lfloor P)} \qquad \frac{(t_1\ \text{then}\ t_2\ \chi\ t_3\ \text{then}\ t_4)}{(P\rfloor t_1\ \text{then}\ P\rfloor t_2\ \chi\ P\rfloor t_3\ \text{then}\ P\rfloor t_4)}$$

$$\frac{(t_1\ \text{then}\ t_2\ \chi\ t_3\ \text{then}\ t_4)}{(t_1\|t\ \text{then}\ t_2\lfloor Q\ \chi\ t_3\lfloor P\ \text{then}\ t_4\|t)} \qquad \frac{(t_1\ \text{then}\ t_2\ \chi\ t_3\ \text{then}\ t_4)}{(t\|t_1\ \text{then}\ Q\rfloor t_2\ \chi\ P\rfloor t_3\ \text{then}\ t\|t_4)}$$

$$\frac{(t_1\ \text{then}\ t_2\ \chi\ t_3\ \text{then}\ t_4),\ (t_1'\ \text{then}\ t_2'\ \chi\ t_3'\ \text{then}\ t_4')}{(t_1\|t_1'\ \text{then}\ t_2\|t_2'\ \chi\ t_3\|t_3'\ \text{then}\ t_4\|t_4')}$$

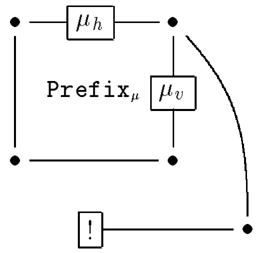
where $t_i : P_i \xrightarrow[k_i]{\mu_i} Q_i$, $t_i' : P_i' \xrightarrow[k_i']{\mu_i'} Q_i'$, and $t : P \xrightarrow[k]{\mu} Q$, □

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 their paper [28], Ferrari and Montanari propose a tile rewrite 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

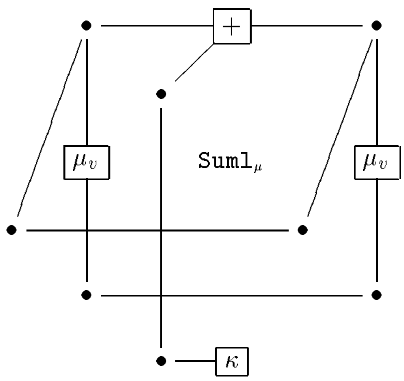
[17] Namely, $(t_1\ \text{then}\ t_2\ \chi\ t_3\ \text{then}\ t_4)$ iff $(t_3\ \text{then}\ t_4\ \chi\ t_1\ \text{then}\ t_2)$.

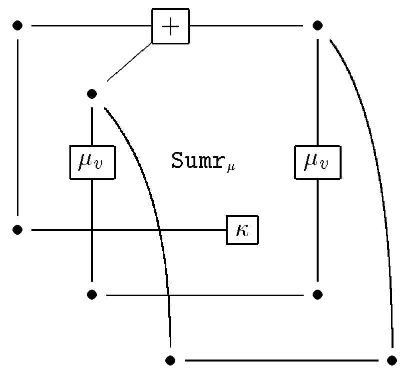
give a graphical representation of their tile system, using *hypergraphs* to model configurations and effects[18]. The labels of horizontal hyperarcs are taken over the signature $\Sigma_S = \{+ : 2 \longrightarrow 1, ! : 0 \longrightarrow 1, \kappa : 1 \longrightarrow 0\} \bigcup_{\mu \in Act} \{\mu_h : 1 \longrightarrow 1\}$, and vertical hyperarcs are labelled over the signature $\Sigma_D = \{T : 2 \longrightarrow 2\} \bigcup_{\mu \in Act} \{\mu_v : 1 \longrightarrow 1\}$. 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[19] of the *strong tile rewrite system* for the simple process calculus of Def. 7.9. The first rule 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. 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.

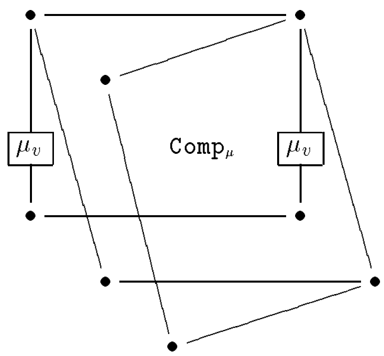


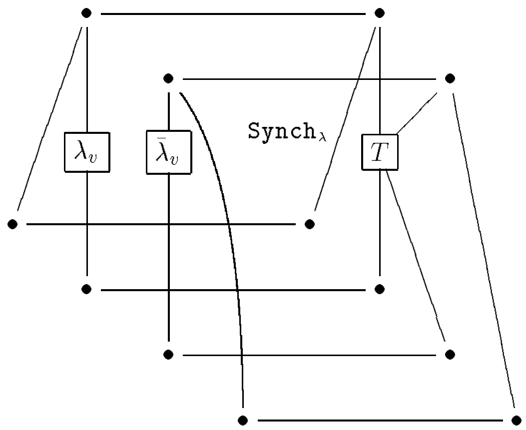


Rule $\texttt{Comp}_\mu$ is the most important. It states that if two processes share the same location and one of them is making a move, then its subprocesses will be allocated to a new location, whereas the parallel process will remain linked to the old location. We do not need to distinguish between a $\texttt{Compl}_\mu$ (left) and a $\texttt{Compr}_\mu$ (right), because they can be obtained one from the other using auxiliary permutation tiles. 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.

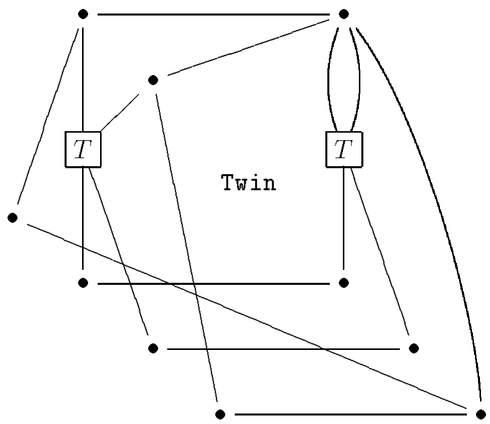
[18] Each *hyperarc* is represented with a labelled box connected to its source and target nodes. Unlabelled arc represents *sharing* (or *aliasing*) in the style of assignments (see Section 2).

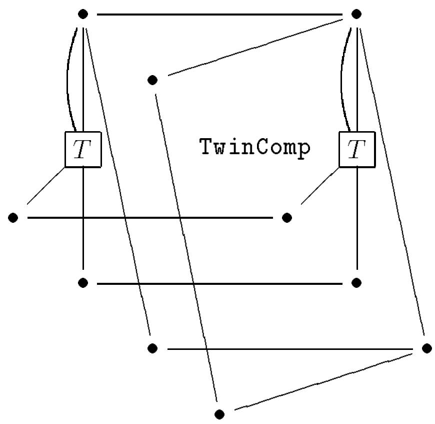
[19] The only auxiliary tiles needed here are permutation tiles.



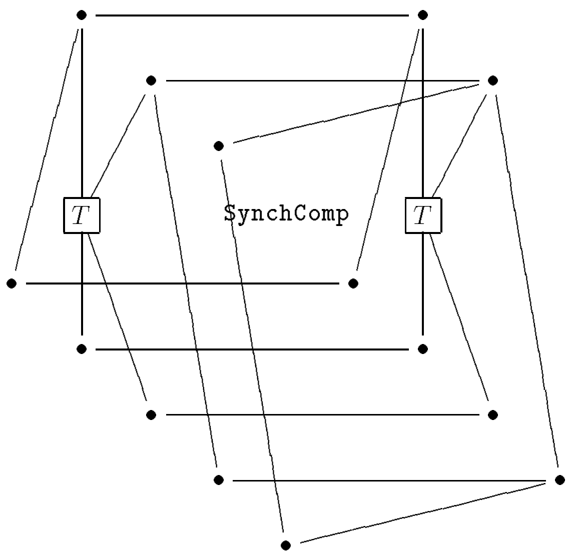


Then two particular cases have to be taken into account. The following rules show how the T events propagate through the sharing. The **Twin** rules state that if the two synchronizing processes were allocated in the same place, then the synchronization is possible, but only one copy of that "old" location has to be mantained.





Notice that a rule such as SynchComp is not needed here because the correct way to synchronize the processes yielding T would consist in using Comp$_\lambda$ and then 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 effects 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 $\{\mathtt{n}(i) \mid i \in \mathbb{N}\}$ (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 some useful operators which allow extracting information of various kinds.

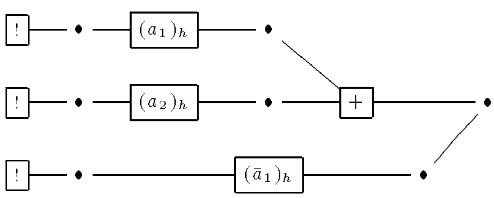
```
*** proj(NL) returns the set of nodes contained in the node list NL
op proj : NodeList -> NodeSet .
*** sources(MS) returns the set of source nodes contained in the hypergraph MS
op sources : EdgeMSet -> NodeSet .
*** targets(MS) returns the set of target nodes contained in the hypergraph MS
op targets : EdgeMSet -> NodeSet .
*** vertices(MS) returns the set of nodes contained in the hypergraph MS
op vertices : EdgeMSet -> NodeSet .
*** 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 $\{\mathsf{a}(i) \mid i \in \mathbb{N}\}$ and their complementary actions $\{\mathsf{bar}(\mathsf{a}(i)) \mid i \in \mathbb{N}\}$ (sort `Channel`), 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 $\mathtt{h}(\mu)$ and $\mathtt{v}(\mu)$. 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 $\mathtt{Suml}_\mu$ and $\mathtt{Sumr}_\mu$.

```
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 7.12 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 6.3, 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 just 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 label 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') .


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''))) .

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 ok(top(MS)) = true
    if and(okHV(MS),acyclic(MS)) .
endm
```

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 Ferrari and Montanari.

Example 7.13 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 7.12. 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) .

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 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.

The graphical representation of the last solution is given in Figure 7.

Six possible combinations have been found, only the synchronization between a_1 and $\bar{a}_1$ (represented in Figure 8) is missing. The reason is that it can only be reached after having visited a successful state, i.e., it is not a topmost solution. However the following query from one of the previous solutions (the last one) leads to the detection of the synchronization (notice the hyperarc with label `t` in the answer).

```
Maude> rew allRew('rewWith[('_[_]['top, ('_[_]['ms, ('_,_[
           ('_[_]['edge, ('_,_[''nil, ''dis, ('_[_]['n, ''1])])]),
```

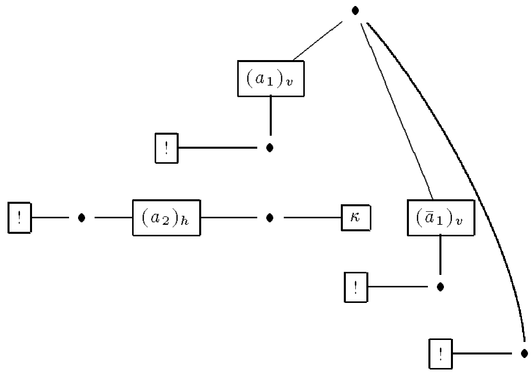


Figure 7: The graphical representation of the parallel execution of the actions a_1 and $\bar{a}_1$ in the process $(a_1.nil + a_2.nil)|\bar{a}_1.nil$.

```
      ('_[_]['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])])])
      ])])]), ('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 7.8), here the number of solutions is smaller than in the other case, because concurrent (interleaving) computations are identified. □

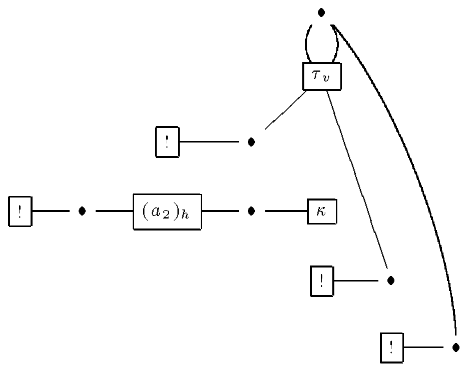


Figure 8: The graphical representation of the synchronization of the actions a_1 and $\bar{a}_1$ in the process $(a_1.nil + a_2.nil)|\bar{a}_1.nil$.

Acknowledgements

We would like to thank Narciso Martí-Oliet for his careful reading of a preliminary version of this report, and his suggestions and comments that have been very precious to improve the exposition in the final version.

References

[1] A. Bastiani and C. Ehresmann. *Multiple Functors I: Limits Relative to Double Categories*, Cahiers de Topologie et Géométrie Différentielle **15** (3), 1974, pp. 545-621.

[2] J.A. Bergstra and J.W. Klop. Process Algebra for Synchronous Comunication. *Information and Computation* 60. 1984. pp. 109–137.

[3] J.A. Bergstra and J.W. Klop. Algebra of Communicating Processes with Abstraction. *Theoretical Computer Science* 37, vol 1. 1985. pp. 77–121.

[4] K.L. Bernstein. *A Congruence Theorem for Structured Operational Semantics of Higher-Order Languages.* Technical Report TR 97-005. School of Computer Science, Telecommunications, and Information Systems. DePaul University. 1997.

[5] B. Bloom, S. Istrail, and A.R. Meyer. Bisimulation can't be Traced. *Journal of the ACM* 42(1):232–268. January 1995.

[6] P. Borovanský, C. Kirchner, and H. Kirchner. Controlling Rewriting by Rewriting In: J. Meseguer, Ed., *Proc. First International Workshop on Rewriting Logic and its Applications.* ENTCS 4. 1996.

[7] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. ELAN: A logical framework based on computational systems. In: J. Meseguer, Ed., *Proc. First International Workshop on Rewriting Logic and its Applications.* ENTCS 4. 1996.

[8] G. Boudol and I. Castellani. Flow Models of Distributed Computations: Three Equivalent Semantics for CCS. *Information and Computation* 114, vol. 2. 1993. pp. 247–314.

[9] G. Boudol, I. Castellani, M. Hennessy, and A. Kiehn. Observing Localities. *Theoretical Computer Science* 114. 1993. pp. 31–61.

[10] 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.* Springer LNCS 1214, 1997. pp. 67–92.

[11] R. Bruni and U. Montanari. Zero-Safe Nets, or Transition Synchronization Made Simple. In: C. Palamidessi, J. Parrow, Eds. *Proceedings of EXPRESS'97.* ENTCS 7. 1997.

[12] R. Bruni and U. Montanari, Zero-Safe Nets: The Individual Token Approach. In: F. Parisi-Presicce, Ed., *Proc. 12th WADT Workshop on Algebraic Development Techniques.* Springer LNCS 1376. 1998. pp.122–140.

[13] M. Clavel. *Reflection in General Logics and in Rewriting Logic with Applications to the Maude Language.* PhD Thesis. Universidad de Navarra, 1998.

[14] M.G. Clavel, F. Duran, S. Eker, P. Lincoln, and J. Meseguer. *An Introduction to Maude (Beta Version).* SRI International, March 1998.

[15] M.G. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In: J. Meseguer, Ed., *Proc. First International Workshop on Rewriting Logic and its Applications.* ENTCS 4. 1996.

[16] M.G. Clavel, F. Durán, S. Eker, and J. Meseguer. *Current Design and Implementation of the Cafe Prover and Knuth-Bendix Checker Tools.* Manuscript, SRI International, October, 1997.

[17] M. Clavel, and J. Meseguer. Reflection and Strategies in Rewriting Logic. In: J. Meseguer, Ed., *Proc. First International Workshop on Rewriting Logic and its Applications.* ENTCS 4. 1996.

[18] M. Clavel, and J. Meseguer. Axiomatizing Reflective Logics and Languages. In: G. Kiczales, Ed., *Proc. Reflection'96.* San Francisco, USA. April 1996. pp. 263–288.

[19] M. Clavel, and J. Meseguer. Internal Strategies in a Reflective Logic. In: B. Gramlich, and H. Kirchner, Eds., *Proc. of the CADE-14 Workshop on Strategies in Automated Deduction*, Townsville, Australia. 1997. pp 1–12.

[20] A. Corradini and F. Gadducci. A 2-categorical Presentation of Term Graph Rewriting. In: E. Moggi, G. Rosolini, Eds., *Proc. CTCS'97.* Springer LNCS 1290. 1997. pp. 87–105.

[21] A. Corradini, F. Gadducci, and U. Montanari. Relating Two Categorical Models of Concurrency. In: *Proc. RTA'95.* Springer LNCS 914. 1995. pp. 225–240.

[22] A. Corradini and U. Montanari. An Algebraic Semantics for Structured Transition Systems and its Application to Logic Programs, *Theoretical Computer Science* 103. 1992. pp. 51–106.

[23] P. Degano, J. Meseguer, and U. Montanari. Axiomatizing the Algebra of Net Computations and Processes. *Acta Informatica* 33(7):641–667. October 1996.

[24] M.C. Eekele, M.J. Plasmeijer, and M.R. Sleep, Eds., *Term Graph Rewriting: Theory and Practice.* Wiley, London. 1993.

[25] C. Ehresmann. Catégories Structurées: *I and II, Ann. Éc. Norm. Sup.* 80, Paris (1963), 349–426; *III, Topo. et Géo. Diff.* V, Paris (1963).

[26] G.L. Ferrari and U. Montanari. Towards the Unification of Models for Concurrency. In *Proceeding CAAP'90.* Springer LNCS 431. 1990, pp. 162–176.

[27] G.L. Ferrari and U. Montanari. A Tile-Based Coordination View of Asynchronous Pi-Calculus. In: I. Prívara, P. Ruzicka, Eds., *Mathematical Foundations of Computer Science 1997.* Springer LNCS 1295, 1997, pp. 52–70.

[28] G.L. Ferrari and U. Montanari. Tiles for Concurrent and Located Calculi. In: C. Palamidessi, J. Parrow, Eds., *Proceedings of EXPRESS'97.* ENTCS Vol.7, 1997.

[29] P. Freyd. Algebra valued functors in general and tensor products in particular. *Coll. Math.*, 14:89–106, 1966.

[30] K. Futatsugi and T. Sawada. Cafe as an extensible specification environment. In *Proc. of the Kunming International CASE Symposium, Kunming, China, November*, 1994.

[31] P. Gabriel and F. Ulmer. Lokal präsentierbare Kategorien. *Springer Lecture Notes in Mathematics* 221. 1971.

[32] F. Gadducci. *On the Algebraic Approach to Concurrent Term Rewriting.* PhD Thesis TD-96-02. Department of Computer Science, University of Pisa, 1996.

[33] F. Gadducci and U. Montanari. Enriched Categories as Models of Computations. In: *Proc. 5th Italian Conference on Theoretical Computer Science, ITCS'95.* World Scientific. 1996. pp. 1–24.

[34] F. Gadducci and U. Montanari. Rewriting Rules and CCS. In: J. Meseguer, Ed., *Proc. First International Workshop on Rewriting Logic and its Applications.* ENTCS 4. 1996.

[35] F. Gadducci and U. Montanari. The Tile Model. In: G. Plotkin, C. Stirling, 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.

[36] J.A. Goguen. A Categorical Manifesto. *Mathematical Structures in Computer Science* 1. 1991. pp. 49–67.

[37] J.F. Groote and F. Vaandrager. Structured Operational Semantics and Bisimulation as a Congruence. *Information and Computation* 100. 1992. pp. 202–260.

[38] S. Haridi, S. Janson, J. Montelius, T. Franzén, P. Brand, K. Boortz, B. Danielsson, B. Carlson, T. Keisu, D. Sahlin, and T. Sjöland. Concurrent Constraint Programming at SICS with the Andorra Kernel Language (Extended Abstract). *PPCP*. 1993. pp. 107–116.

[39] C.A.R. Hoare. *Communication Sequential Processes*. Prentice-Hall. 1985.

[40] A. Kock and G.E. Reyes. Doctrines in Categorical Logic. In: John Bairwise, Ed., *Handbook of Mathematical Logic*. North Holland. 1977. pp. 283–313.

[41] G.M. Kelly and R.H. Street. Review of the Elements of 2-categories. *Lecture Notes in Mathematics 420*. 1974. pp 75–103.

[42] C. Lair. Etude Générale de la Catégorie des esquisses. *Esquisses Math.* 24. 1974.

[43] K.G. Larsen and L. Xinxin. Compositionality Through an Operational Semantics of Contexts. In *Proc. ICALP'90*. LNCS 443, 1990. pp. 526–539.

[44] F.W. Lawvere. Functorial Semantics of Algebraic Theories. In *Proc. National Academy of Science* 50. 1963. pp. 869–872.

[45] F.W. Lawvere. Some algebraic problems in the context of functorial semantics of algebraic theories. In *Proc. 2th Midwest Category Seminar*. Springer Lecture Notes in Mathematics 61. 1968. pp. 41–61

[46] S. MacLane. *Categories for the Working Mathematician*. Springer-Verlag, 1971.

[47] 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*, Kluwer Academic Publishers.

[48] N. Martí-Oliet and J. Meseguer. General Logics and Logical Frameworks. In: D. Gabbay, Ed., *What is a logical system?*. Oxford University Press. 1994. pp. 355–392.

[49] N. Martí-Oliet and J. Meseguer. Rewriting Logic as a Logical and Semantic Framework. In: J. Meseguer, Ed., *Proc. First International Workshop on Rewriting Logic and its Applications*. ENTCS 4. 1996.

[50] 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*.

[51] J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science* 96, 1992. pp. 73–155.

[52] J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In: Gul Agha, Peter Wegner, and Akinori Yonezawa, Eds., *Research Directions in Concurrent Object-Oriented Programming*. MIT Press. 1993. pp. 314–390.

[53] J. Meseguer. Rewriting Logic as a Semantic Framework for Concurrency: A Progress Report, In: U. Montanari, V. Sassone, Eds., *CONCUR'96: Concurrency Theory*, Springer LNCS 1119, 1996, pp. 331-372.

[54] J. Meseguer. *Membership algebra*. Lecture and abstract at the Dagstuhl Seminar on "Specification and Semantics". July 9, 1996.

[55] J. Meseguer, Ed. *Proc. First International Workshop on Rewriting Logic and Applications*. ENTCS 4. 1996.

[56] J. Meseguer. Membership Equational Logic as a Logical Framework for Equational Specification. In: F. Parisi-Presicce, Ed., *Proc. 12th WADT Workshop on Algebraic Development Techniques*. Springer LNCS 1376. 1998. pp. 18–61.

[57] J. Meseguer and U. Montanari. Petri Nets are Monoids. *Information and Computation* 88. 1990. pp. 105–154.

[58] J. Meseguer and U. Montanari. Mapping Tile Logic into Rewriting Logic. In: F. Parisi-Presicce, Ed., *Proc. 12th WADT Workshop on Algebraic Development Techniques*, Springer LNCS 1376. 1998. pp. 62–91.

[59] R. Milner, *Communication and Concurrency*. Prentice-Hall, 1989.

[60] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes (parts I and II). *Information and Computation* 100. 1992. pp. 1–77.

[61] F. Moller. The Nonexistence of Finite Axiomatizations for CCS Congruences. *Proc. LICS'90*, IEEE Press. 1990. pp. 142–153.

[62] F. Moller. The Importance of the Left Merge Operator in Process Algebras. *Proc. ICALP'90*, LNCS 443. 1990. pp. 752–764.

[63] U. Montanari and F. Rossi. Graph Rewriting, Constraint Solving and Tiles for Coordinating Distributed Systems. Draft submitted for publication.

[64] H. Miyoshi. Modelling Conditional Rewriting Logic in Structured Categories. In: J. Meseguer, Ed., *First International Workshop on Rewriting Logic and its Applications*. ENTCS 4. 1996.

[65] G. Plotkin. *A Structural Approach to Operational Semantics*. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University. 1981.

[66] W. Reisig. *Petri Nets*. Springer-Verlag. 1985.

[67] R. de Simone. Higher-level Synchronising Devices in Meije-SCCS. *Theoretical Computer Science* 37. 1985. pp. 245–267.

[68] R. Street. Higher Categories, Strings, Cubes and Simplex Equations. *Applied Categorical Structures* 3. 1995. pp. 29–77.

[69] R. Street. Categorical Structures. In: M. Hazewinkel, Ed., *Handbook of Algebra*. Elsevier. 1996. pp. 529–577.

[70] P. Viry. *Rewriting Modulo a Rewrite System* Technical Report TR-95-20. Department of Computer Science, University of Pisa. 1995.

A The Axioms of Process Tile Logic

Let $\mathcal{R} = \langle \Sigma_H, \Sigma_V, N, R \rangle$ be a pTRS. The class $A_p(\mathcal{R})$ of *abstract process sequents* informally described in Def. 2.24 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 $\_ \cdot \_$.

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 $n \in \mathbb{N}$, and composable arrows $h, g \in \mathbf{S}(\Sigma_H)$, and $v, u \in \mathbf{S}(\Sigma_V)$):

$$1_{v;u} = 1_v \cdot 1_u \qquad 1_{id_n} = 1^{id_n} \qquad 1^{h;g} = 1^h * 1^g$$

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

(for any $n \in \mathbb{N}$, and composable arrows $v, v' \in \mathbf{S}(\Sigma_V)$, and $u, u' \in \mathbf{S}(\Sigma_V)$):

$$\gamma_{v;v',u;u'} = \gamma_{v,u} \cdot \gamma_{v',u'} \qquad \gamma_{id_n,id_m} = 1^{\gamma_{n,m}}$$

(for any $n \in \mathbb{N}$, and composable arrows $h, h' \in \mathbf{S}(\Sigma_H)$, and $g, g' \in \mathbf{S}(\Sigma_H)$):

$$\rho_{h;h',g;g'} = \rho_{h,g} * \rho_{h',g'} \qquad \rho_{id_n,id_m} = 1_{\gamma_{n,m}}$$

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 $v : n \longrightarrow k$, $v' : n' \longrightarrow k' \in \mathbf{S}(\Sigma_V)$, and $h : n \longrightarrow m$, $h' : n' \longrightarrow m' \in \mathbf{S}(\Sigma_H)$):

$$\gamma_{v,v'} \cdot \sigma_{k,k'} = \sigma_{n,n'} \cdot 1_{v' \otimes v} \qquad \rho_{h,h'} * \sigma_{m,m'} = \sigma_{n,n'} * 1^{h' \otimes h}$$

Naturality axioms for σ

(for any $v : n \longrightarrow k$, $v' : n' \longrightarrow k' \in \mathbf{S}(\Sigma_V)$, and $h : n \longrightarrow m$, $h' : n' \longrightarrow m' \in \mathbf{S}(\Sigma_H)$):

$$1_{v \otimes v'} \cdot \sigma'_{k,k'} = \sigma'_{n,n'} \cdot \gamma_{v,v'} \qquad 1^{h \otimes h'} * \sigma'_{m,m'} = \sigma'_{n,n'} \cdot \gamma_{h,h'}$$

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 $n, m, k \in \mathbb{N}$):

$$\sigma_{n \otimes m,k} = ((1_{id_n} \otimes \sigma_{m,k}) * 1^{\gamma_{n,k} \otimes id_m}) \cdot (\sigma_{n,k} \otimes 1_{id_m}) \qquad (\sigma_{n,m} * 1^{\gamma_{m,n}}) \cdot \sigma_{m,n} = 1_{id_{n \otimes m}}$$

Coherence Axioms for σ' (for any $n, m, k \in \mathbb{N}$):

$$\sigma'_{n \otimes m,k} = (1_{id_n} \otimes \sigma'_{m,k}) \cdot (1_{id_n \otimes \gamma_{m,k}} * (\sigma'_{n,k} \otimes 1_{id_m})) \qquad \sigma'_{n,m} \cdot (1^{\gamma_{n,m}} * \sigma'_{m,n}) = 1_{id_{n \otimes m}}$$

Double Coherence Axioms (for any $n, m \in \mathbb{N}$):

$$\sigma'_{n,m} * \sigma_{n,m} = \gamma_{id_n,id_m} \qquad \sigma'_{n,m} \cdot \sigma_{n,m} = \rho_{id_n,id_m}$$

B The Axioms of Term Tile Logic

Let $\mathcal{R} = \langle \Sigma_H, \Sigma_V, N, R \rangle$ be a tTRS. 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[20]:

Associativity Axioms for $\_ \otimes \_$, $\_ * \_$, and $\_ \cdot \_$.

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 $n \in \mathbb{N}$, and composable arrows $v, v' \in \mathbf{A}(\Sigma_V)$, and $u, u' \in \mathbf{A}(\Sigma_V)$):

$$\gamma_{v;v',u;u'} = \gamma_{v,u} \cdot \gamma_{v',u'} \qquad \gamma_{id_n,id_m} = 1^{\gamma_{n,m}}$$

(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 $\alpha : h \xrightarrow[v]{u} g,\ \alpha' : h' \xrightarrow[v']{u'} g' \in P_t(\mathcal{R})$):

$$(\alpha \otimes \alpha') * \gamma_{u,u'} = \gamma_{v,v'} * (\alpha' \otimes \alpha) \qquad (\alpha \otimes \alpha') \cdot \rho_{g,g'} = \rho_{h,h'} \cdot (\alpha' \otimes \alpha)$$

[20] Notice that we make use of the notation of algebraic theories (Section 2.1.1) instead of the equivalent "terms and substitutions" notation used in Def. 2.28 and 2.30.

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 † (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 $n, m, k \in \mathbb{N}$):

$$\begin{array}{ll}
\sigma_{n \otimes m, k} = ((1_{id_n} \otimes \sigma_{m,k}) * 1^{\gamma_{n,k} \otimes id_m}) \cdot (\sigma_{n,k} \otimes 1_{id_m}) & (\pi_n * 1^{\gamma_{n,n}}) \cdot \sigma_{n,n} = \pi_n \\
\pi_{n \otimes m} = ((\pi_n \otimes \pi_m) * 1^{id_n \otimes \gamma_{n,m} \otimes id_m}) \cdot (1_{id_n} \otimes \sigma_{n,m} \otimes 1_{id_m}) & \phi_{n \otimes m} = \phi_n \otimes \phi_m \\
(\pi_n * 1^{\nabla_n \otimes id_n}) \cdot (\pi_n \otimes 1_{id_n}) = (\pi_n * 1^{id_n \otimes \nabla_n}) \cdot (1_{id_n} \otimes \pi_n) & (\pi_n * 1^{id_n \otimes !_n}) \cdot (1_{id_n} \otimes \phi_n) = 1_{id_n} \\
(\sigma_{n,m} * 1^{\gamma_{m,n}}) \cdot \sigma_{m,n} = 1_{id_{n \otimes m}} & \sigma_0 = \pi_0 = \phi_0 = 1_{id_0}
\end{array}$$

Coherence Axioms for σ', τ, and ψ (for any $n, m, k \in \mathbb{N}$):

$$\begin{array}{ll}
\sigma'_{n\otimes m,k} = (1_{id_n} \otimes \sigma'_{m,k}) \cdot (1_{id_n \otimes \gamma_{m,k}} * (\sigma'_{n,k} \otimes 1_{id_m})) & \tau_n \cdot (1^{\nabla_n} * \sigma'_{n,n}) = \tau_n \\
\tau_{n\otimes m} = (\tau_n \otimes \tau_m) \cdot (1^{\nabla_n \otimes \nabla_m} * (1_{id_n} \otimes \sigma'_{n,m} \otimes 1_{id_m})) & \psi_{n\otimes m} = \psi_n \otimes \psi_m \\
\tau_n \cdot (1^{\nabla_n} * (\tau_n \otimes 1_{id_n})) = \tau_n \cdot (1^{\nabla_n} * (1_{id_n} \otimes \tau_n)) & \tau_n \cdot (1^{\nabla_n} * (1_{id_n} \otimes \psi_n)) = 1_{id_n} \\
\sigma'_{n,m} \cdot (1^{\gamma_{n,m}} * \sigma'_{m,n}) = 1_{id_{n\otimes m}} & \sigma'_0 = \tau_0 = \psi_0 = 1_{id_0}
\end{array}$$

Double Coherence Axioms (for any $n, m \in \mathbb{N}$):

$$\begin{array}{ll}
\sigma'_{n,m} * \sigma_{n,m} = \gamma_{id_n,id_m} & \sigma'_{n,m} \cdot \sigma_{n,m} = \rho_{id_n,id_m} \\
\tau_n * \pi_n = \nabla_{id_n} & \tau_n \cdot \pi_n = \delta_{id_n} \\
\psi_n * \phi_n = !_{id_n} & \psi_n \cdot \phi_n = \dagger_{id_n}
\end{array}$$

C Hypertransformations

As explained in Section 3.4, since double categories have two different notions of composition, it is not clear which of them to use for a general notion of double natural transformation. Ehresmann noticed in [25] 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 transformation for n-fold categories we have to consider quartets in all the n dimensions.

Definition C.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 1-vertical category $\mathcal{V}$ of $\mathcal{D}$, and whose composition law is deduced from that of the category $\mathcal{D}^*$ (e.g., there is an arrow α from $P : \mathcal{A} \longrightarrow \mathcal{V}$ to $Q : \mathcal{A} \longrightarrow \mathcal{V}$ iff 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 $\alpha(v) * \alpha'(v)$).

$$
\begin{tikzcd}
 & a \arrow[d, "v"'] & \cdot \arrow[r, "\alpha(a)"] \arrow[d, "Pv"'] \arrow[dr, phantom, "\alpha(v)"] & \cdot \arrow[r, "\alpha'(a)"] \arrow[d, "Qv"] \arrow[dr, phantom, "\alpha'(v)"] & \cdot \arrow[d, "Rv"] & \\
(\mathcal{A}) & b \arrow[d, "u"'] & \cdot \arrow[r] \arrow[d, "Pu"'] \arrow[dr, phantom, "\alpha(u)"] & \cdot \arrow[r] \arrow[d, "Qu"] \arrow[dr, phantom, "\alpha'(u)"] & \cdot \arrow[d, "Ru"] & (\mathcal{D}) \\
 & c & \cdot \arrow[r, "\alpha(c)"'] & \cdot \arrow[r, "\alpha'(c)"'] & \cdot &
\end{tikzcd}
$$

Moreover, T extends to a functor (i.e., 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.2 Let **2** be the category with two objects (0 and 1) and three arrows ($z : 0 \longrightarrow 1$ and the identities for 0 and 1). Then, for any double category $\mathcal{D}$, we have that $T(\mathcal{D},\mathbf{2}) \simeq \mathcal{D}^*$. □

Proposition C.3 *Let $\mathcal{A}$ and $\mathcal{C}$ be two categories. The category $\mathcal{C}^{\mathcal{A}}$ of natural transformations between functors from $\mathcal{A}$ to $\mathcal{C}$ is isomorphic to the category $T(\square\mathcal{C},\mathcal{A})$, where $\square\mathcal{C}$ is the double category of quartets of $\mathcal{C}$. In fact, a functor from $\mathcal{A}$ to the vertical category of $\square\mathcal{C}$ (usually denoted by $\overset{\square}{\mathcal{C}}$) 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.4 [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*:

$$(\mathcal{C}^i,\mathcal{C}^j) \text{ is a double category for each pair } (i,j) \text{ of integers } 1 \le i < j \le n.$$

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, an *n-fold functor* $F : \vec{\mathcal{C}} \longrightarrow \vec{\mathcal{D}}$ is a map $F : \mathcal{C} \longrightarrow \mathcal{D}$ defining 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** and $\mathbf{Cat}_1$ is **Cat**.

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 is denoted by $\vec{\mathcal{C}}^{\gamma}$. If $i_1,\ldots,i_m$ is a sequence of m distinct element 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.5 [Multiple Category] The category **MCat** of *multiple categories* is defined as follows. Its objects are all the small n-fold categories, for every integer n. Let $\vec{\mathcal{C}}$ be an n-fold category and $\vec{\mathcal{D}}$ 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 $\vec{\mathcal{D}}$). $\square$

Definition C.6 [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 void set, otherwise ($n \leq m$) 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 $\mathcal{C}$. $\square$

Remark C.7 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 the compositions are defined pointwise from that of $\vec{\mathcal{D}}$.

Example C.8 If $\mathcal{A}$ and $\mathcal{C}$ are categories, then $Hom(\mathcal{A},\square\mathcal{C})$ is the category $\mathcal{C}^{\mathcal{A}}$ of natural transformations between functors from $\mathcal{A}$ to $\mathcal{C}$. $\square$

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 below), because $F(z,z)$ is just the diagonal of this square.

$$\begin{tikzcd}
(0,0) \arrow[r, "{(0,z)}"] \arrow[d, "{(z,0)}"'] \arrow[dr, "{(z,z)}"] & (0,1) \arrow[d, "{(z,1)}"] \\
(1,0) \arrow[r, "{(1,z)}"'] & (1,1)
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
\cdot \arrow[r, "{F(0,z)}"] \arrow[d, "{F(z,0)}"'] & \cdot \arrow[d, "{F(z,1)}"] \\
\cdot \arrow[r, "{F(1,z)}"'] & \cdot
\end{tikzcd}$$

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 $\square\mathcal{C}$).

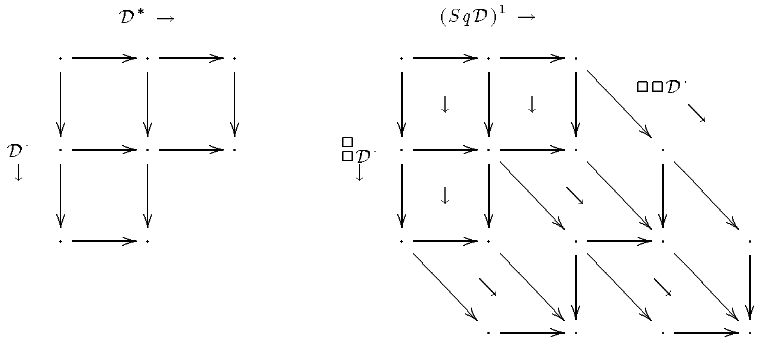
Now let $\vec{\mathcal{C}}$ be an n-fold category ($n > 1$). Consider the functors from $\mathbf{2}\times\mathbf{2}$ to its first category $\mathcal{C}^1$. It follows that, over the set of quartets of $\mathcal{C}^1$ we have not only the double category $\square\mathcal{C}^1$, but also the $(n-1)$-fold category $Hom(\mathbf{2}\times\mathbf{2},\vec{\mathcal{C}})$ whose i-th composition is deduced pointwise from that of $\mathcal{C}^{i+1}$, e.g. 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 $Q;Q' = ((A;_{i+1} A'),(B;_{i+1} B'),(C;_{i+1} C'),(D;_{i+1} D'))$ (if and only if the four composite are defined in $\mathcal{C}^{i+1}$).

Definition C.9 [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: $(Sq\mathcal{C})^{1,\ldots,n-1} = Hom(\mathbf{2}\times\mathbf{2},\vec{\mathcal{C}})$, $(Sq\mathcal{C})^n = \substack{\square\\\square}\mathcal{C}^1$ and $(Sq\mathcal{C})^{n+1} = \square\square\mathcal{C}^{n+1}$. $\square$

The previous construction induces a functor from $\mathbf{Cat}_n$ to $\mathbf{Cat}_{n+1}$ called *Square* and denoted by $Sq_{n,n+1}$ which maps an n-fold functor $F : \vec{\mathcal{C}} \longrightarrow \vec{\mathcal{D}}$ onto the $(n+1)$-fold functor $SqF : Sq\mathcal{C} \longrightarrow Sq\mathcal{D}$ such that $(A,B,C,D) \mapsto (FA,FB,FC,FD)$.

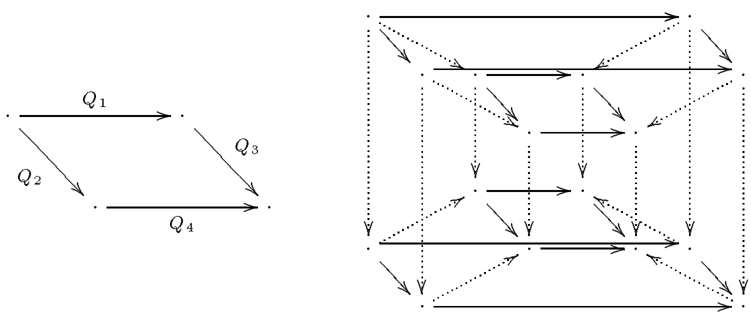
C.1 The 3-fold category $Sq\mathcal{D}$

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 $\mathcal{D}^{*}$.



C.2 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 $\mathcal{D}^{\cdot}$ (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* below:



The 1-st and 2-nd compositions are deduced pointwise from that of $\overset{\square}{\square}(\mathcal{D}^{\cdot})$ and $\square\square(\mathcal{D}^{\cdot})$, 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 $\overset{\square}{\square}((Sq\mathcal{D})^{1})$ and $\square\square((Sq\mathcal{D})^{1})$ (whose compositions are induced from that of $\mathcal{D}^{*}$) and they consist in putting one frame *behind* the other (i.e., the block above has source Q_1 and target Q_4) and one frame *beside* the other (source Q_2 and target Q_3)

C.3 Hypertransformations

Now we try to relate the definition of generalized natural transformation as given in Section 3.4 with the more general notion of *hypertransformation*.

Definition C.10 [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 $i = 1, \ldots, n$ the i-th category of $\square_n\vec{\mathcal{D}}$ is derived from $\overset{\square}{\square}(\mathcal{D}^{i})$, 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 $\vec{\mathcal{D}}$. $\square$

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 $SqSq\mathcal{E}$. We illustrate the double category $Hom(\mathcal{D}, \Box_2\mathcal{E})$ as follows:

- Objects (i.e., vertices) are the double functors from $\mathcal{D}$ to $\mathcal{E}$.
- Horizontal arrows (i.e., objects for the 1-st category of $Hom(\mathcal{D}, \Box_2\mathcal{E})$) are $*$-transformations.
- Vertical arrows (i.e., objects for the 2-nd category of $Hom(\mathcal{D}, \Box_2\mathcal{E})$) are $\cdot$-transformations.
- Cells are generalized transformations between four double functors.
- Compositions are deduced pointwise from the compositions one frame *inside* the other and one frame *beside* the other (we remind that hypertransformations behave functorially w.r.t. the two compositions one frame *below* the other and one frame *behind* the other).

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 in which 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 a module `FRAG-CCS` that corresponds to a fragment of CCS (see Section 7):

```
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
```

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 .
```

and

```
 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 : s1 ... sn -> s .
```

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`,`$P_2 \ldots$`,`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 [14]. 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 .
$\vdots$
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 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 type 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, 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 φ1 and ··· and φm .
```

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 correctness, 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

$$\begin{array}{ll}\texttt{subsorts} & Sl_1 \ \ . \\ & Sl_2 \ \ . \\ & \ \vdots \\ & Sl_n \ \ .\end{array}$$

where each Sl_i is a list of subsort expressions, as a shorthand for

$$\begin{array}{l}\texttt{subsorts}\ Sl_1 \ \ . \\ \texttt{subsorts}\ Sl_2 \ \ . \\ \vdots \\ \texttt{subsorts}\ Sl_n \ \ .\end{array}$$

D.2.3 Membership Assertions

We write

$\texttt{mbs}\ t_1 \ \ \texttt{...}\ \ t_k \ \ \texttt{:}\ \ s \ \ \texttt{.}$

as a shorthand for

$$\begin{array}{l}\texttt{mb}\ t_1 \ \ \texttt{:}\ \ s \ \ \texttt{.} \\ \vdots \\ \texttt{mb}\ t_k \ \ \texttt{:}\ \ s \ \ \texttt{.}\end{array}$$

D.2.4 Using `iff` in a Conditional Sentence

We write

$\texttt{cmb}\ \psi_1 \ \cdots\ \psi_n\ \texttt{iff}\ \phi_1\ \texttt{and}\ \cdots\ \texttt{and}\ \phi_m \ \ \texttt{.}$

where each ψ_i is a membership assertion, as a shorthand for

$$\begin{array}{l}\texttt{cmb}\ \psi_1\ \texttt{if}\ \phi_1\ \texttt{and}\ \cdots\ \texttt{and}\ \phi_m \ \ \texttt{.} \\ \qquad\quad \vdots \\ \texttt{cmb}\ \psi_n\ \texttt{if}\ \phi_1\ \texttt{and}\ \cdots\ \texttt{and}\ \phi_m \ \ \texttt{.} \\ \texttt{cmb/ceq}\ \phi_1\ \texttt{if}\ \psi_1\ \texttt{and}\ \cdots\ \texttt{and}\ \psi_n \ \ \texttt{.} \\ \qquad\quad \vdots \\ \texttt{cmb/ceq}\ \phi_m\ \texttt{if}\ \psi_1\ \texttt{and}\ \cdots\ \texttt{and}\ \psi_n \ \ \texttt{.}\end{array}$$

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

$\texttt{ceq}\ \psi_1 \ \cdots\ \psi_n\ \texttt{iff}\ \phi_1\ \texttt{and}\ \cdots\ \texttt{and}\ \phi_m \ \ \texttt{.}$

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. 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 [14] for its description.

D.3.3 Quoted Identifiers

The module `QID` plays an important role for the 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 reason, 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 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 [14] for the extensive definition of the signature of 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 in `META-LEVEL` as quoted identifiers. 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 6.3. 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 passed as parameter. Furthermore, for the `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 reading to people not acquainted with the Maude implementation).

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 Parametric Modules and Infix Operators

Functional modules can be unparametrized, or they can be parametrized with functional theories as their parameters. However, the present beta version only implements unparametrized 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.