Executable Tile Specifications for Process Calculi*

Roberto Bruni[1], José Meseguer[2] and Ugo Montanari[1]

[1] Dipartimento di Informatica, Università di Pisa, Italia.
[2] Computer Science Laboratory, SRI International, Menlo Park, CA, U.S.A.
bruni@di.unipi.it, meseguer@csl.sri.com, ugo@di.unipi.it

Abstract. *Tile logic* extends *rewriting logic* by taking into account side-effects and rewriting synchronization. These aspects are very important when we model *process calculi*, because they allow us to express the dynamic interaction between processes and "the rest of the world". Since rewriting logic is the semantic basis of several language implementation efforts, we can define an executable specification of tile systems by mapping tile logic back into rewriting logic. In particular, this implementation requires the development of a metalayer to control rewritings, i.e., to discard computations that do not correspond to any deduction in tile logic. Our methodology is applied to *term tile systems* that cover and extend a wide-class of SOS formats for the specification of process calculi. The case study of full CCS, where the term tile format is needed to deal with recursion (in the form of the replicator operator), is discussed in detail.

1 Introduction

This paper reports on the application of tile logic to the specification and execution of process calculi. For the specification part, we take advantage of the synchronization mechanism of tile logic that extends well-known SOS formats. The execution is based on a general translation of tile logic into rewriting logic.

In *rewriting logic* [26], 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 then specified by simple inference rules and deduction in the logic is equivalent to computing in the system. Given this correspondence, a sentence $t \Rightarrow t'$ has two readings: *computationally*, it means that when the system is in a state s, any instance of the pattern t in s can evolve to the corresponding instance of t', possibly in parallel with other changes; *logically*, it means that we can derive the *formula* t' from t. Moreover, the notion of state is entirely user-definable as an algebraic data type satisfying certain equational

* Research partially supported by MURST project Tecniche Formali per Sistemi Software, and by Esprit WG CONFER2. Also partially supported by DARPA through Rome Laboratories Contract F30602-97-C-0312 and NASA Contract NAS2-98073, by Office of Naval Research Contract N00014-96-C-0114, and by National Science Foundation Grants CCR-9505960 and CCR-9633363.

properties. Therefore, rewriting logic has good properties as a *semantic framework* where many different languages, systems and models of computation (e.g., labelled transition systems, grammars, Petri nets and algebraic nets, chemical abstract machine, concurrent objects, actors, graph rewriting, data flow, neural networks, real time systems, and many others) can be nicely expressed by natural encodings. For example, specifications of programming languages given in terms of rewrite theories become *de facto* interpreters for the languages in question. On the other hand, rewriting logic has also very good properties as a *logical framework* where many other logics can be naturally represented (several examples can be found in [25, 24]). Rewriting logic has also been used as a semantic framework for software architectures, providing a formal semantics for *architecture description languages* and their interoperation [31]. Further examples regards *theorem provers* and other formal methods tools, based on *inference systems* that can be specified and prototyped in rewriting logic. Also *communication protocols*, including *secure* ones, are another promising area [13]. Moreover, there exist several languages based on rewriting logic (e.g., Maude, ELAN, CafeObj), developed in different countries, and this growing community has recently organized two workshops to discuss all the aspects of rewriting logic [28, 23]. A progress report on the multiple activities connected to rewriting logic has been the topic of an invited talk at CONCUR'96 [27].

The semantics of process calculi for reactive systems and protocol prototyping are usually presented in the SOS form [35]. Such representation naturally yields a *conditional* rewriting system [25], where the basic rule of the rewrite theory can have the more general form: $t \Rightarrow t'$ if $s_1 \Rightarrow s'_1 \wedge \cdots \wedge s_n \Rightarrow s'_n$.

Unfortunately, the implementation of conditional rules increases the expressive power of rewrite theories as much as the complexity of the underlying rewrite machine. Indeed, conditional rules are not supported by languages based on rewriting logic for efficency reasons. Hence, specifications must be adapted before becoming executable. Of course, such modification can be pursued in an ad hoc fashion for each model, but a better approach consists of having a *methodology* that automatically performs the translation for an entire class of problems.

The *tile model* [17, 19] is a formalism for modular descriptions of the dynamic evolution of concurrent systems. Basically, a set of rules defines the behaviour of certain *open* (e.g., partially specified) *configurations*, which may interact through their *interfaces*. Then, the behaviour of a system as a whole consists of a coordinated evolution of its sub-systems. The name "tile" is due to the graphical representation of such rules, which have the form

$$\begin{tikzcd} \text{inital input interface } \circ \arrow[r, "s"] \arrow[d, "a"'] & \circ \text{ initial output interface} \arrow[d, "b"] \\ \text{final input interface } \circ \arrow[r, "s'"'] & \circ \text{ final output interface} \end{tikzcd}$$

also written $s \xrightarrow[b]{a} s'$, stating that the *initial configuration* s of the system evolves to the *final configuration* s' producing the *effect* b, which can be observed by the rest of the system. However, such a step is allowed only if the subcomponents of s (which is in general an open configuration) evolve to the subcomponents of s', producing the *trigger* a. Triggers and effects are called *observations*. The vertices of each tile are called *interfaces*.

Tiles can be composed horizontally (synchronizing an effect with a trigger), vertically (computational evolutions of a certain component), and in parallel (concurrent steps) to generate larger steps. By analogy with rewriting logic, the tile model also comes equipped with a purely logical presentation [19], where tiles are just considered as (decorated) sequents subject to certain inference rules. 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. As an example, auxiliary tiles may be necessary to represent consistent horizontal and vertical rearrangements of interfaces.

It is clear that tile logic extends (unconditional) rewriting logic, taking into account rewriting with side effects and rewriting synchronization, whereas, in rewriting systems, both triggers and effects are just identities (i.e., rewriting steps may be applied freely). This feature of tile logic has been at the basis of several successful application as a model of computation for reactive systems: varying the algebraic structures of configurations and observations many different aspects can be modelled, ranging from synchronization of net transitions [7], to causal dependencies for located calculi [16], to finitely branching approaches for name-passing calculi [15], to actor systems [34]. Moreover, tile logic allows one to reason about open configurations, in the style of *context systems* [21], whilst ordinary SOS formats work for ground terms only (e.g., *bisimulation* can be generalized to *tile bisimulation* that operates over contexts rather than just over terms).

A main question has concerned how to give an implementation to tile logic. Systems based on rewriting logic are a natural choice, due to the great similarity with the more general framework of tiles. This topic has been extensively investigated in [30], and successively in [4], where the results of [30] have been extended to the cases of *process* and *term tile logic*, where both configurations and effects rely on common auxiliary structures (e.g., for tupling, projecting or permuting interfaces). As a result, the mapping becomes effective provided that "the rewriting engine is able to filter computations". To achieve this, in [5] we make use of the *reflective* capabilities [11, 8] of the Maude language [9] to define suitable *internal strategies* [10, 12], which allow the user to control the computation and to collect the possible results.

In this paper we give a survey of some basic internal strategies, and we show how they can be applied to obtain executable specifications for a rich class of process calculi. To give an example of the implementation mechanism, we instantiate the general idea to the well-known case study of full CCS [32] whose presentation requires the term tile format. While a process calculus (located CCS) that needs *process* tile logic (rather than the simpler *monoidal tile logic* of [30]) has been modelled in [6], this is the first time that *term* tile logic is shown to be indispensable for certain features of process calculi.

Related Works. This work is part of our ongoing research aimed at developing general mechanisms for a uniform implementation of several tile formats. In recent papers, different mathematical structures have been employed to model configurations and observations. Basically, we can distinguish two approaches.

The first approach, proposed in [19], considers models arising from internal constructions in suitable categories with structure. Such "structure", usually determined by the algebra of configurations, is then lifted to tiles (in the horizontal dimension only), whilst the observations just yield a monoid over the sequences of basic actions. Within this class we recall a net model equipped with a synchronization mechanism between transitions, called *zero-safe nets* [7]. This is probably the simplest tile model one can imagine, because its configurations and its observations are just commutative monoids (the monoidal operation models both parallel and sequential compositions). Other examples consist of the monoidal tile system for finite CCS of [30], where discharged choices of nondeterministic sums are managed with explicit garbaging, and the algebraic tile system for finite CCS [19], where configurations have a cartesian structure (corresponding to the term algebra of processes), and free discharging of choices is allowed. Finally, we mention the simple coordination model based on graph rewriting and synchronization of [33] whose configurations, called *open graphs*, have an algebraic characterization as suitable *gs-graphs* (a structure with explicit subterm sharing and garbaging, offering a partial algebraic semantic for modelling graphs with subsets of sharable nodes), used to recast the hard computational problem of tile synchronization into a distributed version of constraint solving.

The second approach considers a richer class of models, where both configurations and observations have similar algebraic structures. Rather than based on internal constructions, such models rely on the notion of *hypertransformation*, which is able to characterize the analogies between the mathematical structures employed in the two dimensions. Within this class we recall the tile model for located CCS, which can take into account causal relationships between the performed actions, by looking at the *locations* where they take place. This model requires some auxiliary tiles for consistent permutations of the elements in the interfaces. Such tiles have been naïvely introduced in [16], and then characterized as suitable hypertransformations in [4, 6]. We also want to mention the tile models sketched in [34] to emphasize the similarities between *actor systems* and calculi with mobility (e.g., π-calculus).

The tile model for full CCS we propose in this paper is based on term structures on both dimensions and clearly belongs to the second class of systems.

The idea of translating tile models based on the first approach above into rewriting logic has been discussed for the first time by two of the authors in [30]. Then, it has been extended to the more general framework (based on hypertransformations) in [4]. Further directions of research have focused on control mechanisms over rewritings that are necessary to support the theoretical results at the implementation level of Maude. To this aim, the definition of a kernel of internal strategies in the language Maude has been fully discussed in [5].

An extensive presentation of the translation of tiles into ordinary rewrite rules can be found in the Technical Report [4] and in the forthcoming PhD Thesis of one of the authors [3]. In particular, we have investigated the similarities between rewriting logic and tile logic from the point of view of their categorical models[1].

Structure of the paper. In Section 2 we give a survey of tile logic and of its translation into rewriting logic. We also propose a brief comparison between several specification formats. In Section 3 we describe some useful internal strategies, written in a self-explanatory Maude-like notation, and in Section 4 we show their application to the field of process calculi. The case study presented in Section 4 consists of a full version of CCS, where the replicator is considered.

2 Mapping Tile Logic into Rewriting Logic

2.1 Tile Logic Specifications

The notions of configuration, observation and interface are the basic ingredients of tile logic, and all of them come naturally equipped with the operation of parallel composition. Moreover, the input (output) interface of the parallel composition of two configurations h and g is just the parallel composition of the two input (output) interfaces of h and g. Similarly for the interfaces of the parallel composition of two observations. To simplify the notation, in this presentation we assume that the parallel composition is associative and has the empty interface (configuration, observation) as neutral element, i.e., we assume that interfaces (configurations, observations) yield a *strict monoid.*

Informally, the interfaces represents connection points between different configurations of the system, and also between consecutive observations of the same component. Therefore, configurations (observations) also have sequential composition as a natural operation. In particular we can assume that configurations and observations form two *strict monoidal categories*, having the same class of objects (i.e., the interfaces). We denote the operators of parallel and of sequential composition by $\_ \otimes \_$ and by $\_;\_$ respectively.

Within tile logic, the basic methodology to specify the model of computation for a concrete system consists of the following steps: (1) define the set of basic configurations of the system; (2) define what the interfaces of each basic configuration are; (3) define the basic events that we want to observe and their interfaces according to the previous steps (if necessary, we can repeatedly apply

[1] A rewriting theory $\mathcal{R}$ yields a *cartesian 2-category* $\mathcal{L}_{\mathcal{R}}$, which does for $\mathcal{R}$ what a Lawvere theory does for a signature. Gadducci and Montanari pointed out in [18], that if also side-effects are introduced, then *double categories* (DC) [14] should be considered as a natural model: tiles are double cells, configurations are horizontal arrows, observations are vertical arrows, and objects model connections between the somehow static horizontal category and the dynamic vertical evolution. Depending on the structures under consideration, either monoidal DC [19], or symmetric DC, or cartesian DC must be considered (the last two notions being introduced in [4]).

these three steps, until the basic structures are chosen); (4) define the set of tiles that describe the basic behaviours of the system accordingly to the framework chosen in the first three steps (again, it could be necessary to iterate all the four steps to obtain a consistent definition of the tile system). The steps (1) and (3) must take into account the fact that the mathematical structures employed to represent configurations and observations are strict monoidal categories. Most of the times, it is convenient to assume that configurations and observations are just the categories freely generated from the basic configurations and from the basic observable actions that the system can perform.

For example, an obvious choice in the definition of tile models for many process algebras is to take the term algebra of processes as the category of configurations, and the free monoid over action strings as the category of observations (e.g., see [19]). This is correct because the tuple of terms can be seen as arrows of a suitable strict monoidal category (namely a cartesian category), where the parallel composition corresponds to tupling of terms and the sequential composition is term substitution. In this case, the structure employed for configurations is richer than the one requested by the framework. Indeed it allows free duplication and projection of terms. We refer to such operators as *auxiliary structures*, because they do not depend on the signature, and instead belong to any term algebra under consideration.

The tile model for full CCS introduced in Section 4 requires a term structure for both configurations and observations. Hence, similar auxiliary structures are necessary on both dimensions and a certain number of auxiliary tiles (for consistent rearrangements of interfaces on both dimensions) must be introduced in the model. We have investigated such kind of tile structures in [4], under the terminology *term tile systems* (tTS).

2.2 Term Tile Systems

In what follows we consider one-sorted signature only. The many-sorted case can be handled very easily in a similar way, but requires a more complex notation that is not necessary for our case study and therefore avoided.

An *algebraic theory* [22] is just a cartesian category having underlined natural numbers as objects. The free algebraic theory associated to a signature Σ is called the *Lawvere theory* for Σ, and is denoted by $\mathbf{Th}[\Sigma]$: the arrows from $\underline{m}$ to $\underline{n}$ are in a one-to-one correspondence with n-tuples of terms of the free Σ-algebra with (at most) m variables, and composition of arrows is term substitution. As a matter of notation, we assume a standard naming of the $\underline{m}$ input variables, namely $x_1, ..., x_m$. When composing two arrows $\boldsymbol{s} : \underline{m} \longrightarrow \underline{k}$ and $\boldsymbol{t} : \underline{k} \longrightarrow \underline{n}$, the resulting term $\boldsymbol{s};\boldsymbol{t}$ is obtained by replacing each occurrence of x_i in $\boldsymbol{t}$ by the i-th term of the tuple $\boldsymbol{s}$, for $i = 1, ..., k$. For example, constants a, b in Σ are arrows from $\underline{0}$ to $\underline{1}$, a binary operator $w(x_1, x_2)$ define an arrow from $\underline{2}$ to $\underline{1}$, and the composition $\langle a, b\rangle; \langle w(x_1, x_2), x_1\rangle; \langle w(x_2, x_1)\rangle$ yields the term $w(a, w(a, b))$, which is an arrow from $\underline{0}$ to $\underline{1}$, in fact:

$$\langle a, b\rangle; \langle w(x_1, x_2), x_1\rangle; \langle w(x_2, x_1)\rangle = \langle w(a, b), a\rangle; \langle w(x_2, x_1)\rangle = \langle w(a, w(a, b))\rangle$$

When no confusion can arise, we avoid the use of angle brackets to denote term vectors. When configurations and observations are terms over two distinct signatures Σ_H and Σ_V, we can assume that each basic tile has the form:

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

with $\boldsymbol{h} \in T_{\Sigma_H}(X_n)^m$, $g \in T_{\Sigma_H}(X_k)$, $\boldsymbol{v} \in T_{\Sigma_V}(X_n)^k$, and $u \in T_{\Sigma_V}(X_m)$, where $X_i = \{x_1, ..., x_i\}$ is a chosen set of variables, totally ordered by $x_{j_1} < x_{j_2}$ if $j_1 < j_2$, and $T_\Sigma(X)^n$ denotes the n-tuples of terms over the signature Σ and variables in X. Due to space limitation, we present tiles more concisely as logic sequents $n \lhd \boldsymbol{h} \xrightarrow[u]{\boldsymbol{v}} g$, where also the number of variables in the "upper-left" corner of the tile is made explicit (the values m and k can be easily recovered from the lengths of $\boldsymbol{h}$ and $\boldsymbol{v}$). The idea is that each interface represents an ordered sequence (i.e., a tuple) of variables; therefore each variable is completely identified by its position in the tuple, and a standard naming $x_1, ..., x_n$ of the variables can be assumed. For example, if the variable x_i appears in the effect u of the above rule, then this means that the effect u depends on the i-th component $\boldsymbol{h}_i$ of the initial configuration. Analogously for the remaining connections. Notice that the same variable x_i, denotes the i-th element of different interfaces when used in each of the four border-arrows of the tile (in particular, only the occurrences of x_i in $\boldsymbol{h}$ and in $\boldsymbol{v}$ denote the same element of the initial input interface $\underline{n}$).

The format of term tiles is very general. In particular, it extends the *positive GSOS* format [1], where multiple testing of the same argument are allowed in the premises, and tested arguments can appear in the target of the transition in the conclusion, i.e., rules can have the more general form

$$\frac{\{x_i \xrightarrow{a_{ij}} y_{ij} \mid 1 \leq i \leq k, 1 \leq n_i\}}{f(x_1, ..., x_k) \xrightarrow{a} C[x_1, y_{11}, ..., y_{1n_1}, ..., x_k, y_{k1}, ..., y_{kn_k}]}$$

where the variables are all distinct, f is a k-ary operator, $n_i \geq 0$, a_{ij} and a are actions and $C[\_{1}, ..., \_{N}]$ is a context that takes $N = \sum_{i=1}^{k}(n_i + 1)$ arguments. The corresponding tiles have the form $k \lhd\ f(x_1, ..., x_k) \xrightarrow[a(x_1)]{\boldsymbol{a}} C[x_1 ... x_N]$, where $\boldsymbol{a} = \langle x_1, a_{11}(x_1), .., a_{1n_1}(x_1), ..., x_k, a_{k1}(x_k), .., a_{kn_k}(x_k)\rangle$ is the vector of triggers (for each argument of f, $\boldsymbol{a}$ contains the idle trigger plus all the actions that are tested in the premises of the GSOS rule for that argument). We remark that the positive GSOS format cannot be handled by tile systems defined using the internal construction approach (see [19]). Term tile logic can also handle rules with *lookahead* as the one defined in [20]:

$$\frac{x \xrightarrow{a^+} y,\ y \xrightarrow{a^-} z}{\text{combine}(x) \xrightarrow{a} \text{combine}(z)} \text{ becomes } 1 \lhd\ \text{combine}(x_1) \xrightarrow[a(x_1)]{a^-(a^+(x_1))} \text{combine}(x_1)\ .$$

Therefore, term tile format appears very expressive and this motivate us to provide an executable framework for it. However, an extensive comparison of existing formats is out of the scope of this paper and is left for future works.

2.3 From Tiles to Rewrite Rules

The comparison between tile logic and rewriting logic takes place by embedding their categorical models in a recently developed, specification framework, called *partial membership equational logic* (PMEqtl) [29]. PMEqtl is particularly suitable for the embedding of categorical structures, first because the sequential composition of arrows is a partial operation, and secondly because membership predicates over a poset of sorts allow the objects to be modelled as a subset of the arrows and arrows as a subset of cells. Moreover, the *tensor product construction* illustrated in [30] can be easily formulated in PMEqtl, providing a convenient definition of the theory of monoidal double categories as the tensor product of the theory of categories (twice) with the theory of monoids.

The advantage of modelling process algebras in tile logic (using the trigger/effect synchronization mechanism of rewritings) should be evident just considering the usual *action prefix* operation, denoted by $\mu.\_$. When applied to a certain process P it returns a process $\mu.P$ which can perform an action μ and then behaves like P. The corresponding tile is represented below (horizontal arrows are process contexts and vertical arrows denote computations). Notice that the horizontal operator $\mu.\_$ and the vertical operator $\mu(\_)$ are very different: the former represents the μ prefix context, which is a syntactic operator, and the latter denotes the execution of the observable action μ. Such tile can be composed horizontally with the identity tile of any process P to model the computation step associated to the action prefix.

$$
\begin{tikzcd}
\underline{1} \arrow[r, "\mu.x_1"] \arrow[d, "id"'] & \underline{1} \arrow[d, "\mu(x_1)"] \\
\underline{1} \arrow[r, "id"'] & \underline{1}
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
\underline{0} \arrow[r, "P"] \arrow[d, "id"'] & \underline{1} \arrow[r, "\mu.x_1"] \arrow[d, "id"] & \underline{1} \arrow[d, "\mu(x_1)"] \\
\underline{0} \arrow[r, "P"'] & \underline{1} \arrow[r, "id"'] & \underline{1}
\end{tikzcd}
=
\begin{tikzcd}
\underline{0} \arrow[r, "\mu.P"] \arrow[d, "id"'] & \underline{1} \arrow[d, "\mu(x_1)"] \\
\underline{0} \arrow[r, "P"'] & \underline{1}
\end{tikzcd}
$$

Now, let *nil* be the inactive process, and consider the process $Q = \mu_1.\mu_2.nil$. If the process Q tries to execute the action μ_2 before executing μ_1 it gets stuck, because there is no tile having $\mu_2(\_)$ as trigger and $\mu_1.\_$ as initial configuration.

$$
\begin{tikzcd}
\underline{0} \arrow[r, "nil"] \arrow[d, "id"'] & \underline{1} \arrow[r, "\mu_2.x_1"] \arrow[d, "id"] & \underline{1} \arrow[r, "\mu_1.x_1"] \arrow[d, "\mu_2(x_1)"] & \underline{1} \\
\underline{0} \arrow[r, "nil"'] & \underline{1} \arrow[r, "id"'] & \underline{1} & ?
\end{tikzcd}
$$

In unconditional rewriting systems, this is not necessarily true, because rewriting steps can be freely contextualized (and instantiated). This problem is well-known in rewriting logic, and some partial solutions have been already proposed in the literature [25, 36]. However, our methodology seems to offer a unifying view for a wide class of related problems. The basic idea is to "stretch" tiles into ordinary rewriting cells as pictured below, maintaining the capability to distinguish between configurations and observations.

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

As a main result, given a tile system $\mathcal{R}$, a sequent $s \xrightarrow[b]{a} s'$ is entailed by $\mathcal{R}$ in tile logic if and only if a sequent $s; b \Rightarrow a; s'$ is entailed by the stretched version of $\mathcal{R}$ in rewriting logic and its proof satisfies some additional constraints (see [30, 4]). Indeed, the forgetful functor from the category of models of the stretched logic (where also the distinction between configuration and observations can be made) to the category of models of the tile logic has a left adjoint. Moreover, for a large class of tile systems (called *uniform*) the additional constraints reduce to check that the border of the sequent can be correctly partitioned into configurations and observations[2] (the source of the sequent must be a configuration followed by an observation, and the target must be an observation followed by a configuration).

It follows that a typical query in a tile system could be: "derive all (some of) the tiles with initial configuration s and effect b" (this corresponds to start with the state $s; b$ and apply the rewritings that simulate a tile computation with vertical source s and horizontal target b). Hence, we need to define some rewriting strategies for exploring the tree of nondeterministic rewritings until a successful configuration is reached. For instance, a general notion of success for uniform tile systems consists of `VH` configurations (i.e., an arrow of the vertical category followed by an arrow of the horizontal category) as we will see in Section 3.3.

3 Internal Strategies in Rewriting Logic

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, and the meaningful sentences are rewrite sequents $t \Rightarrow t'$, where t and t' are Σ-terms. We call *strategy* any computational way of looking for certain proofs of some theorems of T. An *internal strategy language* is a function S that sends each theory T to another theory $S(T)$ *in the same logic*, whose deductions simulate controlled deductions of T. The class of finitely presentable rewrite theories has universal theories, making rewriting logic reflective [10, 8].

[2] If the tile system is not uniform, then also the actual proof term decorating the derivation has to be taken into account. However, since at present we do not have any meaningful example of non uniform systems we are not really interested in having such an implementation.

This means that there exists a finitely representable rewrite theory U able to simulate deductions in all the other rewrite theories, i.e., there is a representation function $\overline{(\_ \vdash \_)}$ that encodes a pair consisting of a rewrite theory T 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. Hence, strategies in $S(U)$ are particularly important, since they represent, at the object level, strategies for computing in the universal theory.

3.1 A Strategy Kernel Language in Maude

Maude [9] 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 Maude implementation supports metaprogramming of strategies via a module-transforming operation which maps a module T to another module `META-LEVEL[`T`]` that is a definitional extension of U [12]. For simplicity, we adopt here a simpler version of the metalevel. In particular, the following operations are defined: `meta-reduce(`$\overline{t}$`)` and `meta-apply(`$\overline{t}$`,`$\overline{l}$`,`n`)`.

`meta-reduce(`$\overline{t}$`)` takes the metarepresentation $\overline{t}$ of a term t and evaluates as follows: (a) $\overline{t}$ is converted to the term it represents; (b) this term is fully reduced using the equations in T; (c) the resulting term t_r is converted to a metaterm which is returned as a result.

`meta-apply(`$\overline{t}$`,`$\overline{l}$`,`n`)` takes the metarepresentations of a term t and of a rule label l, and a natural number and evaluates as follows: (a) $\overline{t}$ is converted to the term it represents; (b) this term is fully reduced using the equations in T; (c) the resulting term t_r is matched against all rules with label l; (d) the first n successful matches are discarded; (e) if there is a $(n+1)$-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 reduced term t'_r is converted to a metaterm and returned as a result, paired with the match used in the reduction (the operator `{_,_}` constructs the pair, and the operator `extTerm` can be used to extract the metaterm from the result).

3.2 Strategies for Nondeterministic Rewritings

We need good ways of controlling the rewriting process – which in principle could go in many undesired directions – using adequate strategies. The importance of similar mechanisms is well-known, and other languages (e.g., ELAN [2]), have built-in functionalities that deal with general forms of nondeterminism. However, the approach based on the definition of suitable internal strategies in Maude is rather general (it is parametric w.r.t. a user-definable success predicate), can be integrated with the built-in membership predicates of Maude (very important for the implementation of uniform tile system based on term structures, as shown in Section 3.3), and allows the customization of the policy adopted.

In [5] we have specified a basic internal strategy language which is able to support nondeterministic specifications, extending the strategy kernel `META-LEVEL`. Such layer provides several kinds of visit mechanisms for the trees of nondeterministic rewritings in T (e.g., breadth-first, depth-first, etc.). A strategy expression has either the form $\texttt{rewWith(}\bar{t}\texttt{,}S\texttt{)}$ 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, $\bar{t}$ is rewritten according to S (and S is reduced into the remaining strategy to be computed). In case of termination, S becomes the trivial strategy `idle`. In doing so, we assume the existence of a user-definable predicate `ok(_)`, defined over the collection of states, such that $\texttt{ok(}st\texttt{) = true}$ if st is successful and $\texttt{ok(}st\texttt{) = false}$ if st is failing.

As an example, we sketch here the depth-first visit with backtracking mechanism. The (meta)expression $\texttt{rewWith(}\bar{t}\texttt{,depthBT(}\bar{l}\texttt{))}$ means that the user wants to rewrite a term t in T using rules with label l, until a solution is found. This corresponds to the evaluation of the expression $\texttt{rewWithBT([(}\bar{t}\texttt{,0)],}\emptyset\texttt{,}\bar{l}\texttt{)}$. The function `rewWithBT` takes as arguments a sequence PL of pairs of the form $(\bar{t}, i)$, where t is a term and i is a natural number, a set of (metarepresentations of) terms TS and the metarepresentation $\bar{l}$ of a label l. The set TS represents the set of visited terms. The sequence PL contains the terms that have to be "checked". If the first argument is the empty sequence, then the function evaluates to *failure*, which means that no solution is reachable. If there is at least one pair $(\bar{t}, i)$ in the sequence, such that $\bar{t} \notin TS$ and $\texttt{ok(}t\texttt{)} \neq \texttt{false}$, then 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. If $\texttt{ok(}t\texttt{) = true}$ then t is a solution: the evaluation returns $\texttt{rewWith(}\bar{t}\texttt{,idle)}$ and the computation ends. The formal definition of such evaluation strategy is given in the Appendix, but we refer to [5] for more details and for the definition of other evaluation strategies.

3.3 Uniform Term Tile Systems

Let Σ_H and Σ_V be two (unsorted) disjoint signatures for configurations and observations. We call *term tile system* (tTS) over Σ_H and Σ_V any tile system whose configuration and observations are terms over Σ_H and Σ_V respectively. Term tile systems are quite close to the ordinary term rewriting framework, and the membership assertions and subsorting mechanism of Maude can be used (together with the internal strategies presented in Section 3.2) to model any uniform tTS $\mathcal{R}$. The idea is to define a rewrite theory $\widehat{\mathcal{R}}$, that simulates $\mathcal{R}$ as described in Section 2, exploiting the membership mechanism of Maude to distinguish the correct computations. The theory $\widehat{\mathcal{R}}$ has the poset of sorts illustrated in Fig. 1.

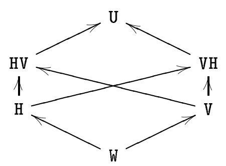

Fig. 1. The sorts of $\widehat{\mathcal{R}}$.

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 signature Σ_V applied to terms over Σ_H (similarly for VH); and the sort U contains terms over the signature $\Sigma_{H\cup V}$. As summarized above, we introduce the following operations and membership assertions, for each $h \in \Sigma_H$ and $v \in \Sigma_V$ (with h of arity n and v of arity m):

```
op h : U^n -> U . op v : U^m -> U .
vars x_1 ... x_max : U .
cmb h(x_1,...,x_n) : H iff x_1 ... x_n : H .
cmb v(x_1,...,x_m) : V iff x_1 ... x_m : V .
cmb h(x_1,...,x_n) : VH iff x_1 ... x_n : VH .
cmb v(x_1,...,x_m) : HV iff x_1 ... x_m : HV .
```

The rewriting rules of $\widehat{\mathcal{R}}$ are the stretched versions of tiles $\boldsymbol{h}\xrightarrow[u]{\boldsymbol{v}}g$ in $\mathcal{R}$.

```
rl [tile] : u(h) => g(v) .
```

The following result characterizes the correctness of our implementation.

Theorem 1. *Given a uniform tTS $\mathcal{R}$, then $\mathcal{R} \vdash \boldsymbol{h}\xrightarrow[u]{\boldsymbol{v}}g \Longrightarrow \hat{\mathcal{R}} \vdash u(\boldsymbol{h}) \Rightarrow g(\boldsymbol{v})$. Moreover, if $\hat{\mathcal{R}} \vdash u(\boldsymbol{h}) \Rightarrow t$ and t :* VH*, then $\exists g$:* H*, $\exists \boldsymbol{v}$:* V *such that $t = g(\boldsymbol{v})$ and $\mathcal{R} \vdash \boldsymbol{h}\xrightarrow[u]{\boldsymbol{v}}g$.*

4 Rewriting CCS Processes via Tiles

Milner's *Calculus for Communicating Systems* (CCS) [32] is among the better well-known and studied concurrency models. In the recent literature, several ways in which CCS can be conservatively represented in rewriting logic have been proposed [25, 36]. We present the executable implementation defined through the translation into Maude of the tile system for full CCS. This work extends the translation given in [19, 30] for a finitary version of CCS (i.e., without replicator).

4.1 CCS and its Operational Semantics

Let Δ (ranged over by α) be the set of *basic actions*, and $\bar{\Delta}$ the set of *complementary actions* (where $\bar{(\_)}$ is an involutive function such that $\Delta = \bar{\bar{\Delta}}$ and $\Delta \cap \bar{\Delta} = \emptyset$). We denote by Λ (ranged over by λ) the set $\Delta \cup \bar{\Delta}$. Let $\tau \notin \Lambda$ be a *distinguished* action, and let $Act = \Lambda \cup \{\tau\}$ (ranged over by μ) be the set of CCS actions. Then, a *CCS process* is a term generated by the following grammar:

$$P ::= nil \mid \mu.P \mid P\backslash\alpha \mid P + P \mid P|P \mid !P.$$

We let P, Q, R, ...range over the set *Proc* of CCS processes. Assuming the reader familiar with the notation, we give just an informal description of CCS algebra operators: the constant *nil* yields the *inactive* process; $\mu.P$ behaves like P but only after the execution of action μ; $P\backslash\alpha$ is the process P with actions α and $\bar{\alpha}$ inhibited by the *restriction* $\_\backslash\alpha$; $P+Q$ is the *nondeterministic sum* of processes P and Q; $P|Q$ is the parallel composition of processes P and Q; finally, $!P$ is the *replicator* of process P. The dynamic behaviour of CCS processes is usually described by a transition system, presented in the SOS style.

Remark 1. To avoid dealing with the metalevel operation of substitution, we have chosen to use the replicator $!P$ instead of the ordinary recursive operator $\mathtt{rec}\,x.P$ of CCS. Our choice does not affect the expressivenes of the calculus, because it is well known that for each agent $\mathtt{rec}\,x.P$ there exists a weak equivalent agent that can simulate it, namely $(\alpha_x.nil|!\bar{\alpha}_x.P')\backslash\alpha_x$, where α_x is a new channel name (i.e., not used by P) and P' is the process obtained by replacing each occurrence of the variable x in P by $\alpha_x.nil$.

Definition 1 (Operational Semantics). *The* CCS transition system *is given by the relation $T \subseteq Proc \times Act \times Proc$ inductively generated from the following set of axioms and inference rules (here and in the following we will omit the obvious symmetric rules for nondeterministic choice and asynchronous communication):*

$$\frac{}{\mu.P \xrightarrow{\mu} P} \qquad \frac{P \xrightarrow{\mu} Q}{P+R \xrightarrow{\mu} Q} \qquad \frac{P \xrightarrow{\mu} Q}{P\backslash\alpha \xrightarrow{\mu} Q\backslash\alpha}\ \mu \notin \{\alpha, \bar{\alpha}\}$$

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

where $P \xrightarrow{\mu} Q$ stands for $(P, \mu, Q) \in T$.

The operational meaning is that a process P may perform an action μ becoming Q if it is possible to inductively construct a sequence of rule applications to conclude that $P \xrightarrow{\mu} Q$. More generally, a process P_0 may evolve to process P_n if there exists a *computation* $P_0 \xrightarrow{\mu_1} P_1 \ldots P_{n-1} \xrightarrow{\mu_n} P_n$.

4.2 A Term Tile System for CCS

In [19] it is shown how to associate a tile system to finite CCS. We adapt their definition to settle the following tTS for the full version of the calculus.

Definition 2 (tTS for CCS). *The tTS $\mathcal{R}_{CCS}$ has the signature Σ_P of CCS processes as horizontal signature, the* action *signature $\Sigma_A = \{\mu(\_) : 1 \longrightarrow 1 \mid \mu \in Act\}$ as vertical signature, and the following basic tiles:*

$$1 \triangleleft \mu.x_1 \xrightarrow[\mu(x_1)]{x_1} x_1 \quad 2 \triangleleft x_1 + x_2 \xrightarrow[\mu(x_1)]{\mu(x_1),x_2} x_1 \quad 1 \triangleleft x_1\backslash\alpha \xrightarrow[\mu(x_1)]{\mu(x_1)} x_1\backslash\alpha \ \ (\mathit{if}\ \mu \notin \{\alpha, \bar{\alpha}\})$$

$$1 \triangleleft\ !x_1 \xrightarrow[\mu(x_1)]{\mu(x_1),x_1} x_1|!x_2 \qquad 2 \triangleleft x_1|x_2 \xrightarrow[\mu(x_1)]{\mu(x_1),x_2} x_1|x_2 \qquad 2 \triangleleft x_1|x_2 \xrightarrow[\tau(x_1)]{\lambda(x_1),\bar{\lambda}(x_2)} x_1|x_2$$

The tile for the action prefix has been already discussed in Section 2. As additional examples, we briefly comment the tile for left nondeterministic choice, and the tile for the replicator, also depicted below.

$$
\begin{tikzcd}
\underline{2} \arrow[r, "x_1+x_2"] \arrow[d, "{\mu(x_1),x_2}"'] & \underline{1} \arrow[d, "\mu(x_1)"] \\
\underline{2} \arrow[r, "x_1"'] & \underline{1}
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
\underline{1} \arrow[r, "!x_1"] \arrow[d, "{\mu(x_1),x_1}"'] & \underline{1} \arrow[d, "\mu(x_1)"] \\
\underline{2} \arrow[r, "x_1|!x_2"'] & \underline{1}
\end{tikzcd}
$$

The meaning of the first tile is that the action μ (i.e., the effect $\mu(x_1)$) can be executed by the sum of two subprocesses (i.e., from the initial configuration) if the left subprocess (i.e., the variable x_1 in the initial input interface) can perform the action μ (i.e., the trigger $\mu(x_1)$), evolving to the same subprocess (i.e., the variable x_1 in the final input interface) that will be reached by the nondeterministic sum after such rewriting (i.e., the final configuration x_1). Notice that we can handle the garbaging of the discarded process in the easiest way, using a discharger to throw it away (thanks to auxiliary structure and tile that were not present in [30]). In our notation, this corresponds to not to mention a variable of the input interface (i.e., variable x_2 of the final input interface).

The second tile can be read in a similar way. The relevant thing is that its trigger refers to the same variable twice. This is not allowed in the model proposed by Gadducci and Montanari, where the structure of observations is just a freely generated strict monoidal category (i.e., it is not cartesian). Such duplication is necessary because in the final configuration we must refer both the process P linked to the variable of the initial input interface and the process Q reached by P after the firing of action μ, which acts as trigger for the rewriting. Hence, tTS can deal with more general format than those considered in [19], and in particular, can embed all the expressive power of full CCS.

Analogously to [19], the following result establishes the correspondence between the ordinary SOS semantics for CCS, and the sequents entailed by $\mathcal{R}_{CCS}$.

Proposition 1. *The tTS $\mathcal{R}_{CCS}$ is uniform, and for any CCS agents P and Q, and action μ:* $P \xrightarrow{\mu} Q \in T \iff \mathcal{R}_{CCS} \vdash 0 \triangleleft P \xrightarrow[\mu(x_1)]{} Q$.

4.3 From Tiles for CCS to Rewrite Rules for CCS

By Proposition 1, it follows immediately that a suitable implementation of $\mathcal{R}_{CCS}$ can be obtained by taking the rewriting theory $\hat{\mathcal{R}}_{CCS}$ defined in Section 3.3, and by defining a suitable success predicate for the metastrategies of Section 3.1. Therefore the rules of $\hat{\mathcal{R}}_{CCS}$ (all labelled by `tile`) are:

$$\mu(\mu.x_1) \Rightarrow x_1 \qquad \mu(x_1 + x_2) \Rightarrow \mu(x_1) \qquad \mu(x_1 \backslash \alpha) \Rightarrow \mu(x_1) \backslash \alpha \text{ (if } \mu \neq \alpha, \bar{\alpha})$$

$$\mu(!x_1) \Rightarrow \mu(x_1)|!x_1 \qquad \mu(x_1|x_2) \Rightarrow \mu(x_1)|x_2 \qquad \tau(x_1|x_2) \Rightarrow \lambda(x_1)|\bar{\lambda}(x_2)$$

and the success predicate is defined by `ceq ok(`t`) = true if` t `: VH .`

Corollary 1. *For any CCS processes P and Q, and action μ:*

$$P \xrightarrow{\mu} Q \Longleftrightarrow \hat{\mathcal{R}}_{CCS} \vdash \mu(P) \Rightarrow Q.$$

A typical (meta)query is `rewWith(`$\overline{\mu(P)}$`,depthBT(`$\overline{\texttt{tile}}$`))`, where $\overline{\mu(P)}$ is the metarepresentation of the test $\mu(P)$ that can be used to see if the CCS process P can perform a transition labelled by μ. Then, the system tries to rewrite $\overline{\mu(P)}$ in all possible ways, until a solution of type `VH` is found (if it exists).

Example 1. Let us consider the CCS process $(\alpha.nil + \beta.nil)\backslash\alpha$. If the rules are applied without any metacontrol, then a possible computation for the test $\beta((\alpha.nil+\beta.nil)\backslash\alpha)$ is: $\beta((\alpha.nil+\beta.nil)\backslash\alpha) \Rightarrow \beta(\alpha.nil+\beta.nil)\backslash\alpha \Rightarrow \beta(\alpha.nil)\backslash\alpha$. Such computation ends in a state that is not a solution (in fact it is the composition of the horizontal arrow $\alpha.nil$, followed by the vertical arrow $\beta(x_1)$, followed by the horizontal arrow $x_1\backslash\alpha$) and that cannot be further rewritten. Therefore, it is discarded in the meta-controlled computation, and the only possible result `rewWith(Q,idle)`, where `Q` is the metarepresentation of $nil\backslash\alpha$, is returned.

5 Concluding Remarks

This work presents a general methodology for the specification and execution of process calculi via term tile systems, which is part of our ongoing research on the relations between tile logic and rewriting logic. We have defined some general metastrategies for simulating tile system specifications on a rewriting machinery equipped with reflective capabilities. We have implemented such strategies in Maude, and have experimented their application to the case study of full CCS (but more complex systems can be represented as well in our format).

Our general methodology for modelling process calculi (and more generally, reactive systems), can be summarized by the following steps: (1) define a tile model of computation of the given system, employing adequate mathematical structures to represent configurations and observations in such a way that the intrinsic modularity and synchronization mechanism of tiles are fully exploited; (2) translate the tiles into rewrite rules; (3) define, if necessary, a notion of successful computation (if the system is uniform, this can be done by just looking at the actual term reached); (4) compute at the metalevel, using the internal strategies that discard wrong computations, until a successful answer is reached. This procedure has been fully illustrated for process tile logic in [6] by the example of located CCS, and for term tile logic in the present paper by the example of full CCS. For each model, we have tested the computations of simple processes. Our experiments are encouraging, because Maude seems to offer a good trade-off between rewriting kernel efficiency and layer-swapping management (from terms to their metarepresentations and viceversa).

Acknowledgements We would like to thank Paolo Baldan and the anonymous referees for useful comments.

References

1. B. Bloom, S. Istrail, and A.R. Meyer, Bisimulation can't be Traced, *Journal of the ACM* **42**(1), 232–268 (1995).
2. P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek, ELAN: A logical framework based on computational systems, in [28].
3. R. Bruni, Tile Logic for Synchronized Rewriting of Concurrent Systems, PhD Thesis, Department of Computer Science, University of Pisa, forthcoming.
4. R. Bruni, J. Meseguer, and U. Montanari, Process and Term Tile Logic, Technical Report SRI-CSL-98-06, SRI International (1998).
5. R. Bruni, J. Meseguer, and U. Montanari, Internal Strategies in a Rewriting Implementation of Tile Systems, in [23].
6. R. Bruni, J. Meseguer, and U. Montanari, Implementing Tile Systems: some Examples from Process Calculi, in *Proc. ICTCS'98*, World Scientific, 168–179 (1998).
7. R. Bruni and U. Montanari, Zero-Safe Nets: Comparing the Collective and the Individual Token Approaches, *Information and Computation*, to appear.
8. M. Clavel, *Reflection in General Logics and in Rewriting Logic with Applications to the Maude Language*, PhD Thesis, Universidad de Navarra (1998).
9. M.G. Clavel, F. Duran, S. Eker, P. Lincoln, and J. Meseguer, An Introduction to Maude (Beta Version), SRI International (1998).
10. M. Clavel and J. Meseguer, Reflection and Strategies in Rewriting Logic, in [28].
11. M. Clavel and J. Meseguer, Axiomatizing Reflective Logics and Languages, in *Proceedings Reflection'96*, San Francisco, USA, 263–288 (1996).
12. M. Clavel and J. Meseguer, Internal Strategies in a Reflective Logic, in *Proc. of the CADE-14 Workshop on Strategies in Automated Deduction*, 1–12 (1997).
13. G. Denker, J. Meseguer, and C. Talcott, Protocol Specification and Analysis in Maude, in *Proc. Workshop on Formal Methods and Security Protocols* (1998).
14. C. Ehresmann, *Catégories Structurées*: I and II, Ann. Éc. Norm. Sup. 80, Paris (1963), 349–426; III, Topo. et Géo. diff. V, Paris (1963).
15. G.L. Ferrari and U. Montanari, A Tile-Based Coordination View of Asynchronous Pi-Calculus, in *Proc. MFCS'97*, Springer *LNCS* **1295**, 52–70 (1997),
16. G.L. Ferrari and U. Montanari, Tiles for Concurrent and Located Calculi, in *Proceedings of EXPRESS'97*, *ENTCS* **7** (1997).
17. F. Gadducci, *On the Algebraic Approach to Concurrent Term Rewriting*, PhD Thesis TD-96-02, Department of Computer Science, University of Pisa (1996).
18. F. Gadducci and U. Montanari, Enriched Categories as Models of Computations, in *Proc. ITCS'95*, World Scientific, 1–24 (1996).
19. F. Gadducci and U. Montanari, The Tile Model, in *Proof, Language and Interaction: Essays in Honour of Robin Milner*, MIT Press, to appear.
20. J.F. Groote, and F. Vaandrager, Structured Operational Semantics and Bisimulation as a Congruence, *Information and Computation* **100**, 202–260 (1992).
21. K.G. Larsen, and L. Xinxin, Compositionality Through an Operational Semantics of Contexts, in *Proc. ICALP'90*, *LNCS* **443**, 526–539 (1990).
22. F.W. Lawvere, Functorial Semantics of Algebraic Theories, in *Proc. National Academy of Science* **50**, 869–872 (1963).
23. C. Kirchner, H. Kirchner, Ed., *Proc. 2nd WRLA'98*, *ENTCS* **15** (1998).
24. N. Martí-Oliet and J. Meseguer, General Logics and Logical Frameworks, in: D. Gabbay, Ed., *What is a logical system?*, Oxford University Press (1994).
25. N. Martí-Oliet and J. Meseguer, *Rewriting Logic as a Logical and Semantic Framework*, SRI Technical Report, CSL-93-05 (1993). To appear in D. Gabbay, Ed., *Handbook of Philosophical Logic*, Kluwer Academic Publishers.

26. J. Meseguer, Conditional Rewriting Logic as a Unified Model of Concurrency, *TCS* **96**, 73–155 (1992).
27. J. Meseguer, Rewriting Logic as a Semantic Framework for Concurrency: A Progress Report, in *Proc. CONCUR'96*, Springer *LNCS* **1119**, 331-372 (1996).
28. J. Meseguer, Ed., *Proc. 1st International Workshop on Rewriting Logic and Applications*, *ENTCS* **4** (1996).
29. J. Meseguer, Membership Equational Logic as a Logical Framework for Equational Specification, in *Proc. 12th WADT'97*. Springer *LNCS* **1376**, 18–61 (1998).
30. J. Meseguer and U. Montanari, Mapping Tile Logic into Rewriting Logic, in *Proc. 12th WADT'97*, Springer *LNCS* **1376**, 62–91 (1998).
31. J. Meseguer and C. Talcott, Using Rewriting Logic to Interoperate Architectural Description Languages (I and II), Lectures at the Santa Fe and Seattle DARPA-EDCS Workshops (1997).
32. R. Milner, *Communication and Concurrency*, Prentice-Hall (1989).
33. U. Montanari and F. Rossi, Graph Rewriting, Constraint Solving and Tiles for Coordinating Distributed Systems, *Applied Categorical Structures*, to appear.
34. U. Montanari and C. Talcott, Can Actors and pi-Agents Live Together?, in *Proceedings HOOTS'97*, *ENTCS* **10** (1998).
35. G. Plotkin, A Structural Approach to Operational Semantics, Technical Report DAIMI FN-19, Computer Science Department, Aarhus University (1981).
36. P. Viry, *Rewriting Modulo a Rewrite System*, Technical Report TR-95-20, Department of Computer Science, University of Pisa (1995).

Appendix. Depth-First Visit with Backtracking

The depth-first strategy with backtracking is defined below in a self-explanatory notation: the Maude-like syntax has been extended by using some ordinary symbols (e.g., `[]`, $\{\}$, $\cup$, $\in$, `0`, `succ`) to deal with lists, sets, natural numbers, etc.

```
var t̄ : Term .  var l̄ : Label .  var n : Nat .
var TS : TermSet .  var PL : TermList .
eq rewWith(t̄,depthBT(l̄)) = rewWithBT([(t̄,0)],∅,l̄) .
eq rewWithBT([],TS,l̄) = failure .
eq rewWithBT([(t̄,n)],TS,l̄) = if t̄∈TS then failure
   else if meta-reduce('ok[t̄]) == 'true then rewWith(t̄,idle)
        else if meta-reduce('ok[t̄]) == 'false then failure
             else if meta-apply(t̄,l̄,n) == error* then failure
                  else rewWithBT([(extTerm(meta-apply(t̄,l̄,n)),0),
                       (t̄,succ(n))],{t̄}∪TS,l̄) fi fi fi fi .
eq rewWithBT([(t̄,n),PL],TS,l̄) = if t̄∈TS then rewWithBT(TL,TS,l̄)
   else if meta-reduce('ok[t̄]) == 'true then rewWith(t̄,idle)
        else if meta-reduce('ok[t̄]) == 'false
             then rewWithBT(PL,{t̄}∪TS,l̄)
             else if meta-apply(t̄,l̄,n) == error*
                  then rewWithBT(PL,{t̄}∪TS,l̄)
                  else rewWithBT([(extTerm(meta-apply(t̄,l̄,n)),0),
                       (t̄,succ(n)),PL],{t̄}∪TS,l̄) fi fi fi fi .
```