An abstract machine for concurrent modular systems: CHARM*

Andrea Corradini, Ugo Montanari and Francesca Rossi

Department of Computer Science, University of Pisa, Corso Italia 40, 56100 Pisa, Italy

Abstract


CHARM (for concurrency and hiding in an abstract rewriting machine) is an abstract machine which allows one to model naturally the behavior of distributed systems consisting of a collection of processes-sharing variables. CHARM is equipped with a clean operational semantics based on term rewriting over a suitable algebra, and it exhibits a sophisticated treatment of concurrency and modularity, which is obtained through the partition of each state into a global and a local part. To show the expressiveness and generality of this abstract machine, three relevant computational formalisms are mapped onto the CHARM framework: graph grammars, concurrent constraint programming, and place/transition Petri nets.

1. Introduction

Various formalisms have been proposed in the last decades for describing and specifying concurrent programs and distributed systems. Among them we recall Petri nets [21], CCS [16], CSP [9], the chemical abstract machine [1], graph grammars [8], and concurrent constraint programming [23]. The high number of such formalisms shows the need for a unifying framework, which should be able to capture the essence of concurrent computations. Such a framework should be general enough so that most of the formalisms already proposed could be embedded in it, but it should be also expressive enough to be able to prove interesting properties about it. We believe that a reasonable balance of generality and expressiveness can be enjoyed by a formalism in which it is possible to express in a simple way both concurrency and...
modularity. In fact, such notions are fundamental in order to describe how concurrent systems interact, synchronize, evolve, compose, or embed in other systems.

In this paper we propose an abstract machine, called CHARM (for concurrency and hiding in an abstract rewriting machine), which is intended to satisfy the above need for a unifying framework for concurrent programming. A CHARM is essentially a structured transition system \[3,5\], i.e., a transition system having an algebraic structure on both states and transitions.

States of a CHARM represent systems consisting of collections of processes interacting through shared variables. Each state is partitioned into a global (i.e., visible) and a local (i.e., hidden) part. The global items of a system are those which allow for interaction with other systems, while the local items are not accessible from other systems and thus are private to the system under consideration.

Transitions of the machine are rewrite rules described by pairs of states with identical global parts. Such a global part represents the part being preserved by the application of the rule (i.e., the part that the rule, being local to the rewritten state, cannot change).

Modularity in this framework comes from the fact that a system (and its evolution) can be obtained by composing its subsystems (and the corresponding evolutions), which in turn can be specified independently. On the other hand, concurrency comes from the fact that different subsystems may evolve in parallel without any need of global synchronization.

On the one hand, the partition of each state into a global and a local part is interesting for modularity reasons, since it makes the operation of parallel composition of states not trivial. On the other hand, it allows a degree of concurrency higher than the one provided, for example, by the chemical abstract machine or by Petri nets. In fact, two (or more) transitions may be applied in parallel not only when the subsystems they affect are disjoint, but also when their intersection is preserved by both of them (i.e., when it belongs to both their global parts).

The technique used for the formal definition of the CHARM follows the algebraic approach introduced in [15] for Petri nets, and further developed for structured transition systems in [3,5] and for concurrent rewriting systems in [14]. This approach is characterized by the fact that states and transitions of a system have the same algebraic structure, which can also be consistently extended to computations. This algebraic construction equips a system with a calculus of computations, which provides a rich modular proof system.

To show the expressiveness and generality of the CHARM computational model, we describe how the classical algebraic approach to graph grammars [8], which has been widely used for algebraic system specification, can be implemented in our framework. Also, the CHARM provides a very natural interpretation of concurrent constraint programming [24,25], since the sharing of variables and the possibility of "asking" (i.e., testing while preserving) a constraint are the two main notions in such a paradigm. Note that the ability to express concurrent constraint programming in the CHARM framework is quite significant, since such a paradigm is already very
An abstract machine for concurrent modular systems: CHARM

general and subsumes many widely used programming paradigms such as logic programming [13], constraint logic programming [11], and concurrent logic programming [22]. Finally, we also show how to model classical P/T Petri nets, as algebraically described in [15], in the CHARM framework.

We first give an informal description of the CHARM in Section 2, and then we present the formal theory underlying our approach by presenting, in Section 3, an algebra for the states of the machine and also for the rewriting rules. We then address the relationship between the CHARM and graph grammars, concurrent constraint programming, and Petri nets in Sections 4, 5, and 6, respectively.

This paper is a revised and extended version of [6]. The major changes with respect to [6] are some new results in Section 3, a more formal treatment of the modeling of graph grammars and cc programs in Sections 4 and 5, respectively, and a new section (Section 6) which shows how to model Petri nets within the CHARM formalism. Section 4 is strongly related to [4], whose results are here restated and extended in the more general framework of the CHARM.

2. An informal description of the abstract machine

In this section we informally give the main ideas underlying the design of the abstract machine we propose (called CHARM in the rest of the paper), and we also emphasize some of its advantages with respect to other transition systems and/or machines which have already been proposed in the literature for describing concurrent systems.

Each state of a CHARM is a (distributed) system, i.e., a collection of processes and a set of (possibly shared) variables where each process is connected to a subset of the variables. This notion of state is very general. In fact, we do not assume any requirement on the structure of processes and variables, which may thus be interpreted in various way. For example, processes may also be thought of as predicates or constraints or relations, and variables may represent communication channels or shared data structures. It is important to note that many of the approaches proposed to represent the evolution of concurrent systems [1,21] cannot directly model the sharing of variables, since a state is simply a multiset of processes.

Each state is partitioned into a local part and a global part, and thus will be informally denoted in the rest of this section by the pair \( S = (G, L) \), where \( G \) stands for the global part and \( L \) for the local part. In terms of distributed systems, we may think of the local (global) part as the hidden (visible) set of variables and processes. Intuitively, the local items are those whose identity is known only to the system under consideration, while the global ones are the interface of the system with the rest of the world and thus may be known by other systems as well. For example, such an interface may contain common data structures, as well as processes implementing services of global utility.
Clearly, the local and the global parts of a state may have some interaction. In fact, a local process may have access to global resources. However, the converse is not allowed, i.e., if a process is global, then all the resources it has access to must be global as well. States satisfying this property are called well formed. This reflects the intuition that if a process is visible, then all its resources are accessible through the process itself. Therefore, they must be specified as global items.

States can be built from smaller states, starting from a collection of elementary states, which we call atoms. An atom is either a variable, or a process together with all its variables. Then the parallel composition of two states (given by the operator $\parallel$) is defined as the state whose global part is the set union of their global parts, and whose local part is the disjoint union of their local parts. This reflects the fact that, as we said above, the identity of the items in the global parts of the two states are known by both of them, so that items with the same name should be identified. Another operator over states allows an item to be moved from the global to the local part. Such an operator is called hiding, and will be denoted by $\backslash$. Parallel composition and hiding, together with a renaming operator (denoted by $[\phi]$), define an algebra whose terms are subject to suitable axioms, which reflect the intuitive meaning of the operators just described.

Such an algebra is parametric with respect to the collection of elementary states. States of a CHARM are defined as equivalence classes of terms of the algebra w.r.t. the axioms.

Although states are closed w.r.t. all the operators, the kind of modularity we are able to model is in some sense "flat". In fact, as we will see in Section 3, every term of the algebra of states is equivalent to a term consisting of the parallel composition of a collection of atoms, where some items are made local by the hiding operator. This reflects the intuition that states can be regarded as collections of processes with a global and a local part.

It must be stressed that the hiding operator of the CHARM is not easily comparable with similar operators in process description languages (such as restriction in CCS and hiding in CSP). Hiding is used here to make certain global items local and not to enforce a hierarchical structure on systems. This will be clear when we will introduce the algebra of transitions (Definition 3.9) of a CHARM, where the inference rule for the hiding operator clearly differs from the corresponding rules in CCS and CSP.

Given an algebra of states $\mathcal{S}$, a specific CHARM $\mathcal{M}$ over $\mathcal{S}$ is defined as a collection of rewrite rules over states, which describe the dynamic behavior of $\mathcal{M}$. Every rewrite rule $R : S \rightarrow S'$ maps its left-hand side state $S = (G, L)$ to its right-hand side $S' = (G, L')$, both having the same global part $G$, which is also called the global part of $R$. Note that both $S$ and $S'$ can be arbitrary states, not necessarily atoms. The graphical representation of a rewrite rule can be seen in Fig. 1. The idea is that $L$ can be deleted and $L'$ can be generated provided that $G$ is present. Thus, our notion of rewriting is context-dependent, the global part of a rule playing the role of the context.

It is worth stressing that the global part $G$ is not affected by the application of $R$, but it is simply tested for existence. The presence of a global part in a rule is necessary for
composing the behaviors of different subsystems in a correct way. Moreover, it is also useful for defining a satisfactory truly concurrent semantics for CHARMs, since it minimizes the causal dependencies among rewrite rules.

Intuitively, the global part $G$ of $R$ contains those items (processes and variables) which are needed for the transformation of the state to take place, but which are not changed by the rewrite rule. For example, we may want to do some operation only if some data structure contains some given information. In this case, the data structure is considered to be global and thus it is not affected by the rewrite rule. The global items of $R$ are also used to embed the state $S$ into a larger state $Q$; this is necessary whenever one wants to describe the evolution of $Q$ in terms of the evolution of $S$.

It is important at this point to note that, unlike our approach, many transition systems or abstract machines proposed in the literature (such as Petri nets and the chemical abstract machine) are context-independent, i.e., they cannot distinguish between the situation where some item is preserved by a rewrite rule, and the one where the same item is deleted and then generated again. For example, the rule “$a$ rewrites to $b$ only if $c$ is present” must be represented in those formalisms as $\{a,c\} \rightarrow \{b,c\}$, which also represents the rule “$a$ and $c$ rewrite to $b$ and $c$”. On the other hand, in these formalisms the global part of a rule is not needed to model the embedding of substates into states because, since states are simply sets or multisets, such embedding is trivial.

Some other formalisms explicitly consider the issue of context-dependent rewriting, and allow one to indicate formally which items should be present for the application of a rule, but are not affected by it. For example, in the algebraic approach to graph grammars the role of the context is played by the so-called “gluing graph” of a graph production, while in concurrent constraint programming the items, the presence of which must be tested, are explicitly mentioned by the use of the “ask” primitive. In later sections of this paper we will explore the relationship between the CHARM and these two formalisms, as well as with place/transition Petri nets.

Since the global items of a state are used to interface it with other states, a closed system $Q$ (i.e., a system which is not supposed to be composed further) is represented by a state with no global part, i.e., $Q = (\emptyset, L_0)$. Now consider a rule $R$ from $S = (G, L) \rightarrow S' = (G, L')$. If $S$ is a substate of $Q$, we may apply $R$ to $Q$. The result of this application is a state $Q'$ obtained by replacing the subpart $L$ of $Q$ with $L'$. In this setting, the role of the global part $G$ of $R$ is to specify the embedding of $L$ and $L'$ in $Q$ and $Q'$, respectively. In fact, if $G$ were empty (i.e., if the rule $R$ had no global part), then $L$ would be
completely isolated within \( Q \); i.e., no process of \( L \) could share any variable with any other process in \( Q \). The graphical representation of the application of \( R \) to \( Q \) can be seen in Fig. 2.

It is worth noting that the substate \( G \) is global in \( R \), but it is local in \( Q \). Therefore, although \( R \) cannot modify \( G \), there may exist another rule \( R' \), rewriting some other substate of \( Q \), for which \( G \) is local (in its left-hand side). In this case an application of \( R' \) to \( Q \) can modify \( G \).

The above construction describes the application of a single rewrite rule to a CHARM state. However, this mechanism is intrinsically concurrent, in the sense that many rewrite rules may be applied in parallel to a state provided that their occurrences do not interfere. In particular, if the occurrences of the rules are pairwise disjoint, we have a degree of parallelism which is also supported by many other models of concurrent computation, such as Petri nets, the Chemical abstract machine, and the concurrent rewriting of [14]. However, our approach provides a finer perception of the causal dependencies among rewrite rules, because rules whose occurrences in a state are not disjoint but intersect only on their global parts can be considered not to depend on each other, and thus can be applied concurrently. This can intuitively be explained by noting that such rules interact only on items which are preserved by all of them. This corresponds to what is called "parallel independence" of production applications in the algebraic theory of graph grammars, which can in fact be faithfully implemented within the CHARM framework, as we will see in Section 4.

The application of one or more rules of a CHARM is modeled, from a technical point of view, by extending the algebra of states (including the axioms) to the rules (for similar approaches in the case of Petri nets or structured transition systems, see [14,3,15]). This is possible because each rule has an associated global part, just like states. The resulting algebra, called the algebra of transitions, contains as elements all the rewrite rules of the abstract machine, an identity rule \( S : S \rightarrow S \) for each state \( S \), and is closed w.r.t. the parallel composition, hiding, and substitution operations. The left- and right-hand sides of a transition (i.e., of an element of the algebra of transitions) faithfully reflect the structure of the transition itself. For example, if \( R : S \rightarrow Q \) and \( R' : S' \rightarrow Q' \) are two rewrite rules, then \( R \parallel R' : S \parallel S' \rightarrow Q \parallel Q' \) is a new parallel transition. As a consequence of this fact, if a rewrite rule \( R \) can be applied to a state \( S \), then any state containing \( S \) can also be transformed through a suitable transition containing \( R \).

![Fig. 2. The application of a rewrite rule.](image-url)
Informally, this can be considered as a meta-rule governing the behavior of a CHARM, and directly corresponds to the so-called “membrane law” of the chemical abstract machine. Note also that, like rewrite rules, transitions preserve the global part of the state they are applied to.

It is now easy to understand how the matching and the embedding of substates into global states can be formally handled. In fact, consider again a state \( Q = (\emptyset, L_Q) \) without a global part, and a rule \( R \) from \( S = (G, L) \) to \( S' = (G, L') \). Matching \( (G, L) \) to \( Q \) means finding a state \( H = (G, L') \) such that \( (S || H) \setminus X = Q \), where \( X \) contains all the items of \( G \). Intuitively, we have \( L' = L_Q - (G \cup L) \) and thus it is unique. Similarly, the new state \( Q' \) will be obtained as \( Q' = (S' || H) \setminus X \). Note that all the items in \( G \) are shared between \( S \) and \( H \) and, after the rewriting, between \( S' \) and \( H \). If we interpret \( H \setminus X \) as “the part of state \( Q \) not modified by the application of \( R \)” (i.e., \( H \setminus X = (\emptyset, L_Q - L) \)), we see how our construction is able to embed the right-hand side \( S' \) of \( R \) into \( H \setminus X \) to get \( Q' \).

A computation of a CHARM is a sequence of transitions, starting from a given initial state. Since each transition preserves the global part of its left-hand side state, the final state of a computation has the same global part as the initial state. Thus, every computation is naturally associated with a global part as well. As for transitions, this will allow us to define an algebra of computations, having the same operations as the algebra of states, plus a sequential composition operation denoted by ;. The elements of the algebra of computations are subject to the same axioms as for states, plus some axioms stating that all the operations distribute over sequential composition. Thus, we have a rich language of computations, where some computations can be proved to be equivalent by using the axioms.

The interesting fact is that the resulting algebra of computations allows one to relate the global evolution of a system to the local behavior of its subsystems. For example, we consider the system \( P = (S || S') \setminus x \), where the two subsystems \( S \) and \( S' \) cooperate through the common global variable \( x \) which is hidden by the use of the \( \setminus x \) operator. Furthermore, consider the computations \( \rho: S \Rightarrow Q \) and \( \rho': S' \Rightarrow Q' \) for \( S \) and \( S' \), respectively. Then by using the algebra of computations it is possible to construct the computation \( \sigma = (\rho || \rho') \setminus x \) which models the evolution of the system \( P \), i.e., \( \sigma: P \Rightarrow P' \), where \( P' = (Q || Q') \setminus x \).

Another relevant advantage of the definition of the algebra of computations of a CHARM is the possibility of providing a truly concurrent semantics in a natural way. In fact, computations differing only in the order in which independent rewrite rules are applied fall within the same equivalence class. For example, considering again the computations \( \rho \) and \( \rho' \) introduced above, we have \( \rho || \rho' = (\rho || S'): (Q || \rho') \), where \( S' \) and \( Q \) stay for the identity computations on such states. This means that \( \rho \) and \( \rho' \) can be performed either in parallel or sequentially, and the two resulting computations are equivalent. Such equivalence classes of computations could therefore provide a truly concurrent semantics for all the formalisms which can be modeled within the CHARM framework; however, this goes beyond the scope of this paper, and it is left as a topic for future research.
3. Formal definitions

In this section we present the formal description of a CHARM following the informal presentation given in the previous section. After introducing the algebra of states, a CHARM will be defined as a collection of rewrite rules over this algebra which preserve the global part of a state. Next we will introduce the algebra of transitions and the algebra of computations of a CHARM.

The states of a CHARM are going to be defined as equivalence classes of terms of an algebra $\mathcal{S}$, which is parametric w.r.t. a fixed tuple $(\mathcal{P}, \mathcal{V}, \mathcal{I}_p, \mathcal{I}_v, pt, vt)$, where the first four elements are disjoint, possibly infinite collections, called process instances, variables, process types, and variable types, respectively, and $pt : \mathcal{P} \rightarrow \mathcal{I}_p$ and $vt : \mathcal{V} \rightarrow \mathcal{I}_v$ are two typing functions for process instances and variables, respectively. The terms of algebra $\mathcal{S}$ include some elementary terms, called “atoms”, and are closed with respect to the operators of parallel composition, renaming, and hiding. The intuitive meaning of these operators is reflected by the axioms listed below in Definition 3.3.

**Definition 3.1 (Terms of the algebra of states).** Let $\mathcal{P}$ be a set of process instances (ranged over by $p, q, \ldots$), $\mathcal{V}$ be a set of variables (ranged over by $v, z, \ldots$), $\mathcal{I}_p$ be a set of process types, $\mathcal{I}_v$ be a set of variable types, and $pt : \mathcal{P} \rightarrow \mathcal{I}_p$ and $vt : \mathcal{V} \rightarrow \mathcal{I}_v$ be two typing functions which associate a unique type to each process instance and variable. Each $x \in (\mathcal{P} \cup \mathcal{V})$ is called an item. Then the terms of the algebra of states $\mathcal{S}$ are generated by the following grammar:

$$S ::= 0 \mid v \mid p(v_1, \ldots, v_n) \mid S \parallel S \mid S[\Phi] \mid S \setminus x,$$

where

- $v, v_1, \ldots, v_n \in \mathcal{V}$;
- $p \in \mathcal{P}$;
- $\parallel$ is called parallel composition;
- $\Phi$ is a (finite domain) substitution, i.e., a function $\Phi : (\mathcal{P} \cup \mathcal{V}) \rightarrow (\mathcal{P} \cup \mathcal{V})$ such that
  - $\Phi(\mathcal{V}) \subseteq \mathcal{V}$ and $\Phi(\mathcal{P}) \subseteq \mathcal{P}$,
  - $vt(\Phi(v)) = vt(v)$ for any $v \in \mathcal{V}$,
  - $pt(\Phi(p)) = pt(p)$ for any $p \in \mathcal{P}$, and
  - the set of items for which $x \neq \Phi(x)$ is finite;
- $\setminus x$ is an item ($\setminus x$ is called a hiding operator).

Terms of the form $0, v$, or $p(v_1, \ldots, v_n)$ are called atoms. Sometimes, in order to make types explicit, we will use the redundant form $v : t$ (where $t = vt(v)$) and $p(v_1 : t_1, \ldots, v_n : t_n) : t$ (where $t_i = vt(v_i)$ for all $1 \leq i \leq n$, and $t = pt(p)$), instead of $v$ and $p(v_1, \ldots, v_n)$, respectively.

Intuitively, term $0$ will denote the empty system, $v$ is the system containing only one variable (or communication channel), and $p(v_1, \ldots, v_n)$ represents a system with one process which has access to $n$ variables. According to the informal description of
Section 2, all atoms represent systems having no local part. Items can be "moved" from the global to the local part of a system by the use of the hiding operator, in fact, \( S \setminus x \) represents the same system as \( S \), except that item \( x \) is local.

The term \( S_1 \parallel S_2 \) represents the parallel composition of system \( S_1 \) and system \( S_2 \). Since the items of the global parts of \( S_1 \) and \( S_2 \) are potential interfaces between the two systems (as discussed in the previous section), the parallel composition should act as set-theoretical union on the global parts (in this way two occurrences of the same global variable or process instance in \( S_1 \) and \( S_2 \) can be identified), but as disjoint union on the local parts, because they are private for each subsystem.

Finally, \( S[\Phi] \) is the system obtained from state \( S \) by changing its global variables and process instances according to substitution \( \Phi \). Note that, by definition, substitution \( \Phi \) does not need to be injective, and moreover it is required to preserve both variable and process types.

The interpretation of the operators of algebra \( \mathcal{F} \) just sketched out constitutes an informal justification for the axioms of Definition 3.3, which will state when two terms of algebra \( \mathcal{F} \) should be intended to represent the same system. Before introducing the conditional axioms, we need a few auxiliary definitions on terms.

**Definition 3.2 (Free items, concrete and closed terms).** Given a term \( S \in S \) (i.e., \( S \) is generated by the grammar of Definition 3.1), its set of free items \( \mathcal{F}(S) \) is inductively defined as

- \( \mathcal{F}(0) = \emptyset \);
- \( \mathcal{F}(v) = \{ v \} \);
- \( \mathcal{F}(p(v_1, \ldots, v_n)) = \{ p, v_1, \ldots, v_n \} \);
- \( \mathcal{F}(S_1 \parallel S_2) = \mathcal{F}(S_1) \cup \mathcal{F}(S_2) \);
- \( \mathcal{F}(S[\Phi]) = \Phi(\mathcal{F}(S)) = \{ \Phi(x) | x \in \mathcal{F}(S) \} \);
- \( \mathcal{F}(S \setminus x) = \mathcal{F}(S) - \{ x \} \).

A term \( S \) is closed iff \( \mathcal{F}(S) = \emptyset \). A term is concrete if no free variable appearing in any of its subterms is restricted. Formally,

- all atoms are concrete;
- \( S_1 \parallel S_2 \) is concrete if both \( S_1 \) and \( S_2 \) are concrete;
- \( S[\Phi] \) is concrete if \( S \) is concrete;
- \( S \setminus x \) is concrete if \( S \) is concrete and \( x \notin \mathcal{F}(S) \).

Intuitively, the free items of a term \( S \) are the process instances and the variables of the global part of the system represented by \( S \). Thus, a closed term represents a system with no global part, while a concrete term corresponds to a system where everything is global.

We now present the axioms that state the relevant algebraic properties of the operators of Definition 3.1. The states of a CHARM over the algebra \( \mathcal{F} \) will be defined as equivalence classes of terms of \( \mathcal{F} \) with respect to these axioms.
Definition 3.3 (Axioms of the algebra of states). The terms of algebra $\mathcal{S}$ introduced in Definition 3.1 are subject to the following conditional axioms.

ACI: $(S_1 \parallel S_2) \parallel S_3 = S_1 \parallel (S_2 \parallel S_3)$, $S_1 \parallel S_2 = S_2 \parallel S_1$, $S \parallel 0 = S$,

ABS: $p(v_1, \ldots, v_n) \parallel v_i = p(v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n)$ for $1 \leq i \leq n$; $S \parallel S = S$ if $S$ is concrete

MAP: $p(v_1, \ldots, v_n)[\Phi] = \Phi(p(\Phi(v_1), \ldots, \Phi(v_n)))$; $v[\Phi] = \Phi(v)$; $0[\Phi] = 0$,

COMP: $S[\varphi][\psi] = S[\psi \varphi]$, $\mathcal{S} \times \mathcal{Y} = \mathcal{S} \times \mathcal{Y}$,

EXC: $S \setminus x \setminus y = S \setminus y \setminus x$,

EL: $S \setminus x = S$ if $x$ is not free in $S$,

DIS: $(S_1 \parallel S_2)[\Phi] = S_1[\Phi] \parallel S_2[\Phi]$,

FAC: $S_1 \setminus x \parallel S_2 = (S_1 \parallel S_2) \setminus x$ if $x$ is not free in $S_2$,

SWAP: $(S \setminus x)[\Phi] = S[\Phi] \setminus \Phi(x)$ if $\forall y \in \mathcal{S}(S \setminus x)$ such that $\Phi(y) = \Phi(x)$,

$\alpha$-CONV: $S[\varphi] = S$ if $\Phi(x) = x \forall x \in \mathcal{S}(S)$.

Two terms $S$ and $S'$ are equivalent (written $S \equiv S'$) if they are in the least congruence relation (w.r.t. all the operators of the algebra) induced by the above axioms.

Let us check that these axioms match the informal meaning of the operators discussed after Definition 3.1. In fact, axioms ACI and ABS state that the parallel composition is commutative and associative, that $0$ is its unit, and that it is idempotent (i.e., $S \parallel S = S$), but only on concrete terms. We will see in Proposition 3.6 that every system can be regarded as the parallel composition of its global part (which is a concrete subterm) and of its local part (which is not concrete). This implies that parallel composition behaves as required, i.e., like disjoint union on local parts, and like set union on global parts.

Axioms COMP, MAP and DIS state that a substitution simply changes the variables and the process instances of a system, distributing over parallel composition. Axioms EXC and EL state that the items of a system can be made local only once and in any order. Axioms FAC and SWAP describe in the expected way the interplay between hiding and the operators for parallel composition and substitution.

Finally, axiom $\alpha$-CONV formalizes the intuition that the names of the hidden items are not meaningful, and thus they can be changed at will. Note, however, that an $\alpha$-conversion may change the variables and the process instances which are local to a system, but not their types, because every substitution is required to preserve types. This fact will be exploited in Sections 5 and 6, where most of the structure of the states of a cc computation or of a Petri net will be coded in the types of the corresponding CHARM states.

These considerations show that the operators of algebra $\mathcal{S}$ behave correctly with respect to the intuition we gave in Section 2. However, these operators are defined on terms, which do not correspond exactly to the states of a CHARM.

Definition 3.4 (States of a CHARM). A state $Q$ (of a CHARM) over algebra $\mathcal{S}$ is an equivalence class of terms of $\mathcal{S}$ w.r.t. the axioms of Definition 3.3, i.e., $Q = [S] = \{S' \in S \mid S \equiv S'\}$ for some $S \in \mathcal{S}$.
The next proposition shows that all the operations of Definition 3.1 (together with the corresponding axioms) can be extended consistently to states. This implies that the informal interpretation of the operators discussed above can also be applied to states.

**Proposition 3.5** (Extending the operators to equivalence classes). Let $Q = [S]$ and $Q' = [S']$ be two states over $S$, and let $x, \phi$ be as in Definition 3.1. We extend the operations on terms of $S$ to states over $S$ in the obvious way, i.e., $Q \parallel Q' = [S \parallel S']$, $Q \backslash x = [S \backslash x]$, and $Q[\phi] = [S[\phi]]$. These operations are well defined and satisfy the axioms of Definition 3.3.

**Proof.** The operations on states are well defined because relation $\approx$ is a congruence for the operators. Moreover, the conditional equations of Definition 3.3 are also meaningful for states, because the side conditions only use the set of free items of a term and the fact that a term is concrete, and both notions can be extended to states. In fact, if $S, S' \in S$ are two terms such that $S \approx S'$, then $F(S) = F(S')$; $S$ is closed iff $S'$ is closed and $S$ is concrete iff $S'$ is concrete. These facts can be checked easily, and it can be seen that they hold when $S$ and $S'$ are replaced by the left- and the right-hand side, respectively, of any axiom. \qed

As anticipated in Section 2, a specific CHARM over $S$ is a set of rewrite rules of the form $R: Q \rightarrow Q'$, where $Q$ and $Q'$ are states over $S$. The fact that states are defined as equivalence classes of terms of $S$ does not cause any problem, because term-rewriting modulo equations has a well understood theory (see e.g. [10]).

The rewrite rules of a CHARM are required to preserve the global part of the states they can be applied to. Therefore, we define a function $G_S$ which extracts from each state a concrete substate corresponding to its global part. Actually, $G_S$ is a partial function on states over $S$, because some state may denote a system whose global part is not a legal system. This happens when a variable is local, but some process instance using it is considered as global. This restriction is motivated by technical reasons, and can be justified by our interpretation of global items as potential interfaces among systems. In fact, if two separate systems specify in their global part the same item, say $v$, then the parallel composition recognizes this fact and identifies the two copies of $v$, allowing the two systems to interact through it (in fact, by idempotence $v \parallel v = v$ because $v$ is concrete). However, if one global process instance is specified by a system as using a local variable, as in term $p(x, y) \backslash x$, this specification cannot be matched by any other system; two instances of it cannot be identified in a parallel composition, because $p(x, y) \backslash x$ is not concrete.

We will call **well formed** the states on which function $G_S$ is defined, and which satisfy another natural constraint, i.e., that every process instance appears at most once in the global and in the local part. The well-formed states and the function $G_S$ are defined by exploiting the existence of a canonical form of terms.
Proposition 3.6 (Canonical form of terms). (1) Every term $S$ of the algebra of states $\mathcal{S}$ has an equivalent canonical form

$$(S_1 \parallel S_2) \setminus x_1 \ldots \setminus x_n,$$

where

- $S_1$ and $S_2$ are either atoms or parallel compositions of distinct atoms; moreover, $x_i \neq x_j$ for $i \neq j$, and $x_i \in \mathcal{F}(S_1 \parallel S_2)$ for all $i \in \{1, \ldots, n\}$;
- $\mathcal{F}(S_1) \cap \{x_1, \ldots, x_n\} = \emptyset$;
- either $S_2 = \emptyset$, or, if $S_2 = S_{21} \parallel \ldots \parallel S_{2k}$ and the $S_{2i}$'s are atoms, then $\mathcal{F}(S_{2i}) \cap \{x_1, \ldots, x_n\} \neq \emptyset$ for all $i \in \{1, \ldots, k\}$;
- for each atom of the form $q(v_1, \ldots, v_m)$ in $S_2$, and for each $i \in \{1, \ldots, m\}$, either $v_i = x_j$ for some $j \in \{1, \ldots, n\}$, or $v_i$ occurs in $S_1$;
- for each $q(v_1, \ldots, v_m)$ in $S_1$, $i \in \{1, 2\}$ and $j \in \{1, \ldots, m\}$, $v_j$ does not appear in $S_i$.

(2) The canonical form of a term is unique up to axioms ACI, EXC, and $\alpha$-CONV. That is, if $(S_1 \parallel S_2) \setminus x_1 \ldots \setminus x_n$ and $(S'_1 \parallel S'_2) \setminus y_1 \ldots \setminus y_k$ are two canonical forms for term $S$ (i.e., both are equivalent to $S$), then $(S_1 \parallel S_2) \setminus x_1 \ldots \setminus x_n \cong (S'_1 \parallel S'_2) \setminus y_1 \ldots \setminus y_k$ can be proved by using only axioms ACI, EXC, and $\alpha$-CONV.

(3) In the hypotheses of the previous point we have that $S_1 \cong S'_1$.

(4) If $S$ is concrete, then any canonical form of $S$ has the structure $S_{11} \parallel \ldots \parallel S_{1h} \parallel 0$, where the $S_{1i}$'s are distinct atoms.

If $S = (S_1 \parallel S_2) \setminus x_1 \ldots \setminus x_n$ is in canonical form, the second condition of Proposition 3.6(2) above guarantees that $S_1 \setminus x_1 \ldots \setminus x_n$ is a concrete subterm of $S$, while the third and the fourth conditions ensure that it is in some sense its "maximal" concrete subterm. By the last result, every state can be regarded as a parallel composition of atoms, some items of which may be hidden. This perfectly matches the informal description of states as collections of processes and variables partitioned into a local and a global part.

Definition 3.7 (Well-formed states and the global part of a state). Let $Q = [S]$ be a state over $\mathcal{S}$, and let $(S_1 \parallel S_2) \setminus x_1 \ldots \setminus x_n$ be a canonical form of $S$. Then state $Q$ is well-formed iff, for each atom of the form $q(v_1, \ldots, v_m) \in S_2$, $q = x_i$ for some $i$, and no process instance appears twice in $S_1 \parallel S_2$. Moreover, if $Q$ is well formed, then its global part is defined as $\mathcal{G}(Q) = [S_1]$, i.e., the equivalence class of terms containing $S_1$.

It can be shown that the definition of a well-formed state is well given because it does not depend on the choice of the term $(S_1 \parallel S_2) \setminus x_1 \ldots \setminus x_n \in Q$, provided that it is in canonical form. Moreover, the definition of function $\mathcal{G}$ is well given by Proposition 3.6(3).

Definition 3.8 (Rewrite rules and CHARM). A rewrite rule $R$ over (states over) $\mathcal{S}$ is a pair of well-formed states of $\mathcal{S}$, $R = (Q, Q')$ (also written $R : Q \rightarrow Q'$) such that
A CHARM $\mathcal{M}$ (over $\mathcal{S}$) is a collection of rewrite rules over $\mathcal{S}$, i.e., $\mathcal{M} = \{R_i : Q_i \rightarrow Q'_i\}_{i \in I}$.

A rewrite rule $R : Q \rightarrow Q'$ describes a possible evolution of a state $Q$. In order to model the application of $R$ to a bigger state that contains $Q$ as a proper substate, we introduce the transitions of a CHARM, which are generated by structural inference rules that embed $R$ in any possible context built with the operators of algebra $\mathcal{S}$. The rules also allow for the application of two or more rewrite rules in parallel.

**Definition 3.9** (The algebra of transitions of a CHARM). Let $\mathcal{M} = \{R_i : Q_i \rightarrow Q'_i\}_{i \in I}$ be a CHARM over $\mathcal{S}$. Then the terms of the algebra of transitions of $\mathcal{M}$, $\mathcal{T}(\mathcal{M})$, are generated by the following inference rules, which also give the left- and the right-hand side of each term,

<table>
<thead>
<tr>
<th>Rule (S)</th>
<th>Left-hand Side (LHS)</th>
<th>Right-hand Side (RHS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_i : Q_i \rightarrow Q'_i$</td>
<td>$Q : Q \rightarrow Q'$</td>
<td></td>
</tr>
<tr>
<td>$T : Q \rightarrow Q_1$, $T' : Q' \rightarrow Q'_1$</td>
<td>$T \parallel T' : Q \parallel Q'_1 \parallel Q'_1$</td>
<td></td>
</tr>
<tr>
<td>$T[\Phi] : Q[\Phi] \rightarrow Q'[\Phi]$</td>
<td>$T : Q \rightarrow Q'$</td>
<td></td>
</tr>
<tr>
<td>$T \setminus x : Q \setminus x \rightarrow Q' \setminus x$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The free items of a term $T \in \mathcal{T}(\mathcal{M})$ are the free items of the left-hand side of $T$ (or, equivalently, of its right-hand side). A term $T$ of $\mathcal{T}(\mathcal{M})$ is concrete iff $T : Q \rightarrow Q$, where $Q$ is a concrete state and $T$ does not include any rewrite rule $R_i \in \mathcal{M}$ as subterm. A transition $T_r$ over $\mathcal{T}(\mathcal{M})$ is an equivalence class $T_r = [T]$ of terms of $\mathcal{T}(\mathcal{M})$ with respect to the axioms of Definition 3.3 (in the axioms, every "$S$" must be replaced by a "$T$", and every atom $Q \in \{0, v, p(v_1, \ldots, v_n)\}$ stays for the corresponding identity $Q : Q \rightarrow Q$). A transition containing at most one basic transition $R \in \mathcal{M}$ is called sequential; otherwise, it is called parallel.

As for states, it can be proved that the operations on terms can be extended consistently to transitions, and that all the axioms also hold for transitions. Moreover, the definition of sequential transition is well given, because if a term $T$ contains exactly one basic transition $R \in \mathcal{M}$, so does any equivalent $T' \approx T$ (in fact, the unique axiom that could delete an occurrence of a transition $R \in \mathcal{M}$ from a term $T'$ is idempotence, but $R \parallel R = R$ does not hold because according to the definitions $R$ is not concrete).

**Definition 3.10** (The algebra of computations of a CHARM). Let $\mathcal{M} = \{R_i : Q_i \rightarrow Q'_i\}_{i \in I}$ be a CHARM over $\mathcal{S}$, and $\mathcal{T}(\mathcal{M})$ be its algebra of transitions. Then the terms of the algebra of computations of $\mathcal{M}$, $\mathcal{C}(\mathcal{M})$, are generated by the following inference rules,
where $\rho: Q \Rightarrow Q'$ means that $\rho$ starts from state $Q$ and ends in state $Q'$:

\[
\frac{\text{Tr} : Q \Rightarrow Q', \rho \Rightarrow Q', \rho' \Rightarrow Q''}{\rho; \rho' : Q \Rightarrow Q''}
\]

\[
\frac{\rho : Q \Rightarrow Q_1, \rho' : Q' \Rightarrow Q_1'}{\rho \parallel \rho' : Q \parallel Q' \Rightarrow Q_1 \parallel Q_1'}
\]

\[
\frac{\rho : Q \Rightarrow Q_1}{\rho \setminus x : Q \setminus x \Rightarrow Q_1 \setminus x}
\]

The free items of a term of $\mathcal{C}(\mathcal{M})$ are the free items of its starting (or ending) state. A term $\rho$ of $\mathcal{C}(\mathcal{M})$ is concrete iff $\rho: Q \Rightarrow Q$, where $Q$ is a concrete state, and all the transitions appearing as subterms in $\rho$ are concrete. A term $\rho \in \mathcal{C}(\mathcal{M})$ is called sequential iff it has the form $\rho = \text{Tr}_1; \ldots; \text{Tr}_n$, where $\text{Tr}_1, \ldots, \text{Tr}_n$ are sequential transitions over $\mathcal{F}(\mathcal{M})$.

A computation $\sigma$ over $\mathcal{C}(\mathcal{M})$ is an equivalence class $\sigma = [\rho]$ of terms of $\mathcal{C}(\mathcal{M})$ with respect to the same axioms as in Definition 3.3, plus the following axioms, valid whenever both sides are defined. These axioms state that the operations of the algebra distribute over sequential composition, that ; is associative, and that there are identities:

- $(\rho \parallel \rho'; (\rho_1 \parallel \rho_1')) = (\rho; \rho_1) \parallel (\rho'; \rho_1')$,
- $(\rho; \rho')[\Phi] = \rho[\Phi]; \rho'[\Phi]$,
- $(\rho; \rho') \setminus x = \rho \setminus x; \rho' \setminus x$,
- $(\rho; \rho'); \rho'' = \rho; (\rho'; \rho'')$,
- $Q; \rho = \rho = Q' \text{ if } \rho : Q \Rightarrow Q'$.

In this case also it is worth stressing that the operations on terms of $\mathcal{C}(\mathcal{M})$ can be extended consistently to equivalence classes (i.e., to computations over $\mathcal{C}(\mathcal{M})$), and that they still satisfy the axioms of Definition 3.3 and those just listed.

4. Modeling graph grammars

The "theory of graph grammars" studies a variety of formalisms which extend the theory of formal languages in order to deal with structures more general than strings, such as graphs and maps. A graph grammar allows one to describe finitely a (possibly infinite) collection of graphs, i.e., those graphs which can be obtained from an initial graph through repeated application of graph productions.

In this section we show how to translate a graph rewriting system $\mathcal{R}$ into a CHARM $\mathcal{M}(\mathcal{R})$, and we prove (via a result of soundness and completeness) that the computations of $\mathcal{M}(\mathcal{R})$ faithfully correspond to those of $\mathcal{R}$. This result extends a similar result presented in a previous paper by two of the authors [4]. Moreover, we will discuss the tight relationship between the parallel composition of transitions of
An abstract machine for concurrent modular systems: CHARM

a CHARM and the notion of parallel independence of direct derivations in the theory of graph grammars.

Throughout this section, for graph we mean globally colored, directed hypergraph, i.e., a tuple \( G = (N, E, c, nl_G, el_G) \), where \( N \) is a set of nodes, \( E \) is a set of edges, \( c : E \rightarrow N^* \) is the connection function (thus each edge can be connected to a list of nodes), and \( nl_G : N \rightarrow N^ \), \( el_G : E \rightarrow E^ \) are two labeling functions associating a color with each node and edge, where \( N^ \) and \( E^ \) are two global sets of node and edge colors, respectively.

Since we only consider finite graphs, we will assume, without loss of generality, that there exist two sets \( \mathcal{N} \) and \( \mathcal{E} \) which include all edges and all nodes, respectively. Moreover, we assume that there exist two global labeling functions \( nl : \mathcal{N} \rightarrow \mathcal{N}^ \) and \( el : \mathcal{E} \rightarrow \mathcal{E}^ \), such that, for each graph \( G = (N, E, c, nl_G, el_G) \), \( nl_G \) is the restriction of \( nl \) to \( N \), and, similarly, \( el_G \) is the restriction of \( el \) to \( E \). Thus, a graph can be presented in a simpler way as a triple \( G = (N, E, c) \).

If \( G' = (N', E', c') \) is another graph, then a graph (homo)morphism \( f : G \rightarrow G' \) is a pair of functions \( (f_N : N \rightarrow N', f_E : E \rightarrow E') \) which preserve the connection functions (i.e., \( c'(f_E(e)) = f_N^*(c(e)) \), where \( f_N^* \) is the extension of \( f_N \) to lists) and the colors (i.e., \( el_G(f_E(e)) = el_G(e) \) and \( nl_G(f_N(n)) = nl_G(n) \)). A graph monomorphism \( f : G \rightarrow G' \) is a graph morphism such that both \( f_N \) and \( f_E \) are injective; moreover, \( f \) is a graph isomorphism if both \( f_N \) and \( f_E \) are isomorphisms.

Following the so-called algebraic approach to graph grammars [8], a graph production \( p = (L \xrightarrow{1} K \xleftarrow{c} R) \) is a pair of graph monomorphisms having as common source a graph \( K \), the gluing graph, indicating which edges and nodes have to be preserved by the application of the production. Graphs \( L \) and \( R \) are called the left-hand side and the right-hand side of \( p \), respectively.

In order to describe how a production can be applied to a graph, we need the following definition.

**Definition 4.1 (Pushout and pushout complement in the category of graphs).** Given two graph morphisms \( b : K \rightarrow B \) and \( d : K \rightarrow D \), a triple \( \langle H, h : B \rightarrow H, c : D \rightarrow H \rangle \) as in Fig. 3 is called a pushout of \( \langle b, d \rangle \) if:

- (commutativity property) \( h \circ b = c \circ d \);
- (universal property) for all graphs \( H' \) and graph morphisms \( h' : B \rightarrow H' \) and \( c' : D \rightarrow H' \), with \( h' \circ b = c' \circ d \), there exists a unique morphism \( f : H \rightarrow H' \) such that \( f \circ h = h' \) and \( f \circ c = c' \), as in Fig. 3.

In this situation, \( H \) is called a pushout object of \( \langle b, d \rangle \). Moreover, given arrows \( b : K \rightarrow B \) and \( h : B \rightarrow H \), a pushout complement of \( \langle b, h \rangle \) is a triple \( \langle D, d : K \rightarrow D, c : D \rightarrow H \rangle \) such that \( \langle H, h, c \rangle \) is a pushout of \( b \) and \( d \). In this case \( D \) is called a pushout complement object of \( \langle b, h \rangle \).

Intuitively, graph \( H \) in Fig. 3 is a pushout object of morphisms \( b \) and \( d \) if it is obtained from the disjoint union of \( D \) and \( B \) by identifying the images of \( K \) in \( D \) and in \( B \).
A production $p$ can be applied to a graph $G$ yielding $H$ (written $G \Rightarrow_p H$) if there is an occurrence of the left-hand side in $G$ (i.e., a graph morphism $g : L \rightarrow G$), and $H$ is obtained as the result of the double-pushout construction of Fig. 4.

This construction may be interpreted as follows. In order to delete the occurrence of $L$ in $G$, we construct the pushout complement of $g$ and $l$, i.e., a triple $\langle D, k : K \rightarrow D, d : D \rightarrow G \rangle$ such that the resulting square is a pushout. Next, we have to embed the right-hand side $R$ in $D$ via a second pushout, which produces graph $H$. If $G \Rightarrow_p H$ we say that there is a direct derivation from $G$ to $H$ via $p$. From this informal explanation of a direct derivation, it should be clear that the gluing graph $K$ of the production is needed to specify how $L$ and $R$ are embedded in the larger graphs $G$ and $H$, respectively.

A graph rewriting system is a set $\mathcal{R}$ of graph productions. A derivation from $G$ to $H$ over $\mathcal{R}$ (shortly $G \Rightarrow^*_{\mathcal{R}} H$), is a finite sequence of direct derivations of the form $G \Rightarrow p_1, G_1 \Rightarrow p_2, \ldots \Rightarrow p_n, G_n = H$, where $p_1, \ldots, p_n$ are in $\mathcal{R}$.

To define the CHARM which implements a given graph rewriting system, we have to define the sets of process instances and of variables, and the corresponding typing functions (see Definition 3.1). Quite obviously, we can regard a colored graph as a distributed system where the edges are process instances, the nodes are variables, and the colors are the corresponding types. Thus, in the rest of this section we will consider the algebra of states $\mathcal{S}_{\text{Graphs}}$ over the set of process instances $\mathcal{E}$, the set of variables $\mathcal{N}$, and with typing functions $\eta : \mathcal{N} \rightarrow \mathcal{N} \mathcal{E}$ and $\epsilon : \mathcal{E} \rightarrow \mathcal{E} \mathcal{C}$, where $\mathcal{E}, \mathcal{N}, \mathcal{N} \mathcal{C}, \mathcal{N} \mathcal{E}, \eta$ and $\epsilon$ are as mentioned above. In [4] the precise relationship between graphs similar to those introduced above and terms of a suitable algebra of states has been explored in the noncolored case. This relationship extends straightforwardly to colored graphs and states over $\mathcal{S}_{\text{Graphs}}$, as summarized below, because the

![Fig. 3. Pushout diagram.](image)

![Fig. 4. Graph rewriting via double-pushout construction.](image)
one-to-one correspondence of process instances and variables with edges and nodes, respectively, can be extended in an obvious way to colors and typing functions. Informally, concrete states over $\mathcal{G}_{\text{Graphs}}$ (see Definition 3.4 and Proposition 3.5) faithfully model finite graphs, while well-formed states (see Definition 3.7) model in a similar way equivalence classes of graph monomorphisms, where the source graph is fixed while the target graph is defined up to isomorphism. Such a class of monomorphisms is called a partially abstract graph in [5], because it can be thought of as a graph with a global and a local part (the one defined up to isomorphisms).

**Definition 4.2** (From states to graph monomorphisms and vice versa). (1) Let $Q = [S]$ be a concrete, well-formed state over $\mathcal{G}_{\text{Graphs}}$, and let $S' \approx S$ be in canonical form (thus $S'$ is a parallel composition of distinct atoms (by Proposition 3.6), where all process instances are distinct (by Definition 3.7)). Then the graph associated with $Q$ is defined as

$$\text{Gr}(Q) = (\{v \mid v \text{ appears in } S'\}, \{p \mid p(v_1, \ldots, v_n) \text{ is an atom in } S'\}, c, n_1', e_1'),$$

where the connection function $c$ is defined as $c(p) = v_1 \cdots v_n$ if $p(v_1, \ldots, v_n)$ is in $S'$, and $n_1', e_1'$ are the restrictions of $n_1: \mathcal{N} \to \mathcal{N}'$ and $e_1: \mathcal{E} \to \mathcal{E}'$ to nodes and edges of $\text{Gr}(Q)$, respectively. Since the sets of colors and the typing functions are fixed once and for all, we will not mention them explicitly in the rest of the definition.

(2) Let $Q = [S]$ be a well-formed state over $\mathcal{G}_{\text{Graphs}}$, and let $(S_1 \parallel S_2) \times \ldots \times x_n$ be a canonical form of $S$. Then

- The concrete graph associated with $Q$ is defined as $\text{CGr}(Q) \equiv \text{Gr}([S_1])$.
- The abstract graph associated with $Q$ is defined as $\text{AGr}(Q) \equiv \text{Gr}([S_1 \parallel S_2])$.
- The graph monomorphism associated with $S$, $m(Q): \text{CGr}(Q) \rightarrow \text{AGr}(Q)$, is the obvious inclusion, since $\text{CGr}(Q)$ is a subgraph of $\text{AGr}(Q)$.

It must be stressed that $\text{AGr}(Q)$ is determined up to renaming of the items $x_1, \ldots, x_n$; thus, depending on the context, it will either denote a set of isomorphic graphs, or an arbitrarily chosen element of that set. Similarly, $m(Q)$ will either denote a set of graph monomorphisms (one for each choice of the target graph), or any element of that set.

(3) Let $G = (N, E, c, n_G, e_G)$ be a graph, with $N = \{n_i\}_{i \leq m}$, $E = \{e_j\}_{j \leq r}$, $c(e_i) = n_{ik}$ for all $1 \leq i \leq r$, $n_G: N \rightarrow \mathcal{N}'$, and $e_G: E \rightarrow \mathcal{E}'$. Then the concrete, well-formed state over $\mathcal{G}_{\text{Graphs}}$ representing $G$ is defined as

$$\text{Tm}(G) = [n_1 \parallel \ldots \parallel n_m \parallel e_1(n_{11}, \ldots, n_{1k_1}) \parallel \ldots \parallel e_r(n_{r1}, \ldots, n_{rk}) \parallel 0].$$

In particular, if $\emptyset$ is the empty graph, then $\text{Tm}(\emptyset) = [0]$.

(4) Let $h: G \subseteq H$ be a graph monomorphism. Then the well-formed state representing $h$ is defined as

$$\text{WFT}(h) = (\text{Tm}(H) \times x_1 \times \ldots \times x_n)[h^{-1}],$$

where $\{x_1, \ldots, x_n\}$ is the set of items of $H$ which are not in the image of $G$ through $h$, and $h^{-1}$ improperly denotes the substitution such that $h^{-1}(y) = x$ if $h(x) = y$, and $h^{-1}(y) = y$ otherwise (which is well defined because $h$ is injective).
The next result states that the above definitions are well given, and clarifies the relationship between terms and graph monomorphisms. For the proof we refer to [4].

**Proposition 4.3** (Equivalence between well-formed terms and monomorphisms). (1) The functions $\text{Gr}, \text{CGr}, \text{AGr}$, and $m$ introduced in Definition 4.2 are well-defined, i.e., they do not depend on the choice of the canonical form (provided that we regard $\text{AGr}(Q)$ and $m(Q)$ as sets of graphs and of monomorphisms, respectively).

(2) Functions $\text{Gr}$ and $\text{Tm}$ are inverses of each other (i.e., $\text{Tm}(\text{Gr}(Q)) = Q$ and $\text{Gr}(\text{Tm}(G)) = G$), and thus they provide an isomorphism between finite graphs and well-formed, concrete states.

(3) If $h$ is a graph monomorphism between finite graphs, then $h \in m(\text{WfT}(h))$. Conversely, if $Q$ is a well-formed state over $\mathcal{G}_{\text{Graphs}}$, then $\text{WfT}(m(Q)) = Q$. Actually, functions $m$ and $\text{WfT}$ provide an isomorphism between suitable equivalence classes of graph homomorphisms and well-formed states.

From Definitions 3.7 and 3.12 it can be checked that if $h: G \hookrightarrow H$, then $\mathcal{G}_P(\text{WfT}(h)) = \text{Tm}(G)$, i.e., that the global part of the state representing $h$ is exactly the concrete state representing $G$. Using this observation, and since a graph production is a pair of graph monomorphisms with common domain, it is easy to associate a CHARM rewrite rule (in the sense of Definition 3.8) with each graph production.

**Definition 4.4** (The CHARM implementing a graph rewriting system). Let $\mathcal{R}$ be a graph rewriting system. For each graph production $p = (L \rightarrow K \rightarrow R)$ in $\mathcal{R}$, its associated rewrite rule $\mathcal{M}(p)$ is defined as $\mathcal{M}(p): \text{WfT}(l) \rightarrow \text{WfT}(r)$. The CHARM implementing $\mathcal{R}$ is defined as $\mathcal{M}(\mathcal{R}) = \{\mathcal{M}(p) | p \in \mathcal{R}\}$.

In order to relate correctly the operational behaviors of a graph rewriting system $\mathcal{R}$ and of its associated CHARM $\mathcal{M}(\mathcal{R})$, we need to introduce a lemma which shows that monadic, concrete contexts (i.e., concrete terms of $\mathcal{G}_{\text{Graphs}}$ with a “hole”) faithfully correspond to pushouts in the category of finite graphs. For the proof we again refer to [5].

**Definition 4.5** (Contexts). A (monadic) term context $c$ over the algebra of states $\mathcal{S}$ is a term generated by the following syntax:

$$c ::= \bullet | c \| Q | c[\Phi] | c \setminus x,$$

where $Q$ is a state over $\mathcal{S}$, and $\Phi$ and $x$ are as in Definition 3.1. A (monadic) context $C$ is an equivalence class of context terms with respect to the equivalence induced by the axioms of Definition 3.3, assuming that $\bullet$ is not concrete and that $\mathcal{F}(\bullet) = \mathcal{S} \cup \mathcal{Y}$. A context is concrete if it includes a context term that does not contain any restriction operator.
If \( Q \) is a state over \( S \), then \( C[Q] \) denotes the state obtained by replacing \( Q \) for \( \bullet \) in \( C \). A context term is in canonical form if it has the form \( c = (\bullet[\Phi] \parallel Q)[\chi_1, \ldots, \chi_n] \), where \( Q \) is any state. Every context term \( c \) is equivalent to a \( c' \) in canonical form.

**Lemma 4.6** (Relating concrete contexts and pushouts). (1) Let \( h: G \rightarrow H \) and \( h': G' \rightarrow H' \) be graph monomorphisms, let \( f: G \rightarrow G' \) be a morphism, and suppose that there exists a morphism \( g: H \rightarrow H' \) such that the diagram in Fig. 5 is a pushout. Then there exists a concrete context \( C \) over \( \mathcal{S}_{\text{Graphs}} \) such that \( C[WfT(h)] = WfT(h') \). More precisely, \( C \) is the context containing the context term \( \bullet[f_N \cup f_E] \parallel \text{tm}(G') \), where \( f_E \) and \( f_N \) are the two components of morphism \( f \). It is worth stressing here that \( C \) is uniquely determined by \( G, G', \) and \( f \).

(2) Let \( Q \) be a well-formed state over \( \mathcal{S}_{\text{Graphs}} \), and let \( C \) be a concrete context over \( \mathcal{S}_{\text{concrete}} \) containing the context term in canonical form \( \bullet[\Phi] \parallel Q' \). Then there exists a graph morphism \( g: \text{AGr}(Q) \rightarrow \text{AGr}(C[Q]) \) such that Fig. 6 is a pushout, where \( f_\Phi \) denotes the obvious graph morphism induced by \( \Phi \).

We are now ready to present a “completeness” result, stating that every direct derivation of a graph rewriting system \( \mathcal{R} \) is faithfully modeled by a sequential transition of the corresponding CHARM. It must be stressed that the graph derivations introduced at the beginning of this section are defined up to isomorphism, i.e., if \( G \Rightarrow_H H \), then \( G' \Rightarrow_{H'} H' \) for each \( G' \cong G \) and \( H' \cong H \). This is due to the fact that the pushout objects of Fig. 4 are defined up to isomorphism. As a consequence, graph derivations actually define a relation among equivalence classes of graphs, rather than among graphs. Such equivalence classes can be represented by closed states; indeed, using the functions introduced in Definition 4.2, if \( G \) is a graph and \( 0_G \) is the unique morphism from the empty graph to \( G \), then \( \text{AGr}(WfT(0_G)) \) is the set of all graphs.
isomorphic to \( G \), i.e., \( \text{AGr}(WfT(0_G)) = \{ G' | G' \cong G \} \). Therefore, we will regard the closed-state \( WfT(0_G) \) as representing the isomorphism class of graphs containing \( G \).

**Theorem 4.7** (Completeness of the CHARM rule implementing a production). Let \( p = (L \overset{i}{\rightarrow} K \overset{r}{\rightarrow} R) \) be a graph production such that there is a direct derivation from graph \( G \) to \( H \) (i.e., \( G \Rightarrow p H \)). Then there exists a sequential transition \( Tr \) over \( \mathcal{T}(\mathcal{M}(\mathcal{R})) \) (the algebra of transitions of \( \mathcal{M}(\mathcal{R}) \), see Definition 3.9) such that \( Tr: WfT(0_G) \rightarrow WfT(0_H) \).

**Proof.** By hypothesis, the double pushout diagram in Fig. 4 can be constructed, and \( \mathcal{M}(p): WfT(l) \rightarrow WfT(r) \) is a transition of \( \mathcal{M}(\mathcal{R}) \). By Lemma 4.6(1) (applied to the left pushout) there exists a concrete context \( C \) over \( \mathcal{I}_{\text{Graphs}} \) such that \( C[WfT(l)] = WfT(d) \). Applying the same lemma to the right pushout we also have that \( C[WfT(r)] = WfT(b) \) for the same context \( C \), because it is uniquely determined by \( K, D, \) and \( k \).

Let now \( C' = C \setminus x_1 \ldots \setminus x_n \), where \( \{ x_1, \ldots, x_n \} = \mathcal{F}(\text{Tm}(D)) \) is the set of free items of both \( WfT(d) \) and \( WfT(b) \). Then clearly \( T \equiv C[\mathcal{M}(p)], C[WfT(l)] \rightarrow C[WfT(r)] \) is a legal, sequential term of algebra \( \mathcal{I}(\mathcal{M}(\mathcal{R})) \), because it can be constructed by using the inference rules of Definition 3.9. Let \( Tr = [T], i.e., Tr \) is the equivalence class of terms of \( \mathcal{I}(\mathcal{M}(\mathcal{R})) \) containing \( T \). It remains to show that \( Tr \) has the correct source and target states. For the source, we have that \( C'[WfT(l)] = C[WfT(l)] \setminus x_1 \ldots \setminus x_n = WfT(d) \setminus x_1 \ldots \setminus x_n = WfT(0_G) \), where the last equivalence follows from the observation that \( G_0 : \emptyset \rightarrow G \in \mathcal{M}(WfT(d) \setminus x_1 \ldots \setminus x_n) \) and by applying Proposition 4.3(3). Similarly, for the target state it can be shown that \( C'[WfT(r)] = WfT(0_H) \). \( \square \)

The last result can easily be extended to entire computations; every derivation of a graph rewriting system \( \mathcal{R} \) is faithfully modeled by a term of the algebra of computations of the corresponding CHARM \( \mathcal{M}(\mathcal{R}) \) (see Definition 3.10).

**Corollary 4.8** (Completeness of the CHARM implementing a graph rewriting system). Let \( \mathcal{R} \) be a graph rewriting system and let \( G \Rightarrow^* H \) be a graph derivation. Then there exists a computation \( \sigma \) over \( \mathcal{C}(\mathcal{M}(\mathcal{R})) \) such that \( \sigma: WfT(0_G) \Rightarrow WfT(0_H) \).

**Proof.** By definition of derivation, there exists a finite sequence of direct derivations \( G = G_0 \Rightarrow p_1 G_1 \Rightarrow p_2 \cdots \Rightarrow p_n G_n = H \). By Theorem 4.7, for each \( i \in \{1, \ldots, n\} \), there exists a sequential transition \( Tr_i \) over \( \mathcal{T}(\mathcal{M}(\mathcal{R})) \) such that \( Tr_i: WfT(0_{G_{i-1}}) \rightarrow WfT(0_{G_i}) \). By sequential composition of these terms, we get the desired computation \( \sigma = [Tr_1; \ldots; Tr_n]: WfT(0_G) \Rightarrow WfT(0_H) \). \( \square \)

We now want to prove the converse, i.e., that every computation of \( \mathcal{M}(\mathcal{R}) \) corresponds to a graph derivation in \( \mathcal{R} \). We will prove this first for sequential transitions.
**Theorem 4.9** (Soundness of the CHARM rule implementing a production). Let \( p = (L \rightarrow K \leftarrow R) \) be a production in \( \mathcal{M} \) and let \( Tr \) be a sequential transition over \( \mathcal{F}(\mathcal{M}(\mathcal{K})) \) containing \( \mathcal{M}(p) \) such that \( Tr: Q \rightarrow Q' \), with \( Q \) and \( Q' \) well-formed and closed. Then \( \text{AGr}(Q) \Rightarrow_p \text{AGr}(Q') \).

**Proof.** Since \( Tr \) is sequential, it contains (as equivalence class) a term of the form \( C[\mathcal{M}(p)]x_1 \ldots x_n \), where \( C \) is a concrete context over \( \mathcal{S}_{\text{Graphs}} \). By the definition of \( \mathcal{M}(p) \) and Definition 3.9 we have \( C[\mathcal{M}(p)]: C[\mathcal{WT}(l)] \rightarrow C[\mathcal{WT}(r)] \). By applying Lemma 4.6(2) to both \( C[\mathcal{WT}(l)] \) and \( C[\mathcal{WT}(r)] \), we have that the double pushout in Fig. 7 can be constructed. This fact implies that \( \text{AGr}(C[\mathcal{WT}(l)]) \Rightarrow_p \text{AGr}(C[\mathcal{WT}(r)]) \), and the statement follows by observing that \( \text{AGr}(C[\mathcal{WT}(l)]) = \text{AGr}(C[\mathcal{WT}(r)]) = \text{AGr}(Q) \), and, similarly, that \( \text{AGr}(C[\mathcal{WT}(r)]) = \text{AGr}(Q') \).

In order to extend the soundness result from sequential transitions to computations, we need the following general fact.

**Proposition 4.10** (Computations can be sequentialized). Given a CHARM \( \mathcal{M} \), every computation over \( \mathcal{C}(\mathcal{M}) \) (regarded as an equivalence class) contains a sequential term of \( \mathcal{C}(\mathcal{M}) \).

**Proof.** Let \( \sigma \) be a computation over \( \mathcal{C}(\mathcal{M}) \), i.e., \( \sigma = [\rho] \) for some \( \rho \in \mathcal{C}(\mathcal{M}) \). We have to show that there exists a sequential term of \( \mathcal{C}(\mathcal{M}) \) equivalent to \( \rho \). In fact, using the axioms of Definitions 3.3 and 3.10, it can be shown that \( \rho \) is equivalent to a term of the form \( Tr_1; \ldots; Tr_n \), where every \( Tr_i \) is a parallel transition of the form \( Tr_i = (R_{il} [\Phi_{i1}] \ldots R_{im} [\Phi_{im}] Q_i) ) x_{i1} \ldots x_{ik_i} \), where each \( R_{ij} \in \mathcal{M} \) is a rewrite rule and \( Q_i \) is a state. The only nontrivial case is when the commutativity between sequential and parallel composition cannot be applied directly, as in \( (\rho; \rho') \parallel Tr \). In this case, if \( Tr: Q \rightarrow Q' \), the term can be transformed as follows: \( (\rho; \rho') \parallel Tr \approx (\rho \parallel Tr) ; (\rho ; Q') \approx (\rho \parallel Tr) ; (\rho' \parallel Q') \).

Next every parallel transition can be transformed into an equivalent sequential term of \( \mathcal{C}(\mathcal{M}) \), using the same axioms. As an example, if \( R_1 [\Phi_1] Q_1 \rightarrow Q'_1 \) and \( R_2 [\Phi_2] Q_2 \rightarrow Q'_2 \), then \( (R_1 [\Phi_1] Q_2) x_1 \ldots x_n \approx ((R_1 [\Phi_1] Q_1) ; Q_2) x_1 \ldots x_n \approx (R_1 [\Phi_1] Q_2) x_1 \ldots x_n \approx (R_1 [\Phi_1] Q_2) x_1 \ldots x_n \).

Fig. 7. The double pushout corresponding to a sequential transition.
Corollary 4.11 (Soundness of the CHARM implementing a graph rewriting system). Let $\mathcal{R}$ be a graph rewriting system and let $\sigma$ be a computation over $\mathcal{C}(\mathcal{M}(\mathcal{R}))$ such that $\sigma: Q \Rightarrow Q'$, with $Q$ and $Q'$ closed and well-formed. Then there exists a graph derivation $\text{AGr}(Q) \Rightarrow \text{AGr}(Q')$.

**Proof.** By Proposition 4.10, $\sigma = [p]$ where $p = Tr_1; \ldots; Tr_n$ is a sequential term of $\mathcal{C}(\mathcal{M}(\mathcal{R}))$. Since $T_i$ is a sequential transition for all $1 \leq i \leq n$, the statement follows by $n$ applications of Theorem 4.9. $\square$

As the last point of this section we want to stress the tight correspondence between the notion of "parallel independence" of derivation steps in the theory of graph grammars, and the operation of parallel composition on transitions of the CHARM.

If $p = (L^L \rightarrow K \rightarrow R)$ and $p' = (L' \rightarrow K' \rightarrow R')$ are two graph productions, $G$ is a graph, and $g: L \rightarrow G$, $g': L' \rightarrow G$ are two occurrences in $G$ of $p$ and $p'$, respectively, then $g$ and $g'$ are called parallel-independent iff whenever an arc or a node of $G$ is in $g(L) \cap g'(L')$ (i.e., it belongs to both occurrences) then it is also in $g(l(K)) \cap g'(l'(K'))$, i.e., it is preserved by both productions. A classical result of the theory of graph grammars states that in this case $p$ and $p'$ can be applied sequentially to $G$ in an arbitrary order, producing the same graph $H$. Moreover, $H$ can also be obtained by applying $p$ and $p'$ in parallel, via the parallel production $p + p' = (L + L' \rightarrow K + K' \rightarrow R + R')$ ($+$ denotes disjoint union) and the obvious occurrence induced by $g$ and $g'$.

Therefore, in the above situation, there are three distinct derivations from $G$ to $H$ (which we denote by $p; p'; p' p$ and $p + p'$, respectively) which are essentially equivalent, in the sense that they modify $G$ in the same way. This fact is faithfully reflected in the algebra of computations of the corresponding CHARM, because the three computations associated by Corollary 4.8 with $p; p'; p' p$ and $p + p'$ turn out to be exactly the same. This fact substantiates that claimed in Section 2, i.e., that two transitions of a CHARM can be applied in parallel not only when they modify disjoint parts of a state, but also when they overlap on items which are preserved by both of them.

The formal correspondence between parallel independent direct derivations of a graph rewriting system and parallel transitions of the associated CHARM is stated in the next proposition. The proof is omitted as it would need a more detailed presentation of the correspondence between terms and graph monomorphisms (discussed in [5]) than the one of Definition 4.2.

**Proposition 4.12** (Parallel transitions and parallel independent direct derivations). (1) Let $Tr$ be a parallel transition over $\mathcal{T}(\mathcal{M}(\mathcal{R}))$, $Tr: Q \rightarrow Q'$, containing exactly two subterms of the form $\mathcal{M}(p)$, say $\mathcal{M}(p_1)$ and $\mathcal{M}(p_2)$. Then $Tr$ induces two occurrences of $p_1$ and $p_2$, respectively, in $\text{AGr}(Q)$ which are parallel-independent, and such that $\text{AGr}(Q) \Rightarrow_{p_1 + p_2} \text{AGr}(Q')$ via those occurrences.
Let $p_1, p_2 \in \mathcal{P}$ be two productions, and $g_1, g_2$ be two parallel-independent occurrences of $p_1$ and $p_2$, respectively, in a graph $G$. Moreover, suppose that $G \models p_1 \Rightarrow p_2 H$ via $g_1$ and $g_2$. Then there exists a transition $Tr$ of the form $Tr = [\mathcal{M}(p_1)[\Phi_1] \parallel \mathcal{M}(p_2)[\Phi_2] \parallel Q\backslash x_1 \ldots \backslash x_n]$ such that $Tr: \text{WF}T(0_g) \rightarrow \text{WF}T(0_H)$.

5. Modeling concurrent constraint programming

The concurrent constraint (cc) programming paradigm [23–25, 7] is an elegant framework which captures and generalizes most of the concepts of logic programming [13], concurrent logic programming [22], and constraint logic programming [11]. The basic idea is that a program is a collection of concurrent agents which share a set of variables, over which they may pose ("tell") or check ("ask") constraints of any kind. Agents are defined by clauses as the parallel composition ($\parallel$), or the existential quantification ($\exists$), or the nondeterministic choice ($+$), of other agents. A computation refines the initial constraint on the shared variables (i.e., the store) through a monotonic addition of information until a stable configuration (if any) is obtained, which is the final constraint returned as the result of such a computation.

The cc paradigm is parametric w.r.t. the kind of constraints that are handled. Any choice of the constraint system (i.e., kind of constraints and solution algorithm) gives a specific cc language. For example, by choosing the Herbrand constraint system we get concurrent logic programming, and by further eliminating concurrency we get logic programming.

In this section we will show how any cc program $P$ can be modeled by a CHARM $\mathcal{M}(P)$ whose computations (on a restricted set of states) are in bijective correspondence with the computations of $P$. The idea is to consider each computation state of a cc program as the current collection of constraints (on the shared variables) and of active agents (together with the variables they involve), and then to represent each computation step as the application of a rewrite rule. More precisely, occurrences of both agents and (primitive) constraints are going to be modeled as process instances, and agents and constraints as process types, while the shared variables are the variables of the abstract machine.

The formal syntax of cc programs and cc agents, as well as the formalization of the constraint system, can be found in the following definitions.

**Definition 5.1 (cc programs).** The following grammar describes the cc language we consider:

$$
P := F.A,
F := p(x): A \mid F.F,
A := \text{success} \mid \text{failure} \mid \text{tell}(c) \Rightarrow A \mid E \mid A \parallel A \mid \exists x.A \mid p(x),
F := \text{ask}(c) \Rightarrow A \mid F+F,
$$
where \( P \) is the class of programs, \( F \) is the class of sequences of procedure declarations, \( A \) is the class of agents, \( c \) ranges over constraints, and \( x \) is a tuple of variables.

Each procedure is defined (at most) once, and thus nondeterminism is expressed via the \( + \) combinator only. We also assume that, in \( p(x)::A \), \( \text{vars}(A) \subseteq x \), where \( \text{vars}(A) \) is the set of all variables occurring free (i.e., not bound by an \( \exists \) in agent \( A \)).

In a program \( P = F.A \), \( A \) is called the initial agent, to be executed in the context of the set of declarations \( F \). This corresponds to the language considered in [25], which allows only guarded nondeterminism. The intuitive behavior of the agents is as follows:

- Agent "\( \text{ask}(c) \rightarrow A \)" checks whether constraint \( c \) is entailed by the current store and then, if so, behaves like agent \( A \). If \( c \) is inconsistent with the current store it fails, and otherwise it suspends until \( c \) is either entailed by the current store or is inconsistent with the store.
- Agent "\( \text{ask}(c_1) \rightarrow A_1 + \text{ask}(c_2) \rightarrow A_2 \)" may behave either like \( A_1 \) or like \( A_2 \) if both \( c_1 \) and \( c_2 \) are entailed by the current store. It behaves like \( A_1 \) if \( c_1 \) only is entailed, it suspends if both \( c_1 \) and \( c_2 \) are consistent with but not entailed by the current store, and it behaves like "\( \text{ask}(c_1) \rightarrow A_1 \)" whenever "\( \text{ask}(c_2) \rightarrow A_2 \)" fails (and vice versa).
- In an "atomic" interpretation of the tell operation, agent "\( \text{tell}(c) \rightarrow A \)" adds constraint \( c \) to the current store and then, if the resulting store is consistent, behaves like \( A \); otherwise it fails. In an "eventual" interpretation of the tell, this same agent adds \( c \) to the store (without any consistency check) and then behaves like \( A \) (if the resulting store is inconsistent this will result in an uncontrolled behavior of the system, since from now on all ask operations will succeed). In this paper we adopt the eventual interpretation of the tell operation.
- Agent \( A_1 \parallel A_2 \) behaves like \( A_1 \) and \( A_2 \) executing in parallel.
- Agent \( \exists x.A \) behaves like agent \( A \), except that the variables in \( x \) are local to \( A \).
- Agent \( p(x) \) is a call of procedure \( p \).

Given a program \( P \), we will refer to \( \text{Ag}(P) \) as the set of all agents (and subagents) occurring in \( P \), i.e., all the elements of type \( A \) occurring in a derivation of \( P \) according to the above grammar.

**Definition 5.2 (Constraint system).** In the cc paradigm, the underlying constraint system can be described [25] as a system of partial information (derived from the information system introduced in [26]) of the form \( \langle D, \vdash \rangle \) where

- \( D \) is a set of tokens (or primitive constraints) and
- \( \vdash \subseteq \varphi(D) \times D \) is the entailment relation which states which tokens are entailed by which sets of other tokens. This relation has to satisfy the following axioms:
  - \( u \vdash -\) if \( x \in u \) (reflexivity), and
  - \( u \vdash -\) if \( u \vdash -\) for all \( y \in v \) and \( v \vdash x \) (transitivity).

Given \( D, |D| \) is the set of all subsets of \( D \) closed under entailment. Then a constraint in a constraint system \( \langle D, \vdash \rangle \) is simply an element of \( |D| \) (that is, a set of tokens).
The state of the computation is a multiset of tokens plus a multiset of agents. We consider multisets instead of sets because a token or an agent may appear more than once in a computation state. It is important to notice that all variables appearing in a cc state have to be considered as existentially quantified, and thus defined up to renaming. This is easy to see in the agent \( p(x, y) \), which is in reality intended as the problem of finding values for \( x \) and \( y \) such that \( p \) holds on them. That is: 
\[
\exists x, y . p(x, y)
\]
Therefore, each cc state will be defined up to an arbitrary renaming of its variables.

**Definition 5.3 (Computation state).** Given a constraint system \( \langle D, \vdash \rangle \) and a program \( P \), a computation state for \( P \) is any multiset \( S = \left\{ n_1 \cdot s_1, \ldots, n_k \cdot s_k \right\} \) (up to renaming of its variables), where, for all \( 1 \leq i \leq k \), \( s_i \in (D \cup Ag(P)) \), and \( n_i \) is a natural number indicating the number of occurrences of \( s_i \) in \( S \).

To define the CHARM corresponding to a given cc program \( P \) (together with its underlying constraint system), we have to define the algebra of its states and its rewrite rules. As for the algebra, the idea is to consider both token and agent occurrences as process instances; the variables they involve as variables, and tokens and agent names as types.

**Definition 5.4 (The algebra of the states).** Given a program \( P \) and a constraint system \( \langle D, \vdash \rangle \), we will consider the algebra of states \( S_{cc} \) over the set of variables \( V \) which occur in \( P \) and \( \langle D, \vdash \rangle \), the set of process types \( Ag(P) \cup D \), and the set of process instances \( \{ \langle i, t \rangle \mid t \in Ag(P) \cup D, i \in \mathbb{N} \} \) (\( \mathbb{N} \) is the set of natural numbers), such that \( pt(\langle i, d \rangle) = d \), for any \( d \in D \), and \( pt(\langle i, a \rangle) = a \) for each \( a \in Ag(P) \). Variable typing is not present in cc programming and thus we will not specify it here (or we could specify the trivial variable typing function \( vt : V \rightarrow \mathcal{T}_v \) where \( \mathcal{T}_v = \{ t_0 \} \), which associates the same type \( t_0 \) to any variable).

Now we give the correspondence between cc states and CHARM states, and we show that all CHARM states corresponding to cc states are closed and without variable atoms. We recall that a CHARM state is an equivalence class of terms of the algebra of states of the considered CHARM. Therefore, whenever we say that a CHARM state has some property, like being closed and without variable atoms, we mean that there is a term in the equivalence class represented by the considered state which has that property.

**Definition 5.5 (From cc computation states to CHARM states).** We associate (the equivalence class of) a term of the algebra \( S_{cc} \) to each cc computation state. Consider a cc computation state \( S = \left\{ n_1 \cdot s_1, \ldots, n_k \cdot s_k \right\} \). Then the state over the algebra
\( \mathcal{E}_{cc} \) corresponding to \( S \) is the equivalence class \( \mathcal{E}_{cc}(S) = [ts] \), where \( ts \) is
\[
(\langle 1, s_1 \rangle, \langle x_1 \rangle) \parallel ... \parallel (\langle 1, s_1 \rangle, \langle x_1 \rangle) \parallel ... \parallel (\langle 1, s_k \rangle, \langle x_k \rangle) \parallel ... \parallel (\langle 1, s_1 \rangle, \langle x_1 \rangle) \parallel ... \parallel (\langle 1, s_k \rangle, \langle x_k \rangle)
\]
This defines a function \( f_S \) from \( cc \) states to CHARM states.

**Proposition 5.6** (\( cc \) states are closed CHARM states without variable atoms). Consider a \( cc \) state \( S \) and the corresponding CHARM state \( Y_{ce}(S) \). Then \( Y_{ce}(S) \) is closed and without variable atoms.

**Proof.** since \( Y_{ce}(S) = [ts] \), it is enough to prove that \( ts \) is closed and without variable atoms. By Definition 5.5, all variables and process instances appearing in \( ts \) are bound by a hiding operator, and thus they are not free. Therefore, \( ts \) is closed. Moreover, all atoms of \( ts \) are of the form \( \langle i, s_j \rangle, \langle x_j \rangle \) where \( s_j \in (D \cup Ag(P)) \). Therefore, no variable atom is present in \( ts \). \( \square \)

**Proposition 5.7** (Closed CHARM states without variable atoms are \( cc \) states). Consider the function \( f_S \) from \( cc \) states to CHARM states, induced by Definition 5.5. Then \( f_S \) is bijective.

Now we must specify the rewrite rules of the CHARM corresponding to a given \( cc \) program \( P \) and constraint system \( \langle D, \vdash \rangle \). The idea is to have one rule for each agent and procedure declaration present in \( P \) and also for each entailment pair present in the constraint system \( \langle D, \vdash \rangle \). In the following, in order to simplify the notation, we will use names like \( r, s, r_1, \ldots \) for the process instances instead of those prescribed by Definition 5.4 of the form \( \langle i, s \rangle \).

**Definition 5.8** (Rewrite rules for \( cc \) programs). Given a \( cc \) program \( P \) and a constraint system \( CS = \langle D, \vdash \rangle \), we define the set \( \mathcal{R}(P, CS) \) of rewrite rules as follows:
- \( A(x_1, \ldots, x_n) = \text{tell}(c(y_1, \ldots, y_m)) \rightarrow A_1(z_1, \ldots, z_k) \) in \( Ag(P) \) implies \( r(x_1, \ldots, x_n): A \xrightarrow{r} (s(y_1, \ldots, y_m): c \parallel r_1(z_1, \ldots, z_k): A_1) \backslash s \backslash r_1 \) in \( \mathcal{R}(P, CS) \);
- \( A(x_1, \ldots, x_n) = \sum_{i=1, \ldots, n} \text{ask}(c_i(y_{1,i}, \ldots, y_{m,i})) \rightarrow A_i(z_{1,i}, \ldots, z_{k,i}) \) in \( Ag(P) \) implies \( (r(x_1, \ldots, x_n): A \parallel s_i(y_{1,i}, \ldots, y_{m,i}): c_i) \xrightarrow{r} (s_i(y_{1,i}, \ldots, y_{m,i}): c_i) \parallel r_i(z_{1,i}, \ldots, z_{k,i}): A_i) \backslash r_i \) in \( \mathcal{R}(P, CS) \), for all \( i = 1, \ldots, n \);
- \( A(x_1, \ldots, x_n) = A_1(y_1, \ldots, y_m) \parallel A_2(z_1, \ldots, z_k) \) in \( Ag(P) \) implies \( r(x_1, \ldots, x_n): A \xrightarrow{r} (r_1(y_1, \ldots, y_m): A_1 \parallel r_2(z_1, \ldots, z_k): A_2) \backslash r_1 \backslash r_2 \) in \( \mathcal{R}(P, CS) \);
An abstract machine for concurrent modular systems: CHARM

1. \( A(x_1, \ldots, x_n) = \exists y_1, \ldots, y_k. A_1(x_1, \ldots, x_n, y_1, \ldots, y_k) \) in \( Ag(P) \) implies
   \[ r(x_1, \ldots, x_n): A \rightarrow r_1(x_1, \ldots, x_n, y_1, \ldots, y_k): A_1 \backslash y_1 \ldots \backslash y_k \text{ in } \mathcal{R}(P, CS); \]

2. \( p(x_1, \ldots, x_n):= A(x_1, \ldots, x_n) \) in \( Ag(P) \) implies
   \[ r(x_1, \ldots, x_n): p \rightarrow s(x_1, \ldots, x_n): A \text{ in } \mathcal{R}(P, CS); \]

3. \( \{t_1(x_1), \ldots, t_n(x_n)\} \vdash t(x) \) implies
   \[ r_1(x_1): t_1 \ldots \parallel r_n(x_n): t_n \rightarrow (r_1(x_1): t_1 \ldots \parallel r_n(x_n): t_n \parallel r(x): t) \backslash r \text{ in } \mathcal{R}(P, CS). \]

Informally, the rule for the tell agent can be written as \( R: (G, L) \rightarrow (G, L') \in \mathcal{R} \), where
\( L \) contains agent \( A \), \( L' \) contains agent \( A_1 \) and constraint \( c \), and \( G \) contains the variables involved in \( A \); such a rule may be seen in Fig. 8. Note that the fact \( c \) is present only in the local part \( L' \) of \( S' \) does not mean that \( c \) is visible only locally. In fact, as we will explicitly see later in an example, the mechanism of rule application in the CHARM formalism makes it possible to treat local items as global ones, and thus to treat \( c \) as all the other constraints. This is due to the fact that all \( cc \) states are represented by closed CHARM states.

Each one of the rewrite rules corresponding to an ask agent can be written informally as \( R_i: (G_i, L_i) \rightarrow (G_i, L'_i) \in \mathcal{R} \), where \( L_i \) contains agent \( A \), \( L'_i \) contains agent \( A_i \), and \( G_i \) contains \( c_i \) and the variables involved by \( A_i \) and \( c_i \); one such rule may be seen in Fig. 9. Note that since the rewrite rule cannot be applied if there is no occurrence of its left-hand side in the current CHARM state, the ask suspension is given for free.

Note that in an “atomic” interpretation tell and ask operations fail if \( c \) is inconsistent with the constraints in the current state. Our rewrite rules model instead the “eventual” interpretation [23], where inconsistency is discovered sooner or later, but possibly not immediately. Thus, immediate failure is not directly modeled.

---

**Fig. 8.** The CHARM rewrite rule for the agent \( A = \text{tell}(c) \rightarrow A_1 \).

\[
\begin{array}{c|c}
S & c \\
G & A \\
L & A_1 \\
\end{array}
\]

**Fig. 9.** The CHARM rewrite rule for the agent \( A = \text{ask}(c) \rightarrow A_1 \).

\[
\begin{array}{c|c}
S & c \\
G & A \\
L & A_1 \\
\end{array}
\]
The rule for an entailment relation pair can be written informally as $R: (G, L) \rightarrow (G, L')$, where $L$ is empty, $G$ contains $t_1, \ldots, t_n$, and $L'$ contains $t$. Note that $L$ is empty, since nothing has to be deleted, and all items involved are either tested for presence and thus preserved ($t_1, \ldots, t_n$) or generated ($t$). This rule may be seen in Fig. 10.

In summary (the eventual interpretation of) a cc program $P$, together with its underlying constraint system $CS = (D, \vdash)$, is modeled by a CHARM $\mathcal{M}(P, CS)$ with as many rewrite rules as agents, procedure declarations, and pairs of the entailment relation. Note that, while the number of agents is always finite, in general there may be an infinite number of pairs in the entailment relation.

**Definition 5.9 (The CHARM implementing a cc program).** Given a cc program $P$ and its underlying constraint system $CS = (D, \vdash)$, the CHARM implementing it is defined as $\mathcal{M}(P, CS) = 9?(P, CS)$.

To state the correspondence between the behavior of the CHARM $\mathcal{M}(P, CS)$ and that of the cc program $P$ with constraint system $CS$, we refer to the graph-oriented model of computation that two of the authors gave to cc programs in [18]. Informally, in that paper each agent and each entailment pair is represented by a graph production (where for graph production we mean what is defined in Section 4), and each computation is represented by a graph derivation, i.e., a sequence of applications of graph productions. The semantics of each of such computations is then defined as a partial order over the set of computation steps (where, informally, agents or constraints which are not related by the partial order can evolve concurrently). Thus, computations which differ only for a different order of execution of concurrent computation steps will be represented by the same partial order. Therefore, a partial order is able to represent all those computations which have the same "dependency pattern".

In view of the results of the previous section and those in [18], it is easy to see that each computation of a cc program $P$ with constraint system $CS$ is here represented by a computation of the CHARM $\mathcal{M}(P, CS)$ starting from a CHARM state which is closed and without variable atoms, which implies that all the states reached during the computation are closed and without variable atoms, as the following theorem shows.

![Fig. 10. The CHARM rewrite rule for the pair $\langle C, t \rangle$ of the entailment relation $\vdash$.](image)
Theorem 5.10 (Properties of CHARM states). Consider a CHARM \( \mathcal{M} \) corresponding to a cc program, and any computation, say \( \rho: \mathcal{S} \Rightarrow \mathcal{Q} \) of such CHARM. Then if \( \mathcal{S} \) is closed and without variable atoms, \( \mathcal{Q} \) is closed and without variable atoms as well.

Proof (outline). It is easy to check that all the rewrite rules defining the given CHARM only involve states with no variable atoms. Then it can be proved by structural induction that the property of being closed and without variable atoms is preserved by the inference rules generating the algebra of CHARM computations. □

We recall that CHARM computations are subject to axioms. Thus, whenever we refer to a CHARM computation we mean its entire equivalence class of computations w.r.t. those axioms.

Since cc computations are defined in [18] as sequences of graph production applications, it means that each computation step involves the evolution of exactly one object (either an agent, which transforms itself into another agent, or a constraint, which entails another constraint). Thus, no parallelism is allowed between the evolution of two or more different objects. On the other hand, CHARM computations are sequences of CHARM transitions, each of which may involve the application of more than one rewrite rule (i.e., they are not sequential in general). As a consequence of that, it can be proved that two different cc computations which have the same dependency pattern (i.e., the same partial order) are represented by the same CHARM computation.

It is important to stress the naturality of the CHARM as an abstract machine for cc programming. In fact, the global part of the rules corresponds exactly to the fact that ask agents need constraints but do not destroy them. Such explicit representation of items which are needed for a rule to take place but are not affected by the rule application is crucial for giving the kind of truly concurrent semantics proposed in [18]. An example can help understanding the main ideas. Consider a cc program \( P \) with initial agent \( A(x, y) \), containing only two agents,

\[
A(x, y) = \text{tell}(c(x)) \rightarrow A_1(x, y),
\]

\[
A_1(x, y) = \text{ask}(c(x)) \rightarrow A_2(y)
\]

and where the underlying constraint system \( CS \) contains only one entailment pair, \( c(x) \vdash \ell(x) \).

Therefore, we have a corresponding CHARM consisting of three rewrite rules, which can be written informally as follows.

\[
R_1 = r(x, y): A \rightarrow r(s(x): c \parallel r_1(x, y): A_1) \setminus s \setminus r_1,
\]

\[
R_2 = s(s(x): c \parallel r_1(x, y): A_1) \setminus r_1 \rightarrow s(s(x): c \parallel r_2(y): A_2) \setminus r_2,
\]

\[
R_3 = s(x): c \rightarrow (s_1(x): c \parallel s_2(x): \ell) \setminus s_2.
\]
These same three rules can be seen informally in Fig. 11. Note that in this figure the notation may be misleading since what seems to be the name of an object (i.e., $A$ or $A_1$ or $A_2$ or $c$ or $t$) is instead formally its type. However, the separation between the global and local part of each of the involved states reflects the formal setting.

Now, if the initial state contains only agent $A$, which is local since everything is local in a CHARM state corresponding to a cc state, one of the possible computations can be seen in Fig. 12. This computation has three steps. In the first one, the agent $A$ is rewritten into the constraint $c$ and the agent $A_1$. In the second one, the agent $A_1$, in the presence of $c$, is rewritten into the agent $A_2$. Finally, in the third one, the presence of the constraint $c$ allows the generation of the constraint $t$. The analysis of this computation allows us to talk about some important issues which have been addressed previously in a formal way and which can now be further clarified by an informal discussion.

First, each computation state has only local objects. This simply means that this computation cannot share anything with other computations, since its states do not have any global object.

Fig. 11. The CHARM rewrite rules for program $P$ and constraint system $CS$.

Fig. 12. Three steps of a computation over $\mathcal{H}(P, CS)$. 
Second, while the rewrite rule applied in the first computation step matches exactly the current state, this is not true for the other two steps. In fact, for example, in the second step, the left-hand side of the rewrite rule is \( (s(x):c \parallel r_1(x,y):A_1) \backslash r_1 \) (i.e., \( c \) is global and \( A_1 \) is local), while the current state is \( (s(x):c \parallel r_1(x,y):A_1) \backslash c \backslash r_1 \) (i.e., both \( c \) and \( A_1 \) are local objects). However, this apparent discrepancy is handled by the inference rules for the algebra of transitions of any CHARM. In fact, the rule \( R_2 = (s(x):c \parallel r_1(x,y):A_1) \backslash r_1 \rightarrow (s(x):c \parallel r_2(y):A_2) \backslash r_2 \) can be modified by “localizing” \( c \) (or preferably, the object \( s \), whose type is \( c \)) and obtaining the rule \( R_3 = (s(x):c \parallel r_1(x,y):A_1) \backslash r_1 \rightarrow (s(x):c \parallel r_2(y):A_2) \backslash r_2 \). The left-hand side of \( R_2 \) exactly matches the current state and thus we can obtain the new state, which consists of the right-hand side of \( R_2 \) where \( c \) (i.e., \( s \)), again, appears as a local object. The same also happens in the third step. In fact, \( s \) is global in the left-hand side of \( R_3 \), while it is local in the current state. Note that such “localization” of some objects is not an additional (invisible) step, but simply reflects the fact that, given a set of rewrite rules (for example, the three rules of Fig. 11), we automatically have the algebra of transitions, which contains all the transitions which are generated by applying the inference rules of Definition 3.9. Then, a computation step, which we informally depict in our figures as the application of one of the rewrite rules to a substate of the current state, is formally a transition from the entire current state to a new state (the right-hand side of the transition).

Another important issue concerns the reason why rewrite rules deal with states with a local and a global part, and not only a local part. In fact, since the initial state of any CHARM computation is a closed state, one could think that the natural thing would be to have rewrite rules where both the left- and the right-hand side are closed states as well. In reality, nothing would change as far as the final state of a computation is concerned. However, if we did that, it would not allow us to apply some subsets of the rules in parallel, and thus we would not be able to express all the concurrency contained in the given cc program. Consider, for instance, rules \( R_2 \) and \( R_3 \). Both of them need the presence of \( c \) to be applied. Also, \( R_2 \) deletes \( A_1 \) and generates \( A_2 \), while \( R_3 \) does not delete anything and generates \( t \). Therefore, it is easy to see that they are independent of each other, and thus they can be applied in parallel to the state resulting from the first computation step. In fact, if \( \Phi[s_1] = s \), then we have

\[
(R_2 \parallel R_3[\Phi]) \backslash s: (s(x):c \parallel r_1(x,y):A_1) \backslash r_1 \backslash s \\
\rightarrow (s(x):c \parallel r_2(y):A_2 \parallel s_2(x):t) \backslash r_2 \backslash s_2 \backslash s,
\]

which is a parallel transition yielding directly the final state of the computation. In [18] this situation was represented by the fact that the computation of Fig. 12 was represented by a partial order where the second and the third steps were concurrent, i.e., unrelated. If, on the contrary, \( R_2 \) and \( R_3 \) had \( c \) in their local parts, then they could be applied in any order, but not simultaneously, to the state obtained by the first computation step. In fact, consider

\[
R_2 \rightarrow R_2 \backslash s - (s(x):c \parallel r_1(x,y):A_1) \backslash r_1 \backslash s \rightarrow (s(x):c \parallel r_2(y):A_2) \backslash r_2 \backslash s
\]
and

\[ R_3 = R_3 \setminus s_1 = s_1(x) : c \setminus s_1 \rightarrow (s_1(x) : c \parallel s_2(x) : t) \setminus s_1 \setminus s_2. \]

Then we have

\[ R_2 \parallel R_3 = (s(x) : c \parallel r_1(x, y) : A_1 \parallel s_1(x) : c) \setminus r_1 \setminus s_1 \]
\[ \rightarrow (s(x) : c \parallel r_2(y) : A_2 \parallel s_1(x) : c \parallel s_2(x) : t) \setminus r_2 \setminus s_1 \setminus s_2. \]

Now, it is easy to see that the left-hand side of such transition does not match with the state obtained by the first computation state (i.e., \((s(x) : c \parallel r_1(x, y) : A_1) \setminus r_1\)), since this state has only one occurrence of \(c\). Summarizing, global parts in rewrite rules are needed whenever we are interested in the formal description of the dependencies between computation steps or rewrite rules, and thus in the specification of the maximal level of parallelism which is contained in the given \(cc\) program.

6. Modeling place/transition Petri nets

Petri nets [20] are the first model of concurrent systems that was developed and the most widely used in many applications. Therefore, we believe that it is of great importance to show that the CHARM can also implement any Petri net and its computations. To do that, we first define a well-known class of Petri nets, the so-called "place/transition" nets, following the algebraic approach proposed in [15]. A Petri net is simply a directed graph equipped with two algebraic operations: a commutative monoid structure on nodes and the concatenation of arcs. Formally, we have the following definition.

**Definition 6.1** (Petri nets and related).

- A graph \(G\) is a set \(T\) of arcs, a set \(V\) of nodes and two functions \(\delta_0\) and \(\delta_1\) called source and target, respectively, such that \(\delta_0, \delta_1 : T \rightarrow V\).
- A commutative monoid \((M, +, 0)\) is a set \(M\) equipped with a binary function \(+ : M \times M \rightarrow M\) which is both associative and commutative, and with a distinguished element \(0 \in M\) (the unit) such that \(0 + m = m\) for all \(m \in M\).
- A (place/transition) Petri net is a graph where the arcs are called transitions and where the set of nodes is the free commutative monoid \(S^\oplus\) over a set of places \(S\), i.e., \(\delta_0, \delta_1 : T \rightarrow S^\oplus\).
- A Petri reflexive commutative monoid \(M\) is a Petri net where the set of transitions is a commutative monoid \((T, +, 0)\), where \(\delta_0, \delta_1 : (T, +, 0) \rightarrow S^\oplus\) are monoid homomorphisms, and where every element \(v\) of \(S^\oplus\) has a transition \(id(v) : v \rightarrow v\) belonging to \(T\).
- A Petri computation model is a reflexive Petri commutative monoid \(C = (\delta_0, \delta_1 : (T, +, 0) \rightarrow S^\oplus, id)\), together with a partial function \(\vdash : T \times T \rightarrow T\) which is defined exactly for those pairs \((\alpha, \beta)\) such that \(\delta_1(\alpha) \vdash \delta_0(\beta)\). In addition, the
following axioms are satisfied:

1. $\delta_0(x; \beta) = \delta_0(x)$ and $\delta_1(x; \beta) = \delta_1(\beta)$;
2. $x; id(\delta_1(x)) = x$ and $id(\delta_0(x)); x = x$;
3. $(x; \beta) \cdot y = x; (\beta; y)$;
4. Given $x: u \rightarrow v$, $x': u' \rightarrow v'$, $\beta: v \rightarrow w$, $\beta': v' \rightarrow w'$, we have
   $(x \cdot x'); (\beta \cdot \beta') = (x; \beta) \cdot (x'; \beta')$.

For each Petri net $N = (\delta_0, \delta_1: T \rightarrow S^\oplus)$, it is possible to construct a corresponding Petri computation model $\mathcal{T}[N]$ which is inductively defined by the axioms 2-4 above, and by the following inference rules:

\[
\begin{array}{c}
\text{t: } u \rightarrow v \in N \\
\text{t: } u \rightarrow v \in \mathcal{T}[N] \\
u \in S^\oplus \\
u \rightarrow u \in \mathcal{T}[N]
\end{array}
\]

\[
\begin{array}{c}
\alpha: u \rightarrow v, \beta: v \rightarrow w \in \mathcal{T}[N] \\
\alpha: u \rightarrow v, x': u' \rightarrow v' \in \mathcal{T}[N] \\
a: u \rightarrow v, \beta: u \rightarrow w \in \mathcal{T}[N] \\
\alpha + x': u \oplus u' \rightarrow v \oplus v' \in \mathcal{T}[N]
\end{array}
\]

Intuitively, $\mathcal{T}[N]$ represents the computations of $N$. In fact, it is a category whose objects are elements of $S^\oplus$, i.e., multisets of places, and whose arrows are computations from one multiset (seen as a marking) to another multiset. A relevant result presented in [15] states that the arrows of category $\mathcal{T}[N]$ from marking $m$ to marking $m'$ are in bijective correspondence with nonsequential processes of $N$ from $m$ to $m'$, as defined in [2]. Thus, $\mathcal{T}[N]$ provides a truly concurrent semantics for net $N$.

We will now model any Petri net $N$ in the CHARM formalism. The idea is to associate with $N$ a CHARM whose states are nodes of $N$ and thus objects of $\mathcal{T}[N]$ (i.e., markings of the net $N$) and whose computations are arrows of $\mathcal{T}[N]$ (i.e., computations of $N$).

The CHARM we associate with $N$, written $\mathcal{M}(N)$, is defined over an algebra of states $\mathcal{S}(N)$ with no variables but only process instances, which represent tokens of the Petri net. The typing function, applied to each one of such tokens, gives the name of the place containing the token.

**Definition 6.2** (Algebra of states for a Petri net). Given a Petri net $N$, the algebra of the states of $\mathcal{M}(N)$, denoted $\mathcal{S}(N)$, has $\mathcal{V} = \emptyset$, $\mathcal{F}_a = \emptyset$, $\mathcal{F}_p = S$, $\mathcal{P} = \{ \langle i, s \rangle | i \in \mathbb{N}, s \in S \}$, and $\text{pt}(\langle i, s \rangle) = s$.

Now, each marking of the Petri net $N$ is going to be represented by a closed state over the just defined algebra of states.

**Definition 6.3** (Markings and CHARM states). Given a Petri net $\delta_0, \delta_1: T \rightarrow S^\oplus$, let us consider any marking, i.e., an element of the set $S^\oplus$, of the form $m = n_1 a_1 \oplus \cdots \oplus n_k a_k$, where $n_i$ is a natural number and $a_i \in S$ for all $i = 1, \ldots, n$. Then the corresponding state
The correspondence between markings of $N$ and closed states over $\mathcal{S}(N)$ is bijective. In fact, since the names of the $x_i$'s do not matter, due to the $\alpha$-conversion axiom, each state can be seen as an element of the free commutative monoid over $S$, just as in the Petri net.

To complete the definition of the CHARM $\mathcal{M}(N)$ we have to define its rewrite rules. Quite obviously, every transition of the given Petri net $N$ is translated into a rewrite rule.

**Definition 6.4 (CHARM associated with a place/transition Petri net).** Given a Petri net $N=(\delta_0, \delta_1; T\rightarrow S^\circ)$, its associated CHARM $\mathcal{M}(N)$ is the CHARM over the algebra of states $\mathcal{S}(N)$ of Definition 6.2, defined as

$$\mathcal{M}(N) = \{\mathcal{M}(t) : st(s_1) \rightarrow st(s_2) | (t : s_1 \rightarrow s_2) \in T\},$$

i.e., $\mathcal{M}(N)$ contains one rewrite rule $\mathcal{M}(t)$ for each transition $t$ of net $N$.

It is now easy to prove that the computations of given Petri net $N$ (i.e., the arrows of category $\mathcal{F}(N)$) are in bijective correspondence with the computations of the CHARM $\mathcal{M}(N)$ just defined. To do that, it is enough to note that the algebra of computations of the CHARM coincide with the algebra of the arrows of category $\mathcal{F}(N)$. In fact, the axioms and the inference rules that such algebras have to satisfy are exactly the same. This can be checked by looking at the axioms given in this section for any Petri computation model (in Definition 6.1), and at those in Section 3 for the algebra of computations (in Definition 3.10).

Since $st(m)$ is a closed state for any marking $m$, this means that all the rewrite rules of the CHARM corresponding to a Petri net deal with closed states only, i.e., each of these rewrite rules rewrites a closed state into a closed state. In terms of Petri nets, this means that each precondition of an event is deleted and each postcondition is generated. Recently, however, condition/event systems (which are a particular variant of place/transition Petri nets) have been extended in [19] to deal with conditions, called context conditions, which are neither deleted nor generated, but are simply checked for existence. That is, a net transition can be written in [19] as $s_1 \xrightarrow{s_3} s_2$, which can be interpreted as the event of going from the marking $s_1$ to the marking $s_2$ only if the marking $s_3$ is present. Clearly, in the CHARM framework it is also possible to model faithfully this extended setting. In fact, such a rule would correspond to a CHARM rewrite rule $st(s_3, s_1) \rightarrow st(s_3, s_2)$, where $st(s, s')$ can be defined as the CHARM state which has $s$ as its global part and $s'$ as its local part.
7. Conclusions and future work

In this paper we introduced an abstract rewriting machine, called CHARM, which is able to model in a faithful way the operational behavior of several formalisms proposed in the literature for describing concurrent systems. A CHARM is essentially a structured transition system, i.e., a transition system where both states and transitions have the same algebraic structure. The computations of a CHARM are intrinsically concurrent, and the algebraic framework provides some handy tools for the definition of a truly concurrent semantics. To show the expressiveness of this abstract machine, three relevant computational formalisms are mapped onto the CHARM framework: graph grammars, concurrent constraint programming, and place/transition Petri nets.

In [12] it has been shown that concurrent constraint programming may encode the lazy and the call-by-value λ-calculus. This encoding exploits a technique similar to the one used by Milner to encode λ-calculus in π-calculus [17], since the mobility of processes (which is one of the main features of π-calculus) can be simulated in cc programming via a clever use of the shared logical variables. This result, combined with our implementation of cc programming in the CHARM, described in Section 5, also suggests that certain higher-order aspects of functional languages may be expressed within the CHARM.

References


