PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT. © ICOT, 1992

CHARM: Concurrency and Hiding in an Abstract Rewriting Machine*

Andrea Corradini **Ugo Montanari** **Francesca Rossi**

University of Pisa
Computer Science Department
Corso Italia 40, 56100 Pisa, Italy
{andrea,ugo,rossi}@dipisa.di.unipi.it

Abstract

CHARM (for Concurrency and Hiding in an Abstract Rewriting Machine) is an abstract machine which allows to naturally model the behaviour 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 expressivness and generality of this abstract machine, two relevant computational formalisms, graph grammars and concurrent constraint programming, are mapped onto the *CHARM* framework.

1 Introduction

Various formalisms have been proposed in the last decades for describing and specifying concurrent programming and distributed systems. Among them we recall Petri nets [Reisig 1985], CCS [Milner 1989], CSP [Hoare 1985], the Chemical Abstract Machine [Berry and Boudol 1990], Graph Grammars [Ehrig 1979], and Concurrent Constraint Programming [Saraswat 1989]. However, 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, so to be able to prove interesting properties about it. We have found that a reasonable balance of generality and expressiveness can be enjoyed by a formalism able to express in a simple way both concurrency and modularity. In fact, such notions are fundamental in order to describe how concurrent systems interact and 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. Such a machine exhibits a sophisticated treatment of the above cited issues of concurrency and modularity, which subsumes and surpasses the corresponding treatment of many other formalisms.

States of a *CHARM* are collections of processes interacting through shared variables. The issue of modularity is addressed basically by partitioning the states into a global (i.e., visible) and a local (i.e., hidden) part. In fact, the global items of a system are those which allow the interaction with other systems, and thus are used to compose them in a non-trivial way. Transitions of the machine are rewrite rules described by pairs of systems with an identical global part, which expresses 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). The presence of a global and a local part allows also a degree of concurrency higher than the one provided, for example, by the Chemical Abstract Machine [Berry and Boudol 1990] or by Petri nets [Reisig 1985]. In fact, two 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.

The technique used for the formal definition of the *CHARM* follows the algebraic approach introduced in [Meseguer and Montanari 1990] for Petri nets, and further developed for structured transition sys-

*Research partially supported by the GRAGRA Basic Research Esprit Working Group n.3299 and by Alenia S.p.A.

tems in [Corradini 1990,Corradini et al. 1990] and for concurrent rewriting systems in [Meseguer 1990]. 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 [Ehrig 1979], 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 [Saraswat and Rinard 1990] [Saraswat et al. 1991], 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. Notice that the ability of expressing concurrent constraint programming in the *CHARM* framework is very significant, since such a paradigm is already very general and subsumes many widely used programming paradigms like logic programming [Lloyd 1987], constraint logic programming [Jaffar and Lassez 1987], and concurrent logic programming [Shapiro 1989].

Although this issue is not addressed in this paper, we are confident that also process description languages, like CCS [Milner 1989] and CSP [Hoare 1985], can be modelled within the *CHARM* framework. This hope is supported by the fact that we use an algebraic approach, where some basic operators of such languages (i.e., parallel composition, hiding, and relabelling) are already present, while other mechanisms (like synchronization and non-deterministic choice) may be coded via suitable techniques, as we will hint in Section 2.

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 and concurrent constraint programming in Sections 4 and Section 5 respectively.

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 enlighten some of its advantages w.r.t. 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 thus may be interpreted in various way. For example, processes may also be though of as predicates or constraints or relations, and variables may represent communication channels or shared data structures. It is important to notice that many of the approaches proposed to represent the evolution of concurrent systems [Berry and Boudol 1990] [Reisig 1985] cannot model directly 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 indicated 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 (resp., global) part as the hidden (resp., 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.

States can be built from smaller states. For example, the parallel composition of two states (given later by the operator "$|$") 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. In the state resulting from the composition, it is possible to force some items, which were global in both the composing sub-states, to become local. This can be done by using a suitable *hiding* operator, which will be denoted by "$\backslash$". Parallel composition and hiding, together with a *renaming* operator (denoted by "$[\Phi]$") define an algebra (introduced in the next section) whose terms are the states of a *CHARM*.

The dynamic behaviour of a *CHARM* is given by a collection of rewrite rules. Every rewrite rule $R : S \to S'$ maps its left-hand side $S = (G, L)$ to its right-hand side $S' = (G, L')$, both having the same global part G, which is also called the global part of R. The graphical representation of a rewrite rule can be seen in Figure 1. The idea is that L can be cancelled 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. Although this goes beyond the scope of this paper, this fact should allow us to define a satisfactory truly concurrent semantics for *CHARM*'s, since it minimizes the causal dependencies among rewrite rules.

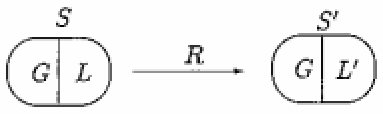

Figure 1: A rewrite rule.

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. It is important at this point to notice that, unlike our approach, many transition systems or abstract machines proposed in the literature (like Petri nets [Reisig 1985] and the Chemical Abstract Machine [Berry and Boudol 1990]) cannot distinguish between the situation where some item is preserved by a rewrite rule, and the one where the same item is cancelled 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, some other formalisms explicitly consider the issue of context-dependent rewriting, and allow one to formally indicate 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 [Ehrig 1979], the role of the context is played by the so-called "gluing graph" of a graph production, while in concurrent constraint programming [Saraswat 1989] [Saraswat and Rinard 1990] [Saraswat et al. 1991], the items the presence of which must be tested are explicitly mentioned by the use of the "ask" primitive. The relationship between the *CHARM* and these two formalisms will be explored deeply in later sections of this paper.

A rewrite rule R from $S = (G, L)$ to $S' = (G, L')$ can be thought of as modelling the evolution of a (small) subsystem, represented by its left hand side S. To apply this rule to a given state $Q = (G_Q, L_Q)$, one first has to find an occurrence of S in Q, i.e., a subsystem of Q "isomorphic" to S (as we shall see later, this requirement will be relaxed in the formal definitions). Following the usual intuition about structured systems, it is evident that all the items which are local for a part of a system are local for the whole system as well, while items which are global for a subsystem can be either global or local for any enclosing system. In the application of a rule like the one above, this observation is formalized by requiring that the occurrence of L in Q is contained in its local part L_Q.

The application of R to a system Q yields a new system $Q' = (G_{Q'}, L_{Q'})$, where $G_{Q'} = G_Q$, i.e., the global part remains unchanged, and $L_{Q'} = (L_Q \setminus L) \cup L'$. In words, the local part of the new state coincides with the local part of the old one, except that the occurrence of L has been replaced by an occurrence of L'. Thus, the part of the state Q which is preserved by the application of R is partitioned in two parts: the occurrence of G, which is necessary to apply R, and the rest, which does not take part in the rewriting. The graphical representation of the application of R to Q can be seen in Figure 2.

The fact that the application of a rewrite rule preserves the global part of the state can be justified by interpreting the items contained in the global part as an interface for a possible composition with other states. Thus, such an interface cannot be modified by any rewrite rule, since a rewrite rule is local to the rewritten state. Notice that, as a consequence of the above considerations, a closed system, i.e., a system which is not supposed to be composed further, is represented by a state with no global part.

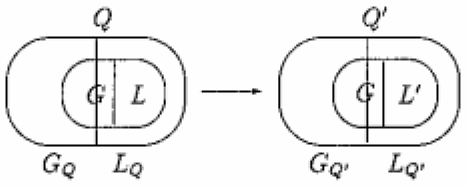

Figure 2: The application of a rewrite rule.

The above construction describes the application of a single rewrite rule of a *CHARM* to a 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 supported also by many other models of concurrent computation, like Petri nets [Reisig 1985], the Chemical Abstract Machine [Berry and Boudol 1990], and the concurrent rewriting of [Meseguer 1990]. 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 fact reflects the intuition, since 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 [Ehrig 1979], which can in fact be faithfully implemented within the *CHARM* framework, as we will see in Section 4.

From a technical point of view, the application of one or more rules of a *CHARM* is modelled by extending the algebra of states to the rules (for similar approaches in the case of Petri nets or structured transition systems see [Meseguer 1990], [Corradini et al. 1990] and [Meseguer and Montanari 1990]). As we shall see in Section 3, 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 $S : S \to S$ for each state S, and it is closed w.r.t. parallel composition, hiding, and substitution operations. The left and the right-hand sides of a transition (i.e., of an element of the algebra of transitions) are easily obtained by structural induction from its syntax. For example, if $R : S \to Q$ and $R' : S' \to Q'$ are two rewrite rules, then $R \mid R' : S \mid S' \to Q \mid Q'$ is a new *parallel* transition. Like rewrite rules, transitions preserve the global part of the state they are applied to.

In this paper we will assume that the algebra of transitions is freely generated by the set of rewrite rules defining a *CHARM*, and by the identity rules for all states. As a consequence of this fact, if a transition can be applied to a state S, then it can be applied to any state containing S as well. Informally, this can be considered as a meta-rule governing the behaviour of a *CHARM*, and directly corresponds to the so-called "membrane law" of the Chemical Abstract Machine [Berry and Boudol 1990].

Although the choice of a free algebra of transitions is satisfactory for the formalisms treated in this paper, more general kinds of algebras would be needed in order to deal with other formalisms, like for example process description languages [Milner 1989] [Hoare 1985]. In fact, some features of those languages (e.g., the parallel composition of agents with synchronization in the presence of restriction, and the description of atomic sequences of actions, useful to provide a low-level implementation of the non-deterministic choice operator "+" [Gorrieri et al. 1990]
[Gorrieri and Montanari 1990]) cannot be modelled adequately within a free algebra of transitions. Nevertheless, as shown in [Ferrari 1990] and [Gorrieri and Montanari 1990] respectively, both those aspects can be faithfully modelled in an algebraic framework by labelling transitions with observations which include an error label and by specifying suitable algebraic theories of computations where the atomic sequences are basic operators. Thus we are confident that, although this goes beyond the scope of this paper, these topics could be fruitfully addressed in the algebraic framework introduced here, by slightly generalizing the construction of the algebra of transitions of *CHARM*'s presented in the next section.

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 of 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 algebra of computations allows one to relate the global evolution of a closed system to the local behaviour of its subsystems. For example, suppose to consider the closed system $P = (S \mid S')\backslash x$, where the two subsystems S and S' cooperate through the common global variable x which is hidden by the use of the $\backslash x$ 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 $\sigma = (\rho \mid \rho')\backslash x$ which models the evolution of the closed system P, i.e., $\sigma : P \Rightarrow P'$, where $P' = (Q \mid Q')\backslash x$.

The algebra of computations provides also some basic mechanisms which should allow to model process synchronization. In fact, consider for example the two computations $\sigma = (\rho \mid \rho')\backslash x$ and $\sigma' = (\rho\backslash x) \mid (\rho'\backslash x)$. Now, $\sigma \neq \sigma'$, since ρ and ρ' can synchronize through the common variable x in σ, but not in σ'.

Another relevant advantage of the definition of the algebra of computations of a *CHARM* consists of 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 computation σ' introduced above, we have that $\sigma' = (\rho\backslash x) \mid (\rho'\backslash x) = ((\rho\backslash x) \mid S')\ ;\ (Q \mid (\rho'\backslash x))$, where S' and Q stand for the identity computations on such states. This means that, since $\rho\backslash x$ and $\rho'\backslash x$ are independent, they can be performed either in parallel or sequentially, and the two resulting computations are equivalent.

With each equivalence class of computations it is possible to associate a partial ordering, recording the causal dependencies among the rewrite rules used in the computations. For the two formalisms we shall consider in this paper (i.e., graph grammars and concurrent constraint programming) the truly concurrent semantics obtained via their translation to a *CHARM* is significant. In fact, it is possible to show that some of the classical results about concurrency and parallelism in graph grammars directly derive from the axioms of our algebra. Also, the truly concurrent semantics proposed in [Montanari and Rossi 1991] for the concurrent constraint programming framework coincides with the one induced by its compilation into a *CHARM*. However, the true concurrency aspects go beyond the scope of this paper.

3 Formal Definitions

In this section we present the formal description of a *CHARM*, following the outline of 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 term. Next we will introduce the algebra of transitions and the algebra of computations of a *CHARM*, respectively.

The states of a *CHARM* are going to be represented by the terms of an algebra $\mathcal{S}$, which is parametric w.r.t. a fixed pair of disjoint infinite collections $(\mathcal{P}, \mathcal{V})$, called *process instances* and *variables* respectively. The terms of this algebra are subject to the axioms presented below in Definition 3.

Definition 1 *Let $\mathcal{P}$ be a set of* process instances *(ranged over by p, q, ...), and $\mathcal{V}$ be a set of* variables *(ranged over by v, z, ...). Each $x \in (\mathcal{P} \cup \mathcal{V})$ is called an* item. *The algebra of states $\mathcal{S}$ is the algebra having as elements the equivalence classes of terms generated by the following syntax, modulo the least equivalence relation induced by the axioms listed in Definition 3:*

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

where v, v_1, ..., $v_n \in \mathcal{V}$; $p \in \mathcal{P}$; "$\mid$" is called parallel composition*; Φ is a (finite domain) substitution, i.e., a function $\Phi : (\mathcal{P} \cup \mathcal{V}) \to (\mathcal{P} \cup \mathcal{V})$ such that $\Phi(\mathcal{V}) \subseteq \mathcal{V}$, $\Phi(\mathcal{P}) \subseteq \mathcal{P}$, and such that the set of items for which $x \neq \Phi(x)$ is finite; and x is an item ($\backslash x$ is called a* hiding *operator). Term of the form 0, v, or $p(v_1, \ldots, v_n)$ are called* atoms.∎

Intuitively, 0 is the empty system, v is the system containing only one variable, and $p(v_1, \ldots, v_n)$ represents a system with one process which has access to n variables. The term $S_1 \mid S_2$ represents the composition of system S_1 and system S_2, $S[\Phi]$ is the system obtained from state S by renaming its items, and $S\backslash x$ is the system which coincides with system S except that item x is local.

Definition 2 *Given a term $S \in \mathcal{S}$, its set of* free items *$\mathcal{F}(S)$ is inductively defined as $\mathcal{F}(0) = \emptyset$; $\mathcal{F}(v) = \{v\}$; $\mathcal{F}(p(v_1, \ldots, v_n)) = \{p, v_1, \ldots, v_n\}$; $\mathcal{F}(S_1 \mid S_2) = \mathcal{F}(S_1) \cup \mathcal{F}(S_2)$; $\mathcal{F}(S[\Phi]) = \Phi(\mathcal{F}(S)) = \{\Phi(x) \mid x \in \mathcal{F}(S)\}$; and $\mathcal{F}(S\backslash x) = \mathcal{F}(S) \setminus \{x\}$ if $x \in \mathcal{F}(S)$ and $\mathcal{F}(S\backslash x) = \mathcal{F}(S)$ otherwise. A term S is* closed *iff $\mathcal{F}(S) = \emptyset$. A term is* concrete *if it does not contain any hiding operator. A term is* open *if no variable appearing in the term is restricted. Formally, all atoms are open; $S_1 \mid S_2$ is open if both S_1 and S_2 are open; $S[\Phi]$ is open if S is open; and $S\backslash x$ is open if S is open and $x \notin \mathcal{F}(S)$. Clearly, all concrete terms are open.*∎

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 an open term corresponds to a system where everything is global. The above interpretation of the operators of the algebra of states is supported by the following axioms, which determine when two terms are equivalent, i.e., represent the same system.

Definition 3 *The terms of algebra $\mathcal{S}$ introduced in Definition 1 are subject to the following conditional axioms.*

ACI: $(S_1 \mid S_2) \mid S_3 = S_1 \mid (S_2 \mid S_3)$; $S_1 \mid S_2 = S_2 \mid S_1$; $S \mid 0 = S$

ABS: $p(v_1, \ldots, v_n) \mid v_i = p(v_1, \ldots, v_n)$, for $1 \leq i \leq n$;
$S \mid S = S$, if S is open
COMP: $S[\Phi][\Psi] = S[\Psi \circ \Phi]$
EXC: $S \backslash x \backslash y = S \backslash y \backslash x$
EL: $S \backslash x = S$, if x is not free in S
MAP: $p(v_1, \ldots, v_n)[\Phi] = \Phi(p)(\Phi(v_1), \ldots, \Phi(v_n))$;
$v[\Phi] = \Phi(v)$; $0[\Phi] = 0$
DIS: $(S_1 \mid S_2)[\Phi] = S_1[\Phi] \mid S_2[\Phi]$
FAC: $S_1 \backslash x \mid S_2 = (S_1 \mid S_2) \backslash x$, if x is not free in S_2
SWAP: $(S \backslash x)[\Phi] = S[\Phi] \backslash \Phi(x)$, if $\not\exists y \in \mathcal{F}(S \backslash x)$ such that $\Phi(y) = \Phi(x)$
α-CONV: $S[\Phi] = S$, if Φ is bijective and $\Phi(x) = x$ $\forall x \in \mathcal{F}(S)$

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

In words, axioms ACI and ABS state that the parallel composition of systems behaves like disjoint union on the local parts of systems, and like set union on the global ones. Axioms $COMP$ and MAP deal with substitution composition and application respectively, while axiom DIS states that substitutions distribute over parallel composition. Axioms EXC and EL state that the items of a system can be made local only once and in any order. Finally, axioms FAC and $SWAP$ describe in an obvious way the interplay between the hiding operator and the operators for parallel composition and substitution, while axiom α-CONV formalizes the intuition that the names of the hidden items are not meaningful.

As anticipated in Section 2, the rewrite rules which define a $CHARM$ must preserve the global part of the states they can be applied to. Therefore we define a function $\mathcal{GP}$ which extracts from each term a concrete subterm, corresponding to its global part. Actually, $\mathcal{GP}$ is a partial function on $\mathcal{S}$, because some term may denote a system whose global part is not a legal system. This happens when a variable is made local, but some process using it is considered as global. Terms on which $\mathcal{GP}$ is defined are called well-formed. The function $\mathcal{GP}$ is defined by exploiting the existence of a canonical form of terms.

Proposition 4 *Every term S of the algebra of states $\mathcal{S}$ has an equivalent canonical form $S_1 \mid S_2 \backslash x_1 \ldots \backslash x_n$, where S_1 and S_2 are parallel compositions of atoms. If $S_2 = S_{21} \mid \ldots \mid S_{2k}$, then $\forall i = 1, \ldots, k$, either $S_{2i} = 0$ or $\mathcal{F}(S_{2i}) \cap \{x_1, \ldots, x_n\} \neq \emptyset$. Moreover, for each atom of the form $q(v_1, \ldots, v_m) \in S_2$, $\forall i = 1, \ldots, m$, either $v_i = x_j$ for some $j = 1, \ldots, n$, or v_i occurs in S_1. If $S_1 \mid S_2 \backslash x_1 \ldots \backslash x_n$ and $S_1' \mid S_2' \backslash y_1 \ldots \backslash y_k$ are two canonical forms for term S, then $S_1 \approx S_1'$.* ■

Definition 5 *Let S be a term and $S_1 \mid S_2 \backslash x_1 \ldots \backslash x_n$ be one of its canonical forms. Term S is* well-formed *if and only if for each atom of the form $q(v_1, \ldots, v_m) \in S_2$, $q = x_i$ for some i. Then, the global part of a well-formed term S is defined as $\mathcal{GP}(S) = S_1$.*■

Definition 6 *A* rewrite rule *R over $\mathcal{S}$ is a pair of well-formed terms of $\mathcal{S}$, $R = (S, S')$ (also written $R : S \rightarrow S'$) such that $\mathcal{GP}(S) \approx \mathcal{GP}(S')$. A* CHARM *$\mathcal{M}$ (over $\mathcal{S}$) is a collection of rewrite rules over $\mathcal{S}$, i.e., $\mathcal{M} = \{R_i : S_i \rightarrow S_i'\}_{i \in I}$.*■

Definition 7 *Let $\mathcal{M} = \{R_i : S_i \rightarrow S_i'\}_{i \in I}$ be a* CHARM *over $\mathcal{S}$. Then the* algebra of transitions *of $\mathcal{M}$, $\mathcal{T}(\mathcal{M})$, is generated by the following inference rules, which also give the left and the right-hand side of each transition.*

$$\frac{i \in I}{R_i : S_i - S_i'} \qquad \frac{S \in \mathcal{S}}{S : S - S}$$

$$\frac{T : S - Q,\ T' : S' - Q'}{T \mid T' : S \mid S' - Q \mid Q'} \qquad \frac{T : S - Q}{T[\Phi] : S[\Phi] - Q[\Phi]}$$

$$\frac{T : S - Q}{T \backslash x : S \backslash x - Q \backslash x}$$

The free items of a transition are the free items of its left (or right) hand side. A transition is open *iff it has the form $S : S \rightarrow S$, with S open. The terms of $\mathcal{T}(\mathcal{M})$ are subject to the same axioms as in Definition 3.*■

Definition 8 *Let $\mathcal{M} = \{R_i : S_i \rightarrow S_i'\}_{i \in I}$ be a* CHARM *over $\mathcal{S}$, and $\mathcal{T}(\mathcal{M})$ be its algebra of transitions. Then the* algebra of computations *of $\mathcal{M}$, $\mathcal{C}(\mathcal{M})$, is generated by the following inference rules, where $\rho : S \Rightarrow S'$ means that computation ρ starts from state S and ends in state S':*

$$\frac{T : S - Q \in \mathcal{T}(\mathcal{M})}{T : S \Rightarrow Q} \qquad \frac{\rho : S \Rightarrow S',\ \rho' : S' \Rightarrow S''}{\rho; \rho' : S \Rightarrow S''}$$

$$\frac{\rho : S \Rightarrow Q,\ \rho' : S' \Rightarrow Q'}{\rho \mid \rho' : S \mid S' \Rightarrow Q \mid Q'} \qquad \frac{\rho : S \Rightarrow Q}{\rho[\Phi] : S[\Phi] \Rightarrow Q[\Phi]}$$

$$\frac{\rho : S \Rightarrow Q}{\rho \backslash x : S \backslash x \Rightarrow Q \backslash x}$$

The free items of a computation are the free items of its starting (or ending) state. The terms of $\mathcal{C}(\mathcal{M})$ are subject to the same axioms as in Definition 3, plus the following functoriality axioms, *valid whenever both sides are defined, stating that the operations of the algebra distribute over sequential composition.*

$$(\rho \mid \rho'); (\sigma \mid \sigma') = (\rho; \sigma) \mid (\rho'; \sigma')$$
$$(\rho; \sigma)[\Phi] = \rho[\Phi]; \sigma[\Phi]$$
$$(\rho; \sigma) \backslash x = \rho \backslash x; \sigma \backslash x$$ ■

4 Modelling 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, like 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 shortly show how to translate a graph grammar into a *CHARM* which faithfully implements its behaviour. Because of space limitations, the discussion will be very informal: a more formal presentation of this translation can be found in [Corradini and Montanari 1991].

Following the so-called *algebraic approach* to graph grammars [Ehrig 1979], a graph production $p = (L \overset{l}{\hookleftarrow} K \overset{r}{\hookrightarrow} R)$ 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. Throughout this section, for graph we mean *unlabelled, directed hypergraph*, i.e., a triple $G = (N, E, c)$, where N is a set of *nodes*, E is a set of *edges*, and $c : E \rightarrow N^*$ is the *connection function* (thus each edge can be connected to a list of nodes). Production p can be applied to a graph G yielding H (written $G \Rightarrow_p H$) if there is an *occurrence* (i.e., a graph morphism) $g : L \rightarrow G$, and H is obtained as the result of the *double pushout* construction of Figure 3.

$$\begin{tikzcd}
L \arrow[d, "g"'] \arrow[dr, phantom, "\texttt{PushOut}"] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d, "k"] \arrow[dr, phantom, "\texttt{PushOut}"] & R \arrow[d, "h"] \\
G & D \arrow[l, "d"] \arrow[r, "b"'] & H
\end{tikzcd}$$

Figure 3: Graph rewriting via double pushout construction.

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., we have to find a graph D (with morphism $k : K \rightarrow D$ and $d : D \rightarrow G$) such that the resulting square is a "pushout". Intuitively, graph G in Figure 3 is the pushout object of morphisms l and k if it is obtained from the disjoint union of L and D by identifying the images of K in L and in D. Next, we have to embed the right-hand side R in D via a second pushout, which produces graph H. In this case we say that there is a *direct derivation* form G to H via p.

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 $G \Rightarrow_{p_1} G_1 \Rightarrow_{p_2} \dots \Rightarrow_{p_n} G_n = H$, where $p_1, \dots, p_n$ 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 (see Definition 1). Quite obviously, we can regard a graph as a distributed system where the edges are processes, and the nodes are variables. Thus we consider a *CHARM* over the pair of sets $(\mathcal{E}; \mathcal{N})$ which are two collections including all edges and all nodes, respectively. The precise relationship between the algebra of states of such a *CHARM* and the graphs introduced above has been explored in [Corradini and Montanari 1991]. It has been shown there that concrete terms of such an algebra (i.e., terms without hiding operators, see Definition 2), faithfully model finite graphs, i.e., if *FGraph* is the collection of all finite graphs and CS is the sub-algebra of concrete terms of $\mathcal{S}$, there are injective functions $Gr : CS \rightarrow FGraph$ and $Tm : FGraph \rightarrow CS$ such that $Gr(Tm(G)) \cong G$ for each graph G. Furthermore, well-formed terms (see Definition 5) model in a similar way "partially abstract graphs", i.e., suitable equivalence classes of graph monomorphisms, where the target graph is defined up to isomorphism. For our goals, it is sufficient to introduce the function WfT which associates a well-formed term with each graph monomorphism.

Definition 9 *Let $G = (N, E, c)$ be a graph, with $N = \{n_i\}_{i \leq m}$, $E = \{e_j\}_{j \leq r}$, and $c(e_i) = n_{i1} \cdot \ldots \cdot n_{ik_i}$ for all $1 \leq i \leq r$. Then the concrete term representing G is defined as*
$Tm(G) = n_1 \mid \dots \mid n_m \mid e_1(n_{11}, \dots, n_{1k_1}) \mid \dots \mid e_r(n_{r1}, \dots, n_{rk_r}) \mid 0.$

Let $h : G \hookrightarrow H$ be a graph monomorphism. Then the well-formed term representing h is defined as

$$WfT(h) = (Tm(H) \backslash x_1 \backslash \dots \backslash x_n)[h^{-1}]$$

where $\{x_1, \dots, x_n\}$ 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). ∎

From the last definition it can be checked that the global part of the well-formed term representing a monomorphism $h : G \hookrightarrow H$ is equivalent to the concrete term representing G, i.e., $Tm(G)$. 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 6) with each graph production.

Definition 10 *Let $\mathcal{R}$ be a graph rewriting system. For each graph production $p = (L \overset{l}{\hookleftarrow} K \overset{r}{\hookrightarrow} R)$ in $\mathcal{R}$, its associated rewrite rule $\mathcal{M}(p)$ is defined as $\mathcal{M}(p) : WfT(l) \to WfT(r)$. The* CHARM *implementing $\mathcal{R}$ is defined as $\mathcal{M}(\mathcal{R}) = \{\mathcal{M}(p_i) \mid p_i \in \mathcal{R}\}$.*■

In order to correctly relate the operational behaviours of a graph rewriting system $\mathcal{R}$ and of its associated *CHARM* $\mathcal{M}(\mathcal{R})$, we have to take care of the translation of the starting graph of a derivation into a term. In fact, if G is such a graph, it would not be sound to take as starting state of $\mathcal{M}(\mathcal{R})$ the concrete term $Tm(G)$. Indeed, we must observe that the graph derivations informally introduced above are defined up to isomorphism, i.e., if $G \Rightarrow_p H$, then $G' \Rightarrow_p H'$ for each $G' \cong G$ and $H' \cong H$. This is due to the fact that the pushout objects of Figure 3 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 are faithfully represented by closed terms of the algebra of states: using Definition 9, the class of all graphs isomorphic to G is represented as $WfT(0_G)$, where 0_G is the unique (mono)morphism from the empty graph to G.

The next theorem states that the translation of a graph rewriting system into a *CHARM* is sound and complete. This result is not trivial, and is based on the fact that every transition of the algebra $\mathcal{T}(\mathcal{M}(\mathcal{R}))$ (see Definition 7) represents a pair of graph monomorphisms with common source, which are the bottom line of a double pushout construction like the one depicted in Figure 3. We refer to [Corradini and Montanari 1991] for the formal proofs.

Theorem 11 *Let $\mathcal{R}$ be a graph rewriting system and $\mathcal{M}(\mathcal{R})$ be the associated* CHARM. Soundness: *If G is a graph and $\rho : WfT(0_G) \Rightarrow Q$ is a term of the algebra of computations of $\mathcal{M}(\mathcal{R})$, i.e., of $\mathcal{C}(\mathcal{M}(\mathcal{R}))$ (see Definition 8), then there is a derivation $G \Rightarrow^*_{\mathcal{R}} H$ such that $WfT(0_H) \approx Q$.* Completeness: *If $G \Rightarrow^*_{\mathcal{R}} H$, then there is a computation ρ in the algebra of computations of $\mathcal{M}(\mathcal{R})$, such that $\rho : WfT(0_G) \Rightarrow WfT(0_H)$.*■

5 Modelling concurrent constraint programming

The concurrent constraint (*cc*) programming paradigm [Saraswat 1989] is a very elegant framework which captures and generalizes most of the concepts of logic programming [Lloyd 1987], concurrent logic programming [Shapiro 1989], and constraint logic programming [Jaffar and Lassez 1987]. 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. Agents are defined by clauses as the parallel composition ("$\|$"), 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. The constraint system is very simply modelled by a *partial information system* [Saraswat et al. 1991], i.e. a pair $< D, \vdash >$, where D is the set of the primitive constraints and $\vdash \subseteq \wp(D) \times D$ is the *entailment relation* which states which tokens are entailed by which sets of other tokens, and which must be reflexive and transitive. Then, a constraint is a set of primitive constraints, closed under entailment.

In this section we will informally show how any *cc* program can be modelled by a *CHARM*. The idea is to consider each state 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, both agents and primitive constraints are going to be modelled as process instances, while the shared variables are the variables of the abstract machine.

Basic computation steps are an ask operation, a tell operation, the decomposition of an agent into other agents, but also the generation of new constraints by the entailment relation. In the following, each agent or constraint always comes together with the variables it involves, even though we sometimes will not say it explicitly.

In a state Q, the agent $A = tell(c) \to A1$ adds constraint c to Q and then transforms itself into agent $A1$. This can be faithfully modelled by a rewrite rule R from $S = (G, L)$ to $S' = (G, L')$ where L contains agent A, L' contains agent $A1$ and constraint c, and G contains the variables involved in A (since these are the only items connecting A to the rest of the state Q). This rule may be seen in Figure 4. Note that the fact that c is present only in the local part L' of S' does not mean that c is visible only locally. In fact, the mechanism of rule application allows to treat a local item as a global one (see Figure 2).

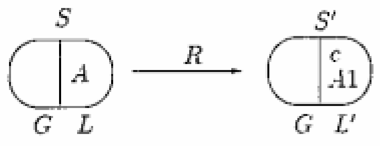

Figure 4: The *CHARM* rewrite rule for the agent $A = tell(c) \rightarrow A1$.

In a state Q, the agent $A = ask(c) \rightarrow A1$ transforms itself into $A1$ if c is in Q and suspends otherwise. The corresponding rewrite rule is R from $S = (G, L)$ to $S' = (G, L')$, where L contains agent A, L' contains agent $A1$, and G contains c. In fact, constraints, once generated, are never cancelled, since the accumulation of constraints is monotonic. Since the rewrite rule cannot be applied if there is no occurrence of the lhs in Q, the ask suspension is given for free. This rule may be seen in Figure 5.

Parallel and nondeterministic composition, as well as existential quantification of agents, are straightforwardly modelled by corresponding rewrite rules. Note that, in an "atomic" interpretation, *tell* and *ask* operations fail if c is inconsistent with the constraints in Q. Our rewrite rules model instead the "eventual" interpretation [Saraswat 1989], where inconsistency is discovered sooner or later, but possibly not immediately. Thus immediate failure is not directly modelled. However, since the difference between the two interpretations basically depends on the way the nondeterministic choice is implemented, the specification of suitable algebraic theories, as suggested in Section 2, could be of help for the implementation of the atomic interpretation of the cc framework.

Each pair $\langle C, t\rangle \in \vdash$ may be modelled by a state change as well. In fact, in a state Q, $\langle C, t\rangle$ can be interpreted as a tell of t whenever C is in Q, and can

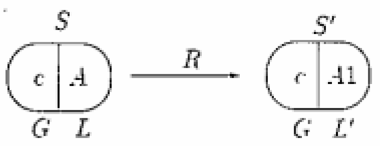

Figure 5: The *CHARM* rewrite rule for the agent $A = ask(c) \rightarrow A1$.

thus be represented by a rewrite rule R from $S = (G, L)$ to $S' = (G, L')$, where L is empty, G contains C, and L' contains t. Note that L is empty, since nothing has to be cancelled, and all items involved are either tested for presence and thus preserved (C) or generated (t). This rule may be seen in Figure 6.

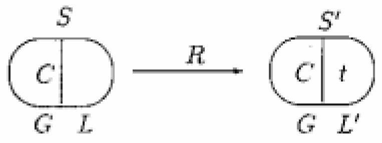

Figure 6: The *CHARM* rewrite rule for the pair $\langle C, t\rangle$ of the entailment relation $\vdash$.

In summary, (the eventual interpretation of) a cc program, together with the underlying constraint system, is modelled in a sound and complete way by a *CHARM* with as many rewrite rules as agents (and subagents) 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). 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 exactly corresponds to the idea that constraints are never cancelled, and thus, once generated locally (by one of the subsystems), are global forever. This description of cc programming within the *CHARM* framework follows a similar one, given in [Montanari and Rossi 1991], where the classical "double-pushout" approach to graph rewriting was used to model cc programs and to provide them with a truly concurrent semantics. Thus, the results of this section are not surprising, given the results in [Montanari and Rossi 1991] and those of the previous section, which show how to model graph grammars through a *CHARM*.

6 Future Work

As pointed out in Section 2, one of the subjects which seem most interesting to investigate is the possibility to provide the *CHARM* with a true-concurrency semantics. Another one is instead the implementation of process description languages onto the *CHARM*. As briefly discussed in sections 2 and 3, both these issues seem to be fruitfully addressable within the algebraic framework we have depicted in this paper.

In [Laneve and Montanari 1991] 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 [Milner et al. 1989], 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, suggests that also higher order aspects of functional languages may be expressed within the *CHARM*.

References

[Berry and Boudol 1990] G. Berry and G. Boudol. The Chemical Abstract Machine. In *Proc. POPL90*. ACM, 1990.

[Corradini 1990] A. Corradini. *An Algebraic Semantics for Transition Systems and Logic Programming.* Ph.D. Thesis TD-8/90. Dipartimento di Informatica, Università di Pisa. Italy. March 1990.

[Corradini et al. 1990] A. Corradini, G. Ferrari, and U. Montanari. Transition Systems with Algebraic Structure as Models of Computations. In *Semantics of Systems of Concurrent Processes*, Guessarian I. ed, Springer-Verlag, LNCS 468. 1990.

[Corradini and Montanari 1991] A. Corradini and U. Montanari. An Algebra of Graphs and Graph Rewriting. In *Proc. 4th Conference on Category Theory and Computer Science.* Springer-Verlag, LNCS, 1991.

[De Boer and Palamidessi 1991] F.S. De Boer and C. Palamidessi. A Fully Abstract Model for Concurrent Constraint Programming. In *Proc. CAAP*, 1991.

[Ehrig 1979] H. Ehrig. Introduction to the Algebraic Theory of Graph Grammars. In *Proc. International Workshop on Graph Grammars*, Springer-Verlag, LNCS 73, 1979.

[Ferrari 1990] G. Ferrari. *Unifying Models of Concurrency.* Ph.D. Thesis, Computer Science Department, University of Pisa. Italy, 1990.

[Gorrieri et al. 1990] R. Gorrieri, S. Marchetti, and U. Montanari. A^2CCS: Atomic Actions for CCS. In *TCS 72*. vol. 2-3, 1990.

[Gorrieri and Montanari 1990] R. Gorrieri and U. Montanari. A Simple Calculus Of Nets. In *Proc. CONCUR90*, Springer-Verlag, LNCS 458, 1990.

[Hoare 1985] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[Jaffar and Lassez 1987] J. Jaffar and J.L. Lassez. Constraint Logic Programming. In *Proc. POPL*. ACM, 1987.

[Laneve and Montanari 1991] C. Laneve and U. Montanari. Mobility in the cc paradigm. Submitted for publication. 1991.

[Lloyd 1987] J.W. Lloyd. *Foundations of Logic Programming.* Springer Verlag, 1987.

[Meseguer 1990] J. Meseguer. Rewriting as a Unified Model of Concurrency. In *Proc. CONCUR90*, Springer-Verlag, LNCS 458, 1990.

[Meseguer and Montanari 1990] J. Meseguer and U. Montanari. Petri Nets are Monoids. *Information and Computation*, vol.88, n.2, 1990.

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

[Milner et al. 1989] R. Milner, J.G. Parrow, and D.J. Walker. *A calculus of mobile processes.* LFCS Reports ECS-LFCS-89-85/86, University of Edinburgh, 1989.

[Montanari and Rossi 1991] U. Montanari and F. Rossi. True Concurrency in Concurrent Constraint Programming. In *Proc. ILPS91*, MIT Press, 1991.

[Reisig 1985] W. Reisig. *Petri Nets: An Introduction.* EATCS Monographs on Theoretical Computer Science, Springer Verlag, 1985.

[Shapiro 1989] E. Shapiro. The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys*, vol.21, n.3, 1989.

[Saraswat 1989] V.A. Saraswat. *Concurrent Constraint Programming Languages.* Ph.D. Thesis, Carnegie-Mellon University, 1989. Also 1989 ACM Dissertation Award, MIT Press.

[Saraswat and Rinard 1990] V.A. Saraswat and M. Rinard. Concurrent Constraint Programming. In *Proc. POPL*, ACM, 1990.

[Saraswat et al. 1991] V.A. Saraswat, M. Rinard, and P. Panangaden. Semantic Foundations of Concurrent Constraint Programming. In *Proc. POPL*, ACM, 1991.