A compositional Semantics for CHR

Maurizio Gabbrielli[1] and Maria Chiara Meo[2]

[1] Dipartimento di Scienze dell'Informazione, Università di Bologna,
Mura A.Zamboni 7, 40127 Bologna, Italy
gabbri@cs.unibo.it

[2] Dipartimento di Scienze, Università di Chieti
Viale Pindaro 42, 65127 Pescara, Italy
cmeo@unich.it

Abstract.
Constraint Handling Rules (CHR) is a committed-choice declarative language which has been designed for writing constraint solvers. A CHR program consists of multi-headed guarded rules which allow to rewrite constraints into simpler ones until a solved form is reached.
CHR has received a considerable attention, both from the practical and from the theoretical side. Nevertheless, due the use of multi-headed clauses, there are several aspects of the CHR semantics which are not been clarified yet. In particular, no compositional semantics for CHR has been defined so far.
In this paper we introduce a fix-point semantics which characterizes the input/output behavior of a CHR program and which is and-compositional, that is, which allows to retrieve the semantics of a conjunctive query from the semantics of its components. Such a semantics can be used as a basis to define incremental and modular analysis and verification tools.

1 Introduction

Constraint Handling Rules (CHR) [9, 10] is a committed-choice declarative language which has been specifically designed for writing constraint solvers. The first constraint logic languages used mainly built-in constraint solvers designed by following a "black box" approach. This made hard to modify, debug, and analyze a specific solver. Moreover, it was very difficult to adapt an existing solver to the needs of some specific applications, and this was soon recognized as a serious limitation since often practical applications involve application specific constraints.

By using CHR one can easily introduce specific user-defined constraints and the related solver into an host language. In fact, a CHR program consists of (a set of) multi-headed guarded simplification and propagation rules which are specifically designed to implement the two most important operations involved in the constraint solving process: Simplification rules allow to replace constraints by simpler ones, while preserving their meaning. Propagation rules are used to add new redundant constraints which do not modify the meaning of the given constraint and which can be useful for further reductions. It is worth noting that the presence of multiple heads in CHR is an essential feature which is needed in order to define reasonably expressive constraint solvers (see the discussion in [10]). However, such a feature, which differentiates this proposal from many existing committed choice logic languages, complicates considerably the semantics of CHR, in particular it makes very difficult to obtain a compositional semantics, as we argue below. This is unfortunate, as compositionality is an highly desirable property for a semantics. In fact, a compositional semantics provides the basis to define incremental and modular tools for software analysis and verification, and these features are essential in order to deal with partially defined components. Moreover, in some cases, modularity allows to reduce the complexity of verification of large systems by considering separately smaller components.

In this paper we introduce a fix-point semantics for CHR which characterizes the input/output behavior of a program and which is and-compositional, that is, which allows to retrieve the semantics of a conjunctive query from the semantics of its components.

In general, due to the presence of synchronization mechanisms, the input/ouput semantics is not compositional for committed choice logic languages and for most concurrent languages in general. Indeed, the need of more complicate semantic structures based on traces was recognized very early as a necessary condition to obtain a compositional model, first for dataflow languages [11] and then in the case of many other paradigms, including imperative concurrent languages [7] and concurrent constraint and logic languages [5].

When considering CHR this basic problem is further complicated: due to the presence of multiple heads the traces consisting of sequences of input/ouput pairs, analogous to those used in the above mentioned works, are not sufficient to obtain a compositional semantics. Intuitively the problem can be stated as follows. A CHR rule $r\ :\ \ A, B \Leftrightarrow c \mid C$ cannot be used to rewrite a goal A, no matter how the variables are constrained (that is, for any input constraint), because the goal consists of a single atom A while the head of the rule contains two atoms A, B. Therefore, if we considered a semantics based on input/ouput traces, we would obtain the empty denotation for the goal A in the program consisting of the rule r plus some rules defining C. Analogously for the goal B. On the other hand, the rule r can be used to rewrite the goal A, B. Therefore, provided that the semantics of C is not empty, the semantics of A, B is not empty and cannot be derived from the semantics of A and B, that is, such a semantics is not compositional. It is worth noting that even restricting to a more simple notion of observable, such as the results of terminating computations, does not simplify this problem. In fact, differently from the case of ccp languages, also the semantic based on these observables (usually called resting points) is not compositional for CHR.

Our solution to obtain a compositional model is to use an augmented semantics based on traces which includes at each steps two “assumptions” on the external environment and two “outputs” of the current process: Similarly to the case of the models for ccp, the first assumption is made on the constraints appearing in the guards of the rules, in order to ensure that these are satisfied and the computation can proceed. The second assumption is specific to our approach and contains atoms which can appear in the heads of rules. This allows us to rewrite a goal G by using a rule whose head H properly contains G: While this is not possible with the standard CHR semantics, we allow that by assuming that the external environment provides the “difference” H minus G and by memorizing such an assumption. The first output element is the constraint produced by the process, as usual. We also memorize at each step also a second output element, consisting of those atoms which are not rewritten in the current derivation and which could be used to satisfy some assumptions (of the second type) when composing sequences representing different computations. Thus our model is based on sequences of quadruples, rather than of simple input/output pairs.

Our compositional semantics is obtained by a fixpoint construction which uses an enhanced transitions system implementing the rules for assumptions described above. We prove the correctness of the semantics w.r.t. a notion of observables which characterizes the input/ouput behavior of terminating computations where the original goal has been completely reduced to built-in constraints. We will discuss later the extensions needed in order to characterize different notions of results, such as the "qualified answers" used in [10].

The remaining of this paper is organized as follows. Next section introduces some preliminaries about CHR and its operational semantics. Section 3 contains the definition of the compositional semantics, while section 4 presents the compositionality and correctness results. Section 5 concludes by discussing directions for future work.

2 Preliminaries

In this section we first introduce some preliminary notions and then define the CHR syntax and operational semantics. Even though we try to provide a self-contained exposition, some familiarity with constraint logic languages and first order logic could be useful.

We first need to distinguish the constraints handled by an existing solver, called built-in (or predefined) constraints, from those defined by the CHR program, called CHR (or user defined) constraints. An atomic constraint is a first-order predicate (atomic formula). By assuming to use two disjoint sorts of predicate symbols we then distinguish built-in atomic constraints from CHR atomic constraints. A built-in constraint c is defined by

$$c ::= a|c \wedge c|\exists_x a$$

where a is an atomic built-in constraint[1]. For built-in constraints we assume given a theory CT which defines their meaning.

On the other hand, according to the usual CHR syntax, we assume that a CHR constraint is a conjunction of atomic CHR constraints. We use c, d to denote built-in constraints, g, h, k to denote CHR constraints and a, b to denote both built-in and CHR constraints (we will call these generically constraints). The capital versions of these notations will be used to denote multisets of constraints. Furthermore we denote by $\mathcal{U}$ the set of user defined constraints and by $\mathcal{B}$ the set of built-in constraints.

We will often use "," rather than $\wedge$ to denote conjunction and we will often consider a conjunction of atomic constraints as a multiset of atomic constraints. In particular, we will use this notation based on multisets in the syntax of CHR. The notation $\exists_{-V}\phi$ where V is a set of variables denotes the existential closure of a formula ϕ with the

[1] We could consider more generally first order formulas as built-in constraints, as far as the results presented here are concerned.

exception of the variables V which remain unquantified. $Fv(\phi)$ denotes the free variables appearing in ϕ and we denote by $\cdot$ the concatenation of sequences and by ε the empty sequence. Furthermore $\uplus$ denotes the multi-set union, while we consider $\setminus$ as an overloaded operator used both for set and multi-set difference (the meaning depends on the type of the arguments).

We are now ready to define the CHR syntax.

Definition 1 (Syntax). *[10] A CHR simplification rule has the form $H \Leftrightarrow c \mid B$ while a CHR propagation rule has the form $H \Rightarrow c \mid B$ where H is a non-empty multiset of user-defined constraints, c is a built-in constraint and B is a possibly empty multi-set of constraints.*

A CHR program is a set of CHR simplification and propagation rules. A CHR goal is a multiset of (both user defined an built-in) constraints.

We prefer to use multisets rather than sequences (as in the original CHR papers) since multisets appear to correspond more precisely to the nature of CHR rules. We denote by $Goals$ the set of all goals.

We describe now the operational semantics of CHR as provided by [10] by using a transition system $T_s = (Conf_s, \longrightarrow_s)$ ($_s$ here stands for "standard", as opposed to the semantics we will use later). Configurations in $Conf_s$ are triples of the form $\langle G, K, d\rangle$ where G are the constraints that remain to be solved, K are the user-defined constraints that have been accumulated and d are the built-in constraints that have been simplified[2].

An *initial configuration* has the form

$$\langle G, \emptyset, \emptyset\rangle$$

and consists of a query G, an empty user-defined constraint and an empty built-in constraint.

A *final configuration* has either the form

$$\langle G, K, \{\mathtt{false}\}\rangle,$$

when it is *failed*, i.e. when it contains an inconsistent built-in constraint store represented by the unsatisfiable constraint `false`, or has the form

$$\langle \emptyset, K, d\rangle$$

when it is successfully terminated, i.e. there are no goals left to solve.

Given a program P, the transition relation $\longrightarrow_s \subseteq Conf \times Conf$ is the least relation satisfying the rules in Table 1 (for the sake of simplicity, we omit indexing the relation with the name of the program). The **Solve** transition allows to update the constraint store by taking into account a built-in constraint contained in the goal. Without loss of generality, we will assume that $Fv(d') \subseteq Fv(c) \cup Fv(d)$. The **Introduce** transition is

[2] In [10] triples of the form $\langle G, K, d\rangle_{\mathcal{V}}$ were used, where the annotation $\mathcal{V}$, which is not changed by the transition rules, is used to distinguish the variables appearing in the initial goal from the variables which are introduced by the rules. We can avoid such an indexing by explicitly referring to the original goal.

Solve	$\dfrac{CT \models c \wedge d \leftrightarrow d' \text{ and c is a built-in constraint}}{\langle (c,G),K,d\rangle \longrightarrow_s \langle G,K,d'\rangle}$
Introduce	$\dfrac{\text{h is a user-defined constraint}}{\langle (h,G),K,d\rangle \longrightarrow_s \langle G,(h,K),d\rangle}$
Simplify	$\dfrac{H \Leftrightarrow c \mid B \in P \quad x = Var(H) \quad CT \models d \rightarrow \exists_x((H = H') \wedge c)}{\langle G, H' \wedge K, d\rangle \longrightarrow_s \langle B \wedge G, K, H = H' \wedge d\rangle}$
Propagate	$\dfrac{H \Rightarrow c \mid B \in P \quad x = Var(H) \quad CT \models d \rightarrow \exists_x((H = H' \wedge c)}{\langle G, H' \wedge K, d\rangle \longrightarrow_s \langle B \wedge G, H' \wedge K, H = H' \wedge d\rangle}$

Table 1. The standard transition system for CHR

used to move a CHR constraint from the goal to the CHR constraint store, where it can be handled by applying CHR rules. The transitions **Simplify** and **Propagate** allow to rewrite CHR constraints (which are in the CHR constraint store) by using rules from the program. As usual, in order to avoid variable names clashes, both these transitions assume that clauses from the program are renamed apart, that is assume that all variables appearing in a program clause are fresh ones. Both the **Simplify** and **Propagate** transitions are applicable when the current store (d) is strong enough to entail the guard of the rule (c), once the parameter passing has been performed (this is expressed by the equation $H = H'$). Note that, due to the existential quantification over the variables x appearing in H, in such a parameter passing the information flow is from the actual parameter (in H') to the formal parameters (H), that is, it is required that the constraints H' which have to be rewritten are an instance of the head H. When applied, both these transitions add the body B of the rule to the current goal and the equation $H = H'$, expressing the parameter passing mechanism, to the built-in constraint store. The difference between **Simplify** and **Propagate** is in the fact that while the former transition removes the constraints H' which have been rewritten from the CHR constraint store, this is not the case for the latter.

Given a goal G, the operational semantics that we consider observes the final stores of computations terminating with an empty goal and an empty user-defined constraint. We call these observables success answers slightly deviating from the terminology of [10] (a goal which has a success answer is called a data-sufficient goal in [10]).

Definition 2 (Success answers). *Let P be a program and let G be a goal. The set $\mathcal{SA}_P(G)$ of success answers for the query G in the program P is defined as follows*

$$\mathcal{SA}_P(G) = \{\langle \exists_{-Fv(G)} d\rangle \mid \langle G, \emptyset, \emptyset\rangle \longrightarrow_s^* \langle \emptyset, \emptyset, d\rangle \not\longrightarrow_P\}.$$

In [10] it is considered also the following different notion of answer, obtained by computations terminating with a user-defined constraint which does not need to be empty.

Definition 3 (Qualified answers). *Let P be a program and let G be a goal. The set $\mathcal{QA}_P(G)$ of qualified answers for the query G in the program P is defined as follows*

$$\mathcal{QA}_P(G) = \{\langle \exists_{-Fv(G)} K \wedge d\rangle \mid \langle G, \emptyset, \emptyset\rangle \longrightarrow_s^* \langle \emptyset, K, d\rangle \not\longrightarrow_P\}.$$

We discuss in Section 5 the extensions needed to characterize also qualified answers. Note that both previous notions of observables characterize an input/output behavior, since the input constraint is implicitly considered in the goal.

In the remaining of this paper we will consider only simplification rules since propagation rules can be mimicked by simplification rules, as far as the results contained in this paper are concerned.

3 A compositional trace semantics

Given a program P, we say that a semantics $\mathcal{S}_P$ is and-compositional if $\mathcal{S}_P(A, B) = \mathcal{C}(\mathcal{S}_P(A), \mathcal{S}_P(B))$ for a suitable composition operator $\mathcal{C}$ which does not depend on the program P. As mentioned in the introduction, due to the presence of multiple heads in CHR, the semantics which associate to a program P the function $\mathcal{SA}_P$ is not and-compositional, since goals which have the same input/ouput behavior can behave differently when composed with other goals. Consider for example the program P consisting of the single rule

$$g, h \Leftrightarrow true|c.$$

(where c is a built-in constraint). According to Definition 3 we have that $\mathcal{SA}_P(g) = \mathcal{SA}_P(k) = \emptyset$, while $\mathcal{SA}_P(g, h) = \{\langle \exists_{-Fv(g,h)} c\rangle\} \neq \emptyset = \mathcal{SA}_P(k, h)$. An analogous example can be made to show that also the semantics $\mathcal{QA}$ is not and-compositional.

The problem exemplified above is different from the classic problem of concurrent languages where the interaction of non-determinism and synchronization makes the input/outuput observables non-compositional. For this reason, considering simply sequences of (input-output) built-in constraints is not sufficient to obtain a compositional semantics for CHR. We have to use some additional information which allow us to describe the behavior of goals in any possible and-composition without, of course, considering explicitly all the possible and-compositions.

The basic idea of our approach is to collect in the semantics also the "missing" parts of heads which are needed in order to proceed with the computation. For example, when considering the program P above, we should be able to state that the goal g produces the constraint c, provided that the external environment (i.e. a conjunctive goal) contains the CHR constraint h. In other words, h is an assumption which is made in the semantics describing the computation of g. When composing (by using a suitable notion of composition) such a semantics with that one of a goal which contains h we can verify that the "assumption" h is satisfied and therefore obtain the correct semantics for g, h. In order to model correctly the interaction of different processes we have to use sequences, analogously to what happens with other concurrent paradigms.

This idea is developed in the following by defining a new transition system which implements this mechanism based on assumptions for dealing with the missing parts of heads. The new transition system allows to generate the sequences appearing in the

Solve'	$$\frac{CT \models c \wedge d \leftrightarrow d'}{\langle c \wedge G^{max=i}, d\rangle \longrightarrow^{\emptyset} \langle G^{max=i}, d'\rangle}$$
Simplify'	$$\frac{\mathsf{cl} = H \leftrightarrow c \mid B \in P \quad x = Fv(H) \quad G^{max=i} \neq \emptyset \quad CT \models d \rightarrow \exists_x((H = (G^{max=i}, K)) \wedge c)}{\langle G^{max=i} \wedge A, d\rangle \longrightarrow^{K} \langle B^{i+1} \wedge A, d \wedge (H = (G, K))\rangle}$$

Table 2. The transition system for the compositional semantics

compositional model by using a standard fix-point construction. As a first step in our construction we modify the notion of configuration used before: Since we do not need to distinguish user defined constraints which appear in the goal from the user defined constraints which have been already considered for reduction, we merge the first and the second components of previous triples. On the other hand, we need the information on the new assumptions, which is added as a label of the transitions.

Thus we define a transition system $T = (Conf, \longrightarrow_P)$ where configurations in $Conf$ are pairs: the first component is a multi-set of indexed atoms (the goal) and the second one is a built-in constraint (the store). Indexes are associated to atoms in order to denote the point in the derivation where they have been introduced. More precisely, atoms in the original goals are labeled by 0, while atoms introduced at the $i - th$ derivation step are labeled by i. Given a program P, the transition relation $\longrightarrow_P \subseteq Conf \times Conf \times \wp(\mathcal{U})$ is the least relation satisfying the rules in Table 2. Note that we consider only **Solve** and **Simplify** rules, as the other rules as previously mentioned are redundant in this context. **Solve'** is the same rule as before, while the **Simplify'** rule is modified to consider assumptions: When reducing a goal G by using a rule having head H, the set of assumption $K = H \setminus G$ (with $H \neq K$) is used to label the transition ($\setminus$ here denotes multiset difference). Indexes allow us to distinguish identical occurrences of atoms which have been introduced in different derivation steps. We will use the notation $G^{max=i}$ to indicate that i is the maximal label occurring in the (non-atomic) goal G and G^i to indicate that all the atoms in G are labeled by i. When no ambiguity arise, to simplify the notation we will use G^i also to denote $G^{max=i}$. In particular, the notation $\langle G^i, c\rangle \rightarrow \langle B^{i+1}, d\rangle$ used in some cases in the transitions always means $\langle G^{max=i}, c\rangle \rightarrow \langle B^{i+1}, d\rangle$, that is, if i is the maximal label occurring in G then all the atoms in B are labeled by $i + 1$. When indexes are not needed we will simply omit them. As before, we assume that program rules to be used in the new simplify rule use fresh variables to avoid names captures.

The semantics domain of our compositional semantics is based on sequences which represent derivations obtained by the transition system in Table 2. More precisely, we first consider "concrete" sequences consisting of tuples of the form $\langle G, c, K, G', d\rangle$: Such a tuple represents exactly a derivation step $\langle G, c\rangle \longrightarrow^{K} \langle G', d\rangle$. The sequences we consider are terminated by tuples of the form $\langle G, c, \emptyset, G, c\rangle$, which represent a terminating step (see the precise definition below). Since a sequence represents a derivation, we assume that the "output" goal G' at step i is equal to the "input" goal G at step $i+1$, that is, we assume that if

$$\ldots\langle G_i, c_i, K_i, G'_i, d_i\rangle\langle G_{i+1}, c_{i+1}, K_{i+1}, G'_{i+1}, d_{i+1}\rangle\ldots$$

appears in a sequence, then $G'_i = G_{i+1}$ holds.

On the other hand, the input store c_{i+1} can be different from the output store d_i produced at previous step, since we need to perform all the possible assumptions on the constraint c_{i+1} produced by the external environment in order to obtain a compositional semantics. However, we assume that if

$$\ldots\langle G_i, c_i, K_i, G'_i, d_i\rangle\langle G_{i+1}, c_{i+1}, K_{i+1}, G'_{i+1}, d_{i+1}\rangle\ldots$$

appears in a sequence then $CT \models c_{i+1} \rightarrow d_i$ holds: This means that the assumption made on the external environment cannot be weaker than the constraint store produced at the previous step. This reflects the monotonic nature of computations, where information can be added to the constraint store and cannot be deleted from it. Finally note that assumptions on CHR constraints (label K) are made only for the atoms which are needed to "complete" the current goal in order to apply a clause. In other words, no assumption can be made in order to apply clauses whose heads do not share any predicate with the current goal.

The set of the above described "concrete" sequences, which represent derivation steps performed by using the new transition system, is denoted by $\mathcal{S}eq$.

From these concrete sequences we extract some more abstract sequences which are the objects of our semantic domain: From each tuple $\langle G, c, K, G', d\rangle$ in a sequence $\delta \in \mathcal{S}eq$ we extract a tuple of the form $\langle c, K, H, d\rangle$ where we consider as before the input and output store (c and d, respectively) and the assumptions (K), while we do not consider anymore the output goal G'. Furthermore, we restrict the input goal G to that part H consisting of all those user-defined constraints which will not be rewritten in the (derivation represented by the) sequence δ. Intuitively H contains those atoms which are available for satisfying assumptions of other goals, when composing two different sequences (representing two derivations of different goals). We also assume that if $\langle c_i, K_i, H_i, d_i\rangle\langle c_{i+1}, K_{i+1}, H_{i+1}, d_{i+1}\rangle$ is in a sequence then $H_i \subseteq H_{i+1}$ holds, since these atoms which will not be rewritten in the derivation can only augment. We then define formally the semantic domain as follows.

Definition 4 (Sequences). *The semantic domain $\mathcal{D}$ containing all the possible sequences is defined as the set*

$$\begin{aligned}\mathcal{D} = \{\langle c_1, K_1, H_1, d_1\rangle \ldots \langle c_n, \emptyset, H_n, c_n\rangle \mid \\ &\textit{for each } j,\ 1 \leq j \leq n \textit{ and for each } i, 1 \leq i \leq n-1,\\ &H_j \textit{ and } K_i \textit{ are multisets of CHR indexed constraints,}\\ &c_j, d_i \textit{ are built-in constraints such that } CT \models d_i \rightarrow c_i,\\ &H_i \subseteq H_{i+1} \textit{ and } CT \models c_{i+1} \rightarrow d_i\}.\end{aligned}$$

In order to define our semantics we need two more notions. First, we define an abstraction operator α which extracts from the concrete sequences in $\mathcal{S}eq$ (representing exactly derivation steps) the sequences used in our semantic domain.

Definition 5. *Let $\delta = \langle G_1, c_1, K_1, G_2, d_1\rangle \dots \langle G_n, c_n, \emptyset, G_n, c_n\rangle$ be a sequence of derivation steps where we assume that atoms are indexed as previously specified. We say that an indexed atom A^j is stable in δ if A^j appears in G_i and in G_{i+1}, for each $1 \leq i \leq n-1$. The abstraction operator $\alpha : \mathcal{S}eq \to \mathcal{D}$ is then defined inductively as*

$$\alpha(\varepsilon) = \varepsilon$$
$$\alpha(\langle G, c, K, G', d\rangle \cdot \delta') = \langle c, K, H, d\rangle \cdot \alpha(\delta')$$

where H is the multiset consisting of all the indexed atoms in G which are stable in $\langle G, c, K, G', d\rangle \cdot \delta'$.

Then we need the notion of "compatibility" of a tuple w.r.t. a sequence. To this aim we first provide some further notation: Given a sequence of derivation steps

$$\delta = \langle G_1, c_1, K_1, G_2, d_1\rangle\langle G_2, c_2, K_2, G_3, d_2\rangle \dots \langle G_n, c_n, \emptyset, G_n, c_n\rangle$$

we denote by $length(\delta)$ and by $instore(\delta)$ the length of the derivation δ and the first input store c_1, respectively. Moreover using t as a shorthand for the tuple $\langle G_1, c_1, K_1, G_2, d_1\rangle$ we define

$V_{loc}(t) = Fv(G_2, d_1) \setminus Fv(G_1, c_1, K_1)$,
$V_{ing}(\delta) = Fv(G_1)$,
$V_{ass}(\delta) = \bigcup_{i=1}^{n-1} Fv(K_i)$,
$V_{stable}(\delta) = Fv(G_n)$,
$V_{constr}(\delta) = \bigcup_{i=1}^{n-1} Fv(d_i) \setminus Fv(c_i)$ and
$V_{loc}(\delta) = \bigcup_{i=1}^{n-1} Fv(G_{i+1}, d_i) \setminus Fv(G_i, c_i, K_i)$.

We then define a compatibility as follows.

Definition 6. *Let $t = \langle G_1, c_1, K_1, G_2, d_1\rangle$ a tuple representing a derivation step for the goal G_1 and let $\delta = \langle G_2, c_2, K_2, G_3, d_2\rangle \dots \langle G_n, c_n, \emptyset, G_n, c_n\rangle$ be a sequence of derivation steps for G_2. We say that t is compatible with δ if the following hold:*

1. $CT \models instore(\delta) \to d_1$,
2. $V_{loc}(\delta) \cap Fv(t) = \emptyset$,
3. *for* $i \in [2, n]$, $V_{loc}(t) \cap Fv(c_i) \subseteq \bigcup_{j=1}^{i-1} Fv(d_j) \cup V_{stable}(\delta)$ *and*
4. $V_{loc}(t) \cap V_{ass}(\delta) = \emptyset$.

Note that if t is compatible with δ then, by using the notation above, $t \cdot \delta$ is a sequence of derivation steps for G_1. We can now define the compositional semantics.

Definition 7 (Compositional semantics). *Let P be a program and let G be a goal. The compositional semantics of G in the program P, $\mathcal{S}_P : Goals \to \wp(\mathcal{D})$, is defined as*

$$\mathcal{S}_P(G) = \alpha(\mathcal{S}'_P(G))$$

where α is the operator introduced in Definition 5 and $\mathcal{S}'_P : Goals \to \wp(\mathcal{S}eq)$ is defined as follows:

$$\begin{array}{rl} \mathcal{S}'_P(G) = \{\langle G, c, K, G', d\rangle \cdot \delta \in \mathcal{S}eq \mid & CT \not\models c \leftrightarrow \texttt{false},\ \langle G, c\rangle \longrightarrow^K \langle G', d\rangle \\ & \textit{and } \delta \in \mathcal{S}_P(G') \textit{ for some } \delta \textit{ such that} \\ & \langle G, c, K, G', d\rangle \textit{ is compatible with } \delta\} \\ \cup & \\ \{\langle G, c, \emptyset, G, c\rangle \in \mathcal{S}eq\}. & \end{array}$$

Formally $\mathcal{S}'_P(G)$ is defined as the least fixed-point of the corresponding operator $\Phi \in (Goals \to \wp(\mathcal{S}eq)) \to Goals \to \wp(\mathcal{S}eq)$ defined by

$$\begin{array}{rl} \Phi(I)(G) = \{\langle G, c, K, G', d\rangle \cdot \delta \in \mathcal{S}eq \mid & CT \not\models c \leftrightarrow \texttt{false},\ \langle G, c\rangle \longrightarrow^K \langle G', d\rangle \\ & \textit{and } \delta \in I(G') \textit{ for some } \delta \textit{ such that} \\ & \langle G, c, K, G', d\rangle \textit{ is compatible with } \delta\} \\ \cup & \\ \{\langle G, c, \emptyset, G, c\rangle \in \mathcal{S}eq\}. & \end{array}$$

In the above definition, the ordering on $Goals \to \wp(\mathcal{S}eq)$ is that of (point-wise extended) set-inclusion. It is straightforward to check that Φ is continuous (on a CPO), thus standard results ensure that the fixpoint can be calculated by $\sqcup_{n\geq 0}\phi^n(\bot)$, where ϕ^0 is the identity map and for $n > 0$, $\phi^n = \phi \circ \phi^{n-1}$ (see for example [8]).

4 Compositionality and correctness

In this section we prove that the semantics defined above is and-compositional and correct w.r.t. the observables $\mathcal{SA}_P$.

In order to prove the compositionality result we first need to define how two sequences describing a computation of A and B, respectively, can be composed in order to obtain a computation of A, B. Such a composition is defined by the (semantic) operator $\|$ which performs an interleaving of the actions described by the two sequences and then eliminates the assumptions which are satisfied in the resulting sequence. For technical reasons, rather than modifying the existing sequences, the elimination of satisfied assumptions is performed on new sequences which are generated by a closure operator η defined as follows.

Definition 8. *Let W be a multiset of indexed atoms and let σ be a sequence in $\mathcal{D}$ of the form*

$$\langle c_1, K_1, H_1, d_1\rangle \dots \langle c_n, K_n, H_n, d_n\rangle.$$

We denote by $\sigma \setminus W$ the sequence

$$\langle c_1, K_1, H_1 \setminus W, d_1\rangle \dots \langle c_n, K_n, H_n \setminus W, d_n\rangle$$

where the multisets difference $H_i \setminus W$ considers indexes.

The operator $\eta : \wp(\mathcal{D}) \to \wp(\mathcal{D})$ is defined as follows. Given $S \in \wp(\mathcal{D})$, $\eta(S)$ is the least set satisfying the following conditions:

1. *$S \subseteq \eta(S)$;*
2. *if $\sigma' \cdot \langle c, K, H, d\rangle \cdot \sigma'' \in S$ then $(\sigma' \cdot \langle c, K \setminus K', H, d\rangle \cdot \sigma'') \setminus W \in \eta(S)$*

where $K' = \{A_1, \ldots, A_n\} \subseteq K$ is a multiset such that there exists a multiset (of indexed atoms) $W = \{B_1^{j_1}, \ldots, B_n^{j_n}\} \subseteq H$ such that $CT \models c \wedge B_i \leftrightarrow c \wedge A_i$, for each $i \in [1, n]$.

A few explanations are in order. The operator η is an upper closure operator[3] which saturates a set of sequences S by adding new sequences where redundant assumptions can be removed: an assumptions a (in K_i) can be removed if a^j appears as a stable atom (in H_i). Once a stable atom is "consumed" for satisfying an assumption it is removed from (the sets of stable atoms of) all the tuples appearing in the sequence, to avoid multiple uses of the same atom. Note that stable atoms are considered without the index in the condition $CT \models c \wedge B_i \leftrightarrow c \wedge A_i$, while they are considered as indexed atoms in the removal operation $H_i \setminus W$. The reason for this slight complication is explained by the following example. Assume that we have the set S consisting of the only sequence $\langle c, \emptyset, \{a^1\}, d\rangle\langle c', \{a\}, \{a^1, a^2\}, d'\rangle$. Such a sequence indicates that at the second step we have an assumption a, while both at the first and at the second step we have produced a stable atom a, which has been indexed by 1 and 2, respectively. In order to satisfy the assumption a we can use either a^1 or a^2. However, depending on what indexed atom we use, we obtain two different simplified sequences in $\eta(S)$, namely $\langle c, \emptyset, \emptyset, d\rangle\langle c', \emptyset, \{a^2\}, d'\rangle$ and $\langle c, \emptyset, \{a^1\}, d\rangle\langle c', \emptyset, \{a^1\}, d'\rangle$, which describes correctly the two different situations.

Before defining the composition operator $\|$ on sequences we need a notation for the sequences in $\mathcal{D}$ analogous to that one introduced for sequences of derivation steps: Let $\sigma = \langle c_1, K_1, H_1, d_1\rangle\langle c_2, K_2, H_2, d_2\rangle \cdots \langle c_n, \emptyset, H_n, d_n\rangle \in \mathcal{D}$ be a sequence for the goal G. We define

$V_{ing}(\sigma) = Fv(G)$ (the free variables of the goal G),
$V_{ass}(\sigma) = \bigcup_{i=1}^{n-1} Fv(K_i)$ (the variables in the assumptions of σ),
$V_{stable}(\sigma) = Fv(H_n) = \bigcup_{i=1}^{n} Fv(H_i)$ (the variables in the stable multisets of σ),
$V_{constr}(\sigma) = \bigcup_{i=1}^{n-1} Fv(d_i) \setminus Fv(c_i)$ (the variables in the output constraints of σ which are not in the corresponding input constraints),
$V_{loc}(\sigma) = (V_{constr}(\sigma) \cup V_{stable}(\sigma)) \setminus (V_{ass}(\sigma) \cup V_{ing}(\sigma))$.

We can now define the composition operator $\|$ on sequences. To simplify the notation we denote by $\|$ both the operator acting on sequences and that one acting on sets of sequences.

Definition 9. *The operator $\|\colon \mathcal{D} \times \mathcal{D} \to \wp(\mathcal{D})$ is defined inductively as follows. Assume that $\sigma_1 = \langle c_1, K_1, H_1, d_1\rangle \cdot \sigma_1'$ and $\sigma_2 = \langle c_2, K_2, H_2, d_2\rangle \cdot \sigma_2'$ are sequences for the goals G_1 and G_2, respectively. If*

$$(V_{loc}(\sigma_1) \cup Fv(G_1)) \cap (V_{loc}(\sigma_2) \cup Fv(G_2)) = Fv(G_1) \cap Fv(G_2)$$

then $\sigma_1 \parallel \sigma_2$ is defined by cases as follows:

[3] $S \subseteq \eta(S)$ holds by definition, and it is easy to see that $\eta(\eta(S)) = \eta(S)$ holds and that $S \subseteq S'$ implies $\eta(S) \subseteq \eta(S')$.

1. *If both* σ_1 *and* σ_2 *have length* 1*, say* $\sigma_1 = \langle c, \emptyset, H_1, c\rangle$ *and* $\sigma_2 = \langle c, \emptyset, H_2, c\rangle$*, then*
$$\sigma_1 \parallel \sigma_2 = \{\langle c, \emptyset, H_1 \uplus H_2, c\rangle\}.$$
2. *If* σ_2 *has length* 1 *and* σ_1 *has length* > 1 *then*
$$\sigma_1 \parallel \sigma_2 = \{\langle c_1, K_1, H_1 \uplus H_2, d_1\rangle \cdot \sigma \mid \sigma \in \sigma_1' \parallel \sigma_2 \text{ and } CT \models instore(\sigma) \to d_1\}.$$
The symmetric case is analogous and therefore omitted.
3. *If both* σ_1 *and* σ_2 *have length* > 1 *then*
$$\begin{aligned}\sigma_1 \parallel \sigma_2 = \{&\langle c_1, K_1, H_1 \uplus H_2, d_1\rangle \cdot \sigma \in \mathcal{D} \mid \sigma \in (\sigma_1' \parallel \sigma_2) \\ &\qquad\qquad\qquad\text{and } CT \models instore(\sigma) \to d_1\} \\ \cup& \\ \{&\langle c_2, K_2, H_1 \uplus H_2, d_2\rangle \cdot \sigma \in \mathcal{D} \mid \sigma \in (\sigma_1 \parallel \sigma_2') \\ &\qquad\qquad\qquad\text{and } CT \models instore(\sigma) \to d_2\}\end{aligned}$$

Finally the composition of sets of sequences $\parallel: \wp(\mathcal{D}) \times \wp(\mathcal{D}) \to \wp(\mathcal{D})$ *is defined by:*

$$\begin{aligned}S_1 \parallel S_2 = \{\sigma \in \mathcal{D} \mid\ & \text{there exist } \sigma_1 \in S_1 \text{ and } \sigma_2 \in S_2 \text{ such that} \\ & \sigma = \langle c_1, K_1, H_1, d_1\rangle \cdots \langle c_n, \emptyset, H_n, c_n\rangle \in \eta(\sigma_1 \parallel \sigma_2), \\ & (V_{loc}(\sigma_1) \cup V_{loc}(\sigma_2)) \cap V_{ass}(\sigma) = \emptyset \text{ and for } i \in [1, n] \\ & (V_{loc}(\sigma_1) \cup V_{loc}(\sigma_2)) \cap Fv(c_i) \subseteq \textstyle\bigcup_{j=1}^{i-1} Fv(d_j) \cup \bigcup_{j=1}^{i} Fv(H_j)\}.\end{aligned}$$

Using this notion of composition of sequences we can show that the semantics $\mathcal{S}_P$ is compositional. Before proving the compositionality theorem we need some technical lemmas.

Lemma 1. *Let* G *be a goal,* $\delta \in \mathcal{S}'_P(G)$ *and let* $\sigma = \alpha(\delta)$*. Then* $V_r(\delta) = V_r(\sigma)$ *holds, where* $r \in \{\, ing,\, ass,\, stable,\, constr,\, loc\,\}$.

Lemma 2. *Let* P *be a program,* G_1 *and* G_2 *be two goals and assume that* $\delta \in \mathcal{S}'_P(G_1, G_2)$*. Then there exists* $\delta_1 \in \mathcal{S}'_P(G_1)$ *and* $\delta_2 \in \mathcal{S}'_P(G_2)$*, such that* $\alpha(\delta) \in \eta(\alpha(\delta_1) \parallel \alpha(\delta_2))$.

Lemma 3. *Let* P *be a program, let* G_1 *and* G_2 *be two goals and assume that* $\delta_1 \in \mathcal{S}'_P(G_1)$ *and* $\delta_2 \in \mathcal{S}'_P(G_2)$ *are two sequences such that the following hold:*

1. $\alpha(\delta_1) \parallel \alpha(\delta_2)$ *is defined,*
2. $\sigma = \langle c_1, K_1, W_1, d_1\rangle \cdots \langle c_n, \emptyset, W_n, c_n\rangle \in \eta(\alpha(\delta_1) \parallel \alpha(\delta_2))$*,*
3. $(V_{loc}(\alpha(\delta_1)) \cup V_{loc}(\alpha(\delta_2))) \cap V_{ass}(\sigma) = \emptyset$*,*
4. *for* $i \in [1, n]$*,* $(V_{loc}(\alpha(\delta_1)) \cup V_{loc}(\alpha(\delta_2))) \cap Fv(c_i) \subseteq \bigcup_{j=1}^{i-1} Fv(d_j) \cup \bigcup_{j=1}^{i} Fv(W_j)$.

Then there exists $\delta \in \mathcal{S}'_P(G_1, G_2)$ *such that* $V_{loc}(\delta) \subseteq V_{loc}(\delta_1) \cup V_{loc}(\delta_2)$ *and* $\sigma = \alpha(\delta)$.

By using the above results we can prove the following theorem.

Theorem 1 (Compositionality). *Let* P *be a program and let* G_1 *and* G_2 *be two goals. Then*
$$\mathcal{S}_P(G_1, G_2) = \mathcal{S}_P(G_1) \parallel \mathcal{S}_P(G_2).$$

4.1 Correctness

In order to show the correctness of the semantics $\mathcal{S}_P$ w.r.t. the (input/output) observables $\mathcal{SA}_P$, we first introduce a different characterization of $\mathcal{SA}_P$ obtained by using the new transition system defined in Table 2.

Definition 10. *Let P be a program and let G be a goal and let $\longrightarrow$ be (the least relation) defined by the rules in Table 2. We define*

$$\mathcal{SA}'_P(G) = \{\exists_{-Fv(G)}c \mid \langle G, \emptyset\rangle \longrightarrow^*_P \langle \emptyset, c\rangle \not\longrightarrow_P\}.$$

The correspondence of $\mathcal{SA}'$ with the original notion $\mathcal{SA}$ is stated by the following proposition, whose proof is immediate.

Proposition 1. *Let P be a program and let G be a goal. Then*

$$\mathcal{SA}_P(G) = \mathcal{SA}'_P(G).$$

The observables $\mathcal{SA}'_P$, and therefore $\mathcal{SA}_P$, describing answers of successful computations can be obtained from $\mathcal{S}$ by considering suitable sequences, namely those sequences which do not perform assumptions neither on CHR constraints nor on built-in constraints. The first condition means that the second components of tuples must be empty, while the second one means that the assumed constraint at step i must be equal to the produced constraint of steps i-1. We call "connected" those sequences which satisfy these requirements:

Definition 11 (Connected sequences). *Assume that*

$$\sigma = \langle c_1, K_1, H_1, d_1\rangle \dots \langle c_n, K_n, H_n, c_n\rangle$$

is a sequence in $\mathcal{D}$. We say that σ is connected if $K_j = \emptyset$ for each j, $1 \leq j \leq n$ and $d_i = c_{i+1}$, for each i, $1 \leq i \leq n-1$.

The proof of the following result derives from the definition of connected sequence and an easy inductive argument. Given a sequence $\sigma = \langle c_1, K_1, H_1, d_1\rangle \dots \langle c_n, K_n, H_n, d_n\rangle$, we denote by $store(\sigma)$ the built-in constraint d_n, by $result(\sigma)$ the constraint $d_n \wedge H_n$ and by $lastg(\sigma)$ the goal H_n.

Proposition 2. *Let P be a program and let G be a goal. Then*

$$\begin{aligned}\mathcal{SA}'_P(G) = \{\exists_{-Fv(G)}c \mid\ & \textit{there exists } \sigma \in \mathcal{S}_P(G) \textit{ such that } instore(\sigma) = \emptyset,\\ & \sigma \textit{ is connected, } lastg(\sigma) = \emptyset \textit{ and } c = store(\sigma)\}.\end{aligned}$$

The following corollary is immediate from Proposition 1.

Corollary 1 (Correctness). *Let P be a program and let G be a goal. Then*

$$\begin{aligned}\mathcal{SA}_P(G) = \{\exists_{-Fv(G)}c \mid\ & \textit{there exists } \sigma \in \mathcal{S}_P(G) \textit{ such that } instore(\sigma) = \emptyset,\\ & \sigma \textit{ is connected, } lastg(\sigma) = \emptyset \textit{ and } c = store(\sigma)\}.\end{aligned}$$

5 Conclusions

In this paper we have introduced a semantics for CHR which is compositional w.r.t. the and-composition of goals and which is correct w.r.t "success answers", a notion of observable which considers the results of successful computations where all the CHR constraints have been rewritten into built-in constraints. We are not aware of other compositional characterizations of CHR answers, so our work can be considered as a first step which can be extended along several different lines.

Firstly, it would be desirable to obtain a compositional characterization also for "qualified answers" obtained by considering computations terminating with a user-defined constraint which does not need to be empty (see Definition 3). This could be done by a slight extension of our model: The problem here is that, given a tuple $\langle G, c, K, G', d\rangle$, in order to reconstruct correctly the qualified answers we need to know whether the configuration $\langle G', d\rangle$ is terminating or not (that is, if $\langle G', d\rangle \not\to$ holds). This could be solved by introducing some termination modes, at the price of a further complication of the traces used in our semantics.

A second possible extension is the investigation of the full abstraction issue. For obvious reasons it would be desirable to introduce in the semantics the minimum amount of information needed to obtain compositionality, while preserving correctness. In other terms, one would like to obtain a results of this kind: $\mathcal{S}_P(G) = \mathcal{S}_P(G')$ if and only if, for any H, $\mathcal{SA}_P(G, H) = \mathcal{SA}_P(G', H)$ (our Corollary 1 only ensures that the "only if" part holds). Such a full abstraction result could be difficult to achieve, however techniques similar to those used in $[5, 2]$ for analogous results in the context of ccp could be considered

It would be interesting also to study further notions of compositionality, for example that one which considers union of program rules rather than conjunctions of goals, analogously to what has been done in [6]. However, due to the presence of synchronization, the simple model based on clauses defined in [6] cannot be used for CHR.

As mentioned in the introduction, the main interest related to a compositional semantics in the possibility to provide a basis to define compositional analysis and verification tools. In our case, it would be interesting to investigate to what extent the compositional proof systems *à la* Hoare defined in $[1, 3]$ for (timed) ccp languages, based on resting points and trace semantics, can be adapted to the case of CHR.

Acknowledgments

We thank Michael Maher for having initially suggested the problem of compositionality for CHR semantics.

References

1. F.S. de Boer, M. Gabbrielli, E. Marchiori and C. Palamidessi. Proving Concurrent Constraint Programs Correct. *Transactions on Programming Languages and Systems (TOPLAS)*, 19(5): 685-725. ACM Press, 1997.
2. F.S. de Boer, M. Gabbrielli, and M.C. Meo. Semantics and expressive power of a timed concurrent constraint language. In G. Smolka. editor, Proc. *Third Int'l Conf. on Principles and*

Practice of Constraint Programming (CP 97), Lecture Notes in Computer Science. Springer-Verlag, 1997.

3. F.S. de Boer, M. Gabbrielli and M.C. Meo. Proving correctness of Timed Concurrent Constraint Programs. *ACM Transactions on Computational Logic*. To appear.
4. F.S. de Boer, J.N. Kok, C. Palamidessi, and J.J.M.M. Rutten. The failure of failures in a paradigm for asynchronous communication. In J.C.M. Baeten and J.F. Groote, editors, *Proceedings of CONCUR'91*, vol. 527 of *LNCS*, pages 111–126. Springer-Verlag, 1991.
5. F.S. de Boer and C. Palamidessi. A Fully Abstract Model for Concurrent Constraint Programming. In S. Abramsky and T.S.E. Maibaum, editors, *Proc. of TAPSOFT/CAAP*, vol. 493 of *LNCS*, pages 296–319. Springer-Verlag, 1991.
6. A. Bossi, M. Gabbrielli, G. Levi, and M. C. Meo. A Compositional Semantics for Logic Programs. *Theoretical Computer Science* 122(1-2): 3–47, 1994.
7. S. Brookes. A fully abstract semantics of a shared variable parallel language. In *Proc. Eighth IEEE Symposium on Logic In Computer Science*. IEEE Computer Society Press, 1993.
8. B.A. Davey and H.A. Priestley. Introduction to Lattices and Order. Cambridge University Press, 1990.
9. T. Früwirth. Introducing simplification rules. TR ECRC-LP-63, ECRC Munich. October 1991.
10. T. Früwirth. Theory and practice of Constraint Handling Rules. *Journal of Logic Programming*, 1994:19, 20:1-679.
11. B. Jonsson. A model and a proof system for asynchronous processes. In *Proc. of the 4th ACM Symp. on Principles of Distributed Computing*, pages 49–58. ACM Press, 1985.
12. V.A. Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, Carnegie-Mellon University, January 1989. Published by The MIT Press, 1991.
13. V.A. Saraswat and M. Rinard. Concurrent constraint programming. In *Proc. of POPL*, pages 232–245. ACM Press, 1990.
14. V.A. Saraswat, M. Rinard, and P. Panangaden. Semantics foundations of Concurrent Constraint Programming. In *Proc. of POPL*. ACM Press, 1991.