Coordination by means of Synchronous and Asynchronous Communication in Concurrent Constraint Programming

Jean-Marie Jacquet\textsuperscript{1,2}

\textit{Institute of Informatics}
\textit{University of Namur}
\textit{Namur, Belgium}

Luboš Brim\textsuperscript{3}

\textit{Department of Computer Science}
\textit{Masaryk University}
\textit{Brno, Czech Republic}

David Gilbert\textsuperscript{4}

\textit{Department of Computing Science}
\textit{University of Glasgow}
\textit{Glasgow, United Kingdom}

Mojmír Křetínský\textsuperscript{5}

\textit{Department of Computer Science}
\textit{Masaryk University}
\textit{Brno, Czech Republic}

\textbf{Abstract}

Concurrent constraint programming has been thought as providing coordination of concurrent processes on the basis of the availability and sharing of information. It classically incorporates a form of asynchronous communication via a shared store. In previous work ([1,2]), we presented a new version of the ask and tell primitives which features asynchronicity and synchronicity, our approach being based on the idea of telling new information just in the case that a concurrently running process is asking for it. In this paper we focus on a semantic study of this new framework, called Scc.
It is first shown to be different in nature from classical concurrent constraint programming and from CCS, a classical reference in traditional concurrency theory. This suggests the interest of new semantics for Scc. To that end, an operational semantics reporting the steps of the computations is presented. A denotational semantics is then proposed. It uses monotonic sequences of labelled pairs of input-output states, possibly containing gaps, and ending – according to the logic programming tradition – with marks reporting success or failure. This denotational semantics is proved to be correct with respect to the operational semantics as well as fully abstract.

1 Introduction

1.1 Concurrency and constraint programming

Concurrent constraint programming has emerged as an important paradigm for concurrent computations (see eg [19,20,21]). It is based on the idea of computing with partial information placed on a shared space, called the store. Accordingly, concurrent processes communicate through this store by telling pieces of information and by asking whether some piece of information is entailed by the current contents of the store. As a legacy of previous proposals for concurrent logic programming languages (Concurrent Prolog, Parlog, GHC, etc.), communication occurs in an asynchronous fashion: tell actions are always allowed to proceed whereas ask operations are blocked when the information on the store is not complete enough to entail the asked constraints.

Following these lines, a natural way of obtaining synchronous communication in concurrent constraint programming is to force the reduction of ask and tell primitives to synchronise. Specifically, our approach considers tell primitives as lazy producers of information and views ask primitives as consumers of this information. From this point of view, a tell operation is reduced when an ask operation requires the told information. Moreover, the reduction of the two primitives is performed simultaneously. However, there is no reason to block ask and tell primitives on information which is already present. Consequently, stress is put on the novelty of the information and hence any tell(c) and ask(c) operations whose constraint argument c is entailed by the current store are reduced without partners.

This framework, called Scc, is presented in [1,2] and its expressiveness has been demonstrated through the coding of a variety of examples. It has been argued that one advantage over related work such as [19,13,11], which

---

1 L. Brim and M. Křetínský thank the Grant Agency of the Czech Republic, grant no.201/00/0400, for supporting their research.
2 Email: jmj@info.fundp.ac.be
3 Email: brim@fi.muni.cz
4 Email: drg@soi.city.ac.uk
5 Email: mojmir@fi.muni.cz
introduce synchronisation by special operators and not by altering the behaviour of tell and ask primitives, is that \texttt{Scc} permits the specification of on \textit{what} information the synchronisation should be made, rather than with \textit{whom}. Synchronisation in \texttt{Scc} is thus data-oriented as opposed to process-oriented.

In order to motivate its interest and to substantiate the need for novel treatments, it is worth stressing the behavioural difference of \texttt{Scc} with, on the one hand, traditional concurrent constraint programming, as exemplified in the \texttt{cc} family of languages ([19]), and, on the other hand, traditional concurrent programming models, as exemplified by CCS ([15]).

It has been argued in [8] that the main difference between CCP and CCS is that complementary actions do not synchronise in CCP. This property is due to the fact that telling a constraint never suspends in \texttt{cc}. In contrast, the action of telling a constraint may suspend until an ask can make use of it. A synchronisation similar to that in CCS is thus produced. However, this synchronisation does not hold in \texttt{Scc} in the case that the told or asked constraints are entailed by the current contents of the store. A novel kind of synchronisation is thus achieved.

Major differences appear between the three frameworks. It is to be expected that these differences call for new treatments as well. In order to formalise our reasoning somewhat, let us turn to the example given in [8]. There CCS and CCP are compared by interpreting the action \texttt{a} as telling the constraint \(x = a\), and the co-action \(\pi\) as asking the constraint \(x = a\). To keep our notations consistent, we shall use “+” for the non-deterministic choice operator and “;” for the sequential composition operator.

\textbf{Example 1.1} [Differentiating CCP and CCS (from [8])] Let \(A_1 = (\pi; \overline{b}) + (\overline{\pi}; r) + (\overline{\pi}; \overline{d})\) and \(A_2 = (\pi; \overline{b}) + (\overline{\pi}; (r + \overline{d}))\). In any compositional semantics for CCS these two processes must be distinguished. Indeed, they behave differently under the context \(A = a; (b + c)\). The process \(A_1\) can deadlock, by choosing the third alternative of the choice, while \(A_2\) cannot. However, in \texttt{cc}, both \(A_1\) and \(A_2\) have the same behaviour. The process \(A_2\) can deadlock by choosing the second alternative, because \(A\) can independently decide to produce \(y = b\) (after \(x = a\)).

\textbf{Example 1.2} [Differentiating CCP and \texttt{Scc}] Using the processes \(A, A_1, A_2\) of the above example, the processes \(A_1\) and \(A_2\) are also distinguished by \(A\) in \texttt{Scc} for the same reason as in CCS.

This example illustrates the difference between \texttt{Scc} and CCP. Stated in other terms, in the CCP paradigm, since the tell operation is asynchronous, the choice guarded by \texttt{tell}(a) is a \textit{local choice} whereas, since the tell operation is synchronous in \texttt{Scc}, this choice is \textit{global} in \texttt{Scc}.

Nevertheless, synchronisation is only forced in \texttt{Scc} in the case that a process tries to tell information which is not already entailed by the store. Otherwise, it can proceed asynchronously. This fact is used subsequently to differentiate CCS and \texttt{Scc}.
Example 1.3 [Differentiating CCS and Scc] Using again the above processes $A, A_1, A_2$, let $B_1 = \overline{b}; A_1$ and $B_2 = \overline{b}; A_2$. In CCS, these two processes can be distinguished by the process $B = b; A$ for the reasons exposed in Example 1.1. However, in Scc, both processes have the same behaviour. The process $B_2$ can now deadlock by choosing the second alternative because $A$ can now independently proceed by the first alternative as $y = b$ is already entailed by the store.

The distinction between Scc and CCS thus appear to be more subtle than the distinction between CCP and CCS. The choice guarded by lazy $tell$ is actually a “mixture” of global and local choice. The choice depends upon actions performed and upon the results of the past behaviour of the system, i.e. upon the constraint contained in the store.

1.2 Results and comparison with related work

The major contribution of this paper consists in a new fully abstract denotational semantics for Scc. For the ease of presentation, we shall focus on a restricted version, called Rscc, where, as opposed to the full version, synchronous communication is performed only between one tell primitive and one ask primitive. It is worth noting that the above examples rely on this restricted form of communication. Moreover, finite goals are only treated here by noting that our results can be extended to recursion using classical techniques such as the ones discussed in [12] and [16].

The peculiarities of our semantic model may be highlighted by contrasting it with fully abstract models proposed for concurrent constraint programming languages and for other more traditional concurrent programming frameworks.

In the context of deterministic concurrent constraint languages, V. Saraswat et al ([21]) have proposed a model based on resting points. Accordingly, the denotation of a goal is the set of stores such that if the goal were initiated in them, it would not be able to produce any new information without receiving more information from its environment. Mathematically, these denotations turn out to be Scott’s closure operators and to enjoy very nice properties, the nicest one being probably that parallel composition amounts to simply intersecting sets. This semantics has been proved to be fully abstract for observables consisting of final stores of computations regardless they are successful or deadlocked. It has been refined in [?] to special sets of closure operators, called bounded trace operators, in order to cope with non-deterministic processes. Essentially, each trace operator records a resting point together with the interactions that the considered goal engages with its environment. This idea has been developed further by Nyström and Jonsson in [17] to tackle infinite computations and fairness.

Concurrently, F. de Boer and C. Palamidessi have proposed in [8] a compositional model based on so-called reactive sequences composed, on the one hand, of sequences of input and output constraints, corresponding respectively
to constraints being told by the environment of a goal and by the goal itself, and, on the other hand, of termination marks corresponding to success, deadlock or failure. A fully abstract model for observables consisting of final stores associated with a termination mark of the form just described has been derived therefrom by introducing saturation conditions, essentially abstracting from the order and the granularity in which constraints are told. The models based on closure operators and of reactive sequences have been proved isomorphic in [10].

The difference between these semantics and our denotational semantics takes its essence in the observation that what is crucial in traditional concurrent constraint languages is the contents of the store rather than with which partner or on which information a goal can synchronize. In Scc both the contents of the store and the information on which a goal can synchronize are important. To reflect this property, our denotational semantics is built upon sequences of steps which start on any possible store, which possibly contain gaps, and which end by pairs composed of a final store and of a termination mark.

The possibility to start on any store and to include gaps allow to treat simultaneously the asynchronous and the synchronous forms of communication embodied in Scc. If enough information is contained in the considered store then the tell and ask primitives proceed by silent moves. Otherwise, two kinds of denotations are provided. One assumes that the tell and ask primitives of the considered goal synchronize with the environment and the computation is continued on this basis. The other denotation reports an ending pair \((\sigma, \delta^- (X))\) which asks to be combined with a similar one.

Compared with [8], the gaps in the denotational sequences play a role similar to the input constraints. However, more abstraction is gained since (maximal) sequences of input constraints are naturally grouped in one gap. Moreover, failure sets, are needed in our context and not in the concurrent constraint framework because of the absence of synchronous communication in this setting. In a sense, the store \(\sigma\) of an ending mark \((\sigma, \delta^- (X))\) represents a resting point: the goal under consideration cannot be reduced further without interaction with the environment. However, a failure set is required to express the synchronization of the considered goal with its environment.

Failure semantics ([6]) is a well-known fully abstract semantics for CCS-like languages based on synchronous semantics. The presence of failure sets in our semantics should thus not be too surprising. Our contribution at this level is however that a tell\((c)\) action does not synchronize with just the corresponding ask\((c)\) action but with any weaker ask operation as well. To model this fact we have introduced the notion of closeness on failure sets.

We shall also be employing the so-called testing technique, which has first been introduced in [12] in an imperative setting. As opposed to the constraint paradigm, this setting does not enjoy the monotonic property of the computation. Whereas monotonicity in general brings propositions which do not hold
in a non-monotonic framework, it makes things more difficult with respect to testing. Indeed, the trace of any auxiliary operation can be removed by using destructing assignments. This is not possible in Scc since no primitive operation is provided to remove auxiliary constraints having been told. Instead, we have introduced the notion of consistency and have shown that, for so-called independent constraints, the addition of auxiliary constraints does not modify substantially the computations.

Full abstraction for a shared variable parallel imperative language is also studied in [5]. However, the observables are composed of the final states of the computation coupled to termination marks. They thus substantially differ from what is returned by our operational semantics and hence, so are the denotational semantics although traces are also used in [5]. Furthermore, the paradigm studied being quite different, there is no counterpart for issues like consistency and closed failure sets.

Recently, de Boer et al [7] proposed a concurrent programming language which combines asynchronous concurrent constraint programming operations with the synchronous imperative communication operations to describe multi-agent systems and gave a fully abstract semantics for their language. The main difference with our approach is that in Scc both the synchronous and the asynchronous forms of communication proceed via a global store. Furthermore, as we have demonstrated in [3], it is possible to naturally model this language in Scc.

1.3 Structure of the paper

The rest of the paper is organised as follows. Section 2 describes the Rscc language. Section 3 presents the operational semantics. Section 4 studies the denotational semantics, which is proved correct in section 5 and fully abstract in section 6.

2 The Language Rscc

As in [21], the constraint system underlying Rscc consists of any system of partial information that supports the entailment relation. Precisely, we assume given a set $C$ of assertions. Each assertion, called an elementary constraint, provides partial information about the state of affairs. By adopting the notation $\mathcal{P}_f(C)$ to denote the set of finite subsets of $S$, constraint systems can be formally defined as follows.

**Definition 2.1** A simple constraint system is a structure $(C, \vdash)$ where $C$ is a non-empty denumerable set of tokens, also called primitive constraints or more simply constraints, and $\vdash \subseteq \mathcal{P}_f(C) \times C$ is a relation, called the entailment relation, satisfying the following properties: for any $\rho, \sigma \in \mathcal{P}_f(C)$ and any $c \in C$,

i) if $c \in \sigma$ then $\sigma \vdash c$
ii) if \( \sigma \models d \) for any \( d \in \rho \) and if \( \rho \models c \) then \( \sigma \models c \).

The relation \( \models \) is extended to \( \mathcal{P}_f(C) \times \mathcal{P}_f(C) \) by stating \( \rho \models \sigma \) iff \( \rho \models c \) for any \( c \in \sigma \).

**Definition 2.2** Stores are defined as finite sets of constraints. Their sets is subsequently denoted as \( S_{store} \).

A property will be required in order to build a fully abstract semantics for the \( \text{R} \text{S} \text{C} \text{C} \) language. It is not guided by the synchronous nature of the language but rather is motivated by the need to find, at some point of a computation, constraints which are not entailed by the current store and which do not affect the rest of the computation. Technically speaking, this basically reduces to requiring that the considered constraint system is not degenerated in the sense that a finite subset of \( C \) entails all the constraints of \( C \). This leads to the following assumption. Note that it holds for most practical constraint systems.

**Assumption 1 (Separating assumption)** For any finite set of constraints \( S \), there is a constraint \( c \) such that
1) \( S \not\models c \)
2) for any \( s \in S \) and any store \( \rho \), if \( \rho \cup \{c\} \models s \) then \( \rho \models s \).

The language description is parametric with respect to \((C, \models)\), and so are the semantic constructions presented.

**Definition 2.3** Goals \( G \in S_{goal} \) are defined by the following grammar
\[
G ::= \Delta \mid NG \\
NG ::= \text{ask}(c) \mid \text{tell}(c) \mid NG; NG \mid NG + NG \mid NG \parallel NG
\]
where \( c \) denotes an elementary constraint, \( \Delta \) denotes the empty goal, and the operators \( ;, +, \parallel \) respectively denote sequential, choice, and parallel composition.

3 The operational semantics \( \mathcal{O} \)

The operational semantics of \( \text{R} \text{S} \text{C} \text{C} \) is defined in Plotkin’s style [18] by means of a labelled transition system. The configurations to be considered are composed here of the goal to be solved coupled to the current store, respectively representing here the statement under consideration and the computed values. The labels are used to indicate communication with the environment. No communication or a silent communication is indicated by \( \tau \). Following the notations of section 1, a tell action gives rise to a \( c \) label where \( c \) is the told constraint whereas a get action gives rise to a \( \bar{c} \) label where \( c \) is the asked constrained. Moreover, we define \( S_{const} \) as the set of constraints \( c \) and of their complements \( \bar{c} \) and take the convention that \( \bar{\bar{c}} \) actually denotes \( c \). We also extend the overline notation on sets by defining \( \bar{S} = \{\bar{s} : s \in S\} \), for any set \( S \subseteq S_{const} \).

**Definition 3.1** Let \( S_{label} \) be the set composed of a fresh symbol \( \tau \) and the
Asynchronous tell

\[
(T_a) \quad \langle \text{tell}(c), \sigma \rangle \xrightarrow{\tau} \langle \triangle, \sigma \rangle
\]

if \( \{ \sigma \vdash c \} \)

Synchronous tell

\[
(T_s) \quad \langle \text{tell}(c), \sigma \rangle \xrightarrow{c} \langle \triangle, \sigma \cup \{ c \} \rangle
\]

if \( \{ \sigma \not\vdash c \} \)

Asynchronous ask

\[
(A_a) \quad \langle \text{ask}(c), \sigma \rangle \xrightarrow{\tau} \langle \triangle, \sigma \rangle
\]

if \( \{ \sigma \vdash c \} \)

Synchronous ask

\[
(A_s) \quad \langle \text{ask}(c), \sigma \rangle \xrightarrow{d} \langle \triangle, \sigma \cup \{ d \} \rangle
\]

if \( \{ \sigma \not\vdash c, \sigma \cup \{ d \} \vdash c \} \)

Fig. 1. \textsc{rscce} transition system: atomic operations

elements of \textsc{Secs}. Define the transition relation \( \rightarrow \) as the smallest relation of \((\text{Sgoal} \times \text{Sstore}) \times \text{Slabel} \times (\text{Sgoal} \times \text{Sstore})\) satisfying the rules of Figures 1 and 2. To simplify the presentation, the following relaxation of the syntax is employed in rules (I), (P), and (S): there, it is understood that when \( G_0, G_1, G_2 \) are empty goals, target goals of the form \( \triangle \parallel G^* \), \( G^* \parallel \triangle \), \( \triangle; G^* \) with \( G^* \not= \triangle \) actually denote \( G^* \) and similarly that \( \triangle \parallel \triangle \) actually represents \( \triangle \). Simplifications are of course operated accordingly.

\textbf{Notation 3.2} To ease the reading, we use the notation \( <G, \sigma> \xrightarrow{l} \) to indicate the fact that there are no \( G' \) and \( \sigma' \) such that \( <G, \sigma> \xrightarrow{l} <G', \sigma'> \).

Following one of the traditions in concurrency theory, the operational semantics specifies the successive non-hypothetical steps of the computation and ends with success if the computation reaches the empty goal and with failure otherwise. Success and failure are respectively indicated by the \( \delta^+ \) and \( \delta^- \) marks.

\textbf{Definition 3.3} Define the operational semantics \( O_h : \text{Sgoal} \rightarrow \mathcal{P}(\text{Sstore}^{<\omega} \times \{ \delta^+, \delta^- \}) \) as the following function: for any goal \( G \)
Choice

\[
\frac{<G, \sigma> \xrightarrow{l} <G', \sigma'>}{<G + G', \sigma> \xrightarrow{l} <G'', \sigma''>}
\]
\[
\frac{<G' + G, \sigma> \xrightarrow{l} <G'', \sigma''>}{(C)}
\]

Sequential composition

\[
\frac{<G, \sigma> \xrightarrow{l} <G'', \sigma''>}{<G ; G', \sigma> \xrightarrow{l} <G''; G', \sigma''>}
\]
\[
(\text{S})
\]

Independent parallelism

\[
\frac{<G, \sigma> \xrightarrow{l} <G'', \sigma''>}{<G \parallel G', \sigma> \xrightarrow{l} <G'' \parallel G', \sigma''>}
\]
\[
\frac{<G' \parallel G, \sigma> \xrightarrow{l} <G'' \parallel G'', \sigma''>}{(I)}
\]

Synchronous communication

\[
\frac{<G_1, \sigma> \xrightarrow{\epsilon} <G'_1, \sigma'>}{<G_2, \sigma> \xrightarrow{\tau} <G'_2, \sigma'>}
\]
\[
\frac{<G_1 \parallel G_2, \sigma> \xrightarrow{\tau} <G'_1 \parallel G'_2, \sigma'>}{(P)}
\]
\[
\frac{<G_2 \parallel G_1, \sigma> \xrightarrow{\tau} <G'_2 \parallel G'_1, \sigma'>}{
\}
\]

Fig. 2. Rscc transition system: composed goals

\[
O_h(G) = \{\sigma_1 \cdots \sigma_n, \delta^+ : <G_0, \sigma_0> \xrightarrow{\tau} \cdots \xrightarrow{\tau} <G_n, \sigma_n>, \quad G_0 = G, \sigma_0 = \epsilon, G_n = \triangle, n \geq 0\} \cup \{\sigma_1 \cdots \sigma_n, \delta^- : <G_0, \sigma_0> \xrightarrow{\tau} \cdots \xrightarrow{\tau} <G_n, \sigma_n> \xrightarrow{\tau}, \quad G_0 = G, \sigma_0 = \epsilon, G_n \neq \triangle, n \geq 0\}
\]

9
It is here worth noting that the operational semantics $O_h$ is not compositional. For instance, taking two independent constraints $c$ and $d$, $O_h(tell(c)) = O_h(tell(d)) = \{\delta^-\}$ whereas $O_h(tell(c) || ask(c)) = \{\{c, \delta^+\} \text{ and } O_h(tell(d) || ask(c)) = \{\delta^-\}$. The purpose of the next section is precisely to define a compositional semantics for $Rscc$ which is correct with respect to the operational semantics but which also contains a “minimal” amount of information to be compositional. In other words, we shall try to define a fully abstract semantics.

4 A denotational semantics

There are two main reasons why the operational semantics $O$ is not compositional. First, after having made a computation step, a store non necessarily empty is produced. A compositional semantics should therefore be defined in order to account for initial stores of any contents. Second, as deduced from the transition system, the computation of the goal $A || B$ amounts to interleave execution steps of $A$ and $B$. A compositional semantics should thus allow for the transition steps made by the environment.

Following [12], we shall model transition steps in the form of pairs of input and output stores and take as semantic domain, sets of sequences of such pairs. These sequences possibly contain gaps, accounting for the action of the environment. Moreover, they will start in any store, allowing previous steps resulting in a possibly non-empty store. However, in contrast to [12], they are required to be monotonic in the sense that information cannot be retracted. Finally, following [6], the deadlock mark $\delta^-$ is generalized to failures – but composed here of closed sets – for compositionality and full abstraction purposes.

4.1 Preliminaries

Before going further, a few notations and concepts are in order. Telling a constraint not entailed by the current store can be viewed as telling any weaker constraint not entailed by the current store since, on the one hand, the update of the store includes the constraint and hence entails weaker constraints, and, on the other hand, tell primitives can synchronize with ask operations for weaker constraints. Conversely, any ask operation can be viewed as asking for stronger constraints. Constraints to be considered as weaker or stronger are formalized as the following sets $c \downarrow \rho$ and $c \uparrow \rho$, respectively.

**Definition 4.1** For any goal $G$ and any store $\sigma$, define

\[
\begin{align*}
c \downarrow \rho &= \{d \in C : \rho \cup \{c\} \vdash d, \rho \not\vdash d\} \\
c \uparrow \rho &= \{d \in C : \rho \cup \{d\} \vdash c\}
\end{align*}
\]

Failure sets will be required to be closed in the following way.

**Definition 4.2** The set $S \subseteq Sconst$ is closed with respect to the store $\sigma$ if, for any constraint $c$, on the one hand, $c \downarrow \sigma \subseteq S$ when $c \in S$ and, on the other
hand, \( \overline{c\sigma} \subseteq S \) when \( \overline{c} \in S \).

The closure of a set \( X \) is defined in a constructive way as follows.

**Definition 4.3** For any set \( X \subseteq \text{Seconst} \) and any store \( \sigma \), the closure of \( X \) with respect to \( \sigma \), denoted \( \overline{X}^\sigma \), is defined as follows: \( \overline{X}^\sigma = \bigcup \{ x|\sigma : x \in X \} \cup \bigcup \{ \overline{x|\sigma} : \overline{x} \in X \} \).

Several properties are worth mentioning.

**Proposition 4.4** For any \( X \subseteq \text{Seconst} \), any constraint \( c \), any stores \( \rho \) and \( \sigma \) such that \( \sigma \not\models c \),

i) the sets \( \text{Seconst} \setminus (c\sigma) \) and \( \text{Seconst} \setminus (c|\sigma) \) are closed wrt \( \sigma \).

ii) \( X \setminus \{ \overline{x} \in X : \rho \vdash x \} \subseteq \overline{X}^\rho \).

iii) if \( X \) is closed wrt \( \rho \), then \( \overline{X}^\rho \subseteq X \).

iv) the set \( \overline{X}^\rho \) is closed wrt \( \rho \).

The notions of steps, ending marks, and sequences of histories are formalized as follows. It is worth noting that, according to the intuition provided by the operational semantics, steps are required to be monotonic in the sense that the output state contains more information than the input state. Denotational histories will be required to be monotonic as well. However, for simplicity purposes, we do not require this property for histories, in general.

**Definition 4.5** Define the set of histories \( \text{Shist} \) as the set \( (\text{Sstep})^\omega \times (\text{Sstore} \times \text{Smark})_{\text{cl}} \) where

i) \( \text{Sstep} = \{(\rho,l,\sigma) : \rho,\sigma \in \text{Sstore}, l \in \text{Slabel}, \rho \subseteq \sigma \} \) denotes the set computational steps;

ii) \( \text{Smark} = \{\delta^+\} \cup \{\delta^- \times \mathcal{P}(\text{Seconst})\} \) denotes the set of ending marks, with any element of \( \{\delta^- \times \mathcal{P}(\text{Seconst})\} \) being rewritten as \( \delta^-(X) \) instead of \( \delta^-(X) \);

iii) where the cl subscript indicates that pairs \( (\sigma,\delta^-(X)) \) for \( X \) closed with respect to \( \sigma \) should only be considered as failing ending marks.

**Notation 4.6** Let \( S \) be a set of histories of \( \text{Shist} \) and \( p \) be a sequence of \( (\text{Sstep})^\omega \). We denote by \( S[p] \) the set \( \{ h : p.h \in S \} \) and by \( S^- \) the set \( \{ h : h = (\sigma,\delta^-(X)) \in S \} \).

**Notation 4.7**

1) Let \( h \) be an history of \( \text{Shist} \). Then

\[
\text{init}(h) = \begin{cases} 
\rho & \text{if } h = (\rho,l,\sigma), h' \\
\sigma & \text{if } h = (\sigma,\delta^+) \\
\sigma & \text{if } h = (\sigma,\delta^-(X)) 
\end{cases}
\]

Moreover, for any set \( S \) of histories and any store \( \sigma \), we denote by \( S\uparrow\sigma \) the set \( \{ h \in S : \sigma \subseteq \text{init}(h) \} \).
2) Let, for \( n \geq 0 \) and \( \delta \in \text{Smark} \), the history

\[
h = (\rho_1, l_1, \sigma_1) \ldots (\rho_n, l_n, \sigma_n)(\rho_{n+1}, \delta)
\]

be in \( \text{Shist} \). Then \( \text{diff}(h) = \bigcup_{i=1}^{n} (\sigma_i \setminus \rho_i) \) and \( \text{const}(h) = \rho_1 \cup \cdots \cup \rho_n \cup \rho_{n+1} \cup \sigma_1 \cup \cdots \cup \sigma_n \). Moreover, abusing notations, we shall lift \( \text{diff} \) and \( \text{const} \) to sets of histories in the expected manner.

**Definition 4.8** The history \( h = (\rho_1, l_1, \sigma_1) \ldots (\rho_n, l_n, \sigma_n)(\rho_{n+1}, \delta) \) is

i) monotonic iff \( \sigma_i \subseteq \rho_{i+1} \) for \( i = 1, \ldots, n \),

ii) continuous iff \( \sigma_i = \rho_{i+1} \) for \( i = 1, \ldots, n \),

iii) real iff \( l_1 = \cdots = l_n = \tau \).

Moreover, \( \overline{h} \) is used to denote the sequence \( \sigma_1 \ldots \sigma_n \delta' \) where \( \delta' = \delta^+ \) if \( \delta = \delta^+ \) and \( \delta' = \delta^- \), otherwise.

4.2 Semantic domain

**Definition 4.9** The semantic domain is defined as the set \( \text{Sdhist} \) of monotonic histories. Its elements are subsequently called denotational histories or, more often, histories, when the context allows this abuse of language.

4.3 Denotational semantics

The semantic domain being specified, defining a compositional semantics consists, on the one hand, in specifying the meaning of elementary statements and, on the other hand, in providing an operator at the semantic level for each syntactic operator. We start by this last task in the following subsection. A compositional semantics is defined next. It is called denotational in view of its compositionality property and the fact that it is defined on denotations only without reference to a transition system.

4.3.1 Semantic operators

There are three operators to combine elementary goals: sequential composition, parallel composition, and choice. Let us examine each of them in turn.

**Sequential composition.** Since the semantic histories may include gaps and start on any input store, composing the meaning of two subgoals amounts to concatenating their histories. This is achieved by the following operator, where further care is taken, in the expected manner, for monotonicity and the termination marks.

**Definition 4.10** Define \( \overline{\sum} : \mathcal{P}(\text{Sdhist}) \times \mathcal{P}(\text{Sdhist}) \rightarrow \mathcal{P}(\text{Sdhist}) \) as the following function: for any subsets \( S_1, S_2 \) of \( \text{Sdhist} \),

\[
S_1 \overline{\sum} S_2 = \{ h_1. h_2 : h_1.(\rho_1, \delta^+) \in S_1, h_2 \in S_2, \rho_1 \subseteq \text{init}(h_2) \}
\]

\[
\cup \{ h_1.(\rho, \delta^- (X)) : h_1.(\rho, \delta^-(X)) \in S_1 \}
\]
Parallel composition. Parallel composition is modelled in an interleaving fashion. Consequently, composing in parallel two semantic histories amounts to their merge or to synchronize their steps. Again care has to be taken to termination marks, as formalized below.

**Definition 4.11** Define the parallel composition of two histories as the function \( \|_h : \mathcal{Sdhist} \times \mathcal{Sdhist} \rightarrow \mathcal{P}(\mathcal{Sdhist}) \) according to their form by means of the following equalities. There, \( \delta \) stands either for \( \delta^+ \) or \( \delta^- (X) \), for some set \( X \).

\[
(p_1, l_1, \sigma_1).h_1 \|_h (p_2, l_2, \sigma_2).h_2
\]

\[
= \{(p_1, l_1, \sigma_1).h : h \in h_1 \|_h (p_2, l_2, \sigma_2).h_2, \sigma_1 \subseteq \rho_2\}
\]

\[
\cup \{(p_2, l_2, \sigma_2).h : h \in (p_1, l_1, \sigma_1).h_1 \|_h h_2, \sigma_2 \subseteq \rho_1\}
\]

\[
\cup \{(\rho, \tau, \sigma).h : \rho_1 = \rho_2 = \rho, l_1 = c_1, l_2 = \overline{c_2},
\sigma = \sigma_1 = \rho \cup \{c_1\}, \sigma_2 = \rho \cup \{c_2\}, \sigma \vdash c_2,
\quad h \in h_1 \|_h h_2\}
\]

\[
\cup \{(\rho, \tau, \sigma).h : \rho_1 = \rho_2 = \rho, l_1 = \overline{c_1}, l_2 = c_2,
\sigma = \sigma_2 = \rho \cup \{c_2\}, \sigma_1 = \rho \cup \{c_1\}, \sigma \vdash c_1,
\quad h \in h_1 \|_h h_2\}
\]

\[
(p_1, l_1, \sigma_1).h_1 \|_h (p_2, \delta_2)
\]

\[
= (p_2, \delta_2) \|_h (p_1, l_1, \sigma_1).h_1
\]

\[
= \{(p_1, l_1, \sigma_1).h : h \in h_1 \|_h (p_2, \delta_2)\}
\]

\[
(p_1, \delta_1) \|_h (p_2, \delta_2)
\]

\[
= \begin{cases} 
\{(p_1, \delta^+)\}, & \text{if } \rho_1 = \rho_2 \text{ and } \delta_1 = \delta_2 = \delta^+, \\
\{(p_1, \delta^-(X))\}, & \text{if } \rho_1 = \rho_2, \delta_1 = \delta^+, \delta_2 = \delta^- (X), \\
\{(p_1, \delta^-(X))\}, & \text{if } \rho_1 = \rho_2, \delta_1 = \delta^- (X), \delta_2 = \delta^+, \\
\{(p_1, \delta^-(X))\}, & \text{if } \rho_1 = \rho_2, \delta_1 = \delta^- (X_1), \delta_2 = \delta^- (X_2), \\
& X \subseteq X_1 \cap X_2, X \text{ closed wrt } \rho_1 \\
& (\text{Seconst} \setminus X_1) \cap (\text{Seconst} \setminus X_2) = \emptyset, \\
& \emptyset, & \text{otherwise.}
\end{cases}
\]

A word on this definition is in order. The first equality involves four sets. The first two correspond to making, as first step, the first step of the histories under consideration. The last two correspond to synchronizing the first steps of these histories.
The second equality basically asserts that ending marks should be combined only and to postpone this combination after all the steps of the two histories have been treated. This is suggested by the operational semantics: when one of the goals ends, the other concurrent goal has to continue.

The last equality deals with combining ending marks. As a general rule, marks may be combined only if they agree on their input store. Note that this is actually no problem since denotational histories may include gaps. All the cases should be clear, except the fourth one. The intuition behind it relies on the standard meaning of failures. A mark of the form $\delta^{-}(X)$ expresses that the considered goal is suspended and that, whatever its environment is, it is unable to perform the actions of $X$. Note that $X$ is not required to contain all the actions that the goal cannot do; it is simply a subset of the set of these actions. Therefore, for two goals $G_1$ and $G_2$ suspended and unable to perform the actions of $X_1$ and $X_2$, respectively, the parallel composition $G_1 \parallel G_2$ is unable to perform the actions of $X_1 \cap X_2$ and is suspended provided $G_1$ and $G_2$ cannot synchronize. A sufficient condition for that is obtained by noting that the actions that $G_i$ can do are in the complement of $X_i$, namely in $\text{Seconst} \setminus X_i$. Consequently, if telling a constraint is understood as telling all the new constraints it subsumes, then $G_1$ and $G_2$ synchronize only if there is an action of $\text{Seconst} \setminus X_1$ which is the complement of an action of $\text{Seconst} \setminus X_2$ ie only if $(\text{Seconst} \setminus X_1) \cap (\text{Seconst} \setminus X_2) \neq \emptyset$. More generally, if $X_i \ (i = 1, 2)$ is exactly the set of all actions that $G_i$ cannot do (and not simply a subset), then, following the same reasoning, it is easy to establish that $G_1$ and $G_2$ do not synchronize if and only if $(\text{Seconst} \setminus X_1) \cap (\text{Seconst} \setminus X_2) \neq \emptyset$. Moreover, the set of actions that $G_1 \parallel G_2$ cannot perform is included in $X_1 \cap X_2$.

**Definition 4.12** Define the parallel composition of two sets of histories as the natural lifting of function $\parallel_h$, namely as the function $\parallel : \mathcal{P}(\text{Sdhist}) \times \mathcal{P}(\text{Sdhist}) \rightarrow \mathcal{P}(\text{Sdhist})$ defined as follows: for any subset $S_1$, $S_2$ of $\text{Sdhist}$, $S_1 \parallel S_2 = \bigcup \{ h_1 \parallel_h h_2 : h_1 \in S_1, h_2 \in S_2 \}$.

**Choice.** Choice is modelled as a global choice, namely a goal formed from the choice of two goals can proceed as any of its components. As before care has to be taken to termination marks. The composed goal fails if the two components do so; it succeeds if at least one of the two components does.

**Definition 4.13** Define $\begin{array}{c} \tilde{\oplus} : \mathcal{P}(\text{Sdhist}) \times \mathcal{P}(\text{Sdhist}) \rightarrow \mathcal{P}(\text{Sdhist}) \end{array}$ as the following function: for any subset $S_1$, $S_2$ of $\text{Sdhist}$, $S_1 \begin{array}{c} \tilde{\oplus} S_2 = (S_1 \setminus S_1^-) \cup (S_2 \setminus S_2^-) \cup (S_1^- \cap S_2^-). \end{array}$

4.3.2 **Definition**

Given the operators $\begin{array}{c} \tilde{\cap}, \parallel, \end{array}$ and $\begin{array}{c} \tilde{\oplus} \end{array}$, defining the denotational semantics amounts to specifying the semantics of the basic constructs $\text{tell}$ and $\text{ask}$, and of the empty goal. This is achieved according to the intuition given by their operational behavior.
Consequently, a \( \text{tell}(c) \) operation makes a silent step \((\rho, \tau, \rho)\) if the input store \(\rho\) entails \(c\). It makes an hypothetical step \((\rho, c, \rho \cup \{c\})\) in case \(\rho \not\vdash c\). Moreover, in that case, a failure mark \(\delta^-(X)\) is registered for any subset \(X\) of actions that \(\text{tell}(c)\) cannot perform i.e. any set disjoint from \(\{d \in C : \rho \cup \{c\} \vdash d, \rho \not\vdash d\}\).

An \(\text{ask}(c)\) operation has a dual behaviour. It makes a silent step \((\rho, \tau, \rho)\) if the input store \(\rho\) entails \(c\). Otherwise, it makes an hypothetical step \((\rho, d, \rho \cup \{d\})\) hoping for the presence of a concurrent \(\text{tell}(d)\) operation with \(\rho \cup \{d\} \vdash c\). Moreover, as before, a failure mark \((X)\) is enclosed for those actions that \(\text{ask}(c)\) cannot perform i.e. for all the actions except \(c\).

All these silent and hypothetical steps are followed by a successfully ending mark \((\sigma, \delta^+)\) however for all the stores \(\sigma \supseteq \rho\) only in order to maintain the monotonicity property of denotational histories.

\textbf{Definition 4.14} Define the denotational semantics as the following function \(\mathcal{D}_h : S_{\text{goal}} \to \mathcal{P}(S_{\text{hist}})\): for any constraint \(c\), for any goals \(G_1, G_2\),

\[
\mathcal{D}_h(\text{tell}(c)) = \{(\rho, \tau, \rho). (\sigma, \delta^+) : \rho, \sigma \in S_{\text{store}}, \rho \vdash c, \rho \subseteq \sigma \}
\]

\[
\cup \{(\rho, c, \rho \cup \{c\}). (\sigma, \delta^+) : \rho, \sigma \in S_{\text{store}}, \rho \not\vdash c, \\
\rho \cup \{c\} \subseteq \sigma \}
\]

\[
\cup \{(\rho, \delta^-(X)) : \rho \in S_{\text{store}}, \rho \not\vdash c, X \subseteq (\text{Seconst} \setminus c\rho), \\
X \text{ closed wrt } \rho \}
\]

\[
\mathcal{D}_h(\text{ask}(c)) = \{(\rho, \tau, \rho). (\sigma, \delta^+) : \rho, \sigma \in S_{\text{store}}, \rho \vdash c, \rho \subseteq \sigma \}
\]

\[
\cup \{(\rho, d, \rho \cup \{d\}). (\sigma, \delta^+) : \rho, \sigma \in S_{\text{store}}, \rho \not\vdash c, \\
\rho \cup \{d\} \vdash c, \rho \cup \{d\} \subseteq \sigma \}
\]

\[
\cup \{(\rho, \delta^-(X)) : \rho \in S_{\text{store}}, \rho \not\vdash c, X \subseteq (\text{Seconst} \setminus c\rho), \\
X \text{ closed wrt } \rho \}
\]

\[
\mathcal{D}_h(\triangle) = \{(\sigma, \delta^+) : \sigma \in S_{\text{store}} \}
\]

\[
\mathcal{D}_h(G_1; G_2) = \mathcal{D}_h(G_1) \circ \mathcal{D}_h(G_2)
\]

\[
\mathcal{D}_h(G_1 || G_2) = \mathcal{D}_h(G_1) \parallel \mathcal{D}_h(G_2)
\]

\[
\mathcal{D}_h(G_1 + G_2) = \mathcal{D}_h(G_1) + \mathcal{D}_h(G_2)
\]

\textbf{5 Correctness}

The semantics \(\mathcal{D}_h\) is compositional by construction. It is also correct with respect to the semantics \(\mathcal{O}_h\) in the sense that this operational semantics can be obtained from it. In fact, it is sufficient to take from \(\mathcal{D}_h\) the real and continuous histories starting in the empty store to get those produced by \(\mathcal{O}_h\).

\textbf{Theorem 5.1} Define \(\alpha : \mathcal{P}(S_{\text{hist}}) \to \mathcal{P}(S_{\text{hist}})\) as follows: for any subset
\( S \subseteq \text{Shist}, \)

\[
\alpha(S) = \{ h : h \in S, h \text{ real and continuous, } \text{init}(h) = \epsilon \}.
\]

Then, for any goal \( G \), \( \mathcal{O}_h(G) = \alpha(\mathcal{D}_h(G)) \).

Note that, as a corollary of this proposition an operational history \( \sigma_1, \ldots, \sigma_n, \delta \) corresponds to a continuous and real denotational history starting in the empty store: \( (\epsilon, \tau, \sigma_1), (\sigma_1, \tau, \sigma_2), \ldots, (\sigma_n, \delta') \). This correspondance is biunivoque if \( \delta = \delta^+ \). In that case, \( \delta' = \delta^- \). It is essentially biunivoque if \( \delta = \delta^+ \). In that case, the prefix up to the last element \( (\sigma_n, \delta') \) is fixed and the only varying part is \( \delta' \) which takes the form \( \delta^-(X) \) for some set \( X \). The set of all such equivalent histories is denoted through the following \([S]\) notation.

**Notation 5.2** For any set \( S \subseteq \text{Shist} \), define \([S]\) as the largest set \( S' \) such that \( \alpha(S') = \alpha(S) \).

6. **Full abstraction**

6.1 **Definition**

Full abstraction consists in requiring that the denotational semantics of two goals are identical iff, from the operational point of view, the two goals behave identically even when they are composed with other goals in all the possible manners. This form of composition is provided by the classical notion of context, which we will not recall here.

**Definition 6.1** The semantics \( \mathcal{D}_h \) is fully abstract with respect to the semantics \( \mathcal{O}_h \) iff the following property holds: for any goals \( G_1, G_2 \), the following assertions are equivalent

1. for any context \( C \), \( \mathcal{O}_h(C[G_1]) = \mathcal{O}_h(C[G_2]) \);
2. \( \mathcal{D}_h(G_1) = \mathcal{D}_h(G_2) \).

6.2 **Intuition**

The compositional property of \( \mathcal{D}_h \) together with theorem 5.1 establish the implication \((ii) \Rightarrow (i)\) of definition 6.1. It thus remains to prove the converse \((i) \Rightarrow (ii)\). This is achieved by contraposition. Given two goals \( G_1, G_2 \) such that \( \mathcal{D}_h(G_1) \neq \mathcal{D}_h(G_2) \) a context \( C \) is constructed such that \( \mathcal{O}_h(C[G_1]) \neq \mathcal{O}_h(C[G_2]) \). The two semantics reporting sets, the construction amounts to constructing from a denotational history \( h \) of one goal, say \( G_1 \), which is not in the denotation of the other \( G_2 \), a context \( C \) and an operational history of \( C[G_1] \) not of \( C[G_2] \). In view of the relation between \( \mathcal{O}_h \) and \( \mathcal{D}_h \) as shown by \( \alpha \) in theorem 5.1, this amounts to establishing the existence of a real and continuous denotational history, starting in \( \epsilon \), which is in \( \mathcal{D}_h(C[G_1]) \) and not in \([\mathcal{D}_h(C[G_2])]\). To that end, following [12], we shall construct from \( h \) a new history \( h' \) and an goal \( T \) such that \( h' \) is in the denotational semantics of \( G_1 \parallel T \) and not in \([\mathcal{D}_h(G_2 \parallel T)]\).
The proof basically proceeds by induction on the length of $h$. In the base case, $h$ takes the form $(\sigma, \delta)$ with $\delta$ being either $\delta^+$ or $\delta^-(X)$. The tester $T$ then basically constructs a real and continuous sequence yielding $\sigma$ from the initial store $e$ in a way that, on the one hand, prevents $G_1$ and $G_2$ to do any intermediary step, and, on the other hand, forces $G_1$ and $G_2$ to do the last step $(\sigma, \delta)$. By hypothesis, this is possible for $G_1$ and not for $G_2$ if $\delta = \delta^+$ or if $\delta = \delta^-$ but $D_h(G_2)^{-} = \emptyset$. In the case where $(\sigma, \delta^-(Y)) \in D_h(G_2)$, for some set $Y$, then a test can be appended to $T$ which forbids $G_2$ to do the last step but yet allow $G_1$ to do it.

In the inductive case, $h$ takes the form $(\rho, l, \sigma).h^*$ for some history $h^*$. Two cases are possible: either there is no history starting by $(\rho, l, \sigma)$ in $D_h(G_2)$ or those which start by $(\rho, l, \sigma)$ cannot end by $h^*$. In the first case, the proof proceeds as in the base case. In the second case, the proof uses induction. However, the induction should be applied for $h^*$ in $D_h(G_1)\{(\rho, l, \sigma)\}$ and not in $D_h(G_2)\{(\rho, l, \sigma)\}$. Because of the way failures are combined by the choice operators, these sets turned out to be basically but not exactly the denotations $D_h(G_1)$ and $D_h(G_2)$, of some goals $G'_1$ and $G'_2$. Consequently the induction needs to be slightly generalized to sets of denotational histories. This extension being discarded here for the sake of simplicity, we thus apply the induction hypothesis for $h^*, G'_1$ and $G'_2$. It points out a tester $T'$ and an history $h''$ which is in $D_h(G_1) \parallel T')$ and not in $[D_h(G_2) \parallel T')]$. From there we should construct a tester $T$ and an history $h'''$ in $D_h(G_1) \parallel T)$ and not in $[D_h(G_2) \parallel T)]$. Basically, the step $(\rho, l, \sigma)$ has to be done before $h''$ and since $h'''$ needs to be continuous, $h'''$ has to start in a possibly non empty store. Hence, we have to generalize the theorem and construct in general from $h$ an history $h'$ which start in any initial store. Given this generalization, the tester $T$ basically consists of first making the steps necessary to produce $\rho$ from the given initial store, then of making an auxiliary transition from $\sigma$ to some $\sigma'$ chosen so as to ensure that $G_1$ and $G_2$ have to do the step $(\rho, l, \sigma)$, and finally consists of $T'$.

A slight extension is needed when $l \neq \tau$. In that case, in order to guarantee $h'$ to be real, the tester $T$ is requested to perform the complementary step $(\rho, \bar{l}, \sigma)$ before making the transition from $\sigma$ to $\sigma'$.

6.3 Auxiliary concepts

The above intuition points out an auxiliary task which consists of making by an auxiliary goal the steps necessary to produce a given target store $\sigma$ from a given initial store $\rho$. These steps are subsequently achieved by means of the following goal $G^S_{\rho_{\sim}^{\sigma}}$.

**Notation 6.2** Let $c_1, \ldots, c_m$ be contraints and let $\rho$ and $\sigma$ be two stores such that $\rho \subseteq \sigma$. Then, we denote by $\sigma \setminus \rho \asymp \{c_1, \ldots, c_m\}$ the following properties

i) $\sigma = \rho \cup \{c_1, \ldots, c_m\}$

ii) $\rho \cup \{c_{i_1}, \ldots, c_{i_k}\} /\!\!\!\not\subset c_i$, for any (possibly empty) subset $\{c_{i_1}, \ldots, c_{i_k}\}$ of the $c_j$’s and for any $c_i \not\in \{c_{i_1}, \ldots, c_{i_k}\}$

17
Note that, as a consequence of point ii above, ρ ∤ c_i, for i = 1, · · · , m

**Definition 6.3** Let S be a finite set of constraints, and ρ and σ be two stores, such that ρ ⊆ σ. Consider, on the one hand, c_1, · · · , c_m constraints such that σ \ ρ ≻ \{c_1, · · · , c_m\} and, on the other hand, a_1, · · · , a_m constraints such that

(i) σ ∪ S \{a_1, · · · , a_i\} ∤ a_{i+1}, for i = 1, · · · , m

(ii) for any store α and any constraint c ∈ S, if α ∪ \{a_1, · · · , a_m\} ⊨ c then α ⊨ c;

(iii) for any store γ ⊆ S, for any constraint c, if γ ∪ \{a_1, · · · , a_m\} ⊨ c then γ ⊨ c or exclusively \{a_1, · · · , a_m\} ⊨ c.

Then, abusing language by forgetting about the constraints c_i ’s in the notation, we denote by \(G_{\rho \rightarrow \sigma}^{S;\{a_1, · · · , a_m\}}\) the following goal

\[
(tell(c_1) \parallel ask(c_1)); (tell(a_1) \parallel ask(a_1)); \\
\cdots \\
(tell(c_m) \parallel ask(c_m)); (tell(a_m) \parallel ask(a_m)).
\]

Moreover, we note by \(\Sigma_{\rho \rightarrow \sigma}^{S;\{a_1, · · · , a_m\}}\) the associated sequence of states

\[(\rho_0, τ, γ_1). (γ_1, τ, ρ_1). · · · (ρ_{i-1}, τ, γ_i). (γ_i, ρ_i). · · · (ρ_{m-1}, τ, γ_m). (γ_m, ρ_m)\]

where \(ρ_0 = ρ, γ_i = ρ_{i-1} \cup \{c_i\}, ρ_i = γ_i \cup \{a_i\}, \) for \(i = 1, · · · , m.\) Note, in particular, that \(ρ_m = σ \cup \{a_1, · · · , a_m\}.\)

Obviously, \(G_{\rho \rightarrow \sigma}^{S;\{a_1, · · · , a_m\}}\) can perform the history \(\Sigma_{\rho \rightarrow \sigma}^{S;\{a_1, · · · , a_m\}}. (ρ_m, δ^+).\) If S is suitably chosen, it also has the property of being responsible for making the steps of \(\Sigma_{\rho \rightarrow \sigma}^{S;\{a_1, · · · , a_m\}}\) when placed in parallel with another goal.

**Proposition 6.4** Let ρ and σ be two stores such that ρ ⊆ σ. Let A be a goal and let S be the set of constraints present in the tell and ask primitives of A.

1) Any history \(h = \Sigma_{\rho \rightarrow \sigma}^{S;\{a_1, · · · , a_m\}}. h'\) of \(\mathcal{D}_h(G_{\rho \rightarrow \sigma}^{S;\{a_1, · · · , a_m\}} \parallel A)\) is from the set \(\Sigma_{\rho \rightarrow \sigma}^{S;\{a_1, · · · , a_m\}}. (γ, δ^+) \parallel_h h_a\) for some store γ and some history \(h_a \in \mathcal{D}_h(A).\)

2) For any goal B, any history \(h = \Sigma_{\rho \rightarrow \sigma}^{S;\{a_1, · · · , a_m\}}. h'\) of \(\mathcal{D}_h((G_{\rho \rightarrow \sigma}^{S;\{a_1, · · · , a_m\}}; B) \parallel A)\) is from the set \(\Sigma_{\rho \rightarrow \sigma}^{S;\{a_1, · · · , a_m\}}. h_b \parallel_h h_a\) for some histories \(h_a \in \mathcal{D}_h(A)\) and \(h_b \in \mathcal{D}_h(B).\)

### 6.4 Key properties

The generalization of denotations to sets of denotations announced in subsection 6.2 requires to characterize these sets by properties verified by the denotation of any goal. We now define the key ones.

**Definition 6.5** The set \(S \subseteq S_{\text{hist}}\) is action-finite at level n if for any prefix
$p \in (\text{Store} \times \text{Label} \times \text{Store})^n$ of length $n$, for any store $\sigma$, the set $\text{Act}(S[p], \sigma)$ is finite. It is uniformly action-finite if it is action-finite at any level.

**Definition 6.6** A set $S \subseteq S_{\text{hist}}$ is extensible from a store $\nu$ if for any stores $\rho, \rho_1, \ldots, \rho_n, \sigma_1, \ldots, \sigma_n \supseteq \nu$, for any labels $l_1, \ldots, l_n$,

1) there is a continuous history in $S$ starting in $\rho$
2) if $S[(\rho_1, l_1, \sigma_1), \ldots, (\rho_n, l_n, \sigma_n)] \neq \emptyset$ and if $\sigma_n \subseteq \rho$ then there is a continuous history in $S[(\rho_1, l_1, \sigma_1), \ldots, (\rho_n, l_n, \sigma_n)]$ starting in $\rho$.

**Definition 6.7** Let $S_1$ and $S_2$ be two sets of constraints. $S_1$ is independent from $S_2$ if the two following conditions hold:

(i) for any constraint $c \in S_1$ and any store $\sigma$, if $\sigma \cup S_2 \vdash c$ then $\sigma \vdash c$;

(ii) for any store $\rho \subseteq S_1$, for any constraint $c$, if $\rho \cup S_2 \vdash c$ then $\rho \vdash c$ or exclusively $S_2 \vdash c$.

Alternatively, $S_2$ is said not to influence $S_1$.

**Definition 6.8** A set of histories $S \subseteq S_{\text{hist}}$ is consistent wrt to the store $\gamma$ if for any superset $S_x$ of $\text{diff}(S)$, for any set of constraints $S_c$ not influencing $S_x$, the following properties hold:

(i) for any labels $l_1, \ldots, l_n$ and for any stores $\rho_1, \ldots, \rho_{n+1}, \sigma_1, \ldots, \sigma_n$ included in $S_x$ and such that $\rho_i \subseteq \sigma_i \subseteq \rho_{i+1}$ for $i = 1, \ldots, n$, and $\gamma \subseteq \rho_i$, $\gamma \subseteq \sigma_j$, for $i = 1, \ldots, n + 1$ and $j = 1, \ldots, n$,

$$(\rho_1, l_1, \sigma_1), \ldots, (\rho_n, l_n, \sigma_n), (\rho_{n+1}, \delta^+) \in S \iff (\rho_1 \cup S_c, l_1, \sigma_1 \cup S_c), \ldots, (\rho_n \cup S_c, l_n, \sigma_n \cup S_c), (\rho_{n+1} \cup S_c, \delta^+) \in S$$

(ii) for any labels $l_1, \ldots, l_n$, for any stores $\rho_1, \ldots, \rho_{n+1}, \sigma_1, \ldots, \sigma_n$ included in $S_x$, and such that $\rho_i \subseteq \sigma_i \subseteq \rho_{i+1}$ for $i = 1, \ldots, n$, and $\gamma \subseteq \rho_i$, $\gamma \subseteq \sigma_j$, for $i = 1, \ldots, n + 1$ and $j = 1, \ldots, n$, for any set $X$ closed wrt $\rho_{n+1}$,

$$(\rho_1, l_1, \sigma_1), \ldots, (\rho_n, l_n, \sigma_n), (\rho_{n+1}, \delta^-(X)) \in S \iff (\rho_1 \cup S_c, l_1, \sigma_1 \cup S_c), \ldots, (\rho_n \cup S_c, l_n, \sigma_n \cup S_c), (\rho_{n+1} \cup S_c, \delta^-(X)) \in S$$

(iii) for any labels $l_1, \ldots, l_n$, for any stores $\rho_1, \ldots, \rho_{n+1}, \sigma_1, \ldots, \sigma_n$ included in $S_x$, and such that $\rho_i \subseteq \sigma_i \subseteq \rho_{i+1}$ for $i = 1, \ldots, n$, and $\gamma \subseteq \rho_i$, $\gamma \subseteq \sigma_j$, for $i = 1, \ldots, n + 1$ and $j = 1, \ldots, n$, for any set $X$ closed wrt $\rho_{n+1} \cup S_c$,

$$(\rho_1, l_1, \sigma_1), \ldots, (\rho_n, l_n, \sigma_n), (\rho_{n+1}, \delta^-(X_{\rho_{n+1}})) \in S \iff (\rho_1 \cup S_c, l_1, \sigma_1 \cup S_c), \ldots, (\rho_n \cup S_c, l_n, \sigma_n \cup S_c), (\rho_{n+1} \cup S_c, \delta^-(X)) \in S$$

(iv) for any store $\rho$, any label $l$, any history $h$ such that $\sigma \cup S_c \subseteq \text{init}(h)$,

$$(\rho, l, \sigma).h \in S \iff (\rho \cup S_c, l, \sigma \cup S_c).h \in S$$

It is uniformly consistent wrt $\gamma$ if it is consistent wrt $\gamma$ and if, for any stores $\rho, \sigma, \gamma'$ such that $\gamma \subseteq \rho, \sigma \subseteq \gamma'$, and for any label $l$ if, it is not empty, the set $S[(\rho, l, \sigma)]$ is uniformly consistent wrt $\gamma'$. 

19
Definition 6.9 The set $S \subseteq S_{\text{dhist}}$ satisfies the disjointness deadlock condition at level $n$ iff for any prefix $p \in (S_{\text{store}} \times S_{\text{label}} \times S_{\text{store}})^n$ of length $n$ and for any store $\sigma$ such that $S[p] \neq \emptyset$ and $(\sigma, \delta^{-}(X)) \in S[p]$ for some set $X$ closed wrt $\sigma$, there are non-empty goals $G_{1}, \ldots, G_{m}$ ($m > 0$) such that

(i) $<G_i, \sigma> \xrightarrow{r} G'$
(ii) for any $Y \subseteq \text{Seconst}$ closed wrt $\sigma$, one has $(\sigma, \delta^{-}(Y)) \in S[p]$ iff $Y \cap \text{Actions}(G_i, \sigma) = \emptyset$ for some $i = 1, \cdots, m$,

where the set $\text{Actions}(G, \rho)$ is composed of all the synchronized tell and ask operations that $G$ can perform:

\[
\text{Actions}(G, \rho) = \bigcup \{ c \mid \rho : <G, \rho> \xrightarrow{c} <G', \sigma>, G' \in S_{\text{goal}}, \sigma \in S_{\text{store}}, c \in C \}
\]

\[
\bigcup \{ \overrightarrow{c} \mid \rho : <G, \rho> \xrightarrow{c} <G', \sigma>, G' \in S_{\text{goal}}, \sigma \in S_{\text{store}}, c \in C \}
\]

The set $S$ satisfies the uniform disjointness deadlock condition if it satisfies the disjointness deadlock condition at any level.

The above properties are summarized in the following notion of coherence.

Definition 6.10 A set of denotational histories is called coherent if its set $\text{diff}(S)$ is finite, if it is uniformly action-finite, if it is extensible wrt the empty store $\epsilon$, if it is uniformly consistent wrt $\epsilon$, and if it enjoys the uniform disjointness deadlock condition.

Proposition 6.11 For any goal $G$, the set $D_{\text{h}}(G)$ is coherent.

6.5 Key proposition

Theorem 6.12 Let $S_1$, $S_2$ be two coherent subsets of $S_{\text{dhist}}$. Then, for any store $\alpha$ such that $(S_1 \uparrow \alpha) \setminus (S_2 \uparrow \alpha) \neq \emptyset$, there is a goal $T$ and a continuous and real history $h \in (S_1 \parallel D_{\text{h}}(T)) \setminus [(S_2 \parallel D_{\text{h}}(T))]$ which starts in $\alpha$.

6.6 Proof of the full abstraction property

We are now in a position to establish the full abstraction property. Assume $D_{\text{h}}(G_1) \neq D_{\text{h}}(G_2)$. Then, since both $D_{\text{h}}(G_1)$ and $D_{\text{h}}(G_2)$ are sets, there is an history $h$ which is in one set and not in the other one. Without lost of generality, we may assume that $h \in D_{\text{h}}(G_1)$ and $h \notin D_{\text{h}}(G_2)$. Then $D_{\text{h}}(G_1) \setminus D_{\text{h}}(G_2) \neq \emptyset$ and, consequently taking as coherent sets $S_1 = D_{\text{h}}(G_1)$, $S_2 = D_{\text{h}}(G_2)$, and $\epsilon$ as initial store $\alpha$, theorem 6.12 establishes that there is a goal $T$ and a real and continuous history $h \in (D_{\text{h}}(G_1) \parallel D_{\text{h}}(T)) \setminus [(D_{\text{h}}(G_2) \parallel D_{\text{h}}(T))]$ which starts in $\epsilon$. Note that, by definition 4.14, $h \in D_{\text{h}}(G_1 \parallel T) \setminus [D_{\text{h}}(G_2 \parallel T)]$. Therefore, by theorem 5.1, $\tilde{h}$ is an operational history of $O_{\text{h}}(G_1 \parallel T)$ which is not in $O_{\text{h}}(G_2 \parallel T)$. There is thus a context $C = \Box \parallel T$, such that $O_{\text{h}}(C[G_1]) \neq O_{\text{h}}(C[G_2])$, which concludes the proof.
7 Conclusion

This paper has presented a fully abstract semantics for a new concurrent constraint language based as in the classical constraint setting on tell and ask primitives but embodying both asynchronous communication and synchronous communication. These two forms of communication are obtained by forcing ask and tell primitive to synchronize except if they deal with constraint already entailed by the considered store, in which case they can proceed asynchronously.

This framework called Scc is proved to be different in nature both from traditional concurrent constraint programming, as exemplified by the cc languages, and from traditional concurrent programming, as exemplified by CCS. It has been shown in [4] to be adequate to program multi-agents applications.

The Scc framework thus calls for new semantic treatments. A first answer towards this long term goal has been given in this paper. An operational semantics reporting the steps of the computations has been presented and then a denotational semantics has been proposed. It uses monotonic sequences of labelled pairs of input-output states, possibly containing gaps, and ending – according to the logic programming tradition – with marks reporting success or failure. This denotational semantics is proved to be correct with respect to the operational semantics as well as fully abstract. Among the major technical innovations to that end are the introduction of a closedness property required for failure sets together with its related development and the study of consistency properties needed for testing purposes.

8 Acknowledgment


References


