A Fully Abstract Model for Concurrent Constraint Programming*

Frank S. de Boer[1] and Catuscia Palamidessi[2]

[1]Technische Universiteit Eindhoven,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands

[2]Dipartimento di Informatica, Università di Pisa,
Corso Italia 40, 56125 Pisa, Italy

[2]Centre for Mathematics and Computer Science,
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands
email: katuscia@cwi.nl

February 19, 1996

Abstract

Recent results [5] have shown that concurrent Logic programming has a very simple model, based on linear sequences, which is fully abstract with respect to the parallel operator and finite observables. This is intrinsically related to the asynchronous and monotonic nature of the communication mechanism, which consists of asking and telling constraints on a common store. We consider here a paradigm for (asynchronous) concurrent programming, based on the above mechanism, and provided with the standard operators of choice, parallelism, prefixing, and hiding of local variables. It comes out that linear sequences still suffice for a compositional description of all the operators. Moreover, we consider the problem of full abstraction. Since our notion of observables implies the removal of silent steps, the presence of the choice operator induces the same problems (for compositionality) as bisimulation in CCS. We show that in our framework this problem has a simple solution which consists of introducing a semantical distinction between the various ways in which deadlock and failure might occur. The resulting semantics is fully abstract and still based on linear sequences.

1 Introduction

The concurrent constraint paradigm [15], [16], [8], [6], [7] represents a considerable improvement with respect to "classical" concurrent logic languages. The combination with Constraint Logic Programming [10] has allowed on one hand to increase the expressiveness, and, on the other hand, to model in a logical manner the synchronization mechanism [12], [14], thus integrating it with the declarative reading of the language. Constraint programming is based on the notion of computing with systems of partial information. The main feature is that the *store* is seen as a constraint on the range of values that variables can assume, rather than a correspondence variables-values. In other words, the store is seen as a (possibly infinite) set of valuations. Constraints are just finite representations of these sets. For instance, a constraint can be a first order formula, like $\{x = f(y)\}$, representing the set $\{\{y = a, x = f(a)\}, \{y = b, x = f(b)\}, \ldots\}$. Constraints are naturally ordered with respect to (reverse) logical implication. As discussed in [15], [16], this

*Part of this work was supported by the ESPRIT BRA project "Integration"

notion of store leads naturally to a paradigm for concurrent programming. All processes share a common store, that represents the constraint established until that moment. Communication is achieved by adding (telling) consistently some constraint to the store, and by checking (asking) if the store entails (implies) a given constraint. Synchronization is based on a blocking ask: a process waits (suspends) until the store is "strong" enough to entail a certain constraint. Notice that the execution of an ask depends monotonically upon the store, which, in turn, is monotonically increased by the execution of a tell. Since ask and tell are the only actions on the store, the store will evolve monotonically: the associated constraint is initially *true* (no restriction upon the possible values for the variables), and it gets more and more refined in the course of the computation (the set of possible values for the variables gets smaller and smaller).

We address here the problem of a compositional and fully abstract semantics for concurrent constraint programming. Compositionality is considered one of the most desirable characteristics of a formal semantics, since it provides a foundation of program verification and modular design. The difficulty in obtaining this property depends upon the *operators* of the language, the behaviour we want to describe (*observables*), and the degree of *abstraction* we want to reach. A compositional model is called *fully abstract* if it identifies programs that behave in the same way under all the possible contexts. A fully abstract model can be considered to be *the* semantics of a language: all the other compositional semantics can be reduced to it by abstracting from the redundant information. Full abstraction is important, for instance, for deciding correctness of program transformation techniques.

There are various reasonable observation criteria that can be adopted for concurrent constraint programming. One may, for instance, consider the sequences of communication actions engaged by a process, or the sequences of intermediate states of the store, etc. We deal with a more abstract one: only the final results, together with the termination modes: success, failure, or deadlock (failure occurs when a process must tell a constraint inconsistent with the store, deadlock occurs when all processes are stucked on the execution of an *ask*). This choice is motivated by the fact that, due to the monotonic evolution of the store, the intermediate states of the computation are just approximations of the final result.

The main operators of concurrent constraint programming are the communication actions, the prefixing, the parallel composition, the nondeterministic choice, and the hiding of local variables [16]. Concerning the control structure, these operators have been regarded just as a particular case of the classical concurrent paradigms, like CCS and TCSP. As a consequence, the structural operational semantics and the problems of compositionality and full abstraction have been approached mostly by the standard methods (failure sets, trees, etc.) of the semantics of concurrency ([1], [11], [2], [3], [7], [16], [9], [8]).

We think that concurrent constraint languages require a different approach. If we compare their computational model with the one of CCS and TCSP we see a relevant difference:

> the communication mechanism is asynchronous, i.e., the execution of a *tell* do not need to synchronize with a (complementary) *ask* engaged by the environment.

In other words, what a process can do does not depend upon the *present behaviour* of the environment, it only depends upon the store (*tell* can always proceed when consistent with the store, and *ask* can always proceed when entailed by the store), i.e. the *past behaviour* of the system. If we want to describe the semantics via a transition system, a rule like the one of CCS, where complementary actions synchronize is therefore not needed.

On the other hand, a compositional model can be achieved by making explicit (in the transition system) the dependency upon the past. This can be done by adding a *passive* rule that does not exists in CCS: an *arbitrary* assumption about a step made by the environment. Arbitrary means *not related* to what the process exactly needs to proceed: it can be either more (so allowing it to proceed), or less (so it can still suspend) or even inconsistent with the constraint that the process is going to ask [tell] (so to generate failure in the next future). This corresponds to considering all the possible interactions between the given process and arbitrary environments, and it leads to a very simple compositional semantics, consisting of sequences of constraints labeled by assume/tell modes. Notice the contrast with CCS and TCSP, where compositionality requires more complicated structures containing branching information, like trees or failure sets. Moreover, we don't need an interleaving operator to compose these sequences. It is sufficient to pick up the ones that "match".

Our sequences must not be confused with similar structures (sequences of ask/tell constraints, input/output substitutions etc.) used in [13], [8], [7] and [16]. Those structures are used there to represent the sequel of actions that a process will engage. There is a basic conceptual difference between an *assume* constraint and an *ask* [*input*] constraint: the second just represents what a process needs to proceed, i.e. the *minimal* assumption that is necessary for the process to go on successfully. As a consequence, sequences of that kind contain less information and they are compositional only for the success set [13], whereas failure and deadlock still require some branching information. In [8] this branching information is given by failure and suspension sets, in [7] and [16] it is explicitly represented by a tree-like structure.

A totally different approach has been developed in [17] for the determinate subcase of constraint logic programming. The basic idea consists of mapping processes into Scott's closure operators, which have the nice property of being representable as the set of their fixpoints. The operators of the language can then be described just as operations on those sets. In particular, parallelism can be modeled simply by intersection. The resulting model is very elegant, but, unfortunately, it only works for the determinate case. In the same paper, the authors present an extension of their model for the indeterminate case. Again, this extended model is based on sequences of ask/tell actions plus some failure information coding the branching structure.

To our knowledge, the first example of a compositional semantics just based on (assume/tell) linear sequences was given in [4], [5][1]. Those works, however, deal with a concurrent logic language, based on substitutions, which is indeterminate, but with no explicit choice operator (it is "hidden" in the clause union). The compositionality is treated only with respect to the parallel operator.

The model $\mathcal{M}$ we present here, for concurrent constraint programming, is compositional with respect to all the operators of the language (prefixing, parallelism, choice, and hiding of local variables). Moreover, the communication mechanism we deal with is more general, since it is based on constraints instead of substitutions. This model is also more simple and elegant, since the hiding of local variables is here modeled by the existential quantifier.

Since it is induced by linear sequence, the congruence associated to $\mathcal{M}$ is coarser than *reactive equality* (a congruence, on tree-like structures, based on strong bisimulation) [16]. One may believe that this model, being linear, has good chances to be fully abstract. This is not the case, for the notion of observables described at the beginning of this section[2]. The reason is that an assume/tell sequence contains information about the *granularity* and the *order* in which constraints are added to the store. Because of the monotonic nature of communication, this granularity and this order cannot be sensed by any context. In other words, it is not possible to define an environment that accepts only a specific sequence of constraints, and blocks otherwise. In fact after any (logically) equivalent sequence the reaction of the environment will be the same.

We show that to achieve full abstraction it is sufficient to *saturate* the denotation S of a process by adding those sequences which are logically equivalent to some sequence in S for any subsequence of constraints produced by the same agent (either the process or the environment). A similar solution was proposed in [5], but the framework we use here (existentially quantified constraints) allows to express the saturation operator in a far more simple and elegant way. However, due to the presence of an explicit nondeterministic choice, the application of this operator is quite delicate in our language, since in the resulting semantics the differences due to silent steps (the actions that do not modify the store) would be deleted. It is well known that this might cause the loss of compositionality with respect to the choice operator. It happens, for instance, in the (weak) bisimulation semantics for CCS, and in the *reactive equivalence* semantics defined in [16]. Essentially this is because the distinction is lost between a computation that deadlocks [fails] immediately, and a computation that deadlocks [fails] after some silent steps. They should be distinguished, since the composition of these computations with a successful alternative choice

[1] The idea of considering arbitrary assumptions (in the framework of substitutions) has been first suggested in [9] for Flat Concurrent Prolog. However, suspension sets, although not necessary, were still present there.

[2] It would be fully abstract if we considered a stronger notion of observables based on sequences of ask/tell actions, similarly to what is done in [17] (for the indeterminate case), but this is out of the scope of this paper.

gives different results (the second may still deadlock [fail] while the first cannot). In our framework it is possible to solve the problem in a quite simple way. We just need to introduce a semantical distinction between two kinds of termination modes, representing the immediate and the not immediate deadlocks [failures]. This solution we propose is, we believe, applicable in general to any language based on asynchronous communication.

The plan of the paper is the following. In the next section we introduce some basic notions. In section 3 we give the syntax of the language, the standard operational model based on a transition system, and we define the notion of observables $\mathcal{O}$. Then we present (section 4) a compositional semantics $\mathcal{M}$ that is shown to be correct (with respect to $\mathcal{O}$), and compositional. In section 5 we present a semantics $\mathcal{A}$ which is a refinement of $\mathcal{M}$. In section 6 we prove that $\mathcal{A}$ is correct and compositional, and in section 7 we prove that $\mathcal{A}$ is fully abstract.

2 Preliminaries

In general, a *constraint system* can be any system of partial information that supports the notion of *consistency* and *entailment*. We consider here simple constraint systems based on first-order theories. Our results, however, can be extended to more general settings.

Let *Var* be a non empty set of *variables*, with typical elements $x, y, z, \ldots$. Let Σ be a (many sorted) first-order alphabet (function symbols $a, b, \ldots, f, g, \ldots$, and predicate symbols, and their signature). A constraint system Γ on (Var, Σ) is a first-order theory. We say that a formula ϕ is *consistent* if $\Gamma \models \exists(\phi)$, where $\exists(\phi)$ is the existential closure of ϕ, and that ϕ_1 *entails* ϕ_2 if $\Gamma \models \phi_1 \Rightarrow \phi_2$. The set of *constraints*, with typical element $c, d, \ldots$, is the set of formulas $\exists X.\vartheta$, where ϑ is a quantifier-free formula and X is a set of variables. $\exists\{x\}.\vartheta$ will be denoted by $\exists x.\vartheta$ and $\exists\emptyset.\vartheta$ by ϑ. The variables in the scope of an existential quantifier are *local*, not visible at the top-level. Usually Γ is assumed to be complete with respect to the notions of concistency and entaiment. In the following we consider Γ to be fixed, so we simply write $\models \exists(c)$ [$\not\models \exists(c)$] and $\models c \Rightarrow d$ [$\not\models c \Rightarrow d$] to represent consistency [inconsistency] and entailment [unentailment].

3 The language

In this section we present a language containing the basic features of concurrent constraint programming. We define its syntax, its (standard) computational model, and the observation criterium we consider.

We use $A, B, \ldots$ to range over the set of processes, $t, u, \ldots$ to range over terms in Σ, $p, q, r, \ldots$ to range over procedure names, and $X, Y, Z, \ldots$ to range over subsets of *Var*. In addition, the notation $\vec{\chi}$ indicates a list of the form $(\chi_1, \ldots, \chi_n)$.

The processes are described by the following grammar

$$A ::= \mathbf{Success} \mid \mathbf{Fail} \mid \mathbf{ask}(c) \rightarrow A \mid \mathbf{tell}(c) \rightarrow A \mid A + A \mid A \parallel A \mid \exists X.A \mid p(\vec{t})$$

Success and **Fail** represent successful and failing termination. **ask** and **tell** are the communication primitives. They are the only actions we consider. The process $\mathbf{ask}(c) \rightarrow A$ waits until the store entails c and then it behaves like A. The process $\mathbf{tell}(c) \rightarrow A$ adds c to the store and then it behaves like A. They both fail when c is inconsistent with the store. The operators $\parallel$ and $+$ are the parallel composition and the nondeterministic choice, respectively. Because of the asynchronism of the communication, the behaviour of $+$ is something in between the (global) nondeterministic choice of CCS and the local nondeterministic choice of TCSP. For instance, $(\mathbf{tell}(c) \rightarrow A) + (\mathbf{tell}(d) \rightarrow B)$ behaves like a local choice when both c and d are consistent with the store, while $(\mathbf{ask}(c) \rightarrow A) + (\mathbf{ask}(d) \rightarrow B)$ behaves like a local choice when both c and d are entailed by it. In the other cases, the choice is global (it depends upon the actions made by the environment). $\exists X.A$ is like the process A, with the variables in X seen as *local*. Finally, $p(\vec{t})$ is a procedure call, p is the name of the procedure, and $\vec{t}$ is the list of the actual parameters. The meaning of a process is given with respect to a set W of procedure declarations of the form $p(\vec{x})$:- A. Given the list of actual parameters $\vec{t}$, an *instantiation* of $p(\vec{x})$:- A is an object of the form $p(\vec{t})$:- A', where A' is obtained from A by simultaneously replacing every (occurrence of a) formal parameter by its corresponding actual parameter, and by renaming all the other variables so to avoid clashes with $\vec{t}$. We denote by $Inst(W)$ the set of the instantiations of the clauses in W. In the sequel, unless stated otherwise, we assume W to be fixed, so we omit reference to it.

3.1 The operational model and the observables $\mathcal{O}$

The operational model is described by a transition system $T = (Conf, \longrightarrow)$. The configurations *Conf* are pairs consisting of a process or a *termination mode*, and a constraint representing the store. The termination modes α are the symbols ss, ff and dd, that denote success, failure and deadlock respectively. The rules of T are described in table 1. For the sake of simplicity, we assume that the variables which are existentially quantified inside a tell [ask] have different names from all the others occurring in the process, or introduced during the computation.

We also assume the presence of a renaming mechanism that takes care of using fresh variables each time an (instance of a) declaration is considered (in **R7**). For the sake of simplicity we do not describe this mechanism in T. In the next section (when describing the compositional semantics) we will show a solution to the problem of renaming. The interested reader can find in [16, 4, 5] various alternative approaches to this problem.

R1 and **R2** indicate that **Success** and **Fail** end immediately in their corresponding termination modes. **R3**-**R6** describe the way in which the communication actions deal with the store. **R7** describes the replacement of a procedure call by the body of the procedure definition (in W). **R8** describes the first step of a process prefixed by a communication action (g stands for **ask** or **tell**). **R9** is the rule for the hiding: a process A in which the variables X are local ($\exists X.A$), first renames the variables in X by fresh variables in Y, and then it behaves like $A[Y/X]$. The renaming is needed in order to avoid clashes with the store and the other variables of A, and it is ensured by the condition $Y \cap var(A, c) = \emptyset$. **R10**-**R14** are the usual rules for the parallel and the choice operators, where the behaviour of a compound process is described in terms of the behaviour of the components. Notice that parallelism is described as interleaving. The main rule for $+$ (**R11**) shows that the indeteminism of our language is sometimes global, sometimes local. In fact, the decision taken by the process will be visible to the external environment only if the store is modified (e.g. if c' is different from c.)

The result of a terminating computation consists of the final store (up to logical equivalence), together with the termination mode. This is formally represented by the notion of *observables*.

Definition 3.1 *The observables are given by the function*

$$\mathcal{O}[\![A]\!] = \{\langle \exists X.c, \alpha\rangle \mid \langle A, true\rangle \longrightarrow^* \langle \alpha, c\rangle\}_{\Leftrightarrow},$$

where X are all the variables in c which do not occur in A, the subscript $\Leftrightarrow$ denotes the closure under logical equivalence, and $\longrightarrow^*$ denotes the transitive closure of $\longrightarrow$.

In the following example, we assume the constraint system to contain the standard equality theory.

Example 3.2 *Consider the process A defined as follows:*

$$A \equiv \mathbf{tell}(x = a) \rightarrow \mathbf{ask}(y = f(a)) \rightarrow \mathbf{Success}$$

Starting with the empty store (true), the process will first tell (add to the store) the constraint $x = a$, and then it will block on the execution of **ask**, *since $x = a$ does not entail $y = f(a)$. Therefore what we observe out of the execution of A is*

$$\mathcal{O}[\![A]\!] = \{\langle x = a, dd\rangle\}_{\Leftrightarrow},$$

Consider now the processes

$$B_1 \equiv \mathbf{tell}(y = f(x)) \rightarrow \mathbf{Success}$$

Table 1: The Transition System T

R1	$\langle \mathbf{Success}, c\rangle \longrightarrow \langle ss, c\rangle$	
R2	$\langle \mathbf{Fail}, c\rangle \longrightarrow \langle ff, c\rangle$	
R3	$\langle \mathbf{ask}(c'), c\rangle \longrightarrow \langle ss, c\rangle$	**if** $\models c \Rightarrow c'$
R4	$\langle \mathbf{ask}(c'), c\rangle \longrightarrow \langle dd, c\rangle$	**if** $\not\models c \Rightarrow c'$
R5	$\langle \mathbf{tell}(c'), c\rangle \longrightarrow \langle ss, c \wedge c'\rangle$	**if** $\models \exists(c \wedge c')$
R6	$\langle \mathbf{tell}(c'), c\rangle \mid \langle \mathbf{ask}(c'), c\rangle \longrightarrow \langle ff, c\rangle$	**if** $\not\models \exists(c \wedge c')$
R7	$\langle p(\vec{t}), c\rangle \longrightarrow \langle A, c\rangle$	**if** $p(\vec{t})$:- $A \in \mathit{Inst}(W)$
R8	$\dfrac{\langle g, c\rangle \longrightarrow \langle ss, c'\rangle \mid \langle \alpha, c\rangle}{\langle g \rightarrow A, c\rangle \longrightarrow \langle A, c'\rangle \mid \langle \alpha, c\rangle}$	**if** $\alpha \in \{ff, dd\}$
R9	$\dfrac{\langle A[Y/X], c\rangle \longrightarrow \langle A', c \wedge c'\rangle}{\langle \exists X.A \longrightarrow \langle A', c \wedge (\exists Y.c')\rangle}$	**if** $Y \cap var(A, c) = \emptyset$
R10	$\dfrac{\langle A, c\rangle \longrightarrow \langle A', c'\rangle \mid \langle ss, c'\rangle}{\begin{array}{c}\langle A \parallel B, c\rangle \longrightarrow \langle A' \parallel B, c'\rangle \mid \langle B, c'\rangle \\ \langle B \parallel A, c\rangle \longrightarrow \langle B \parallel A', c'\rangle \mid \langle B, c'\rangle\end{array}}$	
R11	$\dfrac{\langle A, c\rangle \longrightarrow \langle A', c'\rangle \mid \langle ss, c'\rangle}{\begin{array}{c}\langle A + B, c\rangle \longrightarrow \langle A', c'\rangle \mid \langle ss, c'\rangle \\ \langle B + A, c\rangle \longrightarrow \langle A', c'\rangle \mid \langle ss, c'\rangle\end{array}}$	
R12	$\dfrac{\langle A, c\rangle \longrightarrow \langle dd, c\rangle \quad \langle B, c\rangle \longrightarrow \langle \alpha, c\rangle}{\begin{array}{c}\langle A \parallel B, c\rangle \longrightarrow \langle \alpha, c\rangle \\ \langle B \parallel A, c\rangle \longrightarrow \langle \alpha, c\rangle \\ \langle A + B, c\rangle \longrightarrow \langle dd, c\rangle \\ \langle B + A, c\rangle \longrightarrow \langle dd, c\rangle\end{array}}$	**if** $\alpha \in \{ff, dd\}$
R13	$\dfrac{\langle A, c\rangle \longrightarrow \langle ff, c\rangle}{\begin{array}{c}\langle A \parallel B, c\rangle \longrightarrow \langle ff, c\rangle \\ \langle B \parallel A, c\rangle \longrightarrow \langle ff, c\rangle\end{array}}$	
R14	$\dfrac{\langle A, c\rangle \longrightarrow \langle ff, c\rangle \quad \langle B, c\rangle \longrightarrow \langle ff, c\rangle}{\begin{array}{c}\langle A + B, c\rangle \longrightarrow \langle ff, c\rangle \\ \langle B + A, c\rangle \longrightarrow \langle ff, c\rangle\end{array}}$	

$B_2 \equiv \exists x.\mathbf{tell}(y = f(x)) \rightarrow \mathbf{Success}$

The observables are:

$$\mathcal{O}[\![B_1]\!] = \{\langle y = f(x), ss\rangle\}_{\Leftrightarrow},$$

$$\mathcal{O}[\![B_1]\!] = \{\langle \exists z.y = f(z), ss\rangle\}_{\Leftrightarrow},$$

Let's now consider the processes $A_1 \equiv A \parallel B_1$ and $A_2 \equiv A \parallel B_2$. After the first step of A and the execution of B_1, the store is $x = a \ \wedge \ y = f(x)$, which entails $y = f(a)$. Therefore A_1 can successfully terminate:

$$\mathcal{O}[\![A_1]\!] = \{\langle x = a \ \wedge \ y = f(x), ss\rangle\}_{\Leftrightarrow},$$

This is not the case for A_2, since after the first step of A and the execution of B_2, the store is $x = a \ \wedge \ \exists z.y = f(z)$, which does not entail $y = f(a)$. Therefore A_2 will still deadlock:

$$\mathcal{O}[\![A_2]\!] = \{\langle x = a \ \wedge \ \exists z.y = f(z), dd\rangle\}_{\Leftrightarrow}.$$

4 A compositional semantics based on sequences

We define here a compositional model $\mathcal{M}$ based on sequences of interactions between a process and an arbitrary environment. These interactions are constraints labeled by a (*assume*) or t (*tell*). An *assume* constraint represents an action performed by the environment, while a *tell* constraint represents an action performed by the process itself. We use ℓ to range over $\{\mathrm{a}, t\}$. These sequences encode also the hiding of local variables, by means of existential quantifiers. This will allow to model compositionally the hiding operator, and to express in a more elegant way the saturation procedure so to achieve full abstraction.

A sequence $s = c_1^{\ell_1} \cdot \ldots \cdot c_n^{\ell_n}$ (where each c_i can be of the form $\exists X.c$) has to be interpreted as a conjunction, where the scope of an existential quantifier is the whole subsequence that follows (nesting of local environments). Formally:

Definition 4.1 *We define*

- $Estore(\lambda) = true$ *(λ is the empty sequence)*
- $Estore((\exists X.c)^{\ell} \cdot s) = \exists X(c \wedge Estore(s))$

Furthermore, in order to describe the basic computation steps (in particular, the check for entailment), we need to define the store resulting from a sequence of interactions. This is just the conjunction of all constraints where all the quantifiers are dropped.

Definition 4.2 *We define*

$$\begin{array}{ll} Store(\lambda) & = true \\ Store((\exists X.c)^{\ell} \cdot s) & = c \wedge Store(s) \end{array}$$

For technical convenience we introduce the following notions:

Definition 4.3 *Given a sequence s we define*

- $FV(s)$, *the free variables of s (the global variables),*
- $BV(s)$, *the bound variables of s (the local variables),*
- $BV^{\ell}(s)$ $[FV^{\ell}(s)]$, $\ell \in \{\mathrm{a}, t\}$, *the bound [free] variables of s occurring in constraints labeled by ℓ. The local variables introduced by the process itself are given by $BV\ (s)$ and those introduced by the environment by $BV\ (s)$,*
- $var(\chi)$, *the variables of the object χ (process, sequence ...).*

Table 2: The Transition System T'

R1′	$\langle \mathbf{Success}, s\rangle \longrightarrow \langle ss, s\cdot true \rangle$	
R2′	$\langle \mathbf{Fail}, s\rangle \longrightarrow \langle f\!f, s\rangle$	
R3′	$\langle \mathbf{ask}(c), s\rangle \longrightarrow \langle ss, s\cdot true \rangle$	**if** $\models Store(s) \Rightarrow c$
R4′	$\langle \mathbf{ask}(c), s\rangle \longrightarrow \langle dd, s\rangle$	**if** $\not\models Store(s) \Rightarrow c$
R5′	$\langle \mathbf{tell}(c), s\rangle \longrightarrow \langle ss, s\cdot c' \rangle$	**if** $\models \exists(Store(s\cdot c))$, $\models Store(s\cdot c) \Leftrightarrow Store(s\cdot c')$, and $BV(c') \cap var(s) = \emptyset$
R6′	$\langle \mathbf{tell}(c), s\rangle \mid \langle \mathbf{ask}(c), s\rangle \longrightarrow \langle f\!f, s\rangle$	**if** $\not\models \exists(Store(s\cdot c))$
R7′	$\langle p(\vec{t}), s\rangle \longrightarrow \langle A, s\cdot true \rangle$	**if** $p(\vec{t})$:- $A \in Inst(W)$
R8′	$\dfrac{\langle g, s\rangle \longrightarrow \langle ss, s'\rangle \mid \langle \alpha, s\rangle}{\langle g \rightarrow A, s\rangle \longrightarrow_W \langle A, s'\rangle \mid \langle \alpha, s\rangle}$	**if** $\alpha \in \{f\!f, dd\}$
R9′	$\dfrac{\langle A[Y/X], s\rangle \longrightarrow \langle A', s\cdot c \rangle}{\langle \exists X.A \longrightarrow \langle A', s\cdot(\exists Y.c) \rangle}$	**if** $Y \cap Var(A, s) = \emptyset$
R10′	$\dfrac{\langle A, s\rangle \longrightarrow \langle A', s\cdot c \rangle \mid \langle ss, s\cdot c \rangle}{\begin{array}{c}\langle A \parallel B, s\rangle \longrightarrow \langle A' \parallel B, s\cdot c \rangle \mid \langle B, s\cdot c \rangle \\ \langle B \parallel A, s\rangle \longrightarrow \langle B \parallel A', s\cdot c \rangle \mid \langle B, s\cdot c \rangle\end{array}}$	**if** $BV(c) \cap var(B) = \emptyset$
R11′	$\dfrac{\langle A, s\rangle \longrightarrow \langle A', s\cdot c \rangle \mid \langle ss, s\cdot c \rangle}{\begin{array}{c}\langle A + B, s\rangle \longrightarrow \langle A', s\cdot c \rangle \mid \langle ss, s\cdot c \rangle \\ \langle B + A, s\rangle \longrightarrow \langle A', s\cdot c \rangle \mid \langle ss, s\cdot c \rangle\end{array}}$	
R12′	$\dfrac{\langle A, s\rangle \longrightarrow \langle dd, s\rangle \quad \langle B, s\rangle \longrightarrow \langle \alpha, s\rangle}{\begin{array}{c}\langle A \parallel B, s\rangle \longrightarrow \langle \alpha, s\rangle \\ \langle B \parallel A, s\rangle \longrightarrow \langle \alpha, s\rangle \\ \langle A + B, s\rangle \longrightarrow \langle dd, s\rangle \\ \langle B + A, s\rangle \longrightarrow \langle dd, s\rangle\end{array}}$	**if** $\alpha \in \{f\!f, dd\}$
R13′	$\dfrac{\langle A, s\rangle \longrightarrow \langle f\!f, s\rangle}{\begin{array}{c}\langle A \parallel B, s\rangle \longrightarrow \langle f\!f, s\rangle \\ \langle B \parallel A, s\rangle \longrightarrow \langle f\!f, s\rangle\end{array}}$	
R14′	$\dfrac{\langle A, s\rangle \longrightarrow \langle f\!f, s\rangle \quad \langle B, s\rangle \longrightarrow \langle f\!f, s\rangle}{\begin{array}{c}\langle A + B, s\rangle \longrightarrow \langle f\!f, s\rangle \\ \langle B + A, s\rangle \longrightarrow \langle f\!f, s\rangle\end{array}}$	
R15′	$\langle A, s\rangle \mid \langle ss, s\rangle \longrightarrow \langle A, s\cdot c \rangle \mid \langle ss, s\cdot c \rangle$	**if** $FV(c) \cap BV\ (s) = \emptyset$, $BV(c) \cap var(A, s) = \emptyset$, and $\models \exists(Store(s\cdot c\ ))$

To define the compositional semantics we use a transition system T' (see table 2). The configurations are pairs $\langle A, s\rangle$, such that $Store(s)$ is consistent.

The difference with the transition system T consists mainly in rule **R15**$'$, which models the interaction with the environment. Note that a process A is not immediately affected by actions made by the environment (only its future behaviour will depend on them). An arbitrary constraint can be added (by the environment) consistently to the store without changing the state of A. In other words, A can make an *arbitrary assumption* about the store. However, as the local variables are hidden from the environment, assumptions involving local variables are not allowed. Formally this means that the free variables of an assumption may not occur in the scope of the bound variables introduced by the process itself, i.e., $FV(c) \cap BV(s) = \emptyset$, where c is the assumption about the environment, and s represents the store. The other restriction $BV(c) \cap var(A, s) = \emptyset$ ensures the absence of variable clashes between the local variables of the environment and the variables of the process.

The other rules correspond to the rules of T. In rule **R5**$'$ the possibility of adding to the (sequence representing the) store a constraint c' equivalent to the constraint c to be "told" allows to abstract with respect to the syntactical form of c. Moreover, it allows to rename the variables of c with respect to s, and this will turn out to be convenient to express the saturation procedure. In rule **R10**$'$ the condition $BV(c) \cap var(B) = \emptyset$ allows to avoid clashes between the local variables of A (the ones explicitly hidden by an existential quantifier, and the ones introduced by renaming the clauses of the program) and the variables (local and global) of the other processes (B) in the environment. Notice that this also model a renaming mechanism (for the variables in the declarations) which always uses fresh variables.

The correspondence between T and T' is expressed by the following lemma.

Lemma 4.4 *The rules* **R1**$'$-**R14**$'$ *of* T' *mimic the rules* **R1**-**R14** *of* T*, in the sense that if*

$$\langle A, s\rangle \longrightarrow \langle A', s'\rangle$$

is a **Ri**$'$ *transition step in* T'*, then*

$$\langle A, Store(s)\rangle \longrightarrow \langle A', Store(s')\rangle$$

is a **Ri** *transition step in* T.

Proof Immediate by case analysis of the rules in T'. □

The semantics $\mathcal{M}$, based on the transition system T', delivers sets of sequences of assume/tell constraints, ended by a symbol $\alpha \in \{ss, ff, dd, \perp\}$. The symbol $\perp$ stands for *unfinished*. It is introduced in order to associate a non empty semantics to infinite processes. In fact, even if we are interested only in finite computations, this is necessary for modeling failure compositionally.

$$\begin{array}{rll}
\mathcal{M}[\![A]\!] = & \{s\cdot ss & \mid \langle A, true\rangle \longrightarrow^* \langle ss, s\rangle\} \\
\cup & \{s\cdot ff & \mid \langle A, true\rangle \longrightarrow^* \langle ff, s\rangle\} \\
\cup & \{s\cdot dd & \mid \langle A, true\rangle \longrightarrow^* \langle dd, s\rangle\} \\
\cup & \{s\cdot \perp & \mid \langle A, true\rangle \longrightarrow^* \langle A', s\rangle\}
\end{array}$$

The observables can be obtained from $\mathcal{M}$ as follows. We pick up only the sequences entirely composed by tell constraints and ended by ss, ff, or dd. This amounts to requiring that each constraint we observe has been really produced by the process, and that the computation has reached a final state. Then we abstract from the particular order in which the constraints have been produced, and from the syntactical form of them. This is described by the operator *Result*, defined as follows:

$$Result(S) = \{\langle Estore(s), \alpha\rangle \mid s\cdot\alpha \in S \ \wedge \ s = c_1\cdot\ldots\cdot c_n \ \wedge \ \alpha \in \{ss, ff, dd\}\}.$$

Theorem 4.5 (Correctness of $\mathcal{M}$) $\mathcal{O}[\![A]\!] = Result(\mathcal{M}[\![A]\!])_{/\Leftrightarrow}$

Proof Let A be a process and let $s.\alpha \in \mathcal{M}[\![A]\!]$ such that s contains only tell constraints and $\alpha \neq \perp$. Then there exists in T' a derivation of the form

$$\langle A, \lambda\rangle \;=\; \langle A_0, s_0\rangle \longrightarrow \ldots \langle A_i, s_i\rangle \longrightarrow \ldots \langle A_n, s_n\rangle \;=\; \langle \alpha, s\rangle$$

such that $\alpha \in \{ss, \mathit{ff}, dd\}$ and the rule $\mathbf{R15'}$ is never used.

By lemma 4.4, there exists in T a derivation

$$\langle A, \mathit{true}\rangle \;=\; \langle A_0, \mathit{Store}(s_0)\rangle \longrightarrow \ldots \langle A_i, \mathit{Store}(s_i)\rangle \longrightarrow \ldots \langle A_n, \mathit{Store}(s_n)\rangle \;=\; \langle \alpha, \mathit{Store}(s)\rangle.$$

Therefore $\langle \exists X.\mathit{Store}(s), \alpha\rangle \in \mathcal{O}[\![A]\!]$, where the X is the set of variables in s not occurring in A. The rest follows from the observation that $\exists X.\mathit{Store}(s) = \mathit{Estore}(s)$. □

4.1 Compositionality of $\mathcal{M}$

The operational semantics above defined is compositional with respect to all the operators of the language: $\rightarrow$, $\|$, $+$ and $\exists X$. The corresponding semantic operators $\leadsto$, $\tilde{\|}$, $\tilde{+}$, and $\widetilde{\exists X}$ are defined in below.

To define $\leadsto$ and $\tilde{\|}$, we first define the corresponding partial and nondeterministic operators (still represented by $\leadsto$ and $\tilde{\|}$) on sequences, then we extend them to sets.

tell) Prefixing the action $\mathbf{tell}(c)$ to a process A corresponds to picking up sequences representing computations of A in which c (or an equivalent constraint) is already assumed (initially, i.e. before any tell action), and to replacing the assume mode by the tell mode. A sequence containing an initial constraint inconsistent with c will produce a failure. Sequences of assumptions ended by $\perp$ are not affected. Formally:

let $s = d_1 \cdot \ldots \cdot d_n$.

- $c \leadsto s \cdot c' \cdot s' \cdot \alpha = s \cdot c' \cdot s' \cdot \alpha$ if $\models \mathit{Store}(s \cdot c) \Leftrightarrow \mathit{Store}(s \cdot c')$ and $FV(c') \cap BV(s) = BV(c') \cap FV(s') = \emptyset$
- $c \leadsto s \cdot s' \cdot \alpha = s \cdot \mathit{ff}$ if $\not\models \exists(\mathit{Store}(s \cdot c))$
- $c \leadsto s \cdot \perp = s \cdot \perp$

ask) Prefixing the action $\mathbf{ask}(c)$ to to a process A corresponds to picking up sequences representing computations of A in which c is already assumed and the result is a sequence in which the assumption is replaced by true^t (cfr. $\mathbf{R5'}$). A sequence containing initial assumptions which do not entail c will give a deadlock. The other cases are the same as for **tell**. Formally:

Let $s = d_1 \cdot \ldots \cdot d_n$.

- $c \leadsto s \cdot c \cdot s' \cdot \alpha = s \cdot \mathit{true} \cdot s' \cdot \alpha$ if $\models \mathit{Store}(s) \Rightarrow c$
- $c \leadsto s \cdot s' \cdot \alpha = s \cdot \mathit{ff}$ if $\not\models \mathit{Store}(s \cdot c)$
- $c \leadsto s \cdot s' \cdot \alpha = s \cdot dd$ if $\models \exists(\mathit{Store}(s \cdot c))$ and $\not\models \mathit{Store}(s) \Rightarrow c$
- $c \leadsto s \cdot \perp = s \cdot \perp$

Parallel composition) The operator $\tilde{\|}$, first introduced in [9], allows to combine sequences of assume/tell constraints that are equal at each point, apart from the modes, so modeling the interaction of a process with its environment. It is similar to the (more popular) interleaving operator, the difference is that it applies to sequences containing already all the informations concerning the way in which processes interleave (the assumptions specify "where" and "what"). Hence the application of $\tilde{\|}$ amounts to verify that the assumptions made by one process are indeed validated by the other process (i.e. it tells or assumes the same constraints). In the positive case, the elements of the resulting sequence are labeled by tell whenever they are labeled by tell in at least one of the two sequences, by assume otherwise (a constraint is produced by a pair of parallel processes whenever it is produced by at least one of the two).

- $s_1 \cdot \alpha_1 \tilde{\|} s_2 \cdot \alpha_2 = s_2 \cdot \alpha_1 \tilde{\|} s_1 \cdot \alpha_2$
- $c \ \cdot s_1 \cdot \alpha_1 \tilde{\|} c^{\ell} \cdot s_2 \cdot \alpha_2 = c^{\ell} \cdot (s_1 \cdot \alpha_1 \tilde{\|} s_2 \cdot \alpha_2)$
- $\alpha \tilde{\|} ss = \alpha$
- $\alpha \tilde{\|} f\!f = f\!f$
- $dd \tilde{\|} dd = dd$
- $\perp \tilde{\|} \perp = \perp$

Since arbitrary assumptions can always be made, it is sufficient to consider the cases listed above, and to leave $\rightsquigarrow$ and $\tilde{\|}$ undefined on the sequences of different kind. The extension of these operators to sets is defined in the obvious way:

$$c \ \rightsquigarrow S = \{c \ \rightsquigarrow s \cdot \alpha \mid \ s \cdot \alpha \in S\}$$

$$c \ \rightsquigarrow S = \{c \ \rightsquigarrow s \cdot \alpha \mid \ s \cdot \alpha \in S\}$$

$$S_1 \tilde{\|} S_2 = \{s_1 \cdot \alpha_1 \tilde{\|} s_2 \cdot \alpha_2 \mid \ s_1 \cdot \alpha_1 \in S_1 \ \wedge \ s_2 \cdot \alpha_2 \in S_2\}$$

We now define the semantics operators $\tilde{+}$ and $\widetilde{\exists X}$, corresponding to the choice and the hiding. It is more convenient to define them directly on sets.

Choice) Let S_1 and S_2 be sets of sequences representing the computations of two alternative processes A_1 and A_2. Apart from the cases of deadlock and failure, an alternative choice can always be selected, therefore the successful and unfinished computations of $A_1 + A_2$ are just given by the set union. Concerning failure and deadlock, observe that the successful execution of an action is made visible by telling a constraint. Therefore, when a tell constraint is present, the alternative can be selected (i.e. when the store entails the asked constraints and when it is consistent with the constraints to be told, then the choice is local). We have the set union again. On the other side, sequences consisting of assume constraints and ending in failure and deadlock, are present in the result only if they are present in both sets. Formally:

$$S_1 \tilde{+} S_2 = (S_1 \cup \ \ S_2) \cup (S_1 \cup \ \ S_2) \cup (S_1 \cup \ \ \ S_2) \cup (S_1 \cup_{\perp} S_2)$$

where

$$\begin{array}{lcl}
S_1 \cup \ \ S_2 & = & \{s \cdot ss \mid \ s \cdot ss \in S_1 \cup S_2\} \\
S_1 \cup \ \ S_2 & = & \{s \cdot f\!f \mid \ s \cdot f\!f \in S_1 \cup S_2\}_T \cup \\
 & & \{s \cdot f\!f \mid \ s \cdot f\!f \in S_1 \cap S_2\}_A \\
S_1 \cup \ \ \ S_2 & = & \{s \cdot dd \mid \ s \cdot dd \in S_1 \cup S_2\}_T \cup \\
 & & \{s \cdot dd \mid \ s \cdot dd \in S_1 \cap S_2\}_A \cup \\
 & & \{s \cdot dd \mid \ s \cdot dd \in S_1 \wedge s \cdot f\!f \in S_2\}_A \cup \\
 & & \{s \cdot dd \mid \ s \cdot dd \in S_2 \wedge s \cdot f\!f \in S_1\}_A \cup \\
S_1 \cup_{\perp} S_2 & = & \{s \cdot \perp \mid \ s \cdot \perp \in S_1 \cup S_2\}
\end{array}$$

here S_T denotes the subset of sequences of S in which there occurs a tell constraint, and S_A denotes the subset of sequences of S which consists only of assume constraints.

Hiding) Let $[X/Z, Y/X]$ denote the simultaneous substitution of the free occurrences of a variable of Z by its corresponding variable of X, and a variable of X by its corresponding variable of Y.

- $\widetilde{\exists_A X}.S = \{\exists Y(s[X/Z, Y/X] \cdot \alpha) \mid \ s \cdot \alpha \in S, \ Z \cap var(A) = \emptyset$, and $Y \cap var(s \cdot \alpha) = \emptyset\}$
- For $s = d_1 \cdot \ldots \cdot d_n$ we define $\exists Y(s \cdot c \ \cdot s' \cdot \alpha) = s \cdot \exists Y.c \ \cdot s' \cdot \alpha$ if $Y \cap FV \ (s \cdot s') = \emptyset$

The renaming substitution $[X/Z, Y/X]$ avoids variable clashes between occurrences of the local variables X belonging to different local environments. Some global variables Z of the computation s are interpreted as occurrences of variables of X belonging to a different local environment.

By induction on the length of sequences, and by case analysis, we have following theorem

Theorem 4.6 (Compositionality of $\mathcal{M}$)

$$\begin{array}{lll} \mathcal{M}[\![\mathbf{tell}(c) \to A]\!] = c \rightsquigarrow \mathcal{M}[\![A]\!] & \mathcal{M}[\![A \parallel B]\!] = \mathcal{M}[\![A]\!] \tilde{\parallel} \mathcal{M}[\![B]\!] & \mathcal{M}[\![\exists X.A]\!] = \widetilde{\exists_A X}.\mathcal{M}[\![A]\!] \\ \mathcal{M}[\![\mathbf{ask}(c) \to A]\!] = c \rightsquigarrow \mathcal{M}[\![A]\!] & \mathcal{M}[\![A + B]\!] = \mathcal{M}[\![A]\!] \tilde{+} \mathcal{M}[\![B]\!] & \end{array}$$

Since $\mathcal{M}$ is compositional, the equivalence relation associated to it, defined as

$$A_1 \approx_{\mathcal{M}} A_2 \textit{ iff } \mathcal{M}[\![A_1]\!] = \mathcal{M}[\![A_2]\!]$$

is a congruence:

Corollary 4.7 *If $A_1 \approx_{\mathcal{M}} A_2$ and $B_1 \approx_{\mathcal{M}} B_2$ then*

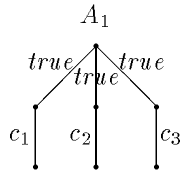
$$\begin{array}{lll} \mathbf{tell}(c) \to A_1 \approx_{\mathcal{M}} \mathbf{tell}(c) \to A_2 & A_1 \parallel B_1 \approx_{\mathcal{M}} A_2 \parallel B_2 & \exists X.A_1 \approx_{\mathcal{M}} \exists X.A_2 \\ \mathbf{ask}(c) \to A_1 \approx_{\mathcal{M}} \mathbf{ask}(c) \to A_2 & A_1 + B_1 \approx_{\mathcal{M}} A_2 + B_2 & \end{array}$$

Proof Standard. □

The congruence above defined is not able to identify processes having the same final results (observables), but it is large enough to capture equivalence of processes that react (stepwise) in the same way to any environment (taking silent steps[3] into account). This notion of equivalence, in concurrent constraint languages, has different properties than the corresponding notion in CCS (modeled by strong bisimulation).

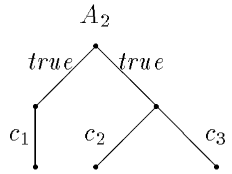
Example 4.8 *Let A_1 and A_2 be defined as follows:*

$$\begin{array}{rl} A_1 \equiv & (\mathbf{tell}(\mathit{true}) \to \mathbf{tell}(c_1) \to \mathbf{Success}) \\ & + \\ & (\mathbf{tell}(\mathit{true}) \to \mathbf{tell}(c_2) \to \mathbf{Success}) \\ & + \\ & (\mathbf{tell}(\mathit{true}) \to \mathbf{tell}(c_3) \to \mathbf{Success}) \end{array}$$



and

$$\begin{array}{rl} A_2 \equiv & (\mathbf{tell}(\mathit{true}) \to \mathbf{tell}(c_1) \to \mathbf{Success}) \\ & + \\ & (\mathbf{tell}(\mathit{true}) \to (\ (\mathbf{tell}(c_2) \to \mathbf{Success}) \\ & \qquad\qquad + \\ & \qquad\qquad (\mathbf{tell}(c_3) \to \mathbf{Success})\ ) \end{array}$$



Assume c_1, c_2 and c_3 to be pairwise inconsistent. The behaviour of A_1 and A_2 with respect to an arbitrary environment will be exactly the same, at each step, and indeed $A_1 \approx_{\mathcal{M}} A_2$. In CCS, where communication is synchronous, these processes can be distinguished by an environment B that accepts (at the second step) c_1 or c_2 (only). In such a situation, $A_1 \parallel B$ can fail (it happens when A_1 follows the third branch), while $A_2 \parallel B$ cannot. This discrimination power of CCS environments, based on the ability to "take decisions together", is reflected by strong bisimulation. A_1 and A_2 indeed are not strongly bisimilar (they are not even bisimilar). On the contrary, in concurrent constraint languages, where decisions are taken asynchronously, either the process or the environment chooses without taking the other into account. Therefore, also A_2 can bring to a failure by choosing c_3.

[3] In this framework, a step is silent if does not modify the store (like, for instance, $\mathbf{tell}(\mathit{true})$).

As a conclusion, $\approx_{\mathcal{M}}$ is coarser than the congruence based on strong bisimulation (*reactive equality*, [16]), and, we think, more appropriate for concurrent constraint.

However, one may argue that this notion of equivalence is too fine and that it would be more interesting, for instance, to abstract from silent steps or, even, from intermediate steps (i.e. to identify processes that give the same final results).

In the next section we introduce a semantics $\mathcal{A}$ that is

- compositional,
- correct, i.e. it does not identify processes which have different observables
- fully abstract, i.e. it identifies processes that give the same observables in any context.

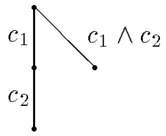
5 A fully abstract semantics

The operational semantics defined in the previous section is not fully abstract with respect to the observables. Like in CCS, silent steps cause unnecessary distinctions, but this is only part of the the reason. More generally, the problem is that sequences contain redundant information about the order and the granularity in which constraints are added to the store.

Example 5.1 *Let B_1 and B_2 be defined as follows:*

$$
\begin{aligned}
A_1 \equiv\ & (\mathbf{tell}(c_1) \rightarrow \mathbf{tell}(c_2) \rightarrow \mathbf{Success}) \\
& + \\
& (\mathbf{tell}(c_1 \wedge c_2) \rightarrow \mathbf{Success})
\end{aligned}
$$

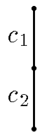
B_1



and

$$A_2 \equiv\ \mathbf{tell}(c_1) \rightarrow \mathbf{tell}(c_2) \rightarrow \mathbf{Success}$$

B_2



Of course, $B_1 \not\approx_{\mathcal{M}} B_2$, but still, in every context, they will produce the same final results. This is because of the monotonicity of communication: we cannot define an environment that accepts both c_1 and c_2 and not $c_1 \wedge c_2$.

In general, the reaction of a process to a certain sequence of actions will only depend upon the (logical) final content of the store. Therefore, we must eliminate distinctions between logically equivalent subsequences. One way to do this is to *saturate* the sets of denotations by adding, for any sequence s, all the ones that differ only for a logically equivalent subsequence (**C1**). This operation, however, is not enough. In $\mathcal{M}$ a sequence can occur only together with all the possible interleavings with arbitrary assumptions. Therefore, also for the new sequences introduced by **C1** we must add all possible interleavings (**C2**).

Definition 5.2 *Given a set of sequences S, the saturation of S, $Sat(S)$, is the minimal set which contain S and satisfies the conditions* **C1** *and* **C2** *of table 3.*

Sometime it will be convenient to apply the saturation operator also on sets of sequences s (without termination mode), with the obvious meaning.

Remark 5.3 *The conditions* **C1** *and* **C2** *preserve the meaning of a sequence. Namely, if $s.\alpha \in Sat(\{s'.\alpha'\})$ (abbrev. $Sat(s'.\alpha')$), then $\models Estore(s) \Leftrightarrow Estore(s')$ holds.*

Table 3: The saturation conditions

C1	if $s_1 \cdot c_1^\ell \ldots c_n^\ell \cdot s_2 \cdot \alpha \in S$ and $\models Estore(s_1 \cdot c_1^\ell \ldots c_n^\ell \cdot s_2) \Leftrightarrow Estore(s_1 \cdot d_1^\ell \ldots d_m^\ell \cdot s_2)$ then $s_1 \cdot d_1^\ell \ldots d_m^\ell \cdot s_2 \cdot \alpha \in S$
C2	if $s_1 \cdot s_2 \cdot \alpha \in S$ and $\models Estore(s_1 \cdot s_2) \Leftrightarrow Estore(s_1 \cdot c \cdot s_2)$ then $s_1 \cdot c\ \cdot s_2 \cdot \alpha \in S$

Remark 5.4 *The saturation operator is idempotent, namely*

$Sat(Sat(S)) = Sat(S).$

We want to define the fully abstract semantics by applying the saturation operator to the semantics $\mathcal{M}$. However, as the closure under the conditions **C1** and **C2** abstracts from silent steps we loose the compositionality with respect to the choice operator. Our solution to this problem consists of introducing the termination modes ff_ι, dd_ι and $\perp_\iota$. The termination modes ff_ι and dd_ι are intended to describe computations in which the process *immediately* fails and suspends respectively. A sequence ending in $\perp_\iota$ will correspond to an unfinished computation in which the process has not performed any step. We introduce the new termination modes by the following modification of the operational semantics $\mathcal{M}$:

$$\begin{array}{rll}
\mathcal{M}_\iota[\![A]\!] = & \{s \cdot ss & \mid \langle A, true\rangle \longrightarrow^* \langle ss, s\rangle\} \\
\cup & \{s \cdot ff & \mid \langle A, true\rangle \longrightarrow^* \langle ff, s\rangle\}_T \\
\cup & \{s \cdot ff_\iota & \mid \langle A, true\rangle \longrightarrow^* \langle ff, s\rangle\}_A \\
\cup & \{s \cdot dd & \mid \langle A, true\rangle \longrightarrow^* \langle dd, s\rangle\}_T \\
\cup & \{s \cdot dd_\iota & \mid \langle A, true\rangle \longrightarrow^* \langle dd, s\rangle\}_A \\
\cup & \{s \cdot \perp & \mid \langle A, true\rangle \longrightarrow^* \langle A', s\rangle\}_T \\
\cup & \{s \cdot \perp_\iota & \mid \langle A, true\rangle \longrightarrow^* \langle A', s\rangle\}_A
\end{array}$$

The semantics $\mathcal{A}$ is defined as follows:

$\mathcal{A}[\![A]\!] = Sat(\mathcal{M}_\iota[\![A]\!])$

In the following two sections we prove the correctness, the compositionality, and the full abstraction of $\mathcal{A}$.

6 The correctness and compositionality of $\mathcal{A}$

The equivalence defined by $\mathcal{A}$ is still stronger than the equivalence induced by $\mathcal{O}$. In fact, the observables can be derived from $\mathcal{A}$, by application of the operator *Result*.

Lemma 6.1 $Result(\mathcal{M}_\iota[\![A]\!]) = Result(\mathcal{M}[\![A]\!])$

Proof The operator *Result* picks up only the sequences with constraints annotated by tell, and for these sequences the definitions of $\mathcal{M}_\iota$ and $\mathcal{M}$ coincide. □

Theorem 6.2 (Correctness of $\mathcal{A}$) $Result(\mathcal{A}[\![A]\!]) = \mathcal{O}[\![A]\!]$

Proof Immediate by lemma 6.1 and remark 5.3. □

Now we consider the compositionality of $\mathcal{A}$. To this purpose, we define the new semantics operators $\rightsquigarrow_\iota$, $\tilde{\|}_\iota$, and $\tilde{+}_\iota$.

The operator $\rightsquigarrow_\iota$ on sequences is a slight modification of the operator $\rightsquigarrow$. It additionally transforms the termination modes ff_ι, dd_ι and $\perp_\iota$ into ff, dd and $\perp$, respectively, and it introduces immediate failure and deadlock.

let $s = d_1 \cdot \ldots \cdot d_n$.

- $c \ \rightsquigarrow_\iota s{\cdot}c' \ \cdot s'{\cdot}\alpha_\iota = s{\cdot}c' \ \cdot s'{\cdot}\alpha$ if $\models Store(s{\cdot}c \ ) \Leftrightarrow Store(s{\cdot}c' \ )$ and
  $FV(c') \cap BV(s) = BV(c') \cap FV(s') = \emptyset$
  for $\alpha_\iota = ss, ff_\iota, dd_\iota, \bot_\iota,$ *and* $\alpha = ss, ff, dd, \bot$
- $c \ \rightsquigarrow_\iota s{\cdot}s'{\cdot}\alpha = s{\cdot}ff_\iota$ if $\not\models \exists(Store(s{\cdot}c \ ))$
- $c \ \rightsquigarrow_\iota s{\cdot}\bot_\iota = s{\cdot}\bot_\iota$
- $c \ \rightsquigarrow_\iota s{\cdot}c \ \cdot s'{\cdot}\alpha = s{\cdot}true \ \cdot s'{\cdot}\alpha$ *if* $\models Store(s) \Rightarrow c$
  for $\alpha_\iota = ss, ff_\iota, dd_\iota, \bot_\iota,$ *and* $\alpha = ss, ff, dd, \bot$
- $c \ \rightsquigarrow_\iota s{\cdot}s'{\cdot}\alpha = s{\cdot}ff_\iota$ if $\not\models Store(s{\cdot}c \ )$
- $c \ \rightsquigarrow_\iota s{\cdot}s'{\cdot}\alpha = s{\cdot}dd_\iota$ if $\models \exists(Store(s{\cdot}c \ ))$ and $\not\models Store(s) \Rightarrow c$
- $c \ \rightsquigarrow_\iota s{\cdot}\bot_\iota = s{\cdot}\bot_\iota$

The parallel operator $\tilde{\|}_\iota$ is defined as the operator $\tilde{\|}$ plus the following:

- $ff_\iota \tilde{\|}_\iota \alpha = ff_\iota,\ \alpha \in \{ff_\iota, dd_\iota, \bot_\iota\}$
- $ff_\iota \tilde{\|}_\iota \alpha = ff,\ \alpha \in \{ss, ff, dd, \bot\}$
- $dd_\iota \tilde{\|}_\iota dd_\iota = dd_\iota$
- $dd_\iota \tilde{\|}_\iota \alpha = dd \tilde{\|} \alpha,\ \alpha \in \{ss, ff, dd\}$
- $\bot_\iota \ \tilde{\|}_\iota \ \bot_\iota = \bot_\iota$
- $\bot_\iota \ \tilde{\|}_\iota \ \bot = \bot$

The extension on sets of $\rightsquigarrow_\iota$ and $\tilde{\|}_\iota$ are defined as usual.

The new operator $\tilde{+}_\iota$ is defined as follows:

$$S_1 \tilde{+}_\iota S_2 = (S_1 \cup \ \ S_2) \cup (S_1 \cup_{\ \iota} S_2) \cup (S_1 \cup_{\ \iota} S_2) \cup (S_1 \cup_{\bot_\iota} S_2)$$

where $S_1 \cup \ \ S_2$ is the same as before, and

$$\begin{array}{lcl} S_1 \cup_{\ \iota} S_2 & = & \{s{\cdot}ff \mid s{\cdot}ff \in S_1 \cup S_2\} \cup \{s{\cdot}ff_\iota \mid s{\cdot}ff_\iota \in S_1 \cap S_2\} \\ S_1 \cup_{\ \iota} S_2 & = & \{s{\cdot}dd \mid s{\cdot}dd \in S_1 \cup S_2\} \cup \{s{\cdot}dd_\iota \mid s{\cdot}dd_\iota \in S_1 \cap S_2\} \cup \\ & & \{s{\cdot}dd_\iota \mid s{\cdot}dd_\iota \in S_1 \wedge s{\cdot}ff_\iota \in S_2\} \cup \{s{\cdot}dd_\iota \mid s{\cdot}dd_\iota \in S_2 \wedge s{\cdot}ff_\iota \in S_1\} \\ S_1 \cup_{\bot_\iota} S_2 & = & \{s{\cdot}\bot \mid s{\cdot}\bot \in S_1 \cup S_2\} \cup \{s{\cdot}\bot_\iota \mid s{\cdot}\bot_\iota \in S_1 \cap S_2\} \end{array}$$

By induction on the length of the sequences, and by case analysis we have the following

Lemma 6.3 (Compositionality of $\mathcal{M}_\iota$)

$$\begin{array}{ll} \mathcal{M}_\iota[\![\mathbf{tell}(c) \to A]\!] = c \ \rightsquigarrow_\iota \mathcal{M}_\iota[\![A]\!] & \mathcal{M}_\iota[\![A \parallel B]\!] = \mathcal{M}_\iota[\![A]\!] \tilde{\|}_\iota \mathcal{M}_\iota[\![B]\!] \\ \mathcal{M}_\iota[\![\mathbf{ask}(c) \to A]\!] = c \ \rightsquigarrow_\iota \mathcal{M}_\iota[\![A]\!] & \mathcal{M}_\iota[\![\exists X.A]\!] = \widetilde{\exists_A X}.\mathcal{M}_\iota[\![A]\!] \\ \mathcal{M}_\iota[\![A + B]\!] = \mathcal{M}_\iota[\![A]\!] \tilde{+}_\iota \mathcal{M}_\iota[\![B]\!] & \end{array}$$

We show now the compositionality of $\mathcal{A}$.

Theorem 6.4 (Compositionality of $\mathcal{A}$)

$$\begin{array}{ll} \mathcal{A}[\![\mathbf{tell}(c) \to A]\!] = Sat(c \ \rightsquigarrow_\iota \mathcal{A}[\![A]\!]) & \mathcal{A}[\![A \parallel B]\!] = Sat(\mathcal{A}[\![A]\!] \tilde{\|}_\iota \mathcal{A}[\![B]\!]) \\ \mathcal{A}[\![\mathbf{ask}(c) \to A]\!] = Sat(c \ \rightsquigarrow_\iota \mathcal{A}[\![A]\!]) & \mathcal{A}[\![\exists X.A]\!] = Sat(\widetilde{\exists_A X}.\mathcal{A}[\![A]\!]) \\ \mathcal{A}[\![A + B]\!] = Sat(\mathcal{A}[\![A]\!] \tilde{+}_\iota \mathcal{A}[\![B]\!]) & \end{array}$$

Proof We prove these results separately by the following lemmata.

Lemma 6.5 $\mathcal{A}[\![\mathbf{tell}(c) \to A]\!] = Sat(c \ \rightsquigarrow_\iota \mathcal{A}[\![A]\!])$.

Proof By definition of $\mathcal{A}$ and the compositionality of $\mathcal{M}_\iota$ it is sufficient to prove $Sat(c \rightsquigarrow_\iota \mathcal{M}_\iota[\![A]\!]) = Sat(c \rightsquigarrow_\iota Sat(\mathcal{M}_\iota[\![A]\!]))$.

$\subseteq$) Trivial.

$\supseteq$) We show that $c \rightsquigarrow_\iota Sat(\mathcal{M}_\iota[\![A]\!]) \subseteq Sat(c \rightsquigarrow_\iota \mathcal{M}_\iota[\![A]\!])$. We treat the following case: Let $s = s_1 \cdot c' \cdot s_2 \cdot \alpha \in c \rightsquigarrow_\iota Sat(\mathcal{M}_\iota[\![A]\!])$, with $s_1 = d_1 \cdot \ldots \cdot d_n$, such that $\models Estore(s_1 \cdot c\ ) \Leftrightarrow Estore(s_1 \cdot c'\ )$ and $s_1 \cdot c' \cdot s_2 \cdot \alpha \in Sat(s' \cdot \alpha)$, where $s' \cdot \alpha \in \mathcal{M}_\iota[\![A]\!]$. It is not difficult to see that $s_1 \cdot c' \cdot s' \cdot \alpha \in \mathcal{M}_\iota[\![A]\!]$. So we have $s_1 \cdot c' \cdot s' \cdot \alpha \in c \rightsquigarrow_\iota \mathcal{M}_\iota[\![A]\!]$. As $s_1 \cdot c' \cdot s_2 \cdot \alpha \in Sat(s')$ we thus can infer that $s_1 \cdot c' \cdot s_1 \cdot c' \cdot s_2 \cdot \alpha \in Sat(c \rightsquigarrow_\iota \mathcal{M}_\iota[\![A]\!])$, from which we derive by **C1** that $s = s_1 \cdot c' \cdot s_2 \cdot \alpha \in Sat(c \rightsquigarrow_\iota \mathcal{M}_\iota[\![A]\!])$.

The case $s = s_1 \cdot f\!f_\iota$, with $s_1 = d_1 \cdot \ldots \cdot d_n$, such that $s_1 \cdot s_2 \in Sat(s')$ and $\not\models Estore(s_1 \cdot c\ )$, where $s' \in \mathcal{M}_\iota[\![A]\!]$, is treated similar. □

Lemma 6.6 $\mathcal{A}[\![\mathbf{ask}(c) \rightarrow A]\!] = Sat(c \rightsquigarrow_\iota \mathcal{A}[\![A]\!])$.

Proof Similar to the proof of the previous lemma. □

Lemma 6.7 $\mathcal{A}[\![A + B]\!] = Sat(\mathcal{A}[\![A]\!] \tilde{+}_\iota \mathcal{A}[\![B]\!])$.

Proof By the definition of $\mathcal{A}$ and the compositionality of $\mathcal{M}_\iota$ it suffices to prove that $Sat(Sat(\mathcal{M}_\iota[\![A]\!]) \tilde{+}_\iota Sat(\mathcal{M}_\iota[\![B]\!])) = Sat(\mathcal{M}_\iota[\![A]\!] \tilde{+}_\iota \mathcal{M}_\iota[\![B]\!])$, the proof of which is straightforward. □

Lemma 6.8 $\mathcal{A}[\![\exists X.A]\!] = Sat(\widetilde{\exists_A X}.\mathcal{A}[\![A]\!])$.

Proof By the definition of $\mathcal{A}$ it suffices to prove $Sat(\widetilde{\exists_A X}.Sat(\mathcal{M}_\iota[\![A]\!])) = Sat(\widetilde{\exists_A X}.\mathcal{M}_\iota[\![A]\!])$. Let $s' \in Sat(s)$, with $s \in \mathcal{M}_\iota[\![A]\!]$. It follows that $s'[X/Z, Y/X] \in Sat(s[X/Z, Y/X])$ ($Z \cap var(A) = \emptyset$). Furthermore, it is not difficult to see that $s'[X/Z, Y/X] \in Sat(s[X/Z, Y/X])$ implies $\exists Y(s'[X/Z, Y/X]) \in Sat(\exists Y(s[X/Z, Y/X]))$. So we have $\widetilde{\exists_A X}.Sat(\mathcal{M}_\iota[\![A]\!]) \subseteq Sat(\widetilde{\exists_A X}.\mathcal{M}_\iota[\![A]\!])$, and thus $Sat(\widetilde{\exists_A X}.Sat(\mathcal{M}_\iota[\![A]\!])) \subseteq Sat(\widetilde{\exists_A X}.\mathcal{M}_\iota[\![A]\!])$. The other inclusion is immediate. □

Lemma 6.9 $\mathcal{A}[\![A \parallel B]\!] = Sat(\mathcal{A}[\![A]\!] \tilde{\parallel}_\iota \mathcal{A}[\![B]\!])$.

Proof By definition of $\mathcal{A}$ and the compositionality of $\mathcal{M}_\iota$ it suffices to prove

$$Sat(Sat(\mathcal{M}_\iota[\![A]\!]) \tilde{\parallel}_\iota Sat(\mathcal{M}_\iota[\![B]\!])) = Sat(\mathcal{M}_\iota[\![A]\!] \tilde{\parallel}_\iota \mathcal{M}_\iota[\![B]\!])$$

To prove this it is sufficient to show that

$$Sat(\mathcal{M}_\iota[\![A]\!]) \tilde{\parallel}_\iota Sat(\mathcal{M}_\iota[\![B]\!]) \subseteq Sat(\mathcal{M}_\iota[\![A]\!]) \tilde{\parallel}_\iota \mathcal{M}_\iota[\![B]\!])$$

Let $s_1 \alpha_1 \in \mathcal{M}_\iota[\![A]\!]$ and $s_2 \alpha_2 \in \mathcal{M}_\iota[\![B]\!]$. Furthermore let $s'_1 \alpha_1 \in Sat(s_1 \alpha_1)$ and $s'_2 \alpha_2 \in Sat(s_2 \alpha_2)$ such that $s'_1 \cdot \alpha_1 \tilde{\parallel}_\iota s'_2 \cdot \alpha_2$ is defined. We prove by induction on the number of applications of **C1-O**, the saturation condition **C1** for $\ell = O$, that $s'_1 \cdot \alpha_1 \tilde{\parallel}_\iota s'_2 \cdot \alpha_2 \in Sat(\mathcal{M}_\iota[\![A]\!] \tilde{\parallel}_\iota \mathcal{M}_\iota[\![B]\!])$. Note that this indeed suffices because $\mathcal{M}_\iota$ is closed under **C2** and the following version of **C1**

C1-I $s_1 \cdot c_1 \cdot \ldots \cdot c_n \cdot s_2 \cdot \alpha \in S \Rightarrow s_1 \cdot c'_1 \cdot \ldots \cdot c'_m \cdot s_2 \cdot \alpha \in S$
if $\models Estore(s_1 \cdot c_1 \cdot \ldots \cdot c_n \cdot s_2) \Leftrightarrow Estore(s_1 \cdot c'_1 \cdot \ldots \cdot c'_m \cdot s_2)$

It is not difficult to see that we may without loss of generality restrict ourselves to modular sequences, namely, to those sequences s such that for every two distinct constraints c and c' of s we have $BV(c) \cap FV(c) = \emptyset$. (Modularize the derivations of s'_1 and s'_2 from s_1, s_2, respectively, prove that $\bar{s}'_1 \cdot \alpha_1 \tilde{\parallel}_\iota \bar{s}'_2 \cdot \alpha_2 \in Sat(\mathcal{M}_\iota[\![A]\!] \tilde{\parallel}_\iota \mathcal{M}[\![B]\!])$, where $\bar{s}'_1$ and $\bar{s}'_2$ are the modularizations of s_1 and s_2. Then, as $s_1 \cdot \alpha_1 \tilde{\parallel}_\iota s_2 \cdot \alpha_2 \in Sat(\bar{s}'_1 \cdot \alpha_1 \tilde{\parallel}_\iota \bar{s}'_2 \cdot \alpha_2)$, we are done).

The case that the number of applications of **C1-O** equals zero follows immediately from the closedness of $\mathcal{M}_\iota$ under **C1-I** and **C2**.

For the sake of a smooth presentation let's introduce the notation $s \Rightarrow s'$ for the derivability of s' from s by one application of an arbitrary saturation condition.

Now let

$$s_1 \Rightarrow^* s_{11} \cdot c_1 \cdot \ldots \cdot c_n \cdot s_{12} \Rightarrow s_{11} \cdot c'_1 \cdot \ldots \cdot c'_m \cdot s_{12} \Rightarrow^* s'_1$$

where $s_{11} c'_1 \ldots c'_m \cdot s_{12} \Rightarrow^* s'_1$ consists only of applications of **C1-I** and **C2**. We have

$$s'_1 = s'_{11} \cdot c'_1 \cdot d_1 \cdot c'_2 \ldots d_{m-1} \cdot c'_m \cdot s'_{12}$$

where the input constraints are introduced by **C2**, $s_{11} \Rightarrow^* s'_{11}$ and $s_{12} \Rightarrow^* s'_{12}$, both derivations using only **C1-I** and **C2**. (The slightly more general case that some of the d_i's introduced are actually sequences of input constraints can be treated in the same way, but requires a more elaborate notation which might obscure the underlying idea.)

So we have

$$s'_2 = s'_{21} \cdot c'_1 \cdot d_1^{\ell_1} \cdot c'_2 \cdot \ldots \cdot d_{m-1}^{\ell_{m-1}} \cdot c'_m \cdot s'_{22}$$

such that $s'_{11} \tilde{\|} s'_{21}$ and $s'_{12} \tilde{\|} s'_{22}$ are defined.

Define

$$r_1 = s'_{11} \cdot c_1 \cdot \ldots \cdot c_n \cdot d_1 \ldots d_{m-1} \cdot s'_{12}$$

It is not difficult to see that $s_1 \Rightarrow^* r_1$, where the number of applications of **C1-O** is one less than that in $s_1 \Rightarrow^* s'_1$: $s_1 \Rightarrow^* s_{11} \cdot c_1 \cdot \ldots \cdot c_n \cdot s_{12} \Rightarrow^* s'_{11} \cdot c_1 \cdot \ldots \cdot c_n \cdot s'_{12} \Rightarrow^* s'_{11} \cdot c_1 \cdot \ldots \cdot c_m \cdot d_1 \cdot \ldots \cdot d_{m-1} \cdot s'_{12}$.

Furthermore let

$$r_2 = s'_{21} \cdot c_1 \cdot \ldots \cdot c_n \cdot d_1^{\ell_1} \cdot \ldots \cdot d_{m-1}^{\ell_{m-1}} \cdot s'_{22}$$

Again it is not difficult to check that $s_2 \Rightarrow^* s'_2 \Rightarrow^* r_2$, with the same number of applications of **C1-O** as in the derivation $s_2 \Rightarrow^* s'_2$.

So we are now in a position which allows us to apply the induction hypothesis: $r_1 \cdot \alpha_1 \tilde{\|}_\iota r_2 \cdot \alpha_2 \in Sat(\mathcal{M}_\iota[\![A]\!] \tilde{\|}_\iota \mathcal{M}_\iota[\![B]\!])$.

Finally, we have $r_1 \cdot \alpha_1 \tilde{\|}_\iota r_2 \cdot \alpha_2 \Rightarrow^* s'_1 \cdot \alpha_1 \tilde{\|}_\iota s'_2 \cdot \alpha_2$, which we leave the reader to verify. □

□

By theorem 6.4 we have that the equivalence relation associated to $\mathcal{A}$, defined as

$$A_1 \approx_{\mathcal{A}} A_2 \textit{ iff } \mathcal{A}[\![A_1]\!] = \mathcal{A}[\![A_2]\!]$$

is a congruence:

Corollary 6.10 *If $A_1 \approx_{\mathcal{A}} A_2$ and $B_1 \approx_{\mathcal{A}} B_2$ then*

$$\begin{array}{ll} \mathbf{tell}(c) \rightarrow A_1 \approx_{\mathcal{A}} \mathbf{tell}(c) \rightarrow A_2 & A_1 \parallel B_1 \approx_{\mathcal{A}} A_2 \parallel B_2 \\ \mathbf{ask}(c) \rightarrow A_1 \approx_{\mathcal{A}} \mathbf{ask}(c) \rightarrow A_2 & \exists X.A_1 \approx_{\mathcal{A}} \exists X.A_2 \\ A_1 + B_1 \approx_{\mathcal{A}} A_2 + B_2 & \end{array}$$

Proof Standard. □

Processes A_1 and A_2 of example 4.8 and processes B_1 and B_2 of example 5.1 are equivalent ($A_1 \approx_{\mathcal{A}} A_2$ and $B_1 \approx_{\mathcal{A}} B_2$). In the following section we prove that $\approx_{\mathcal{A}}$ is the coarsest equivalence that preserve the five operators described above, and that is correct w.r.t. the observables.

7 The full abstraction of $\mathcal{A}$

In this section we prove the full abstraction of $\mathcal{A}$ with respect to our observation criterium. The basic lines of the proof are the following. Given two processes A_1, A_2 with a different semantics $\mathcal{A}$, we build a context that is able to "detect" this difference at the observational level.

An important simplifying observation is that we can restrict ourselves to *modular* sequences. (A sequence s is modular if for every two distinct constraints c and c' of s we have $BV(c) \cap FV(c) = \emptyset$.) Note that the constraint associated with a modular sequence s can be obtained by simply taking the conjunction of the constraints of s. Now if $\mathcal{A}[\![A_1]\!] \neq \mathcal{A}[\![A_2]\!]$ then there exists a modular sequence s, such that, say, $s \cdot \alpha \in \mathcal{A}[\![A_1]\!] \setminus \mathcal{A}[\![A_2]\!]$. This can be seen as follows: Let $s' \cdot \alpha \in \mathcal{A}[\![A_1]\!] \setminus \mathcal{A}[\![A_2]\!]$. Let s be obtained from s' by replacing every constraint c of $s' = s_1 \cdot c^{\ell} \cdot s_2$ by a constraint equivalent to $Estore(s_1 \cdot c^{\ell})$. It follows that s is modular and furthermore that $s \cdot \alpha \in Sat(s' \cdot \alpha)$ by only applications of **C1**, so we have also that $s' \cdot \alpha \in Sat(s \cdot \alpha)$. It follows that $s \cdot \alpha \in \mathcal{A}[\![A_1]\!] \setminus \mathcal{A}[\![A_2]\!]$.

To build up the distinguishing context we will follow different strategies, depending upon the kind of termination mode (immediate or not immediate).

For the sequences ending with the not immediate termination modes ($ss, f\!f, dd$ and $\perp$) the definition of the distinguishing context is *uniform*, in the following sense: given a modular sequence $s \cdot \alpha$ we define a *context* $C(s \cdot \alpha) \parallel [\,]$ such that the process $C(s \cdot \alpha)$ "recognizes" $s \cdot \alpha$. Next we prove that every other sequence $s' \cdot \alpha$ recognized by $C(s \cdot \alpha)$ that gives the same result must generate $s \cdot \alpha$ by application of the saturation operator. Then we reason by contradiction: given a sequence $s \cdot \alpha$ in the semantic difference (say, $s \cdot \alpha \in \mathcal{A}[\![A_1]\!] \setminus \mathcal{A}[\![A_2]\!]$), if the process $C(s \cdot \alpha)$ doesn't induce a difference in the observables, then there exists an other sequence $s' \cdot \alpha$ ($s' \cdot \alpha \in \mathcal{A}[\![A_2]\!]$) recognized by the process that produces the same result. But, since $\mathcal{A}$ is closed, the presence of $s'.\alpha$ implies the presence of $s \cdot \alpha$, and this contradicts the assumption.

Definition 7.1 *Let s be a modular sequence. We define the process $C(s \cdot \alpha)$ by induction on the length of s.*

- $C(ss) = C(f\!f) = C(dd) = \textbf{Success}$, *and* $C(\perp) = \textbf{Fail}$,
- $C(c \;\cdot s \cdot \alpha) = \textbf{ask}(c) \to C(s \cdot \alpha)$,
- $C(c \;\cdot s \cdot \alpha) = \textbf{tell}(c) \to C(s \cdot \alpha)$.

The following proposition states that a process $C(s \cdot \alpha)$ recognizes the sequence $s \cdot \alpha$. Namely, $C(s \cdot \alpha)$ generates $\bar{s} \cdot \bar{\alpha}$, where $\bar{s}$ denotes the "mirror" of s, i.e. $\overline{c \;\cdot s} = c \;\cdot \bar{s}$ and $\overline{c \;\cdot s} = c \;\cdot \bar{s}$, and

$$\bar{\alpha} = \begin{cases} ss & \textit{if } \alpha \in \{ss, f\!f, dd\} \\ f\!f & \textit{otherwise.} \end{cases}$$

Proposition 7.2 *For any modular sequence $s \cdot \alpha$ we have $\bar{s} \cdot \bar{\alpha} \in \mathcal{A}[\![C(s \cdot \alpha)]\!]$.*

Proof Let s' be the sequence obtained from $\bar{s}$ by adding after every assume constraint of $\bar{s}$ *true* . It is straightforward to prove that $s' \cdot \bar{\alpha} \in \mathcal{M}[\![C(s \cdot \alpha)]\!]$. Furthermore, we have by applications of **C1** that $\bar{s} \in Sat(s)$. So by definition of $\mathcal{A}$ we conclude that $\bar{s} \cdot \bar{\alpha} \in \mathcal{A}[\![C(s \cdot \alpha)]\!]$. □

We show now that the process $C(s \cdot \alpha)$ induces $s \cdot \alpha$, in the sense that if $C(s \cdot \alpha)$ interacts with a sequence s' (i.e., for some α' we have $\bar{s}' \cdot \alpha' \in \mathcal{A}[\![C(s \cdot \alpha)]\!]$), which gives the same result as s, i.e., $\models Estore(s) \Leftrightarrow Estore(s')$, then s can be obtained from s' by applying the saturation operator.

The next lemma actually shows that $\bar{s}'$ is in the saturation of $\bar{s}$. The final step is then made by the *mirroring lemma* (cfr. lemma 7.4).

Lemma 7.3 *Given a modular sequence $s \cdot \alpha$, for every $s' \cdot \alpha' \in \mathcal{A}[\![C(s \cdot \alpha)]\!]$ such that $\models Estore(s') \Leftrightarrow Estore(\bar{s})$ we have $s' \in Sat(\bar{s})$.*

Proof By definition we have $\mathcal{A}[\![C(s \cdot \alpha)]\!] = Sat(\mathcal{M}[\![C(s \cdot \alpha)]\!])$. Therefore, by remarks 5.3 and 5.4, it is sufficient to prove that, for $s' \cdot \alpha' \in \mathcal{M}[\![C(s \cdot \alpha)]\!]$ with $\models Estore(s') \Leftrightarrow Estore(\bar{s})$, we have $s' \in Sat(\bar{s})$.

Let $s'.\alpha' \in \mathcal{M}[\![C(s\cdot\alpha)]\!]$ such that $\models Estore(s') \Leftrightarrow Estore(\bar{s})$. Let n be the length of s. It is not so difficult to see that there exists a computation

$$\langle A_0, s'_0\rangle \longrightarrow^* \ldots \longrightarrow^* \langle A_i, s'_i\rangle \longrightarrow^* \ldots \longrightarrow^* \langle A_n, s'_n\rangle$$

such that $s'_0 = \lambda$, $s'_n = s'$, $A_0 = C(s\cdot\alpha)$, and

$$\begin{array}{lll} A_n & = \textbf{Success} & \text{if } \alpha \in \{ss, \mathit{ff}, dd\} \\ & = \textbf{Fail} & \text{otherwise} \end{array}$$

Let $\bar{s}^{(i)}$ denote the suffix of $\bar{s}$ starting from its $(i+1)^{th}$ element. We prove that $s'_i\cdot\bar{s}^{(i)} \in Sat(\bar{s})$ for $0 \leq i \leq n$. We proceed by induction on i.

$\boldsymbol{i = 0}$**)** Obvious, since $s'_0 = \lambda$ and $\bar{s}^{(0)} = \bar{s}$.

$\boldsymbol{i + 1}$**)** By the induction hypothesis we have

$$s'_i\cdot\bar{s}^{(i)} \in Sat(\bar{s}). \tag{1}$$

There are two cases

Case 1: $\bar{s}^{(i)} = c\ \cdot\bar{s}^{(i+1)}$**)** In this case A_i equals $\mathbf{ask}(c) \rightarrow A_{i+1}$. So we have

$$\langle A_i, s'_i\rangle \longrightarrow^* \langle A_{i+1}, s'_{i+1}\rangle$$

where

$$s'_{i+1} = s'_i\cdot c_1\cdot\ldots\cdot c_k\cdot true$$

with

$$\models Store(s'_i\cdot c_1\cdot\ldots\cdot c_k) \Rightarrow c. \tag{2}$$

Next note that

$$\begin{array}{llll} \models & Estore(s') & \Leftrightarrow & \text{(by hypothesis)} \\ & Estore(\bar{s}) & \Leftrightarrow & \text{(by 1 and remark 5.3)} \\ & Estore(s'_i\cdot\bar{s}^{(i)}) & & \end{array}$$

from which we derive

$$\models Estore(s'_i\cdot c_1\cdot\ldots\cdot c_k\cdot c\ \cdot\bar{s}^{(i+1)}) \Leftrightarrow Estore(s'_i\cdot c_1\cdot\ldots\cdot c_k\cdot\bar{s}^{(i)}) \Leftrightarrow Estore(s'_i\cdot\bar{s}^{(i)}) \tag{3}$$

Therefore, since $s'_i.c_1\cdot\ldots\cdot c_k\cdot c\ \cdot\bar{s}^{(i+1)} \in Sat(s'_i\cdot\bar{s}^{(i)})$ (by 3 and **C2**), and $Sat(s'_i\cdot\bar{s}^{(i)}) \subseteq Sat(\bar{s})$ (by 1 and remark 5.4), we obtain

$$s'_i\cdot c_1\cdot\ldots\cdot c_k\cdot c\ \cdot\bar{s}^{(i+1)} \in Sat(\bar{s}) \tag{4}$$

Now we can apply **C1**, using 2, thus obtaining

$$s'_i\cdot c_1\cdot\ldots\cdot c_k\cdot\bar{s}^{(i+1)} \in Sat(s'_i\cdot c_1\cdot\ldots\cdot c_k\cdot c\ \cdot\bar{s}^{(i+1)}). \tag{5}$$

We conclude

$$\begin{array}{lll} s'_{(i+1)}\cdot\bar{s}^{(i+1)} & = & \\ s'_i\cdot c_1\cdot\ldots\cdot c_k\cdot\bar{s}^{(i+1)} & \in & \text{(by 5)} \\ Sat(s'_i\cdot c_1\cdot\ldots\cdot c_k\cdot c\ \cdot\bar{s}^{(i+1)}) & \subseteq & \text{(by 4)} \\ Sat(\bar{s}) & & \end{array}$$

Case 2: $\bar{s}^{(i)} = c \cdot \bar{s}^{(i+1)}$) In this case A_i equals $\mathbf{tell}(c) \to A_{i+1}$. So we have

$$\langle A_i, s'_i \rangle \longrightarrow^* \langle A_{i+1}, s'_{i+1} \rangle$$

where

$$s'_{i+1} = s'_i \cdot c_1 \cdot \ldots \cdot c_k \cdot c$$

Similarly to the previous case, we have

$$\models Estore(s'_i \cdot c_1 \cdot \ldots \cdot c_k \cdot c \cdot \bar{s}^{(i+1)}) \Leftrightarrow Estore(s'_i \cdot c_1 \cdot \ldots \cdot c_k \cdot \bar{s}^{(i)}) \Leftrightarrow Estore(s'_i \cdot \bar{s}^{(i)}) \quad (6)$$

therefore, since $s'_i \cdot c_1 \cdot \ldots \cdot c_k \cdot c \cdot \bar{s}^{(i+1)} \in Sat(s'_i \cdot \bar{s}^{(i)})$ (by 6 and **C2**), and $Sat(s'_i \cdot \bar{s}^{(i)}) \subseteq Sat(\bar{s})$ (by 1 and remark 5.4), we obtain

$$s'_i \cdot c_1 \cdot \ldots \cdot c_k \cdot c \cdot \bar{s}^{(i+1)} \in Sat(\bar{s}) \quad (7)$$

Therefore we can conclude

$$\begin{array}{lll} s'_{(i+1)} \cdot \bar{s}^{(i+1)} & = & \\ s'_i \cdot c_1 \cdot \ldots \cdot c_k \cdot c \cdot \bar{s}^{(i+1)} & \in & \text{(by 7)} \\ Sat(\bar{s}). & & \end{array}$$

□

Lemma 7.4 (Mirroring lemma) *Let s and s' be modular sequences. If $s' \in Sat(s)$, then $\bar{s} \in Sat(\bar{s}')$. (As usual, $\bar{s}$ and $\bar{s}'$ denote the mirror of s, s', respectively.)*

Proof First note that without loss of generality we may assume that s' is obtained from s by a derivation consisting of modular sequences (take a derivation of s' from s and "modularize" all its sequences). So it is sufficient to show that for any set of sequences S, if $Sat(S) = S$, then S satisfies the following property:

C3 $\quad s_1 \cdot c \cdot s_2 \cdot \alpha \in Mod(S) \Rightarrow s_1 \cdot s_2 \cdot \alpha \in Mod(S)$
if $\models Estore(s_1 \cdot s_2) \Leftrightarrow Estore(s_1 \cdot c \cdot s_2)$.

Here $Mod(S)$ denotes the set of modular sequences of S. We then can proceed by induction on the number of applications of the saturation conditions **C1** and **C2**, making use of the fact that **C1** mirrors itself, in the following sense: if s' is derived from s by one application of **C1** then $\bar{s}$ can be derived from $\bar{s}'$ using **C1** again. In same sense an application of **C2** can be mirrored by **C3**.

We prove **C3** by induction on the length of s_2:

$s_2 = \lambda$) In this case we just apply **C1**.

$s_2 = c'^{\ell} \cdot s'_2$) We consider the cases $\ell = I$ and $\ell = O$ separately.

$\ell = O$) By **C1** we have

$$s_1 \cdot c \cdot c' \cdot s'_2 \cdot \alpha \in Mod(S) \Rightarrow s_1 \cdot c' \cdot c \cdot s'_2 \cdot \alpha \in Mod(S)$$

The induction hypothesis then gives us

$$s_1 \cdot c' \cdot s'_2 \cdot \alpha \in Mod(S)$$

$\ell = I$) By **C2** we have

$$s_1 \cdot c \cdot c' \cdot s'_2 \cdot \alpha \in Mod(S) \Rightarrow s_1 \cdot c' \cdot c \cdot c' \cdot s'_2 \cdot \alpha \in Mod(S)$$

An application of **C1** then gives us

$$s_1 \cdot c' \cdot c \cdot s'_2 \cdot \alpha \in Mod(S)$$

By induction hypothesis we obtain

$$s_1 \cdot c' \cdot s'_2 \cdot \alpha \in Mod(S).$$

□

Now we are ready to prove the main theorem

Theorem 7.5 (Full abstraction of $\mathcal{A}$) *For arbitrary processes A_1, A_2 such that $\mathcal{A}[\![A_1]\!] \neq \mathcal{A}[\![A_2]\!]$ there exists a context $C[\,]$ such that $\mathcal{O}[\![C[A_1]]\!] \neq \mathcal{O}[\![C[A_2]]\!]$.*

Proof Assume $s{\cdot}\alpha \in \mathcal{A}[\![A_1]\!] \setminus \mathcal{A}[\![A_2]\!]$. As explained above we may assume without loss of generality that s is modular. We treat first the case that $s = s_1{\cdot}c\;{\cdot}s_2$, such that $\not\models \mathit{Estore}(s_1) \Leftrightarrow \mathit{Estore}(s_1{\cdot}c\;)$. Note that this implies that $\alpha \in \{ss, \mathit{ff}, dd, \perp\}$. Let $A = C(s{\cdot}\alpha)$, and let

$$\bar{\alpha} = \begin{cases} ss & \text{if } \alpha \in \{ss, \mathit{ff}, dd\} \\ \mathit{ff} & \text{otherwise.} \end{cases}$$

By proposition 7.2 we have $\bar{s}{\cdot}\bar{\alpha} \in \mathcal{A}[\![A]\!]$, therefore

$$\mathit{Result}(s{\cdot}\alpha \tilde{\|} \bar{s}{\cdot}\bar{\alpha}) \in \mathit{Result}(\mathcal{A}[\![A \parallel A_1]\!])_{/\Leftrightarrow} = \mathcal{O}[\![A \parallel A_1]\!].$$

Assume now that

$$\mathit{Result}(s{\cdot}\alpha \tilde{\|} \bar{s}{\cdot}\bar{\alpha}) \in \mathcal{O}[\![A \parallel A_2]\!] = \mathit{Result}(\mathcal{A}[\![A \parallel A_2]\!])_{/\Leftrightarrow}$$

By the compositionality of $\mathcal{A}$ and remark 5.3 it follows that there exist $s'\cdot\alpha' \in \mathcal{A}[\![W_2; \bar{A}_2]\!]$ and $\bar{s}'\cdot\alpha'' \in \mathcal{A}[\![W; \bar{A}]\!]$ such that

$$\mathit{Result}(s'\cdot\alpha' \tilde{\|} \bar{s}'\cdot\alpha'') = \mathit{Result}(s{\cdot}\alpha \tilde{\|} \bar{s}{\cdot}\bar{\alpha}). \tag{8}$$

Without loss of generality we may assume that s' is modular. We have

$$\models \mathit{Estore}(\bar{s}') \Leftrightarrow \mathit{Estore}(s' \tilde{\|} \bar{s}') \Leftrightarrow \mathit{Estore}(s \tilde{\|} \bar{s}) \Leftrightarrow \mathit{Estore}(\bar{s}),$$

so by lemma 7.3 we have $\bar{s}' \in \mathit{Sat}(\bar{s})$. An application of lemma 7.4 then yields $s \in \mathit{Sat}(s')$. Therefore $s\cdot\alpha' \in \mathcal{A}[\![A_2]\!]$ holds. By definition of $\mathcal{M}_\iota$ and $\mathcal{A}$, and the assumption that $s = s_1\cdot c\;\cdot s_2$ such that $\not\models \mathit{Estore}(s_1) \Leftrightarrow \mathit{Estore}(s_1\cdot c\;)$, it follows that $s{\cdot}\perp \in \mathcal{A}[\![W_2; \bar{A}_2]\!]$. Furthermore we observe that, by the above mentionated assumption about s, by definition of $\bar{\alpha}$, of the process $C(s{\cdot}\alpha)$, and by 8, we have $\alpha' = \alpha$ if $\alpha \neq \perp$. (Note that the assumption about s excludes the possibility that $\alpha \in \{\mathit{ff}_\iota, dd_\iota, \perp_\iota\}$, because the process $C(s{\cdot}\alpha)$ requires the environment to produce the constraint c.) So we conclude $s\cdot\alpha \in \mathcal{A}[\![A_2]\!]$, and this contradicts our initial assumption.

Next we treat the case that s is *not* of the form $s_1{\cdot}c\;{\cdot}s_2$, such that $\not\models \mathit{Estore}(s_1) \Leftrightarrow \mathit{Estore}(s_1{\cdot}c\;)$. It follows that we may assume without loss of generality, using the saturation conditions, that $s = c\;\cdot\alpha$, for some constraint c. We distinguish the following cases:

$\alpha = ss$) This case is treated as above.

$\alpha = \perp, \perp_\iota$) It is sufficient to consider the case that $c\cdot\perp \in \mathcal{A}[\![A_1]\!] \setminus \mathcal{A}[\![A_2]\!]$, because the case $c\cdot\perp_\iota \in \mathcal{A}[\![A_1]\!] \setminus \mathcal{A}[\![A_2]\!]$ does not occur. Let $C[\;] = \mathbf{tell}(c) \rightarrow ((\mathbf{Fail} \parallel [\;]) + \mathbf{Success})$. It is not difficult to show that $\langle c, \mathit{ff}\rangle \in \mathcal{O}[\![C[A_1]]\!] \setminus \mathcal{O}[\![C[A_2]]\!]$.

$\alpha = \mathit{ff}, dd$) Let $C[\;] = \mathbf{tell}(c) \rightarrow ([\;] + \mathbf{Success})$. It is easy to show that $\langle c, \alpha\rangle \in \mathcal{O}[\![C[A_1]]\!] \setminus \mathcal{O}[\![C[A_2]]\!]$.

$\alpha = \mathit{ff}_\iota, dd_\iota$) First we consider $\alpha = \mathit{ff}_\iota$: Let c' be such that $\not\models c \Rightarrow c'$. Let $C[\;] = \mathbf{tell}(c) \rightarrow (((\mathbf{tell}(\mathit{true}) \rightarrow \mathbf{Fail}) \parallel [\;]) + \mathbf{ask}(c'))$. It follows that $\langle c, dd\rangle \in \mathcal{O}[\![C[A_1]]\!] \setminus \mathcal{O}[\![C[A_2]]\!]$.

Next, we consider $\alpha = dd_\iota$: So let $c\;\cdot dd_\iota \in \mathcal{A}[\![A_1]\!] \setminus \mathcal{A}[\![A_2]\!]$. First we note that $c\;\cdot dd \notin \mathcal{A}[\![A_1]\!]$, because when a process immediately suspends this means that it cannot perform any step. If on the other hand we do have $c\;\cdot dd \in \mathcal{A}[\![A_2]\!]$ we proceed as in the previous case. So suppose now that $c\;\cdot dd \notin \mathcal{A}[\![A_2]\!]$. Let $C[\;] = \mathbf{tell}(c) \parallel [\;]$. We then have $\langle c, dd\rangle \in \mathcal{O}[\![C[A_1]]\!] \setminus \mathcal{O}[\![C[A_2]]\!]$.

□

8 Conclusions and future work

We have presented a fully abstract semantics for concurrent constraint programming which is fully abstract with respect to finite observables and the operators of choice, parallelism, prefixing and hiding. There are a number of open questions that are worth of future investigation. In particular, it would be interesting to extend our results to a notion of observables that includes infinite behaviours.

References

[1] J.W. de Bakker and J.N. Kok. Uniform abstraction, atomicity and contractions in the comparative semantics of Concurrent Prolog. In *Proc. of the International Conference on Fifth Generation Computer Systems*, pages 347–355, Tokyo, Japan, 1988. OHMSHA, LTD. Extended Abstract, full version available as CWI report CS-8834.

[2] F.S. de Boer, J.N. Kok, C. Palamidessi, and J.J.M.M. Rutten. Control flow versus logic: a denotational and a declarative model for Guarded Horn Clauses. In A. Kreczmar and G. Mirkowska, editors, *Proc. of the Symposium on Mathematical Foundations of Computer Science*, volume 379 of *Lecture Notes in Computer Science*, pages 165–176. Springer-Verlag, 1989.

[3] F.S. de Boer, J.N. Kok, C. Palamidessi, and J.J.M.M. Rutten. Semantic models for a version of PARLOG. In Giorgio Levi and Maurizio Martelli, editors, *Proc. of the Sixth International Conference on Logic Programming*, Series in Logic Programming, pages 621–636, Lisboa, 1989. The MIT Press. Extended version to appear in Theoretical Computer Science.

[4] F.S. de Boer and C. Palamidessi. Concurrent logic languages: Asynchronism and language comparison. In *Proc. of the North American Conference on Logic Programming*, Series in Logic Programming. The MIT Press, 1990. Technical Report TR 6/90, Dipartimento di Informatica, Università di Pisa.

[5] F.S. de Boer and C. Palamidessi. On the asynchronous nature of communication in concurrent logic languages: A fully abstract model based on sequences. In *Proc. of Concur 90*, The Netherlands, 1990. Full version available as report at the Technische Universiteit Eindhoven.

[6] M. Falaschi, M. Gabbrielli, G. Levi, and M. Murakami. Nested Guarded Horn Clauses: a language provided with a complete set of Unfolding Rules. In *Proc. of the Japanese National Conference on Logic Programming '89*, 1989.

[7] M. Gabbrielli and G. Levi. Unfolding and fixpoint semantics for concurrent constraint logic programs. In Springer-Verlag, editor, *Proc. of the Second Int. Conf. on Algebraic and Logic Programming*, Lecture Notes in Computer Science, Nancy, France, 1990.

[8] H. Gaifman, M. J. Maher, and E. Shapiro. Reactive Behaviour semantics for Concurrent Constraint Logic Programs. In E. Lusk and R. Overbeck, editors, *North American Conference on Logic Programming*, 1989.

[9] R. Gerth, M. Codish, Y. Lichtenstein, and E. Shapiro. Fully abstract denotational semantics for Concurrent Prolog. In *Proc. of the Third IEEE Symposium on Logic In Computer Science*, pages 320–335. IEEE Computer Society Press, New York, 1988.

[10] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *14th ACM Principles of Programming Languages Conference*, pages 111–119, Munich, F.R.G., 1987. ACM, New York.

[11] J.N. Kok. A compositional semantics for Concurrent Prolog. In R. Cori and M. Wirsing, editors, *Proc. Fifth Symposium on Theoretical Aspects of Computer Science*, volume 294 of *Lecture Notes in Computer Science*, pages 373–388. Springer-Verlag, 1988.

[12] M. J. Maher. Logic semantics for a class of committed-choice programs. In Jean-Louis Lassez, editor, *Proc. of the Fourth International Conference on Logic Programming*, Series in Logic Programming, pages 858–876, Melbourne, 1987. The MIT Press.

[13] V.A. Saraswat. Partial Correctness Semantics for CP($\emptyset, |, \&$). In *Proc. of the Conference on Foundations of Software Computing and Theoretical Computer Science*, volume 206 of *Lecture Notes in Computer Science*, pages 347–368. Springer-Verlag, 1985.

[14] V.A. Saraswat. A somewhat logical formulation of CLP synchronization primitives. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Proc. of the Fifth International Conference on Logic Programming*, Series in Logic Programming, pages 1298–1314, Seattle, USA, 1988. The MIT Press.

[15] V.A. Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, january 1989. To be published by the MIT Press.

[16] V.A. Saraswat and M. Rinard. Concurrent constraint programming. In *Proc. of the seventeenth ACM Symposium on Principles of Programming Languages.* ACM, New York, 1990.

[17] V.A. Saraswat, M. Rinard and P. Panangaden. A model for concurrent constraint programming. July 1990.