Unfolding and fixpoint semantics of concurrent constraint logic programs*

Maurizio Gabbrielli and Giorgio Levi

Dipartimento di Informatica, Università di Pisa, Corso Italia 40, 56125 Pisa, Italy

Abstract


We present a new semantics for a language in the family of concurrent constraint logic languages. The semantics can be based on a notion of clause unfolding, since the language is closed under this transformation. The unfolding semantics is obtained as a limit of an infinite unfolding process. Unfolding is also used to define an immediate consequences operator and, therefore, a fixpoint semantics in the typical logic programming style. The result of both the unfolding and the fixpoint semantics is a set of reactive behaviors, which are trees abstractly representing all the possible computations of a program, including deadlocks and finite failures.

1. Introduction

Committed choice logic languages (CCL), which include GHC [45], PARLOG [4, 22] and Concurrent Prolog (CP) [42, 43], have been considered in the last few years as a very interesting combination of concurrency and logic programming. We assume the reader to be familiar with this class of languages (see [43] for a detailed and updated survey). Let us just recall the basic CCL computational model. The body of a typical CCL clause consists of two components: the guard, which states the conditions which have to be satisfied before the clause can be applied, and the proper body, which specifies the actions to be performed after the (nonbacktrackable) selection of the clause, i.e. after the commitment to that particular clause. Different choices

*This work was supported by ESPRIT Basic Research Action P3020 “Integration” and by C.N.R. Progetto Finalizzato “Sistemi Informatici e Calcolo Parallelo”

0304-3975/92/$05.00 © 1992—Elsevier Science Publishers B.V. All rights reserved
in the definition of the guard and the proper body lead to considerably different computational models with different expressive powers. One relevant feature is flatness. Namely, nonflat languages allow the guards to contain program-defined predicates. On the contrary, in flat languages, only system-defined predicates can appear in the guard. In the class of flat languages, we have, on one side, languages like flat GHC, flat PARLOG and FCP [45, 4, 22, 43], which allow “asking” actions to take place only before the commitment (i.e. languages where a clause is not allowed to export bindings until commitment) and, on the other side, languages like cc(\downarrow,\downarrow) [39] and FCP(\downarrow) [26, 43] where a “tell” action (i.e. an evaluation which affects the external environment) can take place before the commitment. In what follows, we will refer to the latter as the ask: tell languages. The “ask” mechanism depends on the specific language. In general, the notion of “ask” can be reduced to a kind of ask unification, i.e. a modified unification which is performed relatively to a given environment whose variables cannot be instantiated. Typical ask mechanisms are the read-only annotations of CP [42], the one-way unification of PARLOG [4, 22] and the GHC rule of suspension [45]. Different synchronization methods, as, for example, the determinacy conditions of P-Prolog [48] and the similar ones in Alps [32], Andorra Prolog [23] and Pandora [1], are beyond the scope of this paper and will not be considered.

Recently, the concurrent constraint (cc) language paradigm has been proposed in [40, 41] as a computational model which encompasses the flat CCL framework. The cc paradigm integrates concurrent logic programming with constraint logic programming [25] and is based on a logical formulation of the synchronization mechanism, first introduced in [32] and then used in [39] to characterize the synchronization primitives of various concurrent logic languages. In the cc framework, a computation is the concurrent execution of agents which can “ask” and “tell” constraints to a global structure named “store”. Ask operations are performed atomically to ask if a constraint is entailed by the current store and can succeed, fail (if the current store is not consistent with the asked constraint) or suspend (if the store is consistent with the constraint but does not entail it). The set of constraints in the store is monotonically extended by the tell operations which are performed atomically too. If the store represents, by means of constraints, the partial information about the possible values for the program variables, at each step of the computation the store describes a more constrained set of values. The (success, failure or deadlock) result of the computation is the projection of the final store on the variables of the initial state of the computation, the final store being, respectively, the store of a success, a failure or a deadlock state of computation.

A concurrent constraint language cc(\%) is parametric w.r.t. the underlying constraint system %. In particular, if the constraints are equalities over the domain of Herbrand terms, the resulting language is essentially similar to the flat versions of committed choice languages [40, 41], and the various notions of “ask unification” of CCL languages can be considered as particular implementations of the logical formulation of the ask mechanism in terms of entailment.
The semantics of CCL programs was considered by many authors, following several different approaches. Some of these studies consider the operational semantics only (see, for example, [2, 38]). Others can somewhat be considered as examples in the denotational style (see, for example, [21, 6–10, 20, 41]). None of the above approaches is really satisfactory if concurrent logic programs have to be considered as logical and not only as concurrent. In fact, the approach and the formal tools are exactly those that have been used in other areas of concurrency (imperative concurrent programming), while the typical semantic definition style of logic programs (the model-theoretic semantics or the first-order fixpoint semantics based on the “bottom-up” immediate consequences operator [47]) is completely ignored. The “logic” approach was pursued by other authors. For example, [30, 31] defined a fixpoint semantics for deadlock-free programs. This construction was then extended to a fixpoint characterization of the success set for the full language (see [11], the “declarative” semantics in [8] and [35] for a greatest fixpoint definition). If we consider reactive systems, i.e. systems whose computational aim is establishing a permanent interaction with the external environment, a characterization of the success set is not enough, since the results of failed, deadlocked and, in general, nonterminated computations are semantically relevant.

In this paper, which is an extended version of [17], we are concerned with the problem of defining a fixpoint semantics modeling successes, deadlocks and finite failures. We want the semantics to be based on an immediate consequences operator. This can be obtained following the approach originally sketched in [28], already applied in [29] to pure logic programs and in [11] to the success set of CCL programs. This approach has also clearly inspired the denotational definition in [20], which uses a “semantic” form of clauses and a transformation which are exactly the same as those in [11]. The key issue of the approach is that models should allow to derive the observable operational properties. The right notion of model is, informally, that of a (possibly infinite) set of unit clauses, and the relevant operational properties must be observable by executing a goal in the model [12–14]. With this notion of model, a formal semantics can be based on program transformation techniques, i.e. unfolding [3, 44]. Unfolding is, in fact, strongly related to the operational semantics and can be used to formally derive the model, which is also a program. The unfolding semantics and the least-fixpoint semantics are closely related, since the immediate consequences operator is technically very similar to the unfolding.

The first problem is then finding a semantically correct and total (i.e. always applicable) set of unfolding rules. For example, none of the attempts [15, 16, 46] to define a set of unfolding rules for GHC resulted in a total set. As we will show, the problem is related to the fact that the syntax of the language is not powerful enough to express its own semantics, which has to be preserved by the unfolding process. A first solution to this problem was the language NGHC (nested guarded horn clauses) [11], where the syntax of flat GHC was extended so as to allow the definition of a total set of unfolding rules, preserving the success semantics. In the case of reactive systems, however, we want to model also failed and deadlocked computations. Therefore,
following the NGHC direction inspired by Curien’s philosophy [5] of making the syntax akin to the semantics, we defined $cc(\mathcal{H})$, whose main feature is a tree structure of clauses. Due to this improvement in syntax, it is possible to define a total set of unfolding rules which preserve the operational reactive semantics for successes, deadlocks and finite failures. According to the method of [29], an unfolding semantics can then be given. The unfolding semantics of a program $P$ is the interpretation obtained as limit of an infinite unfolding process of $P$, an interpretation being a set of reactive behaviors which are essentially equivalence classes of unit clauses of the language. A fixpoint semantics in the classic logic programming style is defined by making use of an operator $T_{UP}$ defined in terms of unfolding. The fixpoint and unfolding semantics are proved equivalent. $cc(\mathcal{H})$ turns out being very similar to a particular instance of the concurrent constraint languages family [40,41], and we will then consider it in the $cc$ paradigm. Hence, in this paper, we show the semantic construction for a generic $cc(\%)$ language. The semantics of $cc$ languages has been considered in some recent papers [20,9,10,41]. The relation of our results to those contained in these papers will be discussed in Section 9.

The paper is organized as follows. In Section 2 we show the motivations for the definitions of the $cc(\mathcal{H})$ (and $cc(\%)$) language as a language closed under unfolding. In Section 3 the $cc(\%)$ syntax is introduced. Section 4 discusses the operational semantics and contains a formal definition via transition systems. The unfolding rules are shown in Section 5. Section 6 contains the basic definitions of interpretation and ordering on interpretations. The equivalence-preserving properties of the unfolding and the unfolding semantics are introduced in Section 7. An equivalent fixpoint semantics is defined in Section 8. Finally, Section 9 is devoted to a discussion on related and future work. The appendix contains the proof of some technical lemmata and theorems.

2. Towards a concurrent language closed under unfolding

As sketched in the introduction, we are interested in the definition of a correct and total set of unfolding rules. The set of rules is correct if the unfolded program is semantically equivalent to the original one. The set is total if it allows to unfold every procedure call in any program clause. The standard “one-level” guard structure of concurrent logic languages is not powerful enough to allow the definition of such a set of unfolding rules. In fact, it is well known that the semantics of languages with synchronization can be described compositionally only in terms of sequences (of substitutions, constraints, actions) [37, 28]. Therefore, since the semantics has to be preserved by the unfolding process, sometimes it is not possible to apply the unfolding to a clause, because it is not possible (for synchronization reasons) to reduce the sequence of guards defined by the operational semantics to a unique semantically equivalent one level guard. Before giving an example for the flat GHC case, let us informally introduce the language. For an extensive discussion on (flat and nonflat) GHC, see [45,43]. A flat GHC program is a finite set of clauses of the form $H: \leftarrow G | B$, 

where $H$, the head, is an atom, $B$, the body, is a conjunction of atoms and $G$, the guard, is a conjunction of unification atoms of the form $t_1 = t_2$. The computation rule for flat GHC can be considered as an AND parallel resolution, where the synchronization mechanism is defined by the following rules.

- **Rule of suspension.** The evaluation of the clause $p(h_1, \ldots, h_m) :- s_1 = t_1, \ldots, s_n = t_n | B$ on the goal $A = p(l_1, \ldots, l_m)$ succeeds if there exists an mgu $\theta$ for the terms $(h_1, \ldots, h_m, s_1, \ldots, s_n)$ and $(l_1, \ldots, l_m, t_1, \ldots, t_n)$ which does not instantiate any variable occurring in $A$. The evaluation fails if the terms $(h_1, \ldots, h_m, s_1, \ldots, s_n)$ and $(l_1, \ldots, l_m, t_1, \ldots, t_n)$ are not unifiable and suspends otherwise.

- **Rule of commitment.** Let $G_0 = A_1, \ldots, A_n$ be a goal. In order to replace the atom $A_i$, $1 \leq i \leq n$, a clause $H : G|B$ is nondeterministically chosen among those whose evaluation on $A_i$ succeeds. Then $G_0$ is transformed to $(A_1, \ldots, A_{i-1}, B, A_{i+1}, \ldots, A_n) \theta$, where $\theta$ is the result of the successful evaluation of $H : G|B$ on $A_i$.

The evaluation of a goal is the AND parallel evaluation of its atomic components, according to the previous rules. A successful GHC goal refutation computes an answer substitution, as is the case for pure logic programs. Let us now show an example.

**Example 2.1** (Levi [28]). Let $W$ be a Flat GHC program containing the following clauses:

(c1) $s(X, Y) :- \text{true}, X = a, t(a, Y)$.

(c2) $t(X, Y) :- Y = b | \text{true}$.

The existing unfolding rules [15, 16, 46] do not allow the unfolding of $c_1$ using $c_2$. In fact, the clause $c_3$

(c3) $s(X, Y) :- Y = b, X = a$,

which could be the result of the naive unfolding of $c_1$, is equivalent to $c_1$ only if we consider goals consisting of a single call to $s$. Indeed, if $s$ occurs in conjunction with other goals, the order of evaluation of the unification atoms $X = a$ and $Y = b$ is relevant to synchronization. For example, in the program

$$P = \{(c_1) s(X, Y) :- \text{true}, X = a, t(a, Y).$$

$$\quad (c_2) t(X, Y) :- Y = b | \text{true}.$$

$$\quad (c_4) p(X, Y) :- X = a | Y = b. \quad \}$$

the goal $s(X, Y), p(X, Y)$ can only succeed, while it can only lead to a deadlock in the program obtained from $P$ by replacing clause $c_1$ with $c_3$. The correct unfolding of $c_1$ should then be the (nonflat GHC) clause

(c5) $s(X, Y) :- \text{true}, X = a \rightarrow Y = b | \text{true}$.
which tells us that in the evaluation of \( s(X, Y) \) first the tell unification atom \( X = a \) is evaluated and then the ask atom \( Y = b \).

The first step towards the solution of the above problem is the extension of the language syntax, so as to allow multiple guard layers in a clause, as first suggested in [28]. This is exactly the structure of nested GHC (NGHC) clauses [11], later called reactive clauses in [20]. Reactive clauses can be defined in concurrent constraint languages as well [40, 41], where layers are viewed as a syntactic sugar for anonymous procedures. The two main concepts of guarded clauses, namely the synchronization and the commitment rule, have to be adapted to the multiple guard layers of reactive clauses. We first consider synchronization and then show that the GHC suspension rule does not work correctly when applied to reactive clauses obtained by unfolding regular guarded clauses.

**Example 2.2.** Let us consider the following flat GHC program:

\[
R = \{ (c_1) p(X, W) : - X = f(Y), W = b, q(Z) \\
(c_2) q(X) : - X = a, \text{true} \}
\]

The goal \( p(X, Y) \) results in a deadlock because of the guard in \( c_2 \). The unfolding of clause \( c_1 \) is the (nonflat GHC) clause

\[
(c_3) p(X, W) : - X = f(Y), W = b \rightarrow Z = a, \text{true}.
\]

which does not lead to a deadlock for the same goal, under the GHC suspension rule (since the guard \( Z = a \) always succeeds).

A synchronization in terms of a global rule (like that of GHC) would not allow to define a correct unfolding unless the syntax is extended to keep track of the previous unfoldings. A more elegant solution was chosen in NGHC [11] by transforming the (global) synchronization rule into a local synchronization primitive. In the GHC case, this transformation can be achieved by using the one-way unification primitive, introduced in PARLOG [22] and defined as follows.

**Definition 2.3 (One-way unification primitive \( \leq \)).** Let \( s \) and \( t \) be two first-order terms.

(i) If there exists an idempotent mgu \( \theta \) of \( s \) and \( t \) such that \( \theta \) does not bind variables in \( s \) then \( s \leq t \) succeeds with computed answer substitution \( \theta \).

(ii) If \( s \) and \( t \) can be unified only by binding variables in \( s \) then \( s \leq t \) suspends.

(iii) If \( s \) and \( t \) cannot be unified then \( s \leq t \) fails.

Note that if we use explicit synchronization primitives in Example 2.2, clause \( c_2 \) is replaced by

\[
(c'_2) q(X) : X \leq a, \text{true}.
\]
and the unfolding of \( c_1 \)

\[
(c_3) \quad p(X, W) : - X \leq f(Y) \mid W = a \rightarrow Z \leq a \mid \text{true}.
\]

correctly represents the ask constraint of clause \( c_2 \). Note also that the same argument applies to the synchronization of \( cc \) languages. In the general case of constraints, the transformation of the global synchronization rule into a local synchronization primitive requires the local variables to be explicitly quantified when appropriate (see later).

If we consider the Herbrand constraint system, PARLOG one-way unification is exactly all that is needed. The above sketched solution is the one proposed in NGHC [11]. The syntax of an NGHC reactive clause is:

\[
p(X_1, \ldots, X_n) : - \text{ask}_1 | \text{tell}_1 \rightarrow \cdots \rightarrow \text{ask}_m | \text{tell}_m, B_1, \ldots, B_r.
\]

The \( \text{ask}_i \)'s are sets of one-way unification atoms \( (s \leq t) \) which express conditions on the "external environment" (through the variables \( X_1, \ldots, X_n \)) which have to be satisfied without instantiating such variables. The \( \text{tell}_i \)'s are sets of unification atoms \( (s = t) \) which define the values that have to be passed to the external environment. \( B_1, \ldots, B_r \) is the clause body consisting of standard atoms which represent procedure calls. The possibility of syntactically specifying, by means of reactive clauses, arbitrary sequences of ask (one-way) and tell unifications allows the definition of a correct (w.r.t. the success set semantics) and total set of unfolding rules. Given the unfolding rules, an unfolding and an equivalent fixpoint success semantics for NGHC have been defined.

The semantics of a predicate \( p \) in the program \( P \) is a set of sequences of input one-way and output unifications, i.e. a set of NGHC unit clauses for \( p \), which is obtained by the infinite unfolding of the clauses for \( p \) in \( P \) [11].

However, NGHC reactive clauses are not satisfactory to deal with the finite-failures semantics. The problem is related to the commit operator which affects the finite-failures semantics, which must also be preserved by any correct unfolding transformation. Let us consider an example.

Example 2.4. Let \( Q \) be the following NGHC program:

\[
Q = \{(c_1) \; q(X, Y) : - X \leq f(Z) \mid Y = a, p(Z). \\
(c_2) \; p(X) : - X \leq f(Z) \mid Z = a. \\
(c_3) \; p(X) : - X \leq g(Z) \mid Z = a. \}.
\]

The unfolding of \( Q \) is the program \( Q' \), where clause \( c_1 \) has been unfolded to the "set of clauses" \( \{c_4, c_5\} \):

\[
Q' = \{(c_4) \; q(X, Y) : - X \leq f(Z) \mid Y = a \rightarrow Z \leq f(W) \mid W = a. \\
(c_5) \; q(X, Y) : - X \leq f(Z) \mid Y = a \rightarrow Z \leq g(W) \mid W = a. \\
(c_2) \; p(X) : - X \leq f(Z) \mid Z = a. \\
(c_3) \; p(X) : - X \leq g(Z) \mid Z = a. \}.
\]
The goal \( q(f(g(X)), Y) \) always succeeds in \( Q \) (by reducing \( p(g(X)) \) using the only clause whose guard can be satisfied, namely \( c_3 \)). Such a goal could fail in \( Q' \) by nondeterministically selecting for \( q(f(g(X)), Y) \) clause \( c_4 \).

The problems related to finite failures in the above example are instances of the well-known problem of modeling observationally different behaviors in process theory [33] (see Fig. 1). In order to preserve the finite-failures semantics, it is necessary to keep somehow track of the nondeterministic choice structure of the language. Therefore, the semantics of nested commitments of NGHC was defined in such a way that after a commitment at level \( k < m \) in the clause

\[
p(X_1, \ldots, X_n) :- ask_1 \mid tell_1, \ldots, ask_m \mid tell_m \rightarrow B_1, \ldots, B_r.
\]

all the clauses for \( p \) whose prefix is equal to

\[
p(X_1, \ldots, X_n) :- ask_1 \mid tell_1, \ldots, ask_k \mid tell_k.
\]

are still competing for the selection. This operational semantics can better be defined and semantically understood by modifying the language so as to allow to syntactically specify the tree-like nondeterministic choice structure (reactive trees). For example, if + is a (guarded) disjunction operator, the program \( Q \) in Example 2.4 can be unfolded to the following program:

\[
Q'' = \{ (c_6) \ q(X, Y) :- X \leq f(Z) \mid Y = a \rightarrow ((Z \leq f(W) \mid W = a) + (Z \leq g(W) \mid W = a)).
\]

\[
(c_7) \ p(X) :- X \leq f(Z) \mid Z = a.
\]

\[
(c_8) \ p(X) :- X \leq g(Z) \mid Z = a.
\]

where the commit has the natural meaning of nonbacktrackable selection of one of the clause guards among those occurring at the same level, i.e. within the scope of the same + operator. It is worth noting that cc languages allow the definition of reactive trees (called nested clauses in [40]), which are just syntactic notations for multiple clauses.

A different problem, which is not solved in NGHC, is shown by the following program.

\[
R = \{ (c_7) \ r(X, Y) :- X \leq f(Z) \mid Y = a \rightarrow p(Z). \\
(c_8) \ r(X, Y) :- X \leq f(Z) \mid Y = a \rightarrow z(Z). \\
(c_9) \ p(X) :- X \leq f(Z) \mid Z = a. \\
(c_{10}) \ z(X) :- X \leq g(Z) \mid Z = a. \}
\]

Fig. 1. Two observationally different trees.
The unfolding of $R$ is

\[
R' = \{ (c_{11}) \ r(X, Y) : - X \leq f(Z) | Y = a \rightarrow Z \leq f(W) | W = a. \\
(c_{12}) \ r(X, Y) : - X \leq f(Z) | Y = a \rightarrow Z \leq g(W) | W = a. \\
(c_9) \ p(X) : - X \leq f(Z) | Z = a. \\
(c_{10}) \ z(X) : - X \leq g(Z) | Z = a. \}
\]

The goal $r(f(g(X)), Y)$ can fail in the program $R$ by selecting the clause $c_7$, while the same goal always succeeds in $R'$ because, given the operational semantics of NGHC with the previously sketched rule of "multiple commitment", after the (successful) evaluation of the guard $X \leq f(Z) \ Y = a$, both clauses $c_{11}$ and $c_{12}$ are still active and competing for being selected. This problem is solved with the tree structure of clauses using the guarded + operator. The following program corresponds to the NGHC program $R$.

\[
P = \{ (c_{13}) \ r(X, Y) : - (X \leq f(Z) | Y = a \rightarrow p(Z)) \\
\quad + (X \leq f(Z) | Y = a \rightarrow z(Z)). \\
(c_9) \ p(X) : - X \leq f(Z) | Z = a. \\
(c_{10}) \ z(X) : - X \leq g(Z) | Z = a. \}
\]

The unfolding of $P$ is the program

\[
P' = \{ (c_{14}) \ r(X, Y) : - (X \leq f(Z) | Y = a \rightarrow Z \leq f(W) | W = a) \\
\quad + (X \leq f(Z) | Y = a \rightarrow Z \leq g(W) | W = a)). \\
(c_9) \ p(X) : - X \leq f(Z) | Z = a. \\
(c_{10}) \ z(X) : - X \leq g(Z) | Z = a. \}
\]

and the two programs $P$ and $P'$ are equivalent both for the success set and for the finite-failures semantics.

The previous discussion and examples emphasize the fact that a tree structure of clauses is necessary if the language has to be closed under unfolding. Indeed, as shown in [10], it is possible to define a fully abstract reactive semantics for $cc$ languages using linear structures only. However, in this case, the semantic objects do not have the same operational meaning of program clauses and cannot, therefore, be obtained by means of unfolding or as fixpoints of a $T_P$-like operator. Linear structures similar to reactive clauses are also used in [20] to define the reactive semantics of an ask: tell language. These linear structures are derived from the program by using a transformation similar to unfolding. However, they cannot be viewed as clauses, since they contain more information (similar to failure set) necessary to correctly characterize the semantics of finite failures and deadlocks.
In summary, if the language has to be closed under unfolding, it must allow to specify nesting and disjunction of guards and to explicitly represent the synchronization by means of "local" synchronization primitives. These features lead to \textit{cc} languages, where "local" synchronization is obtained by explicit existential quantification of the appropriate local variables. For a detailed discussion on existential quantification, see Section 4.

3. The \textit{cc(\%)} language

In this section we introduce the \textit{cc(\%)} language, where \% is a generic constraint system. \textit{cc(\%)} is, apart from some minor syntactical differences, the \textit{cc(\downarrow, \rightarrow)/\%} language defined in [40]. A constraint system is defined as follows [25,41].

\textbf{Definition 3.1.} A first-order constraint system is a quadruple \% = \langle \Sigma, \mathcal{A}, \text{Var}, \Phi \rangle, where \Sigma is a many-sorted vocabulary with associated set of sorts \mathcal{S} and ranking function \rho, \mathcal{A} is a \Sigma-structure, \text{Var} is an \mathcal{S}-sorted set of variables and \Phi, the set of admissible constraints, is some nonempty subset of (\Sigma, \text{Var})-formulas, closed under conjunction.

As usual, the \mathcal{A} valuation is a mapping from \text{Var} to the domain of \mathcal{A}. A constraint can be considered as the implicit definition of the possibly infinite set of its solutions, i.e. the valuations which satisfy the constraint. A widely used constraint system in logic programming is the Herbrand system, which interprets the vocabulary \Sigma on the free \Sigma-algebra.

A \textit{cc(\%)} clause is an AND/OR tree. The root is the head which has the form \(p(X_1, \ldots, X_n)\), where \(p\) is a predicate symbol and \(X_1, \ldots, X_n\) are distinct variables. \textit{OR} nodes represent alternatives. Each are leaving an \textit{OR} node is labeled by an \textit{ask}: \textit{tell} element, where \textit{ask} and \textit{tell} are two constraints of the constraint system \%. A don't \textit{care} commit operator (\(\rightarrow\)) follows each \textit{ask}: \textit{tell}. \textit{AND} nodes represent sets of (possibly unfolded versions of) procedure calls which have to be evaluated in parallel. A tree leaf is either a procedure call or the terminal element \textit{nil}. Unit clauses are clauses without procedure calls and without \textit{AND} nodes, i.e. \textit{ask}: \textit{tell} labeled \textit{OR} trees.

A legal \textit{cc(\%)} program must have exactly one clause for each predicate. This rule is quite natural, since we no longer need several clauses for the same predicate. All the possible "standard" clauses for a predicate are collected into a unique tree. The formal \textit{cc(\%)} syntax is a minor modification of that one defined in [41].

\textbf{Definition 3.2 (cc(\%) Syntax).} Let \% = \langle \Sigma, \mathcal{A}, \text{Var}, \Phi \rangle be a constraint system.

\begin{align*}
\text{Clause} &::= \text{Head} \leftarrow \text{Agent} \quad \text{Head} ::= p(X_1, \ldots, X_n) \\
\text{ask} &::= c \in \Phi \\
\text{tell} &::= c \in \Phi \\
\text{Proc}_{\text{call}} &::= p(Y_1, \ldots, Y_n)
\end{align*}
Agent ::= nil / \exists \bar{X}. ask : tell\rightarrow Agent / Agent + Agent

where \(X_1, \ldots, X_n, Y_1, \ldots, Y_n\) are distinct variables, \(\rightarrow\) is the commit operator, \(\|\) is the AND-parallel conjunction, \(+\) is the disjunction. We assume that the operators \(\|\) and \(+\) are commutative and associative. \(\bar{X}\) denotes a set of variables. \(Agent\) is the category of agents with typical elements \(A, A_1, \ldots\).

If the \(n\)-adic predicate \(p\) is not defined then the program contains the always-failing clause \(p(X_1, \ldots, X_n) : -c_{\text{fail}} \rightarrow \text{nil}\), where \(c_{\text{fail}}\) is any unsatisfiable constraint in \(\Phi\).

4. The operational semantics of \(cc(\Phi)\)

In this section we informally describe the computation rule. A formal operational semantics via a transition system is defined in Section 4.1.

A \(cc(\Phi)\) computation is the concurrent execution of agents which interact with a global set of constraints named \(\text{store}\). The basic actions performed by agents are asking for a constraint \(c\) being entailed by the store \(\sigma\), and telling a constraint \(c\) to the store \(\sigma\) if \(c \cup \sigma\) is consistent. The computation rule can be considered as a nondeterministic AND-parallel execution of agents w.r.t. a global store \(\sigma\), where the basic computation steps are specified as follows.

- **Evaluation of ask : tell.** Let \(a\) be a possibly existentially quantified ask constraint, i.e. \(a = \exists \bar{X}. \text{ask}\), where \(\text{ask} \in \Phi\) (Definition 3.1). The evaluation of \(a\) in a store \(\sigma\) succeeds if \(\%I = (\forall)(a \Rightarrow a)\) (\(a\) entails \(a\)), fails if \(\%I = (\exists)(\sigma \Rightarrow \neg a)\) (\(\sigma \wedge a\) is inconsistent), suspends if \(\%I = (\exists)(\sigma \wedge a)\) and \(\%I \not= (\forall)(a \Rightarrow a)\) (\(\sigma \wedge a\) is consistent but \(\sigma\) does not entail \(a\)), \((\exists)(F)\) and \((\forall)(F)\) are the existential and the universal closure of the formula \(F\), respectively. The evaluation of a tell constraint \(t\) in the store \(\sigma\) succeeds if \(\%I \not= (\exists)(\sigma \wedge t)\) and fails if \(\%I = (\forall)(\sigma \Rightarrow \neg t)\). The evaluation of an \(a : t\) element is performed as an atomic action. It succeeds if the evaluations of both \(a\) and \(t\) succeed. It fails if the evaluation of either \(a\) or \(t\) fails and suspends otherwise. It \(a : t\) succeeds, the new store is \(\sigma' = \sigma \wedge t\). If the evaluation fails the store is unchanged.

- **Rule of commitment.** At each OR node, an alternative (an edge in the tree) is nondeterministically chosen among those whose \(a : t\) succeeds.

A computation for an atomic goal \(G\) in the \(cc(\Phi)\) program \(P\) proceeds by evaluating the AND and OR nodes of the (unique) clause for \(G\), starting in an initial store \(\sigma\). The evaluation of an OR node \(B\) is performed by committing to a particular edge \(e_t\), as specified by the previous rule of commitment. If the evaluation of every edge coming out from \(B\) is suspended or failed and there exists at least one suspended edge, \(B\) is suspended. If every edge fails, \(B\) fails too. The evaluation of an AND node \(A\) is the parallel evaluation of all the successors of \(A\). If the evaluation of a successor fails, \(A\) fails. If all the successors of \(A\) are suspended, \(A\) is suspended. \text{nil} always succeeds.
The evaluation of the procedure call \( \text{p}(Y_1, \ldots, Y_n) \) in the store \( \sigma \) proceeds by evaluating the root of the renamed version of the (unique) clause \( c = (p(X_1, \ldots, X_n) :- B) \theta \) for \( p \), where \( \theta \) is the set of bindings \( \{X_1/Y_1, \ldots, X_n/Y_n\} \). The computation terminates with failure when the evaluation of a node fails. It terminates with a deadlock when all the nodes are suspended, and terminates with success when all the nodes are successfully evaluated. The result of the computation is the store, restricted to the variables of the goal, obtained when a terminal (success, failure or deadlock) state is reached. The computation for a nonatomic goal can be described by considering a compound goal as an AND node.

Existential quantification and its operational meaning deserve some additional discussion. Existential quantification is introduced to provide the notion of local variable. Note that variables which appear in the store are considered universally quantified in the evaluation of an ask constraint (the condition for the successful evaluation is \( \models (\forall Y)(\sigma \Rightarrow Y = a) \)). For example, let us consider the goal \( p(X) \) in the store \( \sigma = \{X = f(a)\} \) and the clause

\[
(c_1) \quad p(X) :- X = f(Y); \{\} \rightarrow A.
\]

Then the computation is suspended since \( \emptyset \not\models (\forall Y, X)(\sigma \Rightarrow Y = a) \). If the ask constraint \( X = f(Y) \) in the clause has the (natural) meaning of asking \( X \) to be bound to a term of the form \( f(\cdots) \), i.e. if \( Y \) has to be considered as a local variable then \( Y \) has to be existentially quantified. In fact, if we consider the clause

\[
(c_2) \quad p(X) :- \exists Y, X = f(Y); \{\} \rightarrow A.,
\]

then the computation for \( p(X) \) in the store \( \sigma \) is not suspended, because the ask is successfully evaluated since \( \emptyset \models (\forall X)(\sigma \Rightarrow \exists Y, Y = a) \). However, note that the ask evaluation does not perform any binding. For example, in the previous case, even if the evaluation of \( \exists Y, Y = a \) is successful in \( \sigma \), after this evaluation \( Y \) is not bound to \( a \) in the store \( \sigma \). Therefore, when an existentially quantified variable \( Y \) appears in an ask constraint, in order to pass the value of \( Y \) specified by an ask constraint to the remaining part of the program, we have also to “tell” the ask constraint. In other words, a clause like

\[
H :- \exists Y, \text{ask} ; \text{tell} \rightarrow A.,
\]

where \( Y \) appears in ask, has to be changed in

\[
H :- \exists Y, \text{ask} ; \text{ask} \land \text{tell} \rightarrow A.
\]

(note that it is not necessary to check the satisfiability of the constraint ask when performing the evaluation of \( \text{ask} \land \text{tell} \)).

Since variables in the store are universally quantified, an existentially quantified variable \( Y \) has to be considered “bound” by its first occurrence in a tell, since after that occurrence \( Y \) is added to the store. In other words, in the following clause

\[
(c_3) \quad p(W, Z) :- \exists Y, W = f(Y); W = f(Y) \rightarrow Y = a; Z = b.,
\]
given the initial goal $p(X, Z)$ with the initial store $\sigma$, the evaluation of the constraint $Y = a$ means the evaluation of $\forall X \in (\sigma' \wedge Y \neq a)$, where $\bar{X} = \{ Var(\sigma) \} \cup \{ W, Y \}$ and $\sigma' = \{ W = f(Y) \} \cup \sigma$. Then obviously the clauses

\((c_4)\) $p(X, Z) :\exists Y. X = f(a) : true \rightarrow Y = a; Z = b.$

\((c_5)\) $p(X, Z) :\exists Y. X = f(Y) : X = f(Y) \rightarrow Y = a; Z = b.$

have a different semantics, since the first one has the same semantics of $c_4'$:

\((c_4')\) $p(X, Z) : X = f(a) : true \rightarrow \exists Y. Y = a; Z = b.$

i.e. the variable $Y$ in the ask constraint $Y = a$ is existentially quantified. In what follows, we assume that clauses are normalized w.r.t. existential quantification, i.e. $\exists Y$ appears only immediately before a tell constraint containing the variable $Y$. Note that if a $\exists Y$ appears in a clause and $Y$ does not appear in any ask constraint, $\exists Y$ can be removed from the clause without affecting its semantics.

4.1. Formal operational semantics

In this section we define a formal operational semantics for \(cc(\%)\) programs via a transition system.

According to [36], the operational semantics is given by defining a set of configurations $\Gamma$, which describe the possible states of the computation, a set $\Omega$ of terminal configurations and a relation $\rightsquigarrow \subseteq \Gamma \times \Gamma$ which describes the transition relation on configurations. Let us introduce a formal definition.

**Definition 4.1** (Transition system, Plotkin [36]). An unlabeled transition system is a triple $\langle \Gamma, \Omega, \rightsquigarrow \rangle$, where $\Gamma$ is a set of configurations, $\Omega$ is the set of terminal configurations, and $\rightsquigarrow \subseteq \Gamma \times \Gamma$ is the transition relation such that $\forall \gamma \in \Omega, \gamma' \in \gamma \rightsquigarrow \gamma' ; (as \ usual, \rightarrow \ is \ represented \ as \ an \ infix \ operator).$

According to the structural operational style, a configuration can be viewed as consisting of two components. The first component represents the program that has yet to be executed, while the second one contains the “context” in which the program has to be evaluated. In our case, the program to be executed is a partial AND/OR tree and the context is the set of constraints already computed by the program. Concurrency is modeled as nondeterministic interleaving of atomic actions. Hence, a precise definition of the atomic computation step is required. As it will result from the definition of the transition rules, the choice is to consider the evaluation of an ask: tell as an atomic action. Let us now give the definition of the configurations and of the transition relation, the last one being defined as usual by transition rules presented in the “natural deduction” format.
Definition 4.2 (Configurations). The set $\Gamma$ of configurations and the set $\Omega$ of terminal configurations are defined as follows:

$$\Omega = \{ \langle \text{true}, \sigma \rangle \} \cup \{ \langle \text{fail}, \sigma \rangle \} \cup \{ \langle \text{suspend}, \sigma \rangle \},$$

$$\Gamma = \{ \langle \text{Agent}, \sigma \rangle \} \cup \Omega,$$

where Agent is the syntactic category of Definition 3.2, and $\sigma$ is a consistent set of constraints.

The transition rules describe how configurations evolve. Since a configuration contains goals which have to be solved using the program clauses, programs should be included in every configuration. However, since the program does not change in the various steps of computation, we can use [36] a ternary relation to emphasize that the transition $\gamma \rightarrow \gamma'$ is contingent upon the program $P$. In what follows, $P$ will be omitted, when its explicit specification will not strictly be required.

Definition 4.3 (Transition relation). Let ask be a possibly existentially quantified constraint formula, i.e. ask $= \exists \bar{X}. ask'$, where $\bar{X}$ is a (possibly empty) set of variables appearing in ask', and ask $\in \Phi$ is a constraint (Definition 3.1). Let tell be a constraint and let $\sigma$ be a consistent set of constraints for the constraint system $\%$.

The function $\delta(\text{ask: tell}, \sigma)$ which specifies the evaluation rule of constraints is defined as follows:

$$\delta(\text{ask: tell}, \sigma) = \begin{cases} \sigma \land \text{tell} & \text{if } \% \models (\forall)(\sigma \Rightarrow \text{ask}) \text{ and } \% \models (\exists)(\sigma \land \text{tell}), \\
\text{fail} & \text{if } \% \models (\forall)(\sigma \Rightarrow \neg(\text{ask} \land \text{tell})), \\
\text{suspend} & \text{if } \% \models (\exists)(\sigma \land \text{ask} \land \text{tell}) \text{ and } \% \models (\forall)(\sigma \Rightarrow a). 
\end{cases}$$

$(\exists)(F)$ is the existential closure of $F$ and $(\forall)(F)$ is the universal closure of $F$. Then we define the transition relation as the least relation which satisfies the following transition rules:

Agents

1. $\langle \text{nil}, \sigma \rangle \rightarrow \langle \text{true}, \sigma \rangle$,

2. $\langle \text{ask: tell} \rightarrow A, \sigma \rangle \rightarrow \langle A, \sigma A \rangle$ if $\delta(\text{ask: tell}, \sigma) = \sigma A, \sigma A \neq \text{fail}, \text{suspend}$,

3. $\langle \text{ask: tell} \rightarrow A, \sigma \rangle \rightarrow \langle \text{fail}, \sigma \rangle$ if $\delta(\text{ask: tell}, \sigma) = \text{fail}$,

4. $\langle \text{ask: tell} \rightarrow A, \sigma \rangle \rightarrow \langle \text{suspend}, \sigma \rangle$ if $\delta(\text{ask: tell}, \sigma) = \text{suspend}$,
Unfolding and fixpoint semantics

\( \langle A_1, \sigma \rangle \mapsto {\gamma}, \ \gamma \neq \langle \text{fail}, \sigma \rangle, \langle \text{suspend}, \sigma \rangle \)
\( \langle A_1 + A_2, \sigma \rangle \mapsto {\gamma} \)
\( \langle A_2 + A_1, \sigma \rangle \mapsto {\gamma} \)

(6) \( \langle A_1, \sigma \rangle \mapsto \langle \text{fail}, \sigma \rangle \)
\( \langle A_2, \sigma \rangle \mapsto \langle \text{fail}, \sigma \rangle \)
\( \langle A_1 + A_2, \sigma \rangle \mapsto \langle \text{fail}, \sigma \rangle \)
\( \langle A_2 + A_1, \sigma \rangle \mapsto \langle \text{fail}, \sigma \rangle \)

(7) \( \langle A_1, \sigma \rangle \mapsto \langle \text{suspend}, \sigma \rangle \)
\( \langle A_2, \sigma \rangle \mapsto \gamma, \ \gamma \in \{ \langle \text{fail}, \sigma \rangle, \langle \text{suspend}, \sigma \rangle \} \)
\( \langle A_1 + A_2, \sigma \rangle \mapsto \langle \text{suspend}, \sigma \rangle \)
\( \langle A_2 + A_1, \sigma \rangle \mapsto \langle \text{suspend}, \sigma \rangle \)

(8) \( \langle A, \sigma \rangle \mapsto \langle \text{true}, \sigma_1 \rangle \)
\( \langle A || A_1, \sigma \rangle \mapsto \langle A_1, \sigma_1 \rangle \)
\( \langle A_1 || A, \sigma \rangle \mapsto \langle A_1, \sigma_1 \rangle \)

(9) \( \langle A, \sigma \rangle \mapsto \langle \text{fail}, \sigma \rangle \)
\( \langle A || A_1, \sigma \rangle \mapsto \langle \text{fail}, \sigma \rangle \)
\( \langle A_1 || A, \sigma \rangle \mapsto \langle \text{fail}, \sigma \rangle \)

(10) \( \langle A, \sigma \rangle \mapsto \langle \text{suspend}, \sigma \rangle \)
\( \langle A_1, \sigma \rangle \mapsto \langle \text{suspend}, \sigma \rangle \)
\( \langle A || A_1, \sigma \rangle \mapsto \langle \text{suspend}, \sigma \rangle \)
\( \langle A_1 || A, \sigma \rangle \mapsto \langle \text{suspend}, \sigma \rangle \)

(11) \( \langle A, \sigma \rangle \mapsto \langle A_1, \sigma_1 \rangle, \ A_1 \neq \text{true, fail, suspend} \)
\( \langle A || A_2, \sigma \rangle \mapsto \langle A_1 || A_2, \sigma_1 \rangle \)
\( \langle A_2 || A, \sigma \rangle \mapsto \langle A_2 || A_1, \sigma_1 \rangle \)

if \( p(X_1, ..., X_n) :- A \) is a renamed clause in \( P \),

(12) \( \langle A \{X_1/Y_1, ..., X_n/Y_n\}, \sigma \rangle \mapsto {\gamma} \)
\( \langle p(Y_1, ..., Y_n), \sigma \rangle \mapsto {\gamma} \)

It is worth noting that the above definitions are essentially those given in [40]. They are explicitly formalized here because they provide the basis for the definition of the
operational semantics and the corresponding notion of program equivalence. The operational semantics \( \ell \) defined in the sequel considers the results of failing and deadlocked computations. Note that \( \ell \) does not consider partial computation results.

**Definition 4.4 (Store equivalence).** Let \( V \) be a set of variables, \( \sigma_1 \) and \( \sigma_2 \) be two sets of \( \mathcal{C} \)-constraints, where \( \mathcal{C} = (\Sigma, \mathcal{A}, \text{Var}, \Phi) \). Then \( \sigma_1 \equiv_{V} \sigma_2 \) (\( \sigma_1 \) is \( V \)-equivalent to \( \sigma_2 \)) iff

\[
\forall \theta \in \text{Sol}(\sigma_1) \exists \theta' \in \text{Sol}(\sigma_2) \text{ such that } \forall X \in V. X\theta = X\theta' \text{ and vice versa},
\]

where \( \text{Sol}(\sigma) \) denotes the set of the solutions of \( \sigma \), i.e. the set of the \( \mathcal{A} \)-valuations which satisfy the constraints in \( \sigma \).

**Definition 4.5 (Store restriction).** Let \( G \) be a goal, \( \sigma \) a set of constraints, and \( V \) the set of variables in \( G \). Then we define \( \sigma_{G} = S \), where \( S \) is any subset of \( \sigma \), such that \( \sigma \equiv_{V} S \) and \( \sigma \neq_{V} (S \setminus \{ c \}) \) for every \( c \in S \).

**Definition 4.6.** Let \( P \) be a \( \text{cc}(\%) \) program, \( G \) a goal in \( P \), and \( \rightarrow^* \) be the reflexive and transitive closure of \( \rightarrow \). The success, failure and deadlock sets for the program \( P \), denoted by \( SS(P) \), \( FS(P) \) and \( DS(P) \), respectively, are defined as follows:

\[
SS(P) = \{ \langle G, \text{success}, \sigma_{G} \rangle | P \vdash \langle G, \emptyset \rangle \rightarrow^* \langle \text{true}, \sigma \rangle \},
\]

\[
FS(P) = \{ \langle G, \text{failure}, \sigma_{G} \rangle | P \vdash \langle G, \emptyset \rangle \rightarrow^* \langle \text{fail}, \sigma \rangle \},
\]

\[
DS(P) = \{ \langle G, \text{deadlock}, \sigma_{G} \rangle | P \vdash \langle G, \emptyset \rangle \rightarrow^* \langle \text{suspend}, \sigma \rangle \}.
\]

The operational semantics \( \ell (\mathcal{P}) \) of a program \( P \) is defined as:

\[
\ell (P) = \langle SS(P), FS(P), DS(P) \rangle.
\]

If \( \langle G, \Psi, \sigma_{G} \rangle \in SS(P) \cup FS(P) \cup DS(P) \), with \( \Psi \in \{ \text{success}, \text{failure}, \text{deadlock} \} \) then \( \langle \Psi, \sigma_{G} \rangle \) is an answer for the goal \( G \) in the program \( P \).

### 5. Unfolding rules

Since \( \text{cc}(\%) \) allows to define explicit existential quantification, multiple guards and disjunctive nesting, which are called anonymous procedures and nested clauses in \([40]\), \( \text{cc}(\%) \) is closed under unfolding. Because of the tree structure of clauses and since every predicate is defined by a unique clause, unfolding a clause \( c \) in a program \( P \) essentially consists of three steps, namely replacement of procedure calls by procedure bodies, transformation of AND nodes into OR nodes (interleaving), freezing of failed or deadlocked paths. It is worth noting that our interpretations are based on unit clauses, i.e. clauses without procedure calls and without AND nodes. In fact, unit clauses contain all the information we need for defining the (success, deadlock and
finite failure) semantics of programs. The iterative application of unfolding leads to interpretations, since it removes procedure calls and AND nodes.

**Definition 5.1 (Reactive paths).** A reactive path is an object of the form

\[ p(X_1,\ldots,X_n) := \text{ask}_1 : \text{tell}_1 \rightarrow \cdots \rightarrow \text{ask}_n : \text{tell}_n \]

A reactive path is *n-consistent* if

1. \( \mathcal{C} \models (\exists)(\text{ask}_1 \land \text{tell}_1 \land \cdots \land \text{ask}_n \land \text{tell}_n) \) and
2. there exists a constraint \( \sigma \) on the variables \( \{X_1,\ldots,X_n\} \) such that for 
   \[ i = 1, 2, \ldots, n, \mathcal{C} \models (\exists)(\sigma \land \text{ask}_1 \land \text{tell}_1 \land \cdots \land \text{ask}_i \land \text{tell}_i) \]
   \[ \mathcal{C} \models (\sigma \land \text{ask}_1 \land \text{tell}_1 \land \cdots \land \text{ask}_{i-1} \land \text{tell}_{i-1}) \Rightarrow \text{ask}_i. \]

A reactive path of length \( n \) is consistent iff it is \( n \)-consistent. It is *failed* if it is \((n-1)\)-consistent and condition (1) does not hold. It is *deadlocked* if it is \((n-1)\)-consistent, condition (1) holds and condition (2) does not hold.

The detection of consistent, failed and deadlocked paths can be performed by a “compile time” evaluation of the constraints. In the Herbrand constraint system, where constraints are equations on the Herbrand domain, such a constraint evaluation can be based on known results in unification theory, as the solved form of equations [27], extended to deal with the ask mechanism. For the \( cc(\mathcal{H}) \) case (where PARLOG one-way unification [4] is used as a representation of ask equations) we have defined [19] a normal form for constraints and a normalization procedure which computes the normal form of a given sequence of ask and tell equations. This procedure can be used to detect consistent, deadlocked and failed paths. Moreover, it can be used to define equivalence relations on interpretations, since the solved form it computes is unique up to variable renaming. In what follows, we consider the general \( cc(\mathcal{H}) \) case.

**Definition 5.2 (Normal form of clauses).** A clause \( c \rightarrow H : \neg B \) is in normal form if each AND node has at most one OR successor and each reactive path \( H : \neg s \) obtained from a path in \( c \) is consistent, failed or deadlocked.

**Definition 5.3 (Unfolding rules).** Let \( c \) be a clause in the program \( P \). The unfolding of \( c \) in \( P \) is \( \text{Unf}(c, P) = \mu_c(\eta_c(\rho_c(c))) \), where \( \mu_c, \eta_c \) and \( \rho_c \) are defined below.

(i) (replacement of procedure calls by procedure definitions). \( \rho_c : \text{Clause} \rightarrow \text{Clause}, \) given a clause \( c \) builds a clause \( c_1 \), which is the same as \( c \) apart from procedure calls of the form \( q(Y_1,\ldots,Y_n) \), which are replaced by \( \text{Body}\theta, \) where \( q(X_1,\ldots,X_m) := \text{Body} \) is the renamed version of the clause for \( q \) in \( P \) and \( \theta \) is the renaming \( (X_1/Y_1,\ldots,X_m/Y_n) \).

(ii) (transformation of AND nodes to OR nodes). \( \eta_c : \text{Clause} \rightarrow \text{Clause} \) is defined as follows. Given a clause \( c = p(\bar{X}) : A \)

\[ \begin{align*} \text{if } \eta_a(A) &= A \text{ then } \eta_c(c) &= c, \\
\text{else } \eta_c(c) &= \eta_c(p(\bar{X}) := \eta_a(A)) \end{align*} \]
where \( \eta_a : \text{Agent} \rightarrow \text{Agent} \) is defined as follows:

\[
\eta_a(\text{nil}) = \text{nil},
\]

\[
\eta_a(p(\bar{X})) = p(\bar{X}),
\]

\[
\eta_a(a : t \rightarrow A) = a : t \rightarrow \eta_a(A),
\]

\[
\eta_a(A_1 + A_2) = \eta_a(A_1) + \eta_a(A_2);
\]

if \( A_1 = \sum_{i \in I} a_i : t_i \rightarrow A_i \) and \( A_2 = \sum_{j \in J} a'_j : t'_j \rightarrow A'_j \), \( I, J \neq \emptyset \)

then

\[
\eta_a(A_1 \parallel A_2) = \left( \sum_{i \in I} a_i : t_i \rightarrow \eta_a(A_i \parallel A_2) \right) + \left( \sum_{j \in J} a'_j : t'_j \rightarrow \eta_a(A_1 \parallel A'_j) \right),
\]

otherwise

\[
\eta_a(A_1 \parallel A_2) = \eta_a(A_1 \parallel A_2),
\]

Note that the then case of the last rule is one of the equality laws in [34].

(iii) (freezing of failed and deadlocked paths). \( \mu_c : \text{Clause} \rightarrow \text{Clause} \), given a clause \( c \), builds a clause \( c_1 \), which is the same as \( c \) apart from agents of the form \( s \rightarrow a : t \rightarrow A \) (where \( s \) is a reactive path), which are replaced by \( s \rightarrow a : t \rightarrow \text{nil} \) if \( s \rightarrow a : t \) is either failed or deadlocked.

**Definition 5.4** (Unfolding of a program). Let \( P \) be the program \( \{c_1, c_2, ..., c_n\} \). The unfolding of \( P \), denoted by \( \#A \cdot F(P) \), is the collection of clauses \( \bigcup_{i=1}^{n} \text{Unf}(c_i, P) \).

The AND–OR transformation performed in step (ii) can graphically be described as shown in Figs. 2 and 3. The difference between the two figures shows a one-level transformation of the original AND node to an OR node. This transformation process is recursively repeated for the newly generated AND nodes.

Let us now give some examples of unfolding, related to existential quantification. In the following program

\[
P = \{(c_1) \ p(X, Z) : \exists Y. X = f(Y) : \text{true} \rightarrow q(Y, Z),
(c_2) \ q(X, W) : X = a; W = b. \}
\]

the unfolding of clause \( c_1 \) is the clause \( c_3 \):

\[
(c_3) \ p(X, Z) : \exists Y. X = f(Y) : \text{true} \rightarrow Y = a; Z = b.,
\]
whose operational meaning was explained in Section 4. In the program $Q$, 

$$Q = \{ (c_4) \ p(X, Z) :- X = f(a) : true \rightarrow q(Y, Z), \\ (c_5) \ q(X, W) :- \exists Y \ . \ Y = a : W = b. \}$$

the unfolding of $c_4$ is obviously the clause $c_6$:

$$c_6 \quad p(X, Z) :- X = f(a) : true \rightarrow \exists Y \ . \ Y = a : Z = b.$$ 

Note, however, that in the program $R$, 

$$R = \{ (c_7) \ p(X, Z) :- \exists Y \ . \ X = f(a) : true \rightarrow q(Y, Z), \\ (c_8) \ q(X, W) :- X = a : W = b. \}$$

the unfolding of $c_7$ is the clause $c_9$

$$c_9 \quad p(X, Z) :- X = f(a) : true \rightarrow Y = a : Z = b,$$
whose operational semantics is different from that of clause \( c_6 \). Indeed, clause \( c_7 \) is not “normalized” w.r.t. existential quantification, as explained in Section 4, and its “normalized” version \( c'_7 \),

\[
(c'_7) \quad p(X, Z) :- X \neq f(a) : \text{true} \rightarrow q(Y, Z).
\]

has to be considered. Then the unfolding of \( c'_7 \), using clause \( c_8 \), is clause \( c_9 \).

### 6. Interpretations

In this section we introduce the notion of interpretation for \( cc(\%) \) programs. An interpretation for a program \( P \) is a set containing an interpretation for each predicate in \( P \). An interpretation for a predicate \( p \) is a directed downward closed set of reactive behaviors.

**Definition 6.1 (Reactive behavior).** A reactive behavior for the predicate \( p \) is an element \( B_p \) described as follows (up to renaming). Let \( \mathcal{C} = \langle \Sigma, \mathcal{A}, \text{Var}, \Phi \rangle \) be a constraint system.

\[
B_p := \text{Head} :- \text{Agent} \quad \text{Head} := p(X_1, \ldots, X_n)
\]

\[
\text{ask} := c \in \Phi \quad \text{tell} := c \in \Phi
\]

\[
\text{Agent} := \text{nil} \mid \bot \mid \exists X. \text{ask} : \text{tell} \rightarrow \text{Agent} \mid \text{Agent} + \text{Agent}
\]

Moreover, each reactive path in \( B_p \) of length \( n \) must be \( n \)-consistent. The set of reactive behaviors for the predicate \( p \) is denoted by \( \mathcal{R}_p \). The set of reactive behaviors for a program \( P \) will be denoted by \( \mathcal{R}_P \).

Note that a reactive behavior can be represented as an OR tree, whose edges are labeled by constraints, whose root is labeled by the head and whose leaves are labeled by either nil or \( \bot \). The \( \bot \) element stands for unspecified and can operationally be considered as a nonterminating process.

**Definition 6.2 (Preorder on reactive behaviors).** Let \( B_{p_1} = H_1 :- A_1 \) and \( B_{p_2} = H_2 :- A_2 \) be two reactive behaviors for the predicate \( p \). \( B_{p_1} \leq B_{p_2} \) iff there exists a variable renaming \( \rho \) such that \( H_1 = H_2 \rho \) and \( A_1 \subseteq A_2 \rho \), where the relation \( \subseteq \) on agents is the following:

- \( \forall A, \bot \subseteq A \)
- \( \forall A, A \subseteq A \)
- \( A_1 \subseteq A_2 \) iff \( a : t \rightarrow A_1 \subseteq a : t \rightarrow A_2 \)
- \( A_1 \subseteq A_2 \) iff \( A + A_1 \subseteq A + A_2 \)

**Lemma 6.3.** Let \( \mathcal{R}_P \) be the set of reactive behaviors for the predicate \( p \). Then \( \leq \) is a preorder on \( \mathcal{R}_P \).
Proof. $\leq$ is reflexive by definition. The transitivity of $\leq$ can easily be obtained by structural induction on the agents. □

The informal meaning of the preorder is that, if $B_{p_1} \leq B_{p_2}$, then $B_{p_2}$ allows to compute all the (success, failure and deadlock) answers that $B_{p_1}$ computes. Let us now recall some basic definitions.

Definition 6.4. Let $(A, \leq)$ be a preorder. A directed set in $A$ is a subset $D$ of $A$ such that $\forall a, b \in D \exists c \in D$ such that $[a \leq c] \land [b \leq c]$.

Definition 6.5. An ideal is a directed set $S$ which is downward closed, i.e. such that $\forall a \in S [b \leq a \Rightarrow b \in S]$.

The set of ideals of $(P, \leq)$, ordered by set inclusion will be denoted by $(Id(P), \subseteq)$. It is well known that $(Id(P), \subseteq)$ is a cpo and that the set of the finite elements of $(Id(P), \subseteq)$ is a sub-cpo isomorphic to $(P_{\approx}, \leq)$, where $P_{\approx}$ is the quotient set w.r.t. the equivalence relation $\approx$ induced by the preorder $\leq$ (with a standard abuse of notation $\leq$ denotes also the ordering on $P_{\approx}$ induced by the preorder on $P$).

Definition 6.6. $\mathcal{RB}_p$ denotes the set of reactive behaviors for the predicate $p$. $\approx$ denotes the equivalence induced by $\approx$ on the set $\mathcal{RB}_p$ and $\mathcal{RB}_{p, \approx}$ denotes the quotient set of $\mathcal{RB}_p$ w.r.t $\approx$. Moreover, given a reactive behavior $B_p$ for predicate $p$, $B_p^\approx$ denotes its downward closure, i.e. $B_p^\approx = \{ B' \in \mathcal{RB}_p \mid B'_p \subseteq B_p \}$.

Lemma 6.7. $(Id(\mathcal{RB}_p), \subseteq)$ is a complete partial ordering.

Definition 6.8 (Interpretation for a predicate). An interpretation for $p$ is any element $I_p \in (Id(\mathcal{RB}_p), \subseteq)$.

The sub-cpo $(Id(\mathcal{RB}_p), \subseteq)$ of finite elements is isomorphic to the cpo given by the quotient set $(\mathcal{RB}_{p, \approx}, \subseteq)$. Then, in the sequel, we represent the finite elements either by the ideal or by the corresponding equivalence classes. The nonfinite elements will be represented by least upper bounds of directed sets of elements in $(Id(\mathcal{RB}_p), \subseteq)$. $(p(X_1, \ldots, X_n) \leftarrow \bot)^\approx$ (the downward closure of $p(X_1, \ldots, X_n) \leftarrow \bot$, or, equivalently, its equivalence class) is the least element of the lattice $(Id(\mathcal{RB}_p), \subseteq)$. The above construction can easily be extended to programs as follows.

Definition 6.9 (Interpretation). Let $P$ be a program containing the predicates $p_1, \ldots, p_n$. An interpretation $I_P$ for $P$ is an $n$-tuple $I_P = (I_{p_1}, \ldots, I_{p_n})$ such that $I_{p_1} \in (Id(\mathcal{RB}_{p_1}), \subseteq), \ldots, I_{p_n} \in (Id(\mathcal{RB}_{p_n}), \subseteq)$. The set of all the interpretations for $P$ is denoted by $\mathcal{F}_P$.

The following lemma has a straightforward proof.
Lemma 6.10. Let $P$ be a program containing the predicates $p_1, \ldots, p_n$. Let $\leq_x$ be the relation obtained by extending $\leq$ to $n$-tuples of predicate interpretations, i.e.

$$(I_{p_1}, \ldots, I_{p_n}) \leq_x (J_{p_1}, \ldots, J_{p_n}) \text{ iff } I_{p_i} \leq J_{p_i} \land \cdots \land I_{p_n} \leq J_{p_n},$$

Then $(\mathcal{F}_p, \leq_x)$ is a complete partial ordering.

7. Semantic properties of unfolding

This section defines the unfolding semantics of a program and shows its relation to the operational semantics. The first result is the semantic equivalence between a program and its unfolding, stated in terms of the result of a successful, (finitely) failed or deadlocked computation. The proof of the following theorem is in the appendix.

Theorem 7.1 (Equivalence of $P$ and $\mathcal{F}(P)$). Let $P$ be a program. $P$ and $\mathcal{F}(P)$ are equivalent w.r.t. the operational semantics, i.e. if the goal $G$ with initial store $\sigma_G$ terminates in $P$ computing the answer $\langle \Psi, \sigma'_G \rangle$ then $G$ terminates in $\mathcal{F}(P)$ computing the answer $\langle \Psi, \sigma_G \rangle$ with $\langle \Psi, \sigma'_G \rangle$ variant of $\langle \Psi, \sigma_G \rangle$ and vice versa.

7.1. The unfolding semantics

Our unfolding rules are total. Hence, we can consider the infinite unfolding of a program $P$ and define an unfolding semantics, in analogy with the one defined for logic programs in [29] and for NGHC in [11]. At each step $i$ of the (infinite) unfolding process of a program $P$, we have a program $P_i$ from which we can extract a reactive behavior $B_i$ for each predicate $p \in P$. The set of downward closures (or equivalence classes) of the $B_i$'s, $p \in P$, is an interpretation $I_i$ which can be viewed as a partial semantics for $P$. The collection of the interpretations which correspond to the unfoldings of $P$ is a chain in $(\mathcal{F}_p, \leq_x)$ whose least upper bound is the unfolding semantics $\mathcal{F}(P)$ of $P$.

Definition 7.2. Let $c$ be a clause. The reactive behavior associated with $c$, denoted by $\mathcal{R}(c)$, is the subtree obtained from $c$ by replacing each subtree rooted in an AND node and each procedure call by $\bot$.

Lemma 7.3. Given a clause $c \leftarrow p(X_1, \ldots, X_n) :- B$, the downward closure $\mathcal{R}(c)^{\leq}$ of $\mathcal{R}(c)$ is an interpretation for the predicate $p$.

Proof. Straightforward by definition of interpretation. $\square$

Definition 7.4. Given the program $P = \{c_1, \ldots, c_n\}$, the interpretation $I(P)$ for $P$ is $I(P) = ((\mathcal{R}(c_1))^\leq, \ldots, (\mathcal{R}(c_n))^\leq)$. 
Note that we could equivalently consider as interpretation for a predicate the equivalence class of $\mathcal{R}(c)$ w.r.t. the equivalence relation induced by the preordering $\leq$, because of the previously stated isomorphism between finite ideals and the elements of the quotient set.

**Definition 7.5 (Unfolding interpretations).** Let $P = \{c_1, \ldots, c_n\}$ be a program. The following is the definition of a collection of programs which are semantically equivalent:

$$P^1 = P,$$

$$P^{i+1} = \mathcal{U}A\mathcal{F}(P^i).$$

We define a collection of interpretations $I^1, I^2, \ldots$, where $I^i$ denotes the interpretation $I(P^i)$, $i = 2, \ldots$ and $I^1 = I(\{\mu_1(\eta_1((c_1))), \ldots, \mu_n(\eta_n((c_n)))\})$, where $\mu_1, \eta_1$ are as defined in Definition 5.3.

**Lemma 7.6.** The collection of interpretations $I^1, I^2, \ldots$ of Definition 7.5 is a chain in $(\mathcal{A}_P, \leq)$.

**Proof.** The $I^i$'s are interpretations, by Lemma 7.3. Let $I^i = (\mathcal{R}(c_j^i))^\omega, \ldots, R(c_{e_j}^i))^\omega$, $i = 1, 2, \ldots$. By definition of $\leq$, in order to prove that $I^i \leq I^{i+1}$, it is sufficient to prove that $\mathcal{R}(c_j^i) \leq \mathcal{R}(c_j^{i+1})$ for $1 \leq j \leq n$. Since $c_j^{i+1} = \text{Unf}(c_j^i, P)$, $i = 0, 1, \ldots$ the only possible difference between $c_j^i$ and $c_j^{i+1}$ lies in the fact that procedure calls in $c_j^i$ are replaced by their definitions in $c_j^{i+1}$ and some AND nodes are transformed to OR nodes. Therefore, by definition of $\mathcal{R}(c)$, for $1 \leq j \leq n$ the only difference between $\mathcal{R}(c_j^i)$ and $\mathcal{R}(c_j^{i+1})$ lies in the fact that some elements $\bot$ in $\mathcal{R}(c_j^i)$ are replaced by a subtree in $\mathcal{R}(c_j^{i+1})$. Therefore, the thesis holds, by definition of $\leq$. □

**Definition 7.7 (Unfolding semantics).** Let $P$ be a program and let $I^1, I^2, \ldots$ be the chain of interpretations of Definition 7.5. The unfolding semantics of $P$, denoted by $\mathcal{U}(P)$, is defined as $\mathcal{U}(P) = \text{lub}(\{I^1, I^2, \ldots\})$.

The following theorems show the equivalence of the operational and of the unfolding semantics. It should be clear from our discussion and the example that, given a goal, we can “execute” it in an interpretation. In fact, an interpretation is a set of ideals of reactive behaviors, each reactive behavior being a program (interpreting $\bot$ as a nonterminating process). Then we can “execute” a goal $G$ in the interpretation $I = (I_{p_1}, \ldots, I_{p_n})$ for the predicates $\{p_1, \ldots, p_n\}$, by choosing as definition of the predicate $p_i$ one reactive behavior in the ideal $I_{p_i}$. Therefore, our results show that if the evaluation of a goal $G$ in the program $P$ reaches a terminal (success, failure or deadlock) state computing an answer $\langle \Psi, \sigma_G \rangle$, the same answer (up to variable renaming) can be obtained by “executing” $G$ in the model. Let us formalize this notion.
Definition 7.8 (Execution in the model). Let $P = \{c_1, \ldots, c_n\}$ be a program such that $c_i = p_i(X_1, \ldots, X_n) :- A$ for $i = 1, 2, \ldots, n$ and let $I_P = \{I_{p_1}, \ldots, I_{p_n}\}$ be an interpretation for $P$. Then the execution of $G$ in $I_P$ is the execution of $G$ in any program $\{B_{p_1}, \ldots, B_{p_n}\}$, where $B_{p_i} \in I_{p_i}, i = 1, \ldots, n$.

Lemma 7.9. Let $I_P \subseteq J_P$. If the atomic goal $G$ terminates in $I_P$ computing the answer $\langle \Psi, \sigma_{|G} \rangle$ then $G$ terminates in $J_P$ computing the answer $\langle \Psi, \sigma'_{|G} \rangle$, which is a variant of $\langle \Psi, \sigma_{|G} \rangle$.

Proof. If $B_1$ and $B_2$ are two reactive behaviors for the predicate $p$ and $B_1 \leq B_2$, by definition of $\leq$ if $G$ terminates in $B_1$ computing the answer constraint $\langle \Psi, \sigma_{|G} \rangle$ then $G$ terminates in $B_2$ computing the variant answer $\langle \Psi, \sigma'_{|G} \rangle$. Then the thesis follows by definition of $\leq$, and by Definition 7.8 (execution in the model).

Theorem 7.10 (Equivalence of the unfolding and the operational semantics). Let $P$ be a program. If the atomic goal $G$ terminates in $P$ computing the answer $\langle \Psi, \sigma_{|G} \rangle$ then $G$ terminates in $\mathcal{U}(P)$ computing the answer $\langle \Psi, \sigma'_{|G} \rangle$, which is a variant of $\langle \Psi, \sigma_{|G} \rangle$ and vice versa.

Proof. Let $G$ be an atomic goal for a predicate $p$ defined in $P$. Let $P^1 = \mathcal{U}(\mathcal{F}(P), P^2 = \mathcal{U}(\mathcal{F}(P^1), \ldots, P^{i+1} = \mathcal{U}(\mathcal{F}(P^i)$. By Theorem 7.1, programs $P, P^1, P^2, \ldots$ are equivalent. Then by a straightforward induction on the length of the derivation, given an initial store $\pi$, a goal $G$ terminates in $P$ computing the answer $\langle \Psi, \sigma_{|G} \rangle$ (denoted by $P \vdash \langle G, \pi \rangle \rightarrow \langle \Psi, \sigma_{|G} \rangle$) iff there exists $n$ such that $G$ terminates in $P^n$ without evaluating procedure calls and without evaluating AND nodes and computing the answer $\langle \Psi, \sigma'_{|G} \rangle$, which is a variant of $\langle \Psi, \sigma_{|G} \rangle$. Let $c^*_p \in P^n$ be the (unique) clause for predicate $p$. Then we have

\[
P \vdash \langle G, \pi \rangle \rightarrow \langle \Psi, \sigma_{|G} \rangle
\]

\[
\Leftrightarrow P^n \vdash \langle G, \pi \rangle \rightarrow \langle \Psi, \sigma^1_{|G} \rangle \quad \text{(by inductive argument)}
\]

\[
\Leftrightarrow \{c^*_p\} \vdash \langle G, \pi \rangle \rightarrow \langle \Psi, \sigma^2_{|G} \rangle \quad \text{(since $c^*_p \in P^n$ is the clause for $p$)}
\]

\[
\Leftrightarrow \{\mathcal{H}(c^*_p)\} \vdash \langle G, \pi \rangle \rightarrow \langle \Psi, \sigma^3_{|G} \rangle \quad \text{(by definition of $\mathcal{H}$)}
\]

\[
\Leftrightarrow \{I(P^n)\} \vdash \langle G, \pi \rangle \rightarrow \langle \Psi, \sigma^4_{|G} \rangle \quad \text{(by Definition 7.8),}
\]

where $\sigma_{|G}, i = 1, 2, 3, 4$ are variants. Then the only-if part of the thesis holds by Lemma 7.9, since for every $n, I(P^n) \subseteq \mathcal{U}(P)$. If the part holds since $G$ terminates in $\mathcal{U}(P)$ iff it terminates in $I(P^n)$ for a suitable finite $n$. \qed

Let us show an example of the unfolding semantics construction.
Example 7.11. Let us consider a program $P$ in $cc(\mathcal{E})$, where the constraint system consists of equations over Herbrand terms and where $\tau$ denotes the empty (i.e. always satisfiable) constraint.

\[
P = \{ p(X, Y) : Y = b : \tau \rightarrow \text{nil}. \quad q(X, Y) : X = a : Y = b \rightarrow \text{nil} \quad r(X, Y) : \tau : X = a \rightarrow p(X, Y). \quad s(X, Y) : \tau : \tau \rightarrow q(X, Y) \parallel r(X, Y). \} \]

The unfolding of $P$ is

\[
P' = \{ p(x, Y) : Y = b : t \rightarrow \text{nil}. \quad q(X, Y) : X = a : Y = b \rightarrow \text{nil}. \quad r(X, Y) : \tau : X = a \rightarrow Y = b : \tau \rightarrow \text{nil}. \quad s(X, Y) : \tau : \tau \rightarrow (X = a : Y = b : \tau \rightarrow \text{nil}) \quad + (\tau : X = a \rightarrow (p(X, Y) \parallel (X = a : Y = b \rightarrow \text{nil}))). \} \]

The unfolding of $P^1$ is

\[
P^1 = \{ p(X, Y) : Y = b : \tau \rightarrow \text{nil}. \quad q(X, Y) : X = a : Y = b \rightarrow \text{nil}. \quad r(X, Y) : \tau : X = a \rightarrow Y = b : \tau \rightarrow \text{nil}. \quad s(X, Y) : \tau : \tau \rightarrow (X = a : Y = b : \tau \rightarrow \text{nil}) \quad + (\tau : X = a \rightarrow (p(X, Y) \parallel (X = a : Y = b \rightarrow \text{nil}))). \} \]

The reactive behaviors $U$, $U^1$ and $U^2$ associated with $P, P^1$ and $P^2$ are

\[
U = \{ p(X, Y) : Y = b : \tau \rightarrow \text{nil}. \quad q(X, Y) : X = a : Y = b \rightarrow \text{nil}. \quad r(X, Y) : \tau : X = a \rightarrow \bot. \quad s(X, Y) : \tau : \tau \rightarrow \bot. \} \]

\[
U^1 = \{ p(X, Y) : Y = b : \tau \rightarrow \text{nil}. \quad q(X, Y) : X = a : Y = b \rightarrow \text{nil}. \quad r(X, Y) : \tau : X = a \rightarrow Y = b : \tau \rightarrow \text{nil}. \quad s(X, Y) : \tau : \tau \rightarrow (X = a : Y = b : \tau \rightarrow \text{nil}) \quad + (\tau : X = a \rightarrow \bot). \} \]

\[
U^2 = P^2. \]
$U^2$ (the downward closure, or, equivalently, the equivalence class of $U^2$), is the unfolding semantics of $P$ which shows, that the goal $s(X, Y)$ succeeds computing the answer constraint $X = a \land Y = b$, that the goal $s(X, c)$ fails and that the goal $r(X, Y)$ results in a deadlock.

8. Fixpoint semantics

The immediate consequences operator $T_{up}(I)$ on the cpo $(\mathcal{S}_P, \subseteq_x)$ of interpretations, can be defined in a quite natural way by means of unfolding (by interpreting $\bot$ as a procedure call). In what follows for the sake of simplicity, we omit parentheses when this does not cause any ambiguity.

Definition 8.1. Let $P = \{c_{p_1}, c_{p_2}, \ldots, c_{p_n}\}$ be a program, where $c_{p_i}$ is the clause for the predicate $p_i$. Let $J = (J_1, J_2, \ldots, J_n)$ be an interpretation for $P$, where $J_i$ is the interpretation for predicate $p_i$, $i = 1, \ldots, n$. The mapping $T_{up}$ on the set of interpretations of $P$ is defined as follows:

$$T_{up}(J) = (J_{B_1}, J_{B_2}, \ldots, J_{B_n}),$$

where $J_{B_i} = \bigcup_{T \in (J_{B_1}, \ldots, J_{B_i-1})} \mathcal{R}(\text{Unf}(c_i, T))^\times$, $i = 1, \ldots, n$.

Moreover, $\bot_P$ is the interpretation

$$\{p_1(\bar{X}_1) :- \bot, \ldots, p_n(\bar{X}_n) :- \bot\}^\times,$$

which is the bottom element of $(\mathcal{S}_P, \subseteq_x)$.

Theorem 8.4 allows one to define the fixpoint semantics of $cc(\%)$ programs in the standard logic programming style. Let us first introduce a definition and a lemma. The proof of Lemma 8.3 is in the appendix.

Definition 8.2. Let $P$ be a program. Then $\tilde{\rho}_P(c)$ is the transformation which replaces each procedure call $p(Y_1, \ldots, Y_n)$ in the clause $c$ by $Ax$, where $p(X_1, \ldots, X_n) :- A \in P$ is the clause for predicate $p$ and $x = \{X_1 / Y_1, \ldots, X_n / Y_n\}$. $\tilde{\eta}_P(A)$ and $\tilde{\mu}_P(P')$ denote the extensions of $\tilde{\rho}_P$ applied to agents and programs. Moreover, $\eta_P$ and $\mu_P$ denote the extensions of $\tilde{\eta}_c, \mu_c$ (Definition 5.3) to programs.

Lemma 8.3. Let $P = \{c_1, \ldots, c_m\}$ be a program, where $c_h$ is the clause for predicate $p_h$, and let $T = (t_1, \ldots, t_m)$, $T' = (t'_1, \ldots, t'_m)$, where $t_h, t'_h$ are reactive behaviors for the predicate $p_h$ and $t_h \leq t'_h$, $h = 1, 2, \ldots, m$. Then $\mathcal{R}(\text{Unf}(c_h, T)) \leq \mathcal{R}(\text{Unf}(c_h, T'))$, $h = 1, 2, \ldots, m$.

Theorem 8.4. Let $P$ be a program. $T_{up}$ is continuous on $(\mathcal{S}_P, \subseteq_x)$. Hence, there exists the least fixpoint of $T_{up}$:

$$\text{lf}_{P}(T_{up}) = \text{lub}_{n \in \mathbb{N}} T_{up}^n(\bot_P) = T_{up}^{\uparrow_P}.$$
Proof. If $I$ is an interpretation, $T_{UP}(I)$ is an interpretation by definition of $T_{UP}$. We prove the continuity of the $T_{UP}$ operator. Then the existence of the fixpoint and its construction follow from the well-known general results of lattice theory. Let us consider, for the sake of simplicity, the case of one predicate $p$ only, defined by the clause $c_p$ (the extension to the general case is straightforward). Let $I_1 \subseteq I_2 \subseteq \cdots$ be a chain in $(\mathcal{P}^+, \subseteq)$. We must prove that $T_{UP}(\bigcup_i I_i) = \bigcup_i T_{UP}(I_i)$.

By definition of $T_{UP}$ and by definition of unfolding rules, we have

\[
B \in T_{UP} \left( \bigcup_i I_i \right)
\]

\[
\Leftrightarrow B \in \bigcup_{T \in I_n} \mathcal{P}(Unf(T)) \quad \text{(by Definition 8.1)}
\]

\[
\Leftrightarrow B \in \bigcup_{T \in I_n} \mathcal{P}(Unf(T)) \quad \text{(by Definitions 5.3, 6.6 and Lemma 8.3)}
\]

\[
\Leftrightarrow B \in T_{UP}(I_n) \quad \text{(by Definition 8.1)}
\]

\[
\Leftrightarrow B \in \bigcup_i T_{UP}(I_i) \quad \text{(by Definition of \( \bigcup \))}
\]

and the thesis holds. \hfill \square

Definition 8.5 (Fixpoint semantics). The fixpoint semantics $\mathcal{F}(P)$ of a program $P$ is the least fixpoint of the transformation $T_{UP}$ associated with $P$.

Let us show an example of fixpoint construction.

Example 8.6. If $P$ is the program of Example 7.11, the interpretations obtained by $T_{UP}$ are the equivalence classes of the reactive behaviors $F_0, F_1, F_2, F_3, F_4$:

$F_0 = \{ p(X, Y) :- \bot. \}$

$\quad q(X, Y) :- \bot. \$

$\quad r(X, Y) :- \bot. \$

$\quad s(X, Y) :- \bot. \}$

$F_1 = \{ p(X, Y) :- Y = b : \tau \rightarrow \text{nil}. \}$

$\quad q(X, Y) :- X = a : Y = b \rightarrow \text{nil}. \$

$\quad r(X, Y) :- \tau : X = a \rightarrow \bot. \$

$\quad s(X, Y) :- \tau : \tau \rightarrow \bot. \}$
\[ F_2 = \{ p(X, Y) : Y = b : \tau \rightarrow \text{nil}. \]
\[ q(X, Y) : X = a : Y = b : \tau \rightarrow \text{nil}. \]
\[ r(X, Y) : \tau : X = a \rightarrow Y = b : \tau \rightarrow \text{nil}. \]
\[ s(X, Y) : \tau : (X = a : Y = b : \tau : X = a : \bot) \]
\[ + (\tau : X = a : \bot). \}, \]
\[ F_3 = \{ p(X, Y) : Y = b : \tau \rightarrow \text{nil}. \]
\[ q(X, Y) : X = a : Y = b : \tau \rightarrow \text{nil}. \]
\[ r(X, Y) : \tau : X = a \rightarrow Y = b : \tau \rightarrow \text{nil}. \]
\[ s(X, Y) : \tau : (X = a : Y = b : \tau : X = a : \bot) \]
\[ + (\tau : X = a : Y = b : \tau : X = a : \bot). \}, \]
\[ F_4 = F_3, \]
\[ F_4 = U_2^\tau \text{ of Example 7.11} \] is the fixpoint semantics of \( P \).

The following theorem, whose proof is in the appendix, shows the equivalence between the fixpoint and the unfolding semantics.

**Theorem 8.7** (Equivalence of the fixpoint and the unfolding semantics). Let \( P \) be a program. Then \( \#(P) = \mathcal{F}(P) \).

**Corollary 8.8** (Equivalence of the fixpoint and the operational semantics). Let \( P \) be a program. If the goal \( G \) terminates in \( P \) computing the answer \( \langle \Psi, \sigma_G \rangle \) then \( G \) terminates in \( \mathcal{F}(P) \) computing the answer \( \langle \Psi, \sigma_G \rangle \), which is a variant of \( \langle \Psi, \sigma_G \rangle \) and vice versa.

**Proof.** It is a straightforward consequence of Theorems 8.7 and 7.1. \( \square \)

**9. Related and future work**

The most relevant papers on the semantics of concurrent constraint languages are [20, 9, 10, 41]. The semantics definition method adopted by all of the above papers is quite different from ours and can roughly be described as “let us define a suitable abstraction on the description of all the possible program computations”. Our method instead is a real generalization of the fixpoint semantics based on an immediate consequence operator on interpretations, which is typical of logic programming. As already noted, the construction in [20] was inspired by our unfolding approach (already described in [11]), but has a different overall goal, i.e. that of achieving full
abstraction. This is why the semantics is given in terms of sequences of ask and tell constraints (extended with some additional information for failures and deadlocks). A better full abstraction result in terms of sequences of input and output constraints (or substitutions) is achieved in [9, 10], where input constraints are related to (but different from) ask constraints. Finally, [41] uses tree-structured semantic domains very similar to ours in an operational and denotational framework. All of the above papers are strongly concerned with the problem of defining a suitable notion of equivalence on semantic objects.

This problem is not considered at all in this paper. In fact, our notions of equivalence are based on the definition implicit in Theorem 7.1, according to which two equivalent programs have identical answers (modulo variance). This is, in general, not fully satisfactory, since two constraints $c_1$ and $c_2$ are equivalent if they have the same set of solutions. It is worth noting that, according to the constraint logic programming approach [25, 24, 18], one can define both the operational and the declarative semantics by using the notion of consistency only. This is exactly what we have done in the previous sections. However, in real constraint languages we look for suitable transformations on constraints, which allow us to keep reasonable the size of the constraint and to return meaningful answers. One example of such a transformation is the transformation to solved form on sets of equations over the Herbrand Universe [27], which corresponds to the computation of the idempotent most general unifiers. The transformations become the essence of computing (apart from building consistent constraints) and, following the approach in [12–14], they must be taken into account in the equivalence definition. Therefore, if we want to model what we "effectively compute" by means of our operational semantics, we must consider equivalent two constraints $c_1$ and $c_2$ only if there exists an (equivalence-preserving) transformation $\phi$, such that $\phi(c_1)$ is a variant of $\phi(c_2)$ and the transformation $\phi$ is used by the operational semantics to "simplify" the current constraint.

The above argument applies to the constraints which occur inside predicate interpretations. However, it can be applied to predicate interpretations as well. Namely, stronger equivalence relations on predicate interpretations can be defined through equivalence-preserving transformations on trees. For example, it is very easy to adapt to our framework the failure set-like equivalence described in [20]. A similar construction is trivial for the equivalences in [41], which are defined on tree structures as is the case of ours. Our general framework allows us to define a fixpoint semantics for concurrent constraint programs, without being concerned with the equivalence notions on the behaviors. Any reasonable equivalence notion can be applied to the resulting semantics. We are then applying, even to the "hard" concurrency aspects, the CLP approach of handling the model construction and the equivalences separately.

This suggests that tree expressions occurring in predicate interpretations could be viewed as constraints, with their own notions of solutions and consistency and their transformations. We are currently investigating this problem, whose solution would allow us to obtain a model-theoretic semantics as well.
Appendix A

In this appendix, we give the proofs of some technical lemmata and theorems concerning the equivalence results for the unfolding and the fixpoint semantics.

A.1 Equivalence between $\Psi$ and $\Psi_{\mathcal{A}^t F}(P)$

We first need some definitions and lemmata. In what follows, we assume that $P \vdash \gamma \rightarrow R \gamma'$ means that, in program $P$, the configuration $\gamma$ can be transformed in the configuration $\gamma'$ by applying the rule $R$, where $R \in \{1, 2, ..., 12\}$ is one of the transition rules of Definition 4.3.

**Definition A.1.** Let $\langle A_1, \sigma_1 \rangle$ and $\langle A_2, \sigma_2 \rangle$ be two configurations. $\langle A_1, \sigma_1 \rangle$ and $\langle A_2, \sigma_2 \rangle$ are variants iff $\langle A_1, \sigma_1 \rangle = \langle A_2, \sigma_2 \rangle \rho$ (i.e. $A_2 = A_1 \rho$ and $\sigma_2 = \sigma_1 \rho$), where $\rho$ is a renaming. Consider the functions $\eta_\alpha, \rho_\alpha$ and $\mu_\alpha$ defined by Definition 5.3 and let $\rho_\alpha$, $\mu_\alpha$ be the restriction of $\rho_\alpha$, $\mu_\alpha$, respectively, to agents. Then we define the $\approx_{\rho}$ and $\approx_{\mu}$ equivalence relations as follows. If $f \in \{\eta, \rho, \mu\}$ then $\langle A_1, \sigma_1 \rangle \approx_f \langle A_2, \sigma_2 \rangle$ iff one of the following cases holds:

1. $\langle A_1, \sigma_1 \rangle$ and $\langle A_2, \sigma_2 \rangle$ are variants;
2. $\langle \eta_\alpha(A_1), \sigma_1 \rangle$ and $\langle A_2, \sigma_2 \rangle$ are variants;
3. $\langle \rho_\alpha(A_2), \sigma_2 \rangle$ and $\langle A_1, \sigma_1 \rangle$ are variants.

**Lemma A.2.** Let $\gamma'$ and $\gamma'_1$ be two configurations such that $\gamma' \approx_{\rho} \gamma'_1$. Let $P = \{c_1, ..., c_n\}$ be a program and let $P_1 = \{\rho_\alpha(c_1), ..., \rho_\alpha(c_n)\}$, where $\rho_\alpha$ is defined by Definition 5.3. Then $P \vdash \gamma \rightarrow R \gamma'$ iff $P_1 \vdash \gamma_1 \rightarrow R \gamma'_1$, where $\gamma \approx_{\rho} \gamma'_1$.

**Proof.** Let us suppose, without loss of generality, that $\gamma' = \langle A, \sigma \rangle$ and $\gamma'_1 = \langle \rho_\alpha(A), \sigma \rangle \beta$, where $\beta$ is a renaming (if $\gamma$ and $\gamma_1$ are variants, we can apply the same arguments of the proof to $A \beta$ instead of $\eta_\alpha(A) \beta$).

Note that every inference rule can be represented as a tree with two nodes if the rule has a single premise, or three nodes if the rule has two premises. The root of the tree is labeled with the conclusion of the rule and the leaves are labeled with the premises. The premises can be conclusions of other rules. Therefore to each inference step we can associate the proof tree representing all the rules needed for such a step. Note that, because of rules definition, all the trees we consider are finite. We can then define the depth of a proof tree as the length of the longest leaf-root path in the tree. We prove the thesis by induction on the depth $d$ of the proof tree for any transition for any agent $A$ (together with the fact, necessary for the induction, that $d$ is also the depth of the proof tree for $\rho_\alpha(A)$).

**Base case:*** We have the following cases. If $A = \text{nil}$, the thesis follows by definition of Rule 1. If $A = \text{ask} : \text{tell} \rightarrow A_1$, by definition of $\rho_\alpha, \rho_\alpha(A) = \text{ask} : \text{tell} \rightarrow \rho_\alpha(A_1)$. Then for $R \in \{2, 3, 4\}$, $P \vdash \langle \text{ask} : \text{tell} \rightarrow A_1, \sigma \rangle \rightarrow R \gamma$ iff $P_1 \vdash \langle \text{ask} : \text{tell} \rightarrow \rho_\alpha(A_1), \sigma \rangle \beta \rightarrow R \gamma_1$, where $\gamma \approx_{\rho} \gamma_1$, since by definition of $P$, and by Definition 4.3, if $R \in \{3, 4\}$ then $\gamma = \langle \Psi, \sigma \rangle$ and $\gamma_1 = \langle \Psi, \sigma \beta \rangle$, where $\Psi \in \{\text{fail, suspend}\}$. If $R = 2$, by definition of Rule 2), $\gamma = \langle A_1, \sigma \rangle$.\end{document}
and $\gamma_1 = \langle \rho(A_1), \sigma \rangle \beta$. Note that the depth of the proof tree for $\rho_\sigma(A)$ is 1 in all these cases.

**Inductive case:** If Rule 12 is used, $\rho_\sigma(A) = Bx$, where $p(Y_1, \ldots, Y_n) : - B \in P$ and $\alpha = \{ Y_1/X_1, \ldots, Y_n/X_n \}$. Moreover, by definition of $P_1$, $p(Y_1, \ldots, Y_n) : - \rho_\sigma(B) \in P_1$. Then

$$P_1 \vdash \langle p(X_1, \ldots, X_n), \sigma \rangle \rightarrow_{12} \gamma$$

$\Leftrightarrow$  

$$P_1 \vdash \langle Bz, \sigma \rangle \rightarrow_{R} \gamma$$

(by definition of Rule 12)

$\Leftrightarrow$  

$$P_1 \vdash \langle Bz, \sigma \rangle \beta \rightarrow_{R} \gamma \beta$$

(by definition of $\beta$)

$\Leftrightarrow$  

$$P_1 \vdash \langle Bz, \sigma \rangle \beta \rightarrow_{R} \gamma \beta$$

(by inductive hypothesis)

where $\gamma \approx_p \gamma_1$.

If $A = A_1 + A_2$ then we have by definition of $\rho_\sigma$ that $\rho_\sigma(A_1 + A_2) = \rho_\sigma(A_1) + \rho_\sigma(A_2)$. Then by definition of Rules 5–7 and by inductive hypothesis $P_1 \vdash \langle A_1 + A_2, \sigma \rangle \rightarrow_{R} \gamma$ iff $P_2 \vdash \langle \rho_\sigma(A_1) + \rho_\sigma(A_2), \sigma \rangle \beta \rightarrow_{R} \gamma_1$, where $\gamma \approx_p \gamma_1$ and $R \in \{5, 6, 7\}$.

If $A = A_1 \cdot A_2$ then $\rho_\sigma(A_1 \cdot A_2) = \rho_\sigma(A_1) \cdot \rho_\sigma(A_2)$ and, by inductive hypothesis and rules definition, $P_1 \vdash \langle A_1 \cdot A_2, \sigma \rangle \rightarrow_{R} \gamma$ iff $P_1 \vdash \langle \rho_\sigma(A_1), \rho_\sigma(A_2), \sigma \rangle \beta \rightarrow_{R} \gamma_1$, where $\gamma \approx_p \gamma_1$ and $R \in \{8, 9, 10, 11\}$. Note that if, by inductive hypothesis, $d$ is the depth of $A_i$ and $\rho_\sigma(A_i)$, $i = 1, 2$, then $d + 1$ is the depth of $A_1 \cdot A_2$ and $\rho_\sigma(A_i \cdot A_j)$. Analogously in the previous cases. □

**Lemma A.3.** Let $\langle A_1 \parallel A_2, \sigma \rangle$ be a configuration such that $A_1 = \sum_{i \in I} a_i : t_i \rightarrow A_i$ and $A_2 = \sum_{j \in J} a'_j : t'_j \rightarrow A'_j$, $I, J \neq \emptyset$. Let $P_1 = \{ c_1, \ldots, c_n \}$ be a program and let $P_2 = \{ \eta(c_1), \ldots, \eta(c_n) \}$, where $\eta_i$ is defined by Definition 5.3. Then

$$P_1 \vdash \langle A_1 \parallel A_2, \sigma \rangle \rightarrow_{\gamma_1}$$

$\Leftrightarrow$  

$$P_2 \vdash \left( \sum_{i \in I} a_i : t_i \rightarrow \eta_\sigma(A_i \parallel A_2) \right) + \left( \sum_{j \in J} a'_j : t'_j \rightarrow \eta_\sigma(A_1 \parallel A'_j) \right) \rho \rightarrow_{\gamma_2}$$

where $\gamma_1 \approx_\rho \gamma_2$.

**Proof.** The only rules that can be applied to $A_1 \parallel A_2$ are Rules 8–11 (Definition 4.3). Let us consider the case of Rule 11. Then, by Definition 4.3 and by definition of $P_2$,

$$P_1 \vdash \langle A_1 \parallel A_2, \sigma \rangle \rightarrow_{11} \langle A_1 \parallel A_2, \sigma \rangle \quad A_i \neq \text{true, fail, suspend}$$

$\Leftrightarrow$  

$$P_1 \vdash \langle A_1, \sigma \rangle \rightarrow_{5} \langle A_1, \sigma \rangle$$

$\Leftrightarrow$  

$$\exists i \in I \text{ s.t. } P_1 \vdash \langle a_i : t_i \rightarrow A_i, \sigma \rangle \rightarrow_{2} \langle A_1, \sigma \rangle$$

$\Leftrightarrow$  

$$\delta(\text{ask} : \text{tell}, \sigma) = \sigma_1,$$  

$$\sigma_1 \neq \text{fail, suspend}$$

$\Leftrightarrow$  

$$\delta(\text{ask} : \text{tell}, \sigma) \rho = \sigma_1 \rho,$$  

$$\sigma_1 \rho \neq \text{fail, suspend}$$

$\Leftrightarrow$  

$$\exists i \in I \text{ s.t. } P_2 \vdash \langle a_i : t_i \rightarrow \eta_\sigma(A_i \parallel A_2), \sigma \rangle \rho \rightarrow_{2} \langle \eta_\sigma(A_i \parallel A_2), \sigma \rangle \rho$$

$\Leftrightarrow$  

$$P_2 \vdash \left( \sum_{i \in I} a_i : t_i \rightarrow \eta_\sigma(A_i \parallel A_2) \right) + \left( \sum_{j \in J} a'_j : t'_j \rightarrow \eta_\sigma(A_1 \parallel A'_j), \sigma \right) \rho \rightarrow_{5} \langle \eta_\sigma(A_1 \parallel A_2), \sigma \rangle \rho.$$
Lemma A.4. Let $\gamma'_1$ and $\gamma'_2$ be two configurations such that $\gamma'_1 \approx \gamma'_2$. Let $P_1 = \{c_1, \ldots, c_n\}$ be a program and let $P_2 = \{\eta(c_1), \ldots, \eta(c_n)\}$, where $\eta$ is defined by Definition 5.3. Then $P_1 \vdash \gamma'_1 \rightarrow_{R} \gamma'_2$ iff $P_2 \vdash \gamma'_3 \rightarrow_{R} \gamma'_4$, where $\gamma'_1 \approx \gamma'_2$.

Proof. If $\gamma'_1$ and $\gamma'_2$ are variants, the thesis follows in a straightforward way from the definition of transition rules. Then let us suppose, without loss of generality, that $\gamma'_1 = \langle A, \sigma \rangle$ and $\gamma'_2 = \langle \eta(A), \sigma \rangle \rho$, where $\rho$ is a renaming. The proof is by induction on the depth of the proof tree for $A$, defined as in the proof of Lemma A.2.

Base case: If $A = \text{nil}$, the thesis holds by definition of Rule 1.

If $A = \text{ask} : \text{tell} ightarrow A_1, \eta(A) = \text{ask} : \text{tell} \rightarrow \eta(A_1)$ by definition of $\eta$. Then for $R \in \{2, 3, 4\}$, $P_1 \vdash \langle \text{ask} : \text{tell} \rightarrow A_1, \sigma \rangle \rightarrow_{R} \gamma'_1$ iff $P_2 \vdash \langle \text{ask} : \text{tell} \rightarrow \eta(A_1), \sigma \rangle \rho \rightarrow_{R} \gamma'_2$, where $\gamma'_1 \approx \gamma'_2$, since by definition of $P_1, P_2$ and by Definition 4.3, if $R \in \{2, 3, 4\}$ then $\gamma'_1 = \langle \Psi, \sigma \rangle$ and $\gamma'_2 = \langle \Psi, \sigma \rho \rangle$, where $\Psi \in \{\text{fail}, \text{suspend}\}$. If $R = 2$ then $\gamma'_1 = \langle A_1, \sigma \rangle$ and $\gamma'_2 = \langle \eta(A_1), \sigma \rangle \rho$. Note that the depth of the proof tree for $\eta(A)$ is 1 in all previous cases.

Inductive case: Let $A = A_1 + A_2$. By definition of $\eta$, we have $\eta(A_1 + A_2) = \eta(A_1) + \eta(A_2)$. Then by definition of Rules 5–7 and by inductive hypothesis, $P_1 \vdash \langle A_1 + A_2, \sigma \rangle \rightarrow_{R} \gamma'_1$ iff $P_2 \vdash \langle \eta(A_1) + \eta(A_2), \sigma \rangle \rho \rightarrow_{R} \gamma'_2$, where $\gamma'_1 \approx \gamma'_2$ and $R \in \{5, 6, 7\}$.

If $A = A_1 \parallel A_2$ and either $A_1 \neq \sum_{i \in I} a_i : t_i \rightarrow A_i$ or $A_2 \neq \sum_{j \in J} a'_j : t'_j \rightarrow A'_j$ then $\eta(A_1 \parallel A_2) = \eta(A_1) \parallel \eta(A_2)$ by inductive hypothesis and by rules definition $P_1 \vdash \langle A_1 \parallel A_2, \sigma \rangle \rightarrow_{R} \gamma'_1$ iff $P_2 \vdash \langle \eta(A_1) \parallel \eta(A_2), \sigma \rangle \rho \rightarrow_{R} \gamma'_2$, where $\gamma'_1 \approx \gamma'_2$ and $R \in \{8, 9, 10, 11\}$.

If $\eta(A_1 \parallel A_2) = (\sum_{i \in I} a_i : t_i \rightarrow \eta(A_1 \parallel A_2)) + (\sum_{j \in J} a'_j : t'_j \rightarrow \eta(A_1 \parallel A_2))$ then the thesis follows by Lemma A.3.

If Rule 12 is used, $\eta(A) = A$ by definition of $\eta$. By definition of $P_2$, if $p(Y_1, \ldots, Y_n) : B \in P_1$ then $p(Y_1, \ldots, Y_n) : \eta(B) \in P_2$. Let $x = \{Y_1/X_1, \ldots, Y_n/X_n\}$.

$P_1 \vdash \langle p(X_1, \ldots, X_n), \sigma \rangle \rightarrow_{12} \gamma'_1$ (by definition of Rule 12)

$P_2 \vdash \langle \eta(B) \circ \sigma \rangle \rho \rightarrow_{R} \gamma'_2$. (by inductive hypothesis)

with $\gamma'_1 \approx \gamma'_2$. Note that if $d$ is the depth for $B$ and $\eta(B)$, then $d + 1$ is the depth of both $A$ and $\eta(A) (= p(X_1, \ldots, X_n))$. Analogously in the other cases. This completes the proof. □

Theorem 7.1 (Equivalence of $\mathcal{P}$ and $\mathcal{F}(P)$). Let $P$ be a program. $P$ and $\mathcal{F}(P)$ are equivalent w.r.t. the operational semantics, i.e. if the goal $G$ with initial store $\pi$ terminates in $P$ computing the answer $\langle \Psi, \sigma_{G} \rangle$ then $G$ terminates in $\mathcal{F}(P)$ computing the answer $\langle \Psi, \sigma_{G} \rangle$, with $\langle \Psi, \sigma_{G} \rangle$ a variant of $\langle \Psi, \sigma_{G} \rangle$ and vice versa.
Proof. Let $P=\{c_1, ..., c_n\}$, where $c_i, i=1, ..., n$, is the clause for predicate $p_i$. Let $P_1, P_2$ and $P_3 = \mathcal{A}(\mathcal{T}(P))$ be the programs resulting from the application of steps (i)-(iii) of the unfolding procedure to the original program $P$, i.e. $P_1 = \{\rho_i(c_1), ..., \rho_i(c_n)\}$, $P_2 = \{\eta_i(\rho_i(c_1)), ..., \eta_i(\rho_i(c_n))\}$, $P_3 = \{\mu_i(\eta_i(\rho_i(c_1))), ..., \mu_i(\eta_i(\rho_i(c_n)))\}$.

(1) Equivalence of $P$ and $P_1$. By definition of $\approx_\rho$, the equivalence of $P$ and $P_1$ can be proved by showing that, given an initial configuration $\gamma$, $P \triangleright \gamma \rightarrow^* \gamma_0$ iff $P_1 \triangleright \gamma \rightarrow^* \gamma_1$ with $\gamma_0 \approx_\rho \gamma_1$. The proof is straightforward by induction on the length of the derivation by using Lemma A.2.

(2) Equivalence of $P_1$ and $P_2$. Analogously to the previous case, for the definition of $\approx_\eta$ the equivalence of $P_1$ and $P_2$ can be proved by showing that, given an initial configuration $\gamma$, $P_1 \triangleright \gamma \rightarrow^* \gamma_1$ iff $P_2 \triangleright \gamma \rightarrow^* \gamma_2$ with $\gamma_1 \approx_\eta \gamma_2$. Again the proof is straightforward by induction on the length of the derivation by applying Lemma A.4.

(3) Equivalence of $P_2$ and $P_3$. The equivalence between $P_2$ and $P_3$ follows from the definition of $P_3$ and from the definition of the transition rules (Definition 4.3). Indeed, let $s \rightarrow a : t$ be a failed or deadlocked path (Definition 5.1) in a clause $\eta_i(\rho_i(c_1))$ of $P_2$. By definition of $\mu_i$, the path $s \rightarrow a : t \rightarrow A$ is replaced by $s \rightarrow a : t \rightarrow \text{nil}$ in the clause $\mu_i(\eta_i(\rho_i(c_1))) \in P_3$. Then, by definition of deadlocked and failed path and by definition of transition Rules 2-4, $\triangleright_\eta \rightarrow^* \gamma_2$ iff $P_3 \triangleright \gamma \rightarrow^* \gamma_3$ with $\gamma_2 \approx_\eta \gamma_3$ and the thesis holds. \&

A.2. Equivalence between $\mathcal{H}(P)$ and $\mathcal{F}(P)$

Lemma 8.3. Let $P=\{c_1, ..., c_m\}$ be a program, where $c_i$ is the clause for predicate $p_h$, and let $T=(t_1, ..., t_m)$, $T'=(t'_1, ..., t'_m)$, where $t_h, t'_h$ are reactive behaviors for the predicate $p_h$ and $t_h \leq t'_h$, $h=1, 2, ..., m$. Then $\mathcal{A}(\text{Unf}(c_h, T)) \leq \mathcal{A}(\text{Unf}(c_h, T'))$, $h=1, 2, ..., m$.

Proof. Assume $1 \leq h \leq m$. By Definition 5.3 (unfolding rules), $\text{Unf}(c_h, T) = \mu_i(\eta_i(\rho_T(c_h)))$, where $\rho_T$ is defined by Definition 8.2. Let $t_h = p_h(X) : B$ and $t'_h = p_h(X) : B'$ be two renamed versions of the reactive behaviors such that the heads are equal. Since $t_h \leq t'_h$, $B \leq B'$ (Definition 6.2). In what follows, $\equiv$ will denote the extension of $\equiv$ which considers also procedure calls (on which $\equiv$ is just a reflexive relation), and AND nodes ($A \ll A_1, A_2$ iff $A_1 \ll A_2$). Let $\rho_T(c_h) = t_h = p_h(X) : C$ and $\rho_T(c_h) = t'_h = p_h(X) : C'$, where we assume the procedure calls to be replaced by the same renamed version of reactive trees bodies. By definition of $\rho_T : C \subseteq C'$. Moreover, since $T$ and $T'$ contain reactive behaviors (i.e. do not contain AND nodes), if $D_1 \equiv D_2$ is in $C$, then $D_1 \equiv D_2$ is in $C'$ and $D_1 \equiv D_2$, $i=1, 2$. Now let us prove, by structural induction on $C$, that $\mathcal{A}(\eta_n(C)) \leq \mathcal{A}(\eta_n(C'))$.

For the base case we have the following possibilities:

1. If $C = \text{nil}$ then, by definition of $\equiv$ (Definition 6.2), $C' = \text{nil}$.
2. If $C = q(X)$ then, since $C \subseteq C'$, by Definition 6.2, $C' = q(X)$.
3. If $C = \bot$ then, by definition of $\equiv$, for any $C'$, $\mathcal{A}(\eta_n(\bot)) = \bot \leq \mathcal{A}(\eta_n(C'))$.
Thus, the thesis holds for the base case.

For the inductive case we have the following possibilities:

1. If $C = a: t \rightarrow C_1$, by definition of $\sqsubseteq$, $C' = a: t \rightarrow C_1'$. Then

\[
\mathcal{R}(\eta_a(C)) = a: t \rightarrow \mathcal{R}(\eta_a(C_1)) \quad \text{(by Definitions 7.2 and 8.2)}
\]

\[
\sqsubseteq a: t \rightarrow \mathcal{R}(\eta_a(C_1')) \quad \text{(by inductive hypothesis and by Definition of $\sqsubseteq$)}
\]

\[
= \mathcal{R}(\eta_a(C')) \quad \text{(by Definitions 7.2 and 8.2)}
\]

and the thesis holds.

2. If $C = C_1 + C_2$, then, by definition of $\sqsubseteq$, $C' = C_1' + C_2'$. Then

\[
\mathcal{R}(\eta_a(C)) = \mathcal{R}(\eta_a(C_1)) + \mathcal{R}(\eta_a(C_2)) \quad \text{(by Definitions 7.2 and 8.2)}
\]

\[
\sqsubseteq \mathcal{R}(\eta_a(C_1')) + \mathcal{R}(\eta_a(C_2')) \quad \text{(by inductive hypothesis and by definition of $\sqsubseteq$)}
\]

\[
= \mathcal{R}(\eta_a(C_1' + C_2')) \quad \text{(by Definitions 7.2 and 8.2)}
\]

\[
= \mathcal{R}(\eta_a(C')) \quad \text{(by definition of $C'$)}
\]

and the thesis holds.

3. If $C = C_1 \parallel C_2$ then we have two cases:

   i. If either $C_1 \neq \sum_{i \in I} a_i : t_i \rightarrow C_1$ or $C_2 \neq \sum_{j \in J} a_j' : t_j' \rightarrow C_2$, then, by definition of $\eta_a$ (Definition 5.3), $\eta_a(C) = \eta_a(C_1) \parallel \eta_a(C_2)$ and the proof is similar to the one in case 2.

   ii. If $C_1 = \sum_{i \in I} a_i : t_i \rightarrow C_1$, and $C_2 = \sum_{j \in J} a_j' : t_j' \rightarrow C_2$, then, by the hypothesis on $C$ and $C'$, $C' = C_1' \parallel C_2'$, where $C_1' = \sum_{i \in I} a_i : t_i \rightarrow C_1'$, $C_2' = \sum_{j \in J} a_j' : t_j' \rightarrow C_2'$, and $C_1_i \sqsubseteq C_1_i'$, $C_2_i \sqsubseteq C_2_i'$, for $i \in I, j \in J$. Then

\[
\mathcal{R}(\eta_a(C)) = \mathcal{R}\left( \left( \sum_{i \in I} a_i : t_i \rightarrow \eta_a(C_1) \parallel C_2 \right) + \left( \sum_{j \in J} a_j' : t_j' \rightarrow \eta_a(C_1') \parallel C_2' \right) \right)
\]

(by definition of $\eta_a$)

\[
= \mathcal{R}\left( \sum_{i \in I} a_i : t_i \rightarrow \eta_a(C_1) \parallel C_2 \right) + \mathcal{R}\left( \sum_{j \in J} a_j' : t_j' \rightarrow \eta_a(C_1') \parallel C_2' \right)
\]

(by definition of $\mathcal{R}$)

\[
\sqsubseteq \mathcal{R}\left( \sum_{i \in I} a_i : t_i \rightarrow \eta_a(C_1) \parallel C_2 \right) + \mathcal{R}\left( \sum_{j \in J} a_j' : t_j' \rightarrow \eta_a(C_1') \parallel C_2' \right)
\]

(by inductive hypothesis and definition of $\sqsubseteq$)
= \mathcal{R} \left( \sum_{i \in I} a_i \cdot t_i \rightarrow \eta_a(C_1, \parallel C_2) \right) + \left( \sum_{j \in J} a_j' \cdot t_j' \rightarrow \eta_a(C_1', \parallel C_2') \right)

\text{(by definition of } \mathcal{R})

= \mathcal{R}(\eta_a(C')) \quad \text{(by definition of } \eta_a \text{ and by definition of } C')

and the thesis holds.

Since we have considered all the cases, we have proved that \( \mathcal{R}(\eta_a(C)) \subseteq \mathcal{R}(\eta_a(C')) \) and, hence, by definition of \( \eta_a \), that \( \mathcal{R}(\eta_a(t_k)) \subseteq \mathcal{R}(\eta_a(t_k')) \). By definition of } \subseteq \text{ and since } \mathcal{R}(\eta_a(t_k)) \subseteq \mathcal{R}(\eta_a(T_m)), \text{ if } s \rightarrow a : t \text{ is either a failed or a deadlocked reactive path (f-d-path) in } \eta_a(t_k), \text{ then } s \rightarrow a : t \text{ is an f-d-path in } \eta_a(t_k'). \text{ Conversely, if } s \rightarrow a : t \text{ is an f-d-path in } \eta_a(t_k), \text{ then either } s \rightarrow a : t \text{ is an f-d-path in } \eta_a(t_k') \text{ or } s' \rightarrow \bot \text{ is a reactive path in } \eta_a(t_k'). \text{ where } s' \text{ is contained in } s. \text{ Then, by Definition 5.3, } \mu_\rho \text{ replaces } s \rightarrow a : t \text{ by } s \rightarrow a : t \rightarrow \text{nil} \text{, if either } \mu_\rho \text{ replaces } s \rightarrow a : t \rightarrow A \text{ of } \eta_a(t_k) \text{ by } s \rightarrow a : t \rightarrow \text{nil}, \text{ or } s \rightarrow \bot \text{ is in } \eta_a(t_k). \text{ Therefore, } \mathcal{R}(\mu_\rho(\eta_a(t_k))) \subseteq \mathcal{R}(\mu_\rho(\eta_a(t_k'))) \text{ and, by definition of } \leq \text{ and of } \text{ and } \text{ since } \mu_\rho(\eta_a(t_k')) \text{ and this completes the proof. } \square

Lemma A.5. Let } A = A_1 \parallel A_2 \text{ and let } m \text{ be such that } \eta^m_a(A) = \eta^{m+1}_a(A). \text{ Then } \eta^m_a(A_i) = \eta^{m+1}_a(A_i), i = 1, 2 \text{ and } \eta^m_a(A) = \eta^m_a(\eta^m_a(A_1) \parallel \eta^m_a(A_2)).

Proof. The lemma is a straightforward consequence of the definitions of } \eta_a \text{ and of the assumption on } m. \square

Lemma A.6. Let } P, P_1 \text{ be programs and let } \mu_\rho, \eta_\rho \text{ and } \tilde{\rho}_P \text{ be as defined in Definition 8.2. Then } \mu_\rho \eta_\rho \tilde{\rho}_P(P_1) = \mu_\rho \eta_\rho \tilde{\rho}_P \mu_\rho \eta_\rho(P_1).

Proof. By definition of } \mu_\rho, \eta_\rho \text{ and } \tilde{\rho}_P \text{ (Definition 8.2), we have to prove that, given a clause } c \in P_1, \mu_\rho \eta_\rho \tilde{\rho}_P(c) = \mu_\rho \eta_\rho \tilde{\rho}_P \eta_\rho(c). \text{ Let us first prove that } \eta_\rho \tilde{\rho}_P(c) = \eta_\rho \tilde{\rho}_P \eta_\rho(c). \text{ By definition of } \eta_\rho \text{ (Definition 5.3), we only need to prove that for any agent } A, \text{ there exists } m \text{ such that } \eta^m_\rho \tilde{\rho}_P(A) = \eta^m_\rho \tilde{\rho}_P \eta_\rho(A), \text{ where, by definition, } \eta^1_\rho(A) = \eta_\rho(A) \text{ and } \eta^m_\rho(A) = \eta^{m-1}_\rho(A). \text{ The proof is by structural induction on } A. \text{ For the sake of simplicity, in the sequel we omit the subscripts } a, P \text{ when this does not cause any ambiguity.}

Base case: If } A = \text{nil}, \text{ the result is obvious. If } A_1 = p(X), \text{ then } \tilde{\rho}_P(A) = \tilde{\rho}_P \eta_\rho(A) \text{ since, by definition of } \eta_\rho(p(X)) = p(X). \text{ Then the thesis holds for the base case.}

Inductive case: We have the following cases:

1. If } A = a : t \rightarrow A_1, \text{ then assume, by inductive hypothesis, there exists } m \text{ such that } \eta^m_\rho \eta_\rho(A_1) = \eta^m_\rho(A_1). \text{ Then the thesis holds since } \eta^m_\rho \tilde{\rho}_P(A) = a : t \rightarrow \eta^m_\rho \tilde{\rho}_P(A_1) \text{ and } \eta^m_\rho \tilde{\rho}_P(A) = a : t \rightarrow \eta^m_\rho \tilde{\rho}_P(A_1).\]
(2) The case $A = A_1 + A_2$ is similar to the previous one since

$$\eta^n \hat{\rho}(A_1 + A_2) = \eta^n(\hat{\rho}(A_1) + \hat{\rho}(A_2)) \quad \text{(by definition of } \hat{\rho})$$

and analogously

$$\eta^n \hat{\rho} \eta(A_1 + A_2) = \eta^n \hat{\rho} \eta(A_1) + \eta^n \hat{\rho} \eta(A_2).$$

Then, the thesis holds for $m = \max \{m_1, m_2\}$, where, by inductive hypothesis, $\eta^{m_i} \hat{\rho} \eta(A_i) = \eta^{m_i} \hat{\rho}(A_i)$, $i = 1, 2$ (for any $\bar{A}$, if $\eta^m \hat{\rho} \eta(\bar{A}) = \eta^m \hat{\rho}(\bar{A})$ then obviously $\eta^{m+1} \hat{\rho} \eta(\bar{A}) = \eta^{m+1} \rho(\bar{A}))$.

(3) If $A = A_1 \parallel A_2$, let us suppose, by inductive hypothesis, that $\eta^n \hat{\rho} \eta(A_i) = \eta^n \hat{\rho}(A_i)$, $i = 1, 2$ (if the indexes for $A_1$ and $A_2$ are different, we can choose the maximum as in the previous case). Moreover, let us assume $m$ to be such that $\eta^n(\bar{A}) = \eta^{n+1}(\bar{A})$ for any agent $\bar{A}$ such that $\eta^n(\bar{A})$ occurs in the proof. We have the following cases:

(i) Either $A_1 \neq \sum_i a_i \cdot t_i \rightarrow A_1$ or $A_2 \neq \sum_j a_j' \cdot t_j' \rightarrow A_j'$, and either $A_1 \neq p(\bar{X})$ or $A_2 \neq q(\bar{X})$, for any $p, q$. Then

$$\eta^n \hat{\rho} \eta(A_1 \parallel A_2) = \eta^n \hat{\rho} \eta(A_1 \parallel A_2) \quad \text{(by definition of } \eta \text{ and by hypothesis on } A_1, A_2)$$

and the thesis holds.

(ii) If $A_1 = \sum_{i \in I} a_i \cdot t_i \rightarrow A_1$ and $A_2 = \sum_{j \in J} a_j' \cdot t_j' \rightarrow A_j'$, then

$$\eta^n \hat{\rho} \eta(A_1 \parallel A_2) = \eta^n \hat{\rho} \left( \left( \sum_{i \in I} a_i \cdot t_i \rightarrow \eta(A_i \parallel A_2) \right) + \left( \sum_{j \in J} a_j' \cdot t_j' \rightarrow \eta(A_1 \parallel A_j') \right) \right) \quad \text{(by definition of } \eta \text{ and by hypothesis on } A_1, A_2)$$

$$= \eta^n \left( \left( \sum_{i \in I} a_i \cdot t_i \rightarrow \hat{\rho} \eta(A_i \parallel A_2) \right) + \left( \sum_{j \in J} a_j' \cdot t_j' \rightarrow \hat{\rho} \eta(A_1 \parallel A_j') \right) \right) \quad \text{(by definition of } \hat{\rho})$$

$$= \left( \sum_{i \in I} a_i \cdot t_i \rightarrow \eta^n \hat{\rho} \eta(A_i \parallel A_2) \right) + \left( \sum_{j \in J} a_j' \cdot t_j' \rightarrow \eta^n \hat{\rho} \eta(A_1 \parallel A_j') \right) \quad \text{(by definition of } \eta)$$
\[ = \left( \sum_{i \in I} a_i : t_i \rightarrow \eta^m \bar{\rho}(A_1 \parallel A_2) \right) + \left( \sum_{j \in J} a'_j : t'_j \rightarrow \eta^m \bar{\rho}(A_1, A'_2) \right) \]

(by inductive hypothesis)

\[ = \eta^m \left( \left( \sum_{i \in I} a_i : t_i \rightarrow \bar{\rho}(A_i) \right) \parallel \left( \sum_{j \in J} a'_j : t'_j \rightarrow \bar{\rho}(A'_i) \right) \right) \]

(by definition of \( \eta \))

\[ = \eta^m \bar{\rho} \left( \left( \sum_{i \in I} a_i : t_i \rightarrow A_i \right) \parallel \left( \sum_{j \in J} a'_j : t'_j \rightarrow A'_i \right) \right) \]

(by definition of \( \bar{\rho} \))

\[ = \eta^m \bar{\rho}(A_1 \parallel A_2) \]

(by definition of \( A_1 \) and \( A_2 \))

and the thesis holds.

(iii) If \( A_1 = p(X) \), \( A_2 = q(X) \) then

\[ \eta \bar{\rho}(A_1 \parallel A_2) \]

\[ = \eta \bar{\rho}(\eta(A_1) \parallel \eta(A_2)) \]

(by definition of \( \eta \) and by hypothesis on \( A_1, A_2 \))

\[ = \eta \bar{\rho}(A_1 \parallel A_2) \]

(by definition of \( \eta \) and by hypothesis on \( A_1, A_2 \))

and the thesis holds.

We have proved that \( \eta_\alpha \bar{\rho}_p(c) = \eta_\alpha \bar{\rho}_q(c) \) for any clause \( c \). By definition of \( \mu_\alpha \) and by a straightforward inductive argument, for a generic clause \( c, \mu_\alpha \bar{\rho}(c) = \bar{\rho}_\alpha(c) \) and \( \mu_\alpha \eta_\alpha \mu_\alpha(c) = \mu_\alpha \eta_\alpha(c) \). Then \( \mu_\alpha \eta_\alpha \mu_\alpha \eta_\alpha(c) = \mu_\alpha \eta_\alpha \mu_\alpha \eta_\alpha(c) = \mu_\alpha \eta_\alpha \mu_\alpha(c) = \mu_\alpha \eta_\alpha \bar{\rho}(c) \) and this completes the proof. \( \square \)

**Lemma A.7.** Let \( P, P_1 \) be programs, and let \( \bar{\rho} \) be as defined in Definition 8.2. Let \( P^* = \mu_P \eta_P(P) \). Then \( \mu_P \eta_P \bar{\rho}_P(P_1) = \mu_P \eta_P \bar{\rho}_P(P_1) \).

**Proof.** By definition of clauses (Definition 3.2), we only need to prove that, for any agent \( A \), there exists \( m \) such that \( \mu_\alpha \eta^m_\alpha \bar{\rho}_\alpha(A) = \mu_\alpha \eta^m_\alpha \bar{\rho}_\alpha(A) \). For the sake of simplicity, we omit the subscript \( \alpha \) when this does not cause any ambiguity. The proof is by structural induction on \( A \). In what follows, we make the same assumption on \( m \) as we made in the proof of Lemma A.6.

**Base case:** The case \( A = \text{nil} \) is obvious.

If \( A = p(X) \) and \( p(X) : - B \) is the clause for \( p \) in \( P \), then \( p(X) : - \mu \eta^m(B) \in P^* \). Then

\[ \mu \eta^m \bar{\rho}_P(A) \]

\[ = \mu \eta^m \mu \eta \eta^m(B) \]

(by definition of \( \bar{\rho}_P \) and assumption on \( m \))

\[ = \mu \eta^m(B) \]

(by Lemma A.6)

\[ = \mu \eta^m \bar{\rho}_P(A) \]

(by definition of \( \bar{\rho}_P \))

and the thesis holds for the base case.
Inductive case: We have the following possibilities:

(1) If $A = a : t \rightarrow A_1$, two cases exist.
   (i) If $\mu \eta^m \tilde{p}_p(A) = a : t \rightarrow \text{nil}$ then, by definition of $\mu$, $\mu \eta^m \tilde{p}_p(A) = a : t \rightarrow \text{nil}$ and the thesis holds.
   (ii) If $\mu \eta^m \tilde{p}_p(A) = a : t \rightarrow \mu \eta^m \tilde{p}_p(A_1)$ then, by definition of $\mu$, $\mu \eta^m \tilde{p}_p(A) = a : t \rightarrow \mu \eta^m \tilde{p}_p(A_1)$ and the thesis holds since, by inductive hypothesis, $\mu \eta^m \tilde{p}_p(A_1) = \mu \eta^m \tilde{p}_p(A_1)$.

(2) If $A = A_1 + A_2$ then

$$\mu \eta^m \tilde{p}_p(A_1 + A_2) = \mu \eta^m \tilde{p}_p(A_1 + A_2)$$

   (by definition of $\tilde{p}_p$)

$$= \mu (\eta^m \tilde{p}_p(A_1) + \eta^m \tilde{p}_p(A_2))$$

   (by definition of $\eta$)

$$= \mu \eta^m \tilde{p}_p(A_1) + \mu \eta^m \tilde{p}_p(A_2)$$

   (by definition of $\mu$)

and analogously

$$\mu \eta^m \tilde{p}_p(A_1 + A_2) = \mu \eta^m \tilde{p}_p(A_1) + \mu \eta^m \tilde{p}_p(A_2).$$

Then the thesis follows since, by inductive hypothesis, for $i = 1, 2$,

$$\mu \eta^m \tilde{p}_p(A_i) = \mu \eta^m \tilde{p}_p(A_i).$$

(3) If $A = A_1 \| A_2$, as in the previous case,

$$\mu \eta^m \tilde{p}_p(A_1 \| A_2) = \mu \eta^m \tilde{p}_p(A_1 \| A_2)$$

   (by definition of $\tilde{p}_p$)

$$= \mu \eta^m (\eta^m \tilde{p}_p(A_1) \| \eta^m \tilde{p}_p(A_2))$$

   (by Lemma A.5 and by hypothesis on $m$)

$$= \mu \eta^m (\mu \eta^m \tilde{p}_p(A_1) \| \eta^m \tilde{p}_p(A_2))$$

   (by definition of $\mu$)

$$= \mu \eta^m (\mu \eta^m \tilde{p}_p(A_1) \| \mu \eta^m \tilde{p}_p(A_2))$$

   (by definition of $\mu$)

and analogously

$$\mu \eta^m \tilde{p}_p(A_1 \| A_2) = \mu \eta^m (\mu \eta^m \tilde{p}_p(A_1) \| \mu \eta^m \tilde{p}_p(A_2)).$$

Then the thesis follows by inductive hypothesis.

Since we have considered all the cases, the thesis holds. □

**Definition A.8.** Let $P$ be a program. Then the collections of programs $\bar{P}^n$ and $\tilde{P}^n$ are defined as follows:

$$\bar{P}^1 = P,$$

$$\tilde{P}^1 = P,$$

$$P^{n+1} = \bigcup_{j=1, \ldots, m} \text{Unf}(\hat{c}^n_j, P^1),$$

$$\bar{P}^{n+1} = \bigcup_{j=1, \ldots, m} \text{Unf}(\hat{c}^1_j, \tilde{P}^n).$$
where $\tilde{P}^n = \{\tilde{c}^1, \ldots, \tilde{c}^n\}$ and $\hat{P}^n = \{\hat{c}^1, \ldots, \hat{c}^n\}$. Moreover, we define the collections of interpretations $I^n$ and $\tilde{I}^n$ as follows:

$$
\tilde{I} = I(\mu_P \eta_P^n(P)), \quad \hat{I} = I(\mu_P \eta_P^n(P)),
$$

$$
\tilde{I}^n = I(\tilde{P}^n), \quad \hat{I}^n = I(\hat{P}^n), \quad n = 2, 3, \ldots,
$$

where $I(P)$ is defined as in Definition 7.4 and $\mu_P, \eta_P$ are defined in Definition 8.2.

**Lemma A.9.** Let $\tilde{P}^n$ and $\hat{P}^n$ be as defined in Definition A.8. Then, for $n = 1, 2, \ldots$, $\tilde{P}^n = \hat{P}^n$.

**Proof.** The proof is by induction on $n$. The base case is obvious. Let us suppose by inductive hypothesis that $\tilde{P}^n = \hat{P}^n$. Let $\tilde{\rho}_P, \mu_P, \eta_P$ be defined as in Definition 8.2. Then,

$$
\tilde{P}^{n+1} = \bigcup_{j = 1, \ldots, m} \text{Unf}(\tilde{c}_j, P) \quad \text{(by definition of $P$)}
$$

$$
= \mu_P \eta_P \tilde{\rho}_P \ldots \mu_P \eta_P \tilde{\rho}_P (P) \quad \text{(by Definition 5.3 and by definition of $\tilde{c}^n$)}
$$

$$
= \mu_P \eta_P \tilde{\rho}_P \ldots \tilde{\rho}_P (P). \quad \text{(by Lemma A.6)}
$$

Let $P^n = \tilde{\rho}_P \ldots \tilde{\rho}_P (P)$. By inductive hypothesis, $\hat{P}^n = \tilde{P}^n = \mu_P \eta_P P^{n-1}$. Then

$$
\hat{P}^{n+1} = \mu_P \eta_P \hat{\rho}_n (P) \quad \text{(by definition of $\hat{\rho}$)}
$$

$$
= \mu_P \eta_P \hat{\rho}_n (P) \quad \text{(by inductive hypothesis)}
$$

$$
= \mu_P \eta_P \rho_P \ldots \rho_P (P) \quad \text{(by Lemma A.7)}
$$

$$
= \mu_P \eta_P \rho_P \ldots \rho_P (P). \quad \text{(by definition of $\hat{\rho}$ and of $P^n$)}
$$

and the thesis holds. □

**Lemma A.10.** Let $P$ be a program, and let $I^n, n = 1, 2, \ldots$, be the collection of interpretations defined in Definition A.8. Then $I^n = T^n_{\tilde{P}}(\perp_P), n = 1, 2, \ldots$.

**Proof.** Assume $1 \leq h \leq m$. Let $P = \{c_1, c_2, \ldots, c_m\}$, where $c_h$ is the clause for the predicate $p_h$, and let $\tilde{P}^n = \{\tilde{c}_1, \ldots, \tilde{c}_m\}, n = 1, 2, \ldots$, be as defined in Definition A.8. Let $T^n_{\tilde{P}}(\perp_P) = (J^n_1, J^n_2, \ldots, J^n_m)$ and $I^n = (\hat{I}^n_1, \hat{I}^n_2, \ldots, \hat{I}^n_m), n = 1, 2, \ldots$, where $\hat{I}^n_h$ and $J^n_h$ are interpretations for the predicate $p_h$. The proof is by induction on $n$. 
Base case. By Definition 8.1 ($T_{UP}$ operator), $J_h^1 = \mathcal{R}(Unf(c_h, \perp_p))^\ast$ and by Definition A.8 (interpretation $\tilde{I}_h^1$), \[ \mathcal{R}(\mu_p \eta_p(c_p)) = \tilde{I}_h^1. \] By Definition 5.3 (unfolding rules), Definition 7.2 ($\mathcal{R}(c)$ transformation) and Definition 8.1 ($\mu_p \eta_p(c_p)$), \[ \mathcal{R}(Unf(c_h, \perp_p)) = \mathcal{R}(\mu_p \eta_p(c_p)) = \tilde{I}_h^1. \] Then $J_h^1 = \mathcal{R}(Unf(c_h, \perp_p))^\ast$ and the thesis holds for the base case.

Inductive case. Assume, by inductive hypothesis, $J_h^i = \tilde{I}_h^i$. By definition of $\tilde{I}_h^i$, $\tilde{I}_h^i = \mathcal{R}(\tilde{c}_h^i)$. Then

\begin{align*}
J_{h+1}^1 &= \bigcup_{T \in (\tilde{c}_h^1)^*} \mathcal{R}(Unf(c_h, T))^\ast \quad \text{(by definition of $T_{UP}$)} \\
&= \bigcup_{T \in (\tilde{c}_h^1)^*} \mathcal{R}(Unf(c_h, T))^\ast \quad \text{(by inductive hypothesis)} \\
&= \bigcup_{T \in (\tilde{c}_h^1)^*} \mathcal{R}(Unf(c_h, T))^\ast \quad \text{(by definition of $\tilde{I}_h^i$)}.
\end{align*}

Let us define $T^* = (\mathcal{R}(\tilde{c}_1^i), \ldots, \mathcal{R}(\tilde{c}_m^i))$. Then

\begin{align*}
\tilde{I}_{h+1}^i &= \mathcal{R}(\tilde{c}_h^{i+1}) \quad \text{(by definition of $\tilde{I}_h^i$)} \\
&= \mathcal{R}(Unf(c_h, \tilde{I}_h^i)) \quad \text{(by definition of $\tilde{c}_h^{i+1}$)} \\
&= \mathcal{R}(Unf(c_h, T^*)) \quad \text{(by definition of $\mathcal{R}$ and $T^*$)},
\end{align*}

where the $m$-tuple of reactive behaviors $T^*$ is interpreted as a program in the obvious way. By definition of $\simeq$ (Definition 6.6), $T^* \in (\mathcal{R}(\tilde{c}_1^i))^\ast \times \cdots \times (\mathcal{R}(\tilde{c}_m^i))^\ast$ and, therefore,

\[ \mathcal{R}(Unf(c_h, T^*)) \subseteq \bigcup_{T \in (\tilde{c}_h^1)^*} \mathcal{R}(Unf(c_h, T))^\ast \]

and, hence, $\tilde{I}_{h+1}^i \subseteq J_{h+1}^i$. Conversely, by definition of $\simeq$ and of the preorder $\leq$ (Definition 6.2), $\forall T \in (\mathcal{R}(\tilde{c}_1^i))^\ast \times \cdots \times (\mathcal{R}(\tilde{c}_m^i))^\ast$, if $T = (t_1, \ldots, t_m)$ then $t_h \leq (\tilde{c}_1^i)$ for $h = 1, 2, \ldots, m$. Then, by Lemma 8.3, $\forall T \in (\mathcal{R}(\tilde{c}_1^i))^\ast \times \cdots \times (\mathcal{R}(\tilde{c}_m^i))^\ast$, $\mathcal{R}(Unf(c_h, T)) \leq \mathcal{R}(Unf(c_h, T^*))$. Therefore,

\[ \bigcup_{T \in (\tilde{c}_h^1)^*} \mathcal{R}(Unf(c_h, T))^\ast \leq \mathcal{R}(Unf(c_h, T^*)) \]

and, hence, $J_{h+1}^i \subseteq \tilde{I}_{h+1}^i$; this completes the proof. \[ \square \]

Lemma A.11. Let $P = \{c_1, \ldots, c_m\}$ be a program and let $c_h^i$ and $\tilde{c}_h^i$ be defined as follows $(h = 1, \ldots, m)$:

\begin{align*}
c_1^1 &= c_h, & \tilde{c}_1^1 &= c_h, \\
c_h^i &= \tilde{p}_{p-1}(c_h^{i-1}), & \tilde{c}_h^i &= \tilde{p}(\tilde{c}_h^{i-1}),
\end{align*}

where $P^i = \{c_1^i, \ldots, c_m^i\}$ and $\tilde{P}^i = \{\tilde{c}_1^i, \ldots, \tilde{c}_m^i\}$ and $\tilde{p}$ is defined as in Definition 8.2. Then $\forall n \exists j \geq n$ such that $P^n = \tilde{P}^j$. \[ \square \]
Proof. The proof is by induction on \( n \). The base case is obvious since \( \bar{P}^1 = P^1 \). Let \( P^n = \bar{P}^n \) by inductive hypothesis. Assume \( 1 \leq h \leq m \). Then

\[
\begin{align*}
\bar{c}^{n+1}_h &= \bar{p}_n(c^n_h) \quad \text{(by definition of } c^{n+1}_h) \\
&= \bar{p}_n(c^1_h) \quad \text{(by inductive hypothesis)} \\
&= \bar{p}_n(c_h) \quad \text{(by definition of } \bar{p}_j \text{ for a suitable } k \geq j) \\
&= c^k_h \quad \text{(by definition of } c^k_h) 
\end{align*}
\]

and the thesis holds. \( \square \)

Lemma A.12. Let \( \bar{I}^1, \bar{I}^2, \ldots \) be the collection of interpretations defined by Definition A.8 and let \( I^n, \bar{I}^n, \bar{I}_n, \bar{I}_n \) be the collection of interpretations defined by Definition 7.5. Then \( \forall n I^n \subseteq \bar{I}^n \) and \( \forall n \exists j(\geq n) \) such that \( I^n \subseteq \bar{I}^j \).

Proof. Assume \( 1 \leq h \leq m \). Let \( \bar{P}^1, \bar{P}^2, \ldots \) and \( P^1, P^2, \ldots \) be the collections of programs defined by Definitions A.8 and 7.5, respectively, where, for \( n = 1, 2, \ldots, \bar{P}^n = \{c^n_1, \ldots, c^n_m\} \) and \( P^n = \{c^n_1, \ldots, c^n_m\} \). Let \( \mu_r, \eta_r \) and \( \rho_r \) be as defined in Definitions 5.3 and 8.2. Then

\[
\begin{align*}
\bar{c}^{n+1}_h &= Unf(c^n_h, P) \\
&= \mu_r(\eta_r \rho_r(c^n_h)) \quad \text{(by Definition 5.3)} \\
&= \mu_r(\eta_r \rho_r(c^n_h)) \quad \text{(by Definition of } \bar{c}^n_h \text{)} \\
&= \mu_r(\eta_r \rho_r(c^n_h)) \quad \text{(by Lemma A.6)}
\end{align*}
\]

and

\[
\begin{align*}
c^{n+1}_h &= Unf(c^n_h, P^n) \quad \text{(by definition of } P^i) \\
&= \mu_r(\eta_r \rho_n(c^n_h)) \quad \text{(by Definition 5.3)} \\
&= \mu_r(\eta_r \rho_n(c^n_h)) \quad \text{(by definition of } c^n_h \text{)} \\
&= \mu_r(\eta_r \rho_n(c^n_h)) \quad \text{(by Lemma A.7),}
\end{align*}
\]

where for \( i = 1, 2, \ldots, n \), \( \mu_r \eta_r(P^{*i}) = P^i \). By Lemma A.11, there exists \( j \geq n \) such that

\[
\begin{align*}
\bar{\rho}_r \ldots \bar{\rho}_r(c_h) - \bar{\rho}_r \ldots \bar{\rho}_r(c_h).
\end{align*}
\]
Then $\forall n \exists j$ such that $R(e^*_n) = R(\tilde{c}^*_n)$ and, by definition of $\tilde{I}^n$ and $I^n$, $\forall n \exists j(\geq n)$ such that $I^n \subseteq \tilde{I}^j$. Moreover, since $\tilde{I}^1, \tilde{I}^2, \ldots$ is a chain, $\tilde{I}^n \subseteq \tilde{I}^j$ for any $j \geq n$. Then $\forall n$, $\tilde{I}^n \subseteq \tilde{I}^1 = I^n$, where $j \geq n$ is chosen as before and this completes the proof. \[ \square \]

**Theorem 8.7** (equivalence of the fixpoint and the unfolding semantics). Let $P$ be a program. Then $\mathcal{U}(P) = \mathcal{F}(P)$.

**Proof.** By Definition 7.5, $B \in \mathcal{U}(P)$ iff $\exists n$ such that $B \in I^n$ (where $I^n$ is an interpretation of the chain as defined by Definition 7.5). By Definition 8.5, $B \in \mathcal{F}(P)$ iff $\exists m$ such that $B \in T^m_{\mathcal{F}(P)}(\bot_P)$. Then in order to prove the theorem, we only need to prove that, for $n=1,2,\ldots$, if $B \in I^n$ then $\exists m$ such that $B \in T^m_{\mathcal{F}(P)}(\bot_P)$ and vice versa. Let $\tilde{I}^1, \tilde{I}^2, \ldots$ be as defined in Definition A.8. Assume $B \in I^n$. Then, by Lemma A.12, there exists $m(\geq n)$ such that $B \in \tilde{I}^m$. Moreover, by Lemma A.9, $\tilde{I}^m = I^m$, and by Lemma A.10, $I^n = T^n_{\mathcal{F}(P)}(\bot_P)$. Therefore, $B \in T^m_{\mathcal{F}(P)}(\bot_P)$. Conversely, if $B \in T^m_{\mathcal{F}(P)}(\bot_P)$, by Lemma A.10, $B \in I^n$, by Lemma A.9, $B \in \tilde{I}^m$, and by Lemma A.12, $B \in I^n$; this completes the proof. \[ \square \]

**References**


