URL: http://www.elsevier.nl/locate/entcs/volume16.html 18 pages

Two semantics for Timed Default Concurrent Constraint Programming [1]

Simone Tini and Andrea Maggiolo-Schettini

Dipartimento di Informatica, Università di Pisa, Corso Italia 40, 56125 Pisa, Italy.
e_mail: {*tini, maggiolo*}*@di.unipi.it*

Abstract

In this paper we present a general approach to give semantics of synchronous languages. By applying this approach, we define two semantics for Timed Default Concurrent Constraint Programming.

1 Introduction

Nondeterministic concurrent process languages are languages for the description of interactive systems, namely systems interacting with their environment at their own rate. In [8] and [9] Letichevsky and Gilbert present a general theory for such languages. They introduce *Action Language* as a common model for nondeterministic concurrent process languages and define two semantics for it, an *intensional semantics* and an *interactive semantics*. The intensional semantics of a program gives its behavior by abstracting from the behavior of the environment. The idea of interactive semantics is that the meaning of a program is a transformation of its environment, which corresponds to inserting the program into the environment. If a notion of behavior of the environment is defined, then the interactive semantics of a program is a function from behaviors of the environment to behaviors of the environment.

In this paper, following [8] and [9], we present a general theory of the class of *synchronous languages* [2,5]. Synchronous languages have been developed to program *reactive systems* [7], namely systems which interact continuously with their environment at a rate controlled by this. Execution in a reactive system proceeds in bursts of activity. In each phase, the environment stimulates

[1] Research partially supported by ESPRIT Working Group "Concurrent Constraint Programming for Time Critical Applications" (COTIC), Project Number 23677, and by MURST Progetto "Tecniche Formali per la Specifica, l'Analisi, la Verifica, la Sintesi e la Trasformazione di Sistemi Software".

the system with an input and the system reacts computing a response. The environment, which does not evolve during reactions, expects that responses are computed in a bounded time. Synchronous languages are based on the so called *synchronous hypothesis* [4], namely the assumption that the system is able to instantaneously react to prompts from its environment. This amounts to saying that the underlying machine is infinitely fast and takes no time to execute operations involved in instructions sequencing, process handling and interprocess communication. The synchronous hypothesis is an abstraction which relies on the idea that both the environment and the system are discrete and the system is faster than the environment. In order to make the synchronous hypothesis realistic, efficient implementations of synchronous languages into automata and hardware components have been proposed. As demonstrated in [3], implementations in hardware realized by directly translating programs into circuits are more efficient than implementations obtained by translating firstly programs into automata and then automata into circuits. Differently with respect to interactive systems where there is a perfect symmetry between system and environment, a reactive system and its environment are clearly two distinct entities, each having its own rôle. This is the main reason for which the results in [8,9] are not directly applicable to the class of synchronous languages.

Our aim is to endow synchronous languages with both an intensional and an interactive semantics. The domain of the intensional semantics is an algebra of behaviors which is parametric w.r.to an algebra of actions. When a particular language is considered, both the set of actions and the relations over them must be instantiated. The intensional semantics permits to consider the structure of a reactive system at the wanted level of detail, by choosing appropriate equivalences over the algebra of behaviors. As an example, suppose that one wants to directly translate a program into hardware components, as in [3]. In this case, as the concurrent structure of programs is reflected in the circuits, one wants a semantics that gives an account of such structure. On the contrary, if programs are translated into sequential automata, where concurrency disappears, it is reasonable to have a set of axioms over the algebra of behaviors such that the intensional meaning of a program is equivalent to the intensional meaning of a sequential program.

The domain of the interactive semantics is an algebra of transformations of behaviors of the environment. To give the interactive meaning of a program it is therefore necessary to define what the behavior of the environment is. To this purpose, we consider an algebra of behaviors of the environment over an algebra of its actions. The interactive semantics stresses how programs interact with the environment, abstracting from details such as their concurrent structure and communication among subprograms. Two programs have the same interactive meaning iff they cannot be distinguished by any environment. We define a function from the domain of the intensional semantics to the domain of the interactive semantics such that the interactive meaning of a

program can be obtained by applying this function to its intensional meaning.

Our approach offers a uniform algebraic framework in which different synchronous languages can be compared: constructs of languages can be characterized, expressiveness of different languages can be established, equivalences of programs in the different formalisms can be proved.

The synchronous formalism we are interested in is Timed Default Concurrent Constraint Programming (*tdccp*) introduced in [12–14]. We obtain an intensional semantics for *tdccp* by an instantiation of the algebra of actions. As *tdccp* is implemented by sequential automata, we choose a set of axioms over the algebra of behaviors such that the intensional meaning of a program is a behavior that does not reflect its concurrent structure. This semantics is shown to agree with the operational semantics of [12]. We define a function such that the interactive meaning of a *tdccp* agent is obtained by applying this function to its intensional meaning.

2 Timed Concurrent Constraint Programming

In this section we recall Timed Default Concurrent Constraint Programming. For clarity, we firstly present Concurrent Constraint Programming (*ccp*) ([11]).

Concurrent Constraint Programming replaces the traditional notion of a store as a valuation of variables with the notion of a store consisting of pieces of information which restrict the possible values of variables. A program consists in a multiset of agents which run concurrently and interact by adding information to the store (*tell* operation) and querying the store about validity of some information in it (*ask* operation). It is not possible to subtract information from the store, which is therefore supposed to increase monotonically. Agents run asynchronously and ask operations are used for synchronization, as, if a query is not answered positively, the inquiring agent waits until there is enough information in the store to entail the information required.

We give now the notion of constraint system. A constraint system $\mathcal{D}$ is a system of partial information consisting of a set of primitive constraints (first order formulas) or *tokens* D, closed under conjunction and existential quantification, and an inference relation $\vdash$ relating tokens to tokens. We use $a, a', b \ldots$ and $a_1, a_2, \ldots$ to range over tokens. The entailment relation induces, through symmetric closure, the logical equivalence relation $\approx$. Formally:

Definition 2.1 *A* constraint system *is a structure* $(D, \wedge, \vdash, Var, \{\exists_X \mid X \in Var\})$ *such that:*

- D *is a set of* tokens *closed under* conjunction *($\wedge$). The* entailment relation $\vdash \subseteq D \times D$ *satisfies:*
  - $a \vdash a$
  - $a_1 \vdash a_2$ *and* $a_2 \wedge a_3 \vdash a_4$ *implies* $a_1 \wedge a_3 \vdash a_4$
  - $a_1 \wedge a_2 \vdash a_1$ *and* $a_1 \wedge a_2 \vdash a_2$
  - $a_1 \vdash a_2$ *and* $a_1 \vdash a_3$ *implies that* $a_1 \vdash a_2 \wedge a_3$

- *Var is an infinite set of* variables. *For each variable $X \in Var$, $\exists_X : D \to D$ is an operation satisfying usual laws on existential quantification:*
  - $a \vdash \exists_X a$
  - $\exists_X(a_1 \wedge \exists_X a_2) \approx \exists_X a_1 \wedge \exists_X a_2$
  - $\exists_X \exists_Y a \approx \exists_Y \exists_X a$
  - *$a_1 \vdash a_2$ implies that $\exists_X a_1 \vdash \exists_X a_2$*
- *$\vdash$ is decidable.*

The last condition is necessary to have an operational semantics which is effective.

A *constraint* is an entailment closed subset of D. For any set of tokens S, we let $\mathcal{E}(S)$ stand for the constraint $\{a \in D \mid \exists\{a_1, \ldots, a_n\} \subseteq S \text{ s.t. } a_1 \wedge \ldots \wedge a_n \vdash a\}$. For any token a, $\mathcal{E}(a)$ denotes $\mathcal{E}(\{a\})$.

The set of constraints ordered by inclusion ($\subseteq$) form a complete algebraic lattice with least upper bounds induced by $\wedge$, least element **true** $= \{a \mid \forall a' \in D.\, a' \vdash a\}$ and greatest element **false** $= D$.

We present now the combinators considered in *ccp*. In the following $U, V, U_1, \ldots$ range over agents.

Tell. Agent "a" adds token a to the store.

Ask. Agent "**if** a **then** U" queries the store about the validity of token a. If the store entails a then **if** a **then** U behaves as U. If the store does not entail a then **if** a **then** U waits until a is entailed by the store.

Parallel Composition. Agent "(U_1, U_2)" is the parallel asynchronous composition of U_1 and U_2.

Hiding. Agent "**new** X **in** U" behaves as U, provided that X is local to U. This means that all assumptions on X must be generated by some evolution of U and that the external world cannot see X.

We obtain Default Concurrent Constraint Programming (*dccp*) by considering also defaults for negative information. A new combinator is defined as follows.

Negative ask. Agent "**if** a **else** U" queries the store about the validity of token a. If the store does not entail a then **if** a **else** U behaves as U. If the store entails a then the computation of **if** a **else** U terminates.

Starting with an initial store, an agent U is supposed to evolve by adding tokens to the store, until no more information is produced that is not entailed by the store. In this case we say that U *converges* on such store. In order to give the operational semantics of *dccp* we consider configurations, which are multisets of agents, and binary transition relations $\rightarrow_b$, indexed by token b, over configurations. Token b is the guess about the final store. This means that the operational semantics computes the result of an agent running in a given store only if the final store is known beforehand. The nondeterminism which arises can be bounded for finite agents and made effective by backtracking. For any configuration Γ, let us denote by $\sigma(\Gamma)$ the subset of all tokens in Γ. In order to give the operational semantics of agent U starting with an initial

store, we consider the configuration $\Gamma = (U, \Delta)$, where Δ is the set of tokens in the initial store. In this case, token a is in $\sigma(\Gamma)$ either if a is an agent in U or if a is in Δ. The relation $\rightarrow_b$ is defined below:

$$\frac{\sigma(\Gamma) \vdash a}{\Gamma,\ \textbf{if}\ a\ \textbf{then}\ U \rightarrow_b \Gamma, U} \qquad \frac{b \not\vdash a}{\Gamma,\ \textbf{if}\ a\ \textbf{else}\ U \rightarrow_b \Gamma, U}$$

$$\Gamma,\ \textbf{new}\ X\ \textbf{in}\ U \rightarrow_b \Gamma, U[Y/X]\ (Y \text{ not free in } U, \Gamma)$$

$$\Gamma, (U_1, U_2) \rightarrow_b \Gamma, U_1, U_2.$$

We do not need any rule for the combinator **Tell**, as if agent a is in configuration Γ, then token a is in $\sigma(\Gamma)$.

For any agent U and input token a, the function r_o defined as follows gives the set of possible output tokens b:

$$r_o(U)(a) = \{b \in D \mid \exists b' \in D.\, (U, a) \rightarrow_{b'}^{*} \Gamma \not\rightarrow_{b'}, b' = \sigma(\Gamma), \exists_{\vec{Y}} b' \approx b\}.$$

Here $\vec{Y}$ are the new local variables in $\sigma(\Gamma)$ introduced during the derivation.

In [12] a denotational semantics is defined and proved to be fully abstract w.r.to the operational semantics described above.

Timed Default Concurrent Constraint Programming enriches *dccp* with a notion of discrete time. Concretely, the temporal construct "**next** U" is introduced. The intuitive meaning is that **next** U imposes the constraints of U at the time instant after the current one. The operational meaning is that if **next** U is invoked at time t, then a copy of U is invoked at time $t+1$. According to the synchronous hypothesis principle, combinators derived from *dccp* do not consume time. How a *tdccp* agent U works can be explained as follows. At each instant of time the environment adds an input token to the empty store and U reacts instantaneously by enriching the store and by computing the agent to be activated at the next instant of time. The reaction consists in running a *dccp* program. The store is completely discharged between two instants of time.

An agent *tdccp* may be also a procedure call $p(V_1, \ldots, V_n)$, with p a procedure name and each V_i is an agent. The procedure p is defined as $p(x_1, \ldots, x_n) = U$, where x_i is an agent variable and U is an agent. It is possible to have recursive definitions of the form

$p_1(x_1, \ldots, x_n) = U_1$

$\vdots$

$p_m(x_1, \ldots, x_n) = U_m,$

where calls of procedures $p_1, \ldots, p_m$ may appear in the body of $U_1, \ldots, U_m$. In this case, the variables $x_1, \ldots, x_n$ must be in the scope of a **next**, namely recursion is guarded. This is needed to have computation bounded by the size of the program in each time step. To ensure that at run-time there are only

boundedly many different procedure calls, it is required that any recursive procedure call takes exactly the same parameters as the procedure definition.

We give now the operational semantics of *tdccp*. We consider configurations consisting of multisets of agents and a binary relation $\rightsquigarrow$ over configurations such that $\Gamma \rightsquigarrow \Gamma'$ means that if agents in Γ are active at the current instant of time, then agents in Γ' are activated at the next instant of time. To define the relation $\rightsquigarrow$ we need a set of rules to compute both the output at the current instant of time and the agents to be activated at the next instant of time. To this purpose, let us consider configurations consisting of pairs whose elements are a multiset of agents currently active and a "continuation", which is the multiset of agents that will be activated at the subsequent time. We define binary transition relations $\rightarrow_b$ over such configurations analogously to what is done for *dccp*. Each relation $\rightarrow_b$ relates two configurations which are active at the same instant of time. The rules for such relation are the following:

$$\frac{\sigma(\Gamma) \vdash a}{((\Gamma,\ \textbf{if } a \textbf{ then } U), \Delta) \rightarrow_b ((\Gamma, U), \Delta)} \qquad \frac{b \nvdash a}{((\Gamma,\ \textbf{if } a \textbf{ else } U), \Delta) \rightarrow_b ((\Gamma, U), \Delta)}$$

$$((\Gamma, (U_1, U_2)), \Delta) \rightarrow_b ((\Gamma, U_1, U_2), \Delta)$$

$$((\Gamma,\ \textbf{next } U), \Delta) \rightarrow_b (\Gamma, (U, \Delta))$$

$$((\Gamma,\ \textbf{new } X \textbf{ in } U), \Delta) \rightarrow_b ((\Gamma, U[Y/X]), \Delta) \quad (Y \text{ not free in } U, \Gamma)$$

$$((\Gamma, p(V_1, \ldots, V_n)), \Delta) \rightarrow_b ((\Gamma, U[V_1/x_1, \ldots, V_n/x_n]), \Delta).$$

In the last rule we assume that $p(x_1, \ldots, x_n) = U$ is the definition of the procedure p. As expected, the only rule which modifies the second component of configurations is the rule for **next**.

We can now define the binary transition relation $\rightsquigarrow$ over configurations consisting in multisets of agents. The relation $\rightsquigarrow$ is computed by exploiting the relation $\rightarrow_b$, as stated by the following rule:

$$\frac{\exists b \in D.(\Gamma, \emptyset) \rightarrow_b^* (\Gamma', \Delta) \not\rightarrow_b \qquad \sigma(\Gamma') = b}{\Gamma \rightsquigarrow \textbf{ new } \vec{Y} \textbf{ in } \Delta}$$

As in the case of *dccp*, $\vec{Y}$ are the variables introduced during the computation. Given an agent U and a sequence of tokens s as input, the sequence of outputs s' can be computed by the function rt_o defined as follows:

$$\begin{aligned} rt_o(U)(s) = \{s' \mid\ & |s'| = |s| = n, U \stackrel{def}{=} U_0 \\ & \forall i < n\, (U_i, s(i)) \rightsquigarrow U_{i+1} \\ & s'(i) = r_o(U_i)(s(i))\} \end{aligned}$$

The output at each instant of time is computed by relations $\rightarrow_b$.

In [12] it is argued that, according to the semantics above, a program may have zero or more evolution paths. An agent U has no evolution path for input a if and only if $(U, a) \not\rightsquigarrow$. In this case the agent has a *non reactive* behavior

and fails. As an example, let us consider the agent $U = ($ **if** $X = 1$ **else** $Y = 1$, **if** $Y = 1$ **then** $X = 1)$. If U is executed in the empty store, $U, \emptyset \not\rightarrow$. An agent U has more than one evolution path for input a if either $(U, a) \rightsquigarrow U_1$ and $(U, a) \rightsquigarrow U_2$ for $U_1 \neq U_2$, or $|r_o(U)(a)| > 1$. In this case we say that the agent has a *non deterministic* behavior. As an example, let us consider the agents $U_1 =$ **if** $X = 1$ **else** $X = 2$ and $U_2 =$ **if** $X = 2$ **else** $X = 1$. Let us consider the agent $U = (U_1, U_2)$. If the store entails neither token $X = 1$ nor token $X = 2$, then there is a nondeterministic choice between adding $X = 1$ or $X = 2$ to the store.

An agent is said to be *determinate* iff it has exactly one evolution path for each input token. An algorithm for checking determinacy of agents is given in [12].

Example 2.2 As a running example we use a simplified specification of the central locking system for a two-door car given in [10]. Doors can be either locked or unlocked. Doors can be locked and unlocked either from outside the car with a key or from inside the car by pushing a button. The system consists of three components: a central controller and a controller for each of the two doors. Here we specify only the central controller. In [10] the complete system is specified. The central controller can be in three internal states: *Ready*, *Lock* and *Unlock*. When state *Ready* is active, then the central controller is waiting for a signal to lock or unlock the doors. We assume that when the doors are locked either from outside or from inside the car, a signal $ldoors$ is received by the central controller which sends signals $lleft$ and $lright$ to the controllers of the left and right door respectively. Then the central controller moves to the state *Lock*; it returns to the state *Ready* when the two door controllers send signals *lack* and *rack* respectively. Analogously, when the central controller is in state *Ready* and it receives the signal $udoors$, it sends to the door controllers the signals $uleft$ and $uright$, and moves to the state *Unlock*; it returns to the state *Ready* when it receives the *lack* and *rack* signals. The *tdccp* agents corresponding to the central controller is the agent $U = Ready$, where *Ready* is a procedure without parameters defined as follows:

$Ready =$ (**if** $ldoors$ **then** ($lleft, lright,$ **next** $Lock$),
if $ldoors$ **else if** $udoors$ **then** ($uleft, uright,$ **next** $Unlock$),
if $ldoors$ **else if** $udoors$ **else next** $Ready$).

$Lock =$ (**if** $lack \wedge rack$ **then next** $Ready$,
if $lack \wedge rack$ **else next** $Lock$).

$Unlock =$ (**if** $lack \wedge rack$ **then next** $Ready$,
if $lack \wedge rack$ **else next** $Unlock$).

3 Algebras of behaviors

In this section we define the domain of the intensional semantics of synchronous languages.

Let us consider the 2-sorted signature $\Sigma = (S, \Omega)$ such that:

- $S = \{A, B\}$
- $\Omega = Act \cup \{0, \delta :\to A\} \cup \{\mathrm{O}, \Delta :\to B\} \cup \{\cdot : A \times B \to B\} \cup \{\theta_{f_i} : A \to A \,|\, i \in I\} \cup \{\rho_{f_i} : B \to B \,|\, i \in I\} \cup \{+, \|\colon B \times B \to B\}$, where Act is a set of constants of sort A.

Sorts A and B are the sort of *actions* and the sort of (reactive) *behaviors*, respectively. The intuition is that each action in A corresponds to a reaction of a reactive system. Actions are temporally atomic, in correspondence with instantaneous executions or reactions. We assume the set Act as a parameter of our definition, which must be instantiated when a particular language is considered. We assume the *empty action* δ and the *disaster action* 0 to represent the reaction that does not affect the environment and the reaction that causes a failure, respectively. We consider a *composition function* $\gamma : A \times A \to A$ satisfying the following requirements:

- for every $x, y \in A$: $\gamma(x, y) = \gamma(y, x)$
- for every $x, y, z \in A$: $\gamma(\gamma(x, y), z) = \gamma(x, \gamma(y, z))$
- for every $x \in A$: $\gamma(x, x) = x$, $\gamma(x, \delta) = x$ and $\gamma(x, 0) = 0$.

For each pair of actions x, y corresponding to two reactions in two different sequential components of a system, $\gamma(x, y)$ corresponds to the compound reaction.

The carrier set of behaviors constitute the domain of the intensional semantics. We assume the following operations:

- $\cdot : A \times B \to B$ is the *prefixing*. The behavior $\sigma \cdot p$ consists in the action σ at the current instant of time followed by the behavior p at the next instant of time. Operation $\cdot$ is needed to model sequencing operators like the operator **next** in *tdccp*.
- $+ : B \times B \to B$ is the *alternate composition*. The behavior $p + q$ may be either p or q. If we consider languages having an operator of nondeterministic internal choice, such as Statecharts [6], then $+$ models such operator. If we consider languages without any operator of nondeterministic internal choice, such as *tdccp*, then the operation $+$ models the external choice, namely a choice completely dependent on the environment.
- $\|\colon B \times B \to B$ is the *merge*. Behavior $p \,\|\, q$ is the synchronous running of p and q. Operation $\|$ is needed to model operators of parallel composition.

We assume the *empty behavior* Δ and the *disaster behavior* O satisfying the equations $\Delta = \delta \cdot \Delta$ and $\mathrm{O} = 0 \cdot \mathrm{O}$, respectively.

Finally, we consider a family $(f_i)_{i \in I}$ of *renaming functions*, $f_i : A \to A$.

$u+v$	$=$	$v+u$	(A1)	$\rho_{f_i}(u+v)$	$=$	$\rho_{f_i}(u)+\rho_{f_i}(v)$	(A6)
$(u+v)+w$	$=$	$u+(v+w)$	(A2)	$\rho_{f_i}(x\cdot u)$	$=$	$\theta_{f_i}(x)\cdot\rho_{f_i}(u)$	(A7)
$u+\mathrm{O}$	$=$	u	(A3)	$\theta_{f_i}(x)$	$=$	$f_i(x)$	(A8)
O	$=$	$0\cdot u$	(A4)	$(u+v)\parallel w$	$=$	$u\parallel w+v\parallel w$	(A9)
Δ	$=$	$\delta\cdot\Delta$	(A5)	$u\parallel(v+w)$	$=$	$u\parallel v+u\parallel w$	(A10)
				$x\cdot u\parallel y\cdot v$	$=$	$\gamma(x,y)\cdot(u\parallel v)$	(A11)

Table 1
The set of axioms Eq.

This family of functions is needed in order to model operators like **hiding** of *tdccp*. Given a function f_i, $i\in I$, we define the operations $\theta_{f_i}:A\to A$ and $\rho_{f_i}:B\to B$, such that $\theta_{f_i}(\sigma)=f_i(\sigma)$ for each $\sigma\in A$ and ρ_{f_i} is the extension of θ_{f_i} to behaviors.

We assume the set of axioms Eq over Σ in Table 1, where variables are intended to be universally quantified. Our convention is that $x,y\ldots$ range over actions and $u,v\ldots$ range over behaviors. We denote by $Mod_\Sigma(Eq)$ the class of Σ-algebras that are models of Eq.

As we shall see in the following, axioms A9-A11 imply that each term t of sort B can be rewritten into a term $t'=\sum_{i<n}\sigma_i\cdot t_i$, where σ_i is an action and t_i a term, $i<n$. This result is standard in a non truly-concurrent approach. This is reasonable for synchronous languages oriented to their implementation by means of automata. The choice of axioms should be different if one wanted a semantics oriented to implementation in hardware. As an example, let us assume a program P in a deterministic synchronous language having the operator "|" of parallel composition. If the language is translated into automata, as in the case of *tdccp*, then P and $P|P$ are implemented by the same automaton, and therefore if the behavior p is the intensional meaning of P, it is reasonable to have $p=p\parallel p$. On the contrary, if the language is compositionally translated into hardware, the circuits corresponding to P and $P|P$ are different and therefore it is less reasonable to have $p=p\parallel p$.

All terms of sort B are interpreted as infinite behaviors. This corresponds to the fact that reactive systems do not terminate. In order to have cyclic behaviors, we need recursive specifications.

A *recursive equation* over (Σ,Eq) is an equation of the form:

$$u=s(u)$$

where $s(u)$ is a term of sort B containing the variable u.
A *solution of a recursive equation* $u=s(u)$ in a Σ-algebra in $Mod_\Sigma(Eq)$ is a behavior p such that $p=s(p)$, namely p satisfies the equation in the Σ-algebra. In this case we say that p *substitutes* u.

A *recursive specification* E over (Σ,Eq) is a set of recursive equations over

(Σ, Eq). For a set of variables U, it holds that for each $u \in U$ there is an equation of the form

$$u = s_u(U)$$

and one of the variables in U is called the *root variable*.
A *solution of a recursive specification* E in a Σ-algebra in $Mod_\Sigma(Eq)$ is a behavior p such that there is a set of behaviors satisfying the equations in the Σ-algebra, and p substitutes the root variable.

Given a recursive specification E and a recursion variable u, we denote by $< u|E >$ the behavior that substitutes u.

As usual, we are interested in guarded recursive specifications.

Let s be a term of sort B containing a variable u of sort B. An occurrence of u is *guarded* in s if s has a subterm of the form $\sigma \cdot t$, where $\sigma \in Act$ and t is a term of sort B containing the considered occurrence. The *term* s is *completely guarded* if all occurrences of all variables are guarded. A *recursive specification* E is *completely guarded* if all right hand sides of all equations in E are completely guarded terms.

In general a Σ-algebra in $Mod_\Sigma(Eq)$ may have zero, one or more solutions for a guarded recursive specification. There exists a subclass of $Mod_\Sigma(Eq)$ of algebras having exactly one solution for each guarded recursive specification (for an argument see [1]). We can consider an arbitrary algebra $\mathcal{A}$ in this subclass and we consider the carrier set of behaviors $\mathcal{A}(B)$ as domain of the intensional semantics. In order to have the intensional semantics of a particular synchronous language, we need to instantiate the set Act of actions, the family $(f_i)_{i \in I}$ of functions, and the function γ. Given a program P, we shall denote by $\mathcal{I}(P)$ its intensional semantics.

We say that a behavior p is *finitely definable* if and only if p is obtained from constants in Σ by means of operations in Σ and guarded recursive specifications with finitely many equations.

Following [1], we can prove the following proposition.

Proposition 3.1 *A finitely definable behavior p can be written in* head normal form *as follows:*

$$p = \sum_{i<n} \sigma_i \cdot p_i$$

where σ_i is an action, $\sigma_i \neq 0$, and p_i is a behavior for each $i < n$.

The convention is that $p = \mathrm{O}$ if $n = 0$.
For each process p the head normal form is unique, modulo associativity and commutativity of $+$.
Given a behavior $p = \sum_{i<n} \sigma_i \cdot p_i$, we say that $\sigma_i \cdot p_i$ is a *summand* of p for each $i < n$.

4 An intensional semantics for *tdccp*

In this section we define a semantic function $\mathcal{I}$ such that for each *tdccp* agent U, $\mathcal{I}(U)$ is its intensional semantics. As said in the previous section, $\mathcal{I}(U)$ is an element of the carrier set of behaviors of an algebra $\mathcal{A}$ in $Mod_{\Sigma}(Eq)$.

We begin with defining a set of actions $Act(D)$, parametric w.r.to the set of tokens D, and a function $\mathcal{C}$, and then we instantiate Act to $Act(D)$ and γ to $\mathcal{C}$.

Given a token $a \in D$, let $\overline{a}$ denote the fact that a is not entailed by the store. For a subset D' of D, let $\overline{D'}$ denote the set $\{\overline{a} \mid a \in D'\}$. Moreover, with abuse of notation, we assume that for each token $a \in D$, $\overline{\overline{a}}$ denotes a.

Definition 4.1 *Let D be a set of tokens. We define $Act(D)$ as the set of tuples $(l, \mathcal{O})$, such that:*

- $l \in 2^{D \cup \overline{D}}$, $\mathcal{E}(l \cap D) \cap (\overline{l} \cap D) = \emptyset$
- *$\mathcal{O}$ is a set of orderings $2^l \times 2^l$ such that for each $\prec \in \mathcal{O}$:*
  - *$\prec$ is an irreflexive ordering relation*
  - *for each $C, C' \in 2^l$ such that $C' \subseteq C$, if $C'' \prec C'$ then $C[C''/C'] \prec C$.*

The action $(l, \mathcal{O})$ corresponds to a reaction of a *tdccp* agent. If a token $a \in D$ is in l, then a is added to the store by either the environment or the agent. If $\overline{a}$ is in l, then the fact that a is not entailed by the store is among the causes of the reaction. This motivates the request that $\mathcal{E}(l \cap D) \cap (\overline{l} \cap D) = \emptyset$.

The orderings reflect *causality* among the tokens in l, so that given $\prec \in \mathcal{O}$, $\{a, a''\} \prec \{a'\}$ means that a' is added to the store if both a and a'' are entailed by the store. On the other hand, $\{a\} \prec \{a'\}$ and $\{a''\} \prec \{a'\}$ means that either a or a'' is sufficient to add a' to the store.

As an example, let us consider the *tdccp* agent **if** a **then if** b **then** c. The action $(\{a, b, c\}, \{\{(\{a, b\}, \{c\})\}\})$ corresponds to the successful request about entailment of tokens a and b and adding c to the store. The second component of the action contains only one ordering. This always holds for actions corresponding to reactions of sequential components. The action $(\{\overline{a}\}, \emptyset)$ corresponds to the unsuccessful request to the store about the entailment of token a.

Given an action $\sigma = (l, \mathcal{O}) \in Act(D)$ and an ordering $\prec \in \mathcal{O}$ we denote by $trigger(\sigma, \prec)$ the set $\{a \in l \mid \nexists C \in 2^l \text{ s.t. } C \prec \{a\}\}$ and we denote by $added(\sigma, \prec)$ the set $l \cap (trigger(\sigma, \prec))^c$. Now, $trigger(\sigma, \prec) \cap D$ is the set of tokens required to be entailed by the store for enabling the reaction. On the contrary, $\overline{trigger(\sigma, \prec) \cap \overline{D}}$ is the set of tokens required not to be entailed by the store for enabling the reaction. Finally, $added(\sigma, \prec)$ is the set of tokens added to the store during the reaction.
Given a set of tokens $D' \subseteq D$, an action $\sigma = (l, \mathcal{O})$ and an ordering $\prec \in \mathcal{O}$, we say that D' *triggers* σ *w.r.to* $\prec$ iff $trigger(\sigma, \prec) \cap D \subseteq \mathcal{E}(D')$ and $\overline{trigger(\sigma, \prec) \cap \overline{D}} \cap \mathcal{E}(D') = \emptyset$. If D' triggers σ w.r.to some ordering $\prec$, then

the reaction corresponding to σ is a reaction to the store D'. Moreover, if $\mathcal{O} = \emptyset$, then D' triggers $(l, \mathcal{O})$ if $\mathcal{E}(D') \supseteq l \cap D$ and $\mathcal{E}(D') \not\supseteq \overline{l \cap \overline{D}}$.

An action $\sigma = (l, \mathcal{O}) \in Act(D)$ is a *basic action* if there are disjoint sets $A \subseteq D \cup \overline{D}$ and $B \subseteq D$ such that:

- $l = A \cup B$
- $\mathcal{O} = \{\prec\}$; $\prec = \{(A, b) \mid b \in B\}$.

Such a basic action is the reaction to the store A such that, for each token $b \in B$, the set of tokens that cause b is precisely A. An action corresponding to a reaction of a sequential *tdccp* agent is a basic action. As an example, the action $(\{a\}, \{\{(\emptyset, \{a\})\}\})$ is the basic action corresponding to the reaction of agent a.

The definition of function $\mathcal{C}$ is quite complex.

Given two orderings $\prec_1, \prec_2$, we denote by $(\prec_1 \cup \prec_2)^+$ the ordering $\prec$, such that $\prec_1 \cup \prec_2 \subseteq \prec$ and such that, if $C \prec C'$ and $C' \prec C''$, then $C \prec C''$ and, for each C, C', if $C' \subseteq C$ and $C'' \prec C'$, then $C[C''/C'] \prec C$.

Definition 4.2 *Given actions $\sigma_1 = (l_1, \mathcal{O}_1), \sigma_2 = (l_2, \mathcal{O}_2) \in Act(D)$, let $(l, \mathcal{O})$ be the pair such that:*

- $l = l_1 \cup l_2$
- $\mathcal{O} = \{\prec \mid \exists \prec_1 \in \mathcal{O}_1, \exists \prec_2 \in \mathcal{O}_2$ *s.t.* $\prec \subseteq (\prec_1 \cup \prec_2)^+$ *and for each* $\prec'$ *with* $\prec \subset \prec' \subseteq (\prec_1 \cup \prec_2)^+$ $\prec'$ *has circularities* $\}$.

If $(l, \mathcal{O}) \in Act(D)$ then $\mathcal{C}(\sigma_1, \sigma_2) = (l, \mathcal{O})$, else $\mathcal{C}(\sigma_1, \sigma_2) = 0$.

Consider the actions $\sigma_1 = (l_1, \mathcal{O}_1)$ and $\sigma_2 = (l_2, \mathcal{O}_2)$ and the pair $(l, \mathcal{O})$ as in the definition above. If $(l, \mathcal{O})$ is in $Act(D)$, then $\mathcal{C}(\sigma_1, \sigma_2)$ is formed by the union of tokens $l = l_1 \cup l_2$ and the orderings that are the maximal subsets of $(\prec_1 \cup \prec_2)^+$ which are not reflexive, where $\prec_1 \in \mathcal{O}_1$ and $\prec_2 \in \mathcal{O}_2$.

As an example, let us consider the action $\sigma_1 = (\{a, b\}, \{\{(\{a\}, \{b\})\}\})$ and the action $\sigma_2 = (\{a, b\}, \{\{(\{b\}, \{a\})\}\})$ corresponding to reactions of agents $U_1 =$ **if** a **then** b and $U_2 =$ **if** b **then** a, respectively. Now, according to Def. 4.2, we have that $\mathcal{C}(\sigma_1, \sigma_2) = (\{a, b\}, \{\{(\{a\}, \{b\})\}, \{(\{b\}, \{a\})\}\})$. The action $\sigma = \mathcal{C}(\sigma_1, \sigma_2)$ corresponds to a reaction of agent (U_1, U_2). Note that σ has two orderings, each reflecting a different causality relation. The idea is that the first ordering reflects the fact that the reaction is caused by token a, the second ordering reflects the fact that the reaction is caused by token b. Assume now the action $\sigma_3 = (\{a, \overline{b}\}, \{\{(\{\overline{b}\}, \{a\})\}\})$ corresponding to a reaction of agent $U_3 =$ **if** b **else** a. We have that $\mathcal{C}(\sigma_1, \sigma_3) = 0$. This corresponds to the fact that σ_1 and σ_3 are incompatible, in the sense that they correspond to two mutual exclusive reactions. Note that agent (U_1, U_3) is not determinate, as it cannot react to any store entailing neither a nor b. As an example, we have that $(U_1, U_3), \emptyset \not\rightarrow$, namely agent (U_1, U_3) fails if the environment does not add any token to the store.

We assume that δ corresponds to the action $(\emptyset, \emptyset)$. We assume also that

for each action σ, $\mathcal{C}(\sigma, 0) = 0 = \mathcal{C}(0, \sigma)$. It is immediate that the function $\mathcal{C}$ satisfies the requirements for the composition function.

We define now a function $pref : D \cup \overline{D} \times A \to A$ such that if action σ corresponds to a reaction of agent U, then $pref(a, \sigma)$ and $pref(\overline{a}, \sigma)$ correspond to the same reaction of agents **if** a **then** U and **if** a **else** U, respectively.

Definition 4.3 *Given $a \in D \cup \overline{D}$ then:*

- $pref(a, (l, \mathcal{O})) = (l \cup \{a\}, \{\{(A \cup \{a\}, B - \{a\}) \mid (A, B) \in \prec\} \mid \ \prec \in \mathcal{O}\})$
- $pref(a, 0) = 0$.

According to Def. 4.3, token a must be in $trigger(pref(a, \sigma), \prec)$ for each ordering $\prec$ of $pref(a, \sigma)$. As an example, let us assume the action $\sigma = (\{b\}, \{\{(\emptyset, \{b\})\}\})$ which, as we will see in the following, corresponds to the reaction of agent b. If the store entails a, then $pref(a, \sigma) = (\{a, b\}, \{\{(\{a\}, \{b\})\}\})$ is the action corresponding to the reaction of agent **if** a **then** b.

We extend now the function $pref$ to behavior terms. Given a term $t = \sum_{i<n} \sigma_i \cdot t_i$ of sort B, we have that $pref(a, t) = \sum_{i<n} pref(a, \sigma_i) \cdot t_i$.

Let us suppose that action σ corresponds to a reaction of an agent U. For each variable X we define a function loc_X such that $loc_X(\sigma)$ is the action corresponding to the same reaction of agent **new** X **in** U. If we denote by D_X the subset of tokens in D having a free occurrence of variable X, then $loc_X(\sigma)$ must satisfy two requirements. The first is that no token in D_X is visible in $loc_X(\sigma)$, as X is a local variable. The second is that if a token in D_X is among the causes of σ, then $loc_X(\sigma) = 0$. The reason is that it is not possible that the environment adds to the store tokens entailing constraints on a variable local to an agent.

Definition 4.4 *For $X \in Var$, $loc_X : A \to A$ is the function such that:*

- $loc_X(l, \mathcal{O}) = (l[Y/X], \mathcal{O}[Y/X])$ *if there exists an ordering $\prec \in \mathcal{O}$ such that no token in D_X is in $trigger(\sigma, \prec)$ and Y is "fresh"*
- $loc_X(l, \mathcal{O}) = 0$ *if for each ordering $\prec \in \mathcal{O}$ there exists a token $a \in D_X$ such that $a \in trigger(\sigma, \prec)$*
- $loc_X(0) = 0$.

We assume that the family of functions $(f_i)_{i \in I}$ is instantiated to the family of functions $(loc_X)_{X \in Var}$.

We define now a function $\mathcal{I}'$ such that for each *tdccp* agent U, $\mathcal{I}'(U)$ is a term of sort B.

Definition 4.5 *The functions $\mathcal{I}' : tdccp \to B$ is inductively defined as follows:*

- $\mathcal{I}'(a) = (\{a\}, \{\{(\emptyset, \{a\})\}\}) \cdot \Delta$

- $\mathcal{I}'(\ \textbf{\textit{if}}\ a\ \textbf{\textit{then}}\ U) = (\{\overline{a}\}, \emptyset) \cdot \Delta + pref(a, \mathcal{I}'(U))$
- $\mathcal{I}'(\ \textbf{\textit{if}}\ a\ \textbf{\textit{else}}\ U) = (\{a\}, \emptyset) \cdot \Delta + pref(\overline{a}, \mathcal{I}'(U))$
- $\mathcal{I}'(\ \textbf{\textit{next}}\ U) = (\emptyset, \emptyset) \cdot \mathcal{I}'(U)$
- $\mathcal{I}'(U_1, U_2) = \mathcal{I}'(U_1) \parallel \mathcal{I}'(U_2)$
- $\mathcal{I}'(\ \textbf{\textit{new}}\ X\ \textbf{\textit{in}}\ U) = \rho_{loc_X}(\mathcal{I}'(U))$
- $\mathcal{I}'(p_i(V_1, \ldots, V_n)) =< z_i \mid E >$, *if we have the guarded recursive definition*

  $p_1(x_1, \ldots, x_n) = U_1$
  $\vdots$
  $p_m(x_1, \ldots, x_n) = U_m$
  and we consider the completely guarded recursive specification E

  $z_1 = \mathcal{I}'(U_1[V_1/x_1, \ldots, V_n/x_n, z_1/p_1(x_1, \ldots, x_n), \ldots, z_m/p_m(x_1, \ldots, x_n)])$
  $\vdots$
  $z_m = \mathcal{I}'(U_m[V_1/x_1, \ldots, V_n/x_n, z_1/p_1(x_1, \ldots, x_n), \ldots, z_m/p_m(x_1, \ldots, x_n)])$
  where $\mathcal{I}'(z_i) = z_i$ *for each variable* z_i.

The initial action of $\mathcal{I}'(a)$ says that the token a is added to the store during the reaction. The behavior term $\mathcal{I}'(\ \textbf{if}\ a\ \textbf{then}\ U)$ is the alternate composition of two behavior terms: the initial action of the first corresponds to the negative response by the store about the entailment of token a, the second is obtained from $\mathcal{I}'(U)$ by adding a among the causes of its first action. The behavior term $\mathcal{I}'(\ \textbf{if}\ a\ \textbf{else}\ U)$ is defined analogously. The behavior term $\mathcal{I}'(\ \textbf{next}\ U)$ is obtained by prefixing $\mathcal{I}'(U)$ with the action δ. The behavior term $\mathcal{I}'((U_1, U_2))$ corresponds to the merge of $\mathcal{I}'(U_1)$ and $\mathcal{I}'(U_2)$. The behavior term $\mathcal{I}'(\ \textbf{new}\ X\ \textbf{in}\ U)$ is obtained by applying the operation ρ_{loc_X} to $\mathcal{I}'(U)$. The behavior term $\mathcal{I}'(p_j(V_1, \ldots, V_n))$ is defined as follows. A recursive specification E is constructed by starting from the guarded recursive definition of $p_1, \ldots, p_m$. We replace each agent variable x_i by V_i, $1 \leq i \leq n$, and we replace $p_i(V_1, \ldots, V_n)$ by the behavior variable z_i, $1 \leq i \leq m$. Then we apply $\mathcal{I}'$ to the right hand sides of the equations. As the recursive definition of $p_1, \ldots, p_m$ is guarded, then E is completely guarded. Now, $\mathcal{I}'(p_i(V_1, \ldots, V_n)) =< z_i \mid E >$.

We give now the definition of $\mathcal{I}$.

Definition 4.6 *Given a tdccp agent* U, *we define* $\mathcal{I}(U)$ *as* $\mathcal{A}(\mathcal{I}'(U))$.

Note that function $\mathcal{I}$ is well defined, as $\mathcal{A}$ has precisely one solution for each completely guarded recursive specification.

Example 4.7 Let us consider the *tdccp* agent U defined in Example 2.2. The intensional semantics of U is $\mathcal{A}(< ready | E >)$, where E is the following recursive specification:

$ready = (\{ldoors, lleft, lright\}, \{\{(\{ldoors\}, \{lleft\}), (\{ldoors\}, \{lright\})\}\}) \cdot$

$$
\begin{aligned}
&lock\\
&+ (\{\overline{ldoors}, udoors, uleft, uright\}, \{\{(\{\overline{ldoors}, udoors\}, \{uleft\}),\\
&(\{\overline{ldoors}, udoors\}, \{uright\})\}\}) \cdot unlock\\
&+ (\{\overline{ldoors}, \overline{udoors}\}, \emptyset) \cdot ready
\end{aligned}
$$

$$lock = (\{lack \wedge rack\}, \emptyset) \cdot ready + (\{\overline{lack \wedge rack}\}, \emptyset) \cdot lock$$

$$unlock = (\{lack \wedge rack\}, \emptyset) \cdot ready + (\{\overline{lack \wedge rack}\}, \emptyset) \cdot unlock.$$

The following propositions demonstrate that our semantics is well defined with respect to the operational semantics in [12].

Proposition 4.8 *Given an agent U and a token a, we have that $(U, a) \rightsquigarrow U'$ and $rt_o(U)(a) = b'$ if and only if $\mathcal{I}'(U)$ has a summand $\sigma \cdot \mathcal{I}'(U')$, where either a triggers σ w.r.to an ordering $\prec$ and $b' \approx \exists_{\vec{Z}}(a \wedge (added(\sigma, \prec)))$, provided that $\vec{Z}$ are the fresh variables in σ, or $\sigma = (l, \emptyset)$, a triggers σ and $b' \approx a$.*

Proof. By structural induction over U. □

Proposition 4.9 *Given an agent U with $\mathcal{I}'(U) = \sum_{i<n} \sigma_i \cdot \mathcal{I}'(U_i)$, then U is determinate if and only if for each set of tokens $D' \subseteq D$ there exists precisely one σ_i, $i < n$, such that either D' triggers $\sigma_i = (l_i, \emptyset)$, or D' triggers σ_i w.r.to an ordering $\prec_i$.*

Proof. Follows directly from Prop. 4.8. □

As an example, let us assume $U = (U_1, U_2)$, where $U_1 =$ **if** $X = 1$ **else** $X = 2$ and $U_2 =$ **if** $X = 2$ **else** $X = 1$. We have that $\mathcal{I}'(U) = \sigma_1 \cdot \Delta + \sigma_2 \cdot \Delta + \sigma_3 \cdot \Delta$, where $\sigma_1 = (\{\overline{X = 1}, X = 2\}, \{\{(\{\overline{X = 1}\}, \{X = 2\})\}\})$, $\sigma_2 = (\{\overline{X = 2}, X = 1\}, \{\{(\{\overline{X = 2}\}, \{X = 1\})\}\})$ and $\sigma_3 = (\{X = 1, X = 2\}, \emptyset)$. As $\emptyset$ triggers both σ_1 w.r.to $\{(\{\overline{X = 1}\}, \{X = 2\})\}$ and σ_2 w.r.to $\{(\{\overline{X = 2}\}, \{X = 1\})\}$, U is not determinate.

5 An interactive semantics for *tdccp*

In this section we define the domain of the interactive semantics for synchronous languages and we explain how the interactive semantics of a program can be obtained from its intensional semantics. Then we give the interactive semantics of *tdccp*.

We assume an algebra of environment actions F and an algebra of environment behaviors E over F. We consider the constants $0, \delta, \mathrm{O}, \Delta$ and the operations $+$ and $\cdot$ as in Σ and the axioms A1-A5 as in Table 1. Operation $+$ corresponds to the nondeterministic choice and operation $\cdot$ corresponds to the sequencing. However, one may consider further operations.

In the case of *tdccp* we assume that $F = \{\delta, 0\} \cup Act'(D)$, where $Act'(D)$ is the set of actions $(l, \mathcal{O})$ in $Act(D)$ such that the empty set of tokens triggers

$(l, \mathcal{O})$ w.r.to some ordering $\prec$ in $\mathcal{O}$. The idea is that an action $f \in F$, with $f = (l, \mathcal{O})$, corresponds to prompting the reactive system by adding l to the empty store.

We consider the *reaction function react* : $F \times A \to F$ such that $react(f, \sigma)$ is the action f "enriched" by the reaction corresponding to σ. The function *react* describes the transformation of the action of the environment due to the interaction with the reactive system.

The function *react* induces an equivalence relation $\sim$ over A such that two $\sim$-equivalent actions cannot be distinguished by the environment:

$$\sigma_1 \sim \sigma_2 \text{ iff } \forall f \in F \; react(f, \sigma_1) = react(f, \sigma_2).$$

We require that the function *react* satisfies the following conditions:

- $\sim$ is a congruence, namely if $\sigma_1 \sim \sigma_2$ then for each σ we have $\gamma(\sigma_1, \sigma) \sim \gamma(\sigma_2, \sigma)$ and for each f_i we have $f_i(\sigma_1) \sim f_i(\sigma_2)$
- $\sigma = 0$ iff for each $f \in F$ $react(f, \sigma) = 0$
- $\sigma \sim \delta$ iff for each $f \in F$ $react(f, \sigma) = f$.

The interactive semantics $\mathcal{R}(P)$ of a program P is a transformation from environment behaviors to environment behaviors. According to this idea, we consider the algebra of behavior transformations of the type $\phi : E \to E$.

The algebra of behavior transformations has the same type of the algebra of behaviors defined in the previous sections. We consider the signature $\Sigma = (S, \Omega)$ and we replace the constants O and Δ by ϕ_0 and I, respectively. The *identity transformation* I is such that $I(e) = e$, for each $e \in E$, and the *zero transformation* ϕ_0 is such that $\phi_0(e) = \mathrm{O}$, for each $e \in E$.
Given the transformations ϕ and ψ and the action σ, we consider the transformations $\sigma \cdot \phi$ such that:

$$(\sigma \cdot \phi)(e + e') = (\sigma \cdot \phi)(e) + (\sigma \cdot \phi)(e') \text{ for each } e, e' \in E$$
$$(\sigma \cdot \phi)(f \cdot e) = react(f, \sigma) \cdot \phi(e) \text{ for each } f \in F, e \in E$$

and the transformation $\phi + \psi$ such that:

$$(\phi + \psi)(e) = \phi(e) + \psi(e) \text{ for each } e \in E.$$

Then we consider the transformation $\rho_{f_i}(\phi)$ as completely defined by axioms A6-A8 in Table 1. Finally, we consider the transformation $\phi \parallel \psi$ as completely defined by axioms A9-A11 in Table 1.

It is immediate to prove the following proposition.

Proposition 5.1 *The algebra of transformations satisfies the set of equations in Table 1, where* O *and* Δ *are replaced by* ϕ_0 *and* I*, respectively.*

Let $\mathcal{A}$ be the Σ-algebra in $Mod_\Sigma(Eq)$ such that $\mathcal{A}(B)$ is the domain of the intensional semantics. We consider the homomorphism *trans* of $\mathcal{A}$ to the algebra of transformations such that $trans(\Delta) = I$, $trans(\mathrm{O}) = \phi_0$ and $trans(\sigma) = \sigma$ for each action σ.

Note that for each transformation ϕ there exists a head normal form

$\sum_{i<n} \sigma_i \cdot \phi_i$ such that $\phi = \sum_{i<n} \sigma_i \cdot \phi_i$. This head normal form is unique modulo equivalence relation $\sim$ over actions and commutativity and associativity of $+$. This fact and the property of congruence of $\sim$ imply that the homomorphism *trans* is well defined.

We define $\mathcal{R}(P)$ as follows.

Definition 5.2 *Given a program P, we define $\mathcal{R}(P)$ as $trans(\mathcal{I}(P))$.*

In order to give an interactive semantics of a language it is sufficient to define its intensional semantics and to instantiate the function *react*.

Now, in the case of *tdccp*, let us take the function *react* defined as follows.

Definition 5.3 *The function react: $F \times A \to F$ is such that:*

$$react(f,\sigma) = \begin{cases} \mathcal{C}(f,\sigma) & \text{if } \mathcal{C}(f,\sigma) \in Act'(D) \\ 0 & \text{otherwise.} \end{cases}$$

We can prove the following proposition.

Proposition 5.4 *The equivalence $\sim$ induced by function react is a congruence w.r.to functions $\mathcal{C}$ and loc_X, $X \in Var$.*

If we consider the function *react* as in Def. 5.3, we have immediately the interactive semantics of *tdccp*.

Example 5.5 Let us consider the agent U of Example 2.2. In Example 4.7 we have defined its intensional semantics $\mathcal{I}(U)$. Let $\phi = \mathcal{R}(U) = trans(\mathcal{I}(U))$ be the interactive semantics of U. As an example, if we assume the behavior of the environment $e_1 =$
$(\{ldoors\}, \{\{(\emptyset, \{ldoors\})\}\}) \cdot (\emptyset, \emptyset) \cdot (\{lack, rack\}, \{\{(\emptyset, \{lack\}), (\emptyset, \{rack\})\}\}) \cdot$
e_1', then we have that $\phi(e_1) =$
$(\{ldoors, lleft, lright\}, \{\{(\emptyset, \{ldoors\}), (\{ldoors\}, \{lleft\}), (\{ldoors\}, \{lright\})\}\}) \cdot$

$(\overline{lack \wedge rack}, \emptyset) \cdot (\{lack, rack\}, \{\{(\emptyset, \{lack\}), (\emptyset, \{rack\})\}\}) \cdot \phi(e_1')$.
This shows how the agent U interacts with the environment.

References

[1] Baeten, J.C.M. and Weijland, W.P.: Process Algebra. Cambridge Tracts in Theoretical Computer Science, **18**, 1990.

[2] Benveniste, A. and Berry, G. editors: Another Look at Real-Time Systems. Special Issue of Proceedings of the IEEE, September 1991.

[3] Berry, G.: A Hardware Implementation of Pure Esterel. Sadhana, Academic Proceedings in Engineering Sciences, Indian Academic of Sciences, **17**, 1992, pp. 95-130.

[4] Berry, G. and Gonthier, G.: The Esterel Synchronous Programming Language: Design, Semantics, Implementation. Science of Computer Programming, **19**,

1992, pp. 87-152.

[5] Halbwachs, N.: Synchronous Programming of Reactive Systems. The Kluwer International Series in Engineering and Computer Science, Kluwer Academic Publishers, 1993.

[6] Harel, D.: Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming, **8**, 1987, pp. 231-274.

[7] Harel, D. and Pnueli, A.: On the Development of Reactive Systems. In K.R. Apt, editor, Logic and Models of Concurrent Systems, NATO, ASI-13, Springer, 1985, pp. 477-498.

[8] Letichevsky, A. and Gilbert, D.: Towards an Interactive Semantics of Nondeterministic Concurrent Programming Languages. Presented at the Second workshop of the INTAS-93-1702 project "Efficient Symbolic Computing", St Petersburg, Russia, October 1996.

[9] Letichevsky, A. and Gilbert, D.: A General Theory of Action Languages. Cybernetics and System Analysis, **1**, 1998, pp. 16-37.

[10] Philipps, J. and Scholz, P.: Compositional Specification of Embedded System with Statecharts. Proc. of Theory and Practise of Software Development, TAPSOFT '97, Lecture Notes in Computer Science 1214, Springer, 1987.

[11] Saraswat, V.A.: Concurrent Constraint Programming. The MIT Press. 1993.

[12] Saraswat, V.A., Jagadeesan, R. and Gupta, V.: Timed Default Concurrent Constraint Programming. Journal of Symbolic Computation, **11**, 1996, pp. 1-46.

[13] Saraswat, V.A., Jagadeesan, R. and Gupta, V.: Default Timed Concurrent Constraint Programming. Proc. of Twenty Second ACM Symposium on Principles of Programming Languages, San Francisco, 1995.

[14] Saraswat, V.A., Jagadeesan, R. and Gupta, V.: Programming in Timed Concurrent Constraint Languages. In B. Mayoh, E. Tougu, J. Penjain editors, Computer and System Sciences, NATO, ASI-131, Springer, 1994, pp. 477-498.