Timed Soft Concurrent Constraint Programs*

Stefano Bistarelli[1,2], Maurizio Gabbrielli[3], Maria Chiara Meo[1] and Francesco Santini[2,4]

[1] Dipartimento di Scienze, Università "G. D'Annunzio" di Chieti-Pescara, Italy
bista@sci.unich.it,cmeo@unich.it
[2] Istituto di Informatica e Telematica (CNR), Pisa, Italy
stefano.bistarelli@iit.cnr.it,francesco.santini@iit.cnr.it
[3] Dipartimento di Scienze dell'Informazione, Università di Bologna, Italy
gabbri@cs.unibo.it
[4] IMT - Institute for Advanced Studies, Lucca, Italy
f.santini@imtlucca.it

Abstract. We propose a timed and soft extension of Concurrent Constraint Programming. The time extension is based on the hypothesis of *bounded asynchrony*: the computation takes a bounded period of time and is measured by a discrete global clock. Action prefixing is then considered as the syntactic marker which distinguishes a time instant from the next one. Supported by soft constraints instead of crisp ones, *tell* and *ask* agents are now equipped with a preference (or consistency) threshold which is used to determine their success or suspension. In the paper we provide a language to describe the agents behavior, together with its operational and denotational semantics, for which we also prove the compositionality and correctness properties. Agents negotiating *Quality of Service* can benefit from this new language, by coordinating among themselves and mediating their preferences.

1 Introduction

Time is a particularly important aspect of cooperative environments. In many "real-life" computer applications, the activities have a temporal duration (that can be even interrupted) and the coordination of such activities has to take into consideration this timeliness property. The interacting actors are mutually influenced by their actions, meaning that A reacts accordingly to the timeliness and "quality" of B's behavior and vice versa. In fact, these interactions can be often related to quantities to be measured or minimized/maximized, in order to take actions depending from this result: consider, for example, some generic communicating-agents that need to negotiate a desired *Quality of Service* (*QoS*). In this case, they both need to coordinate through time-dependent decisions and to quantify and publish their respective requirements. These agents can be instantiated to concrete instances, such as web services, internet QoS architectures and mechanisms that provide QoS, workflows and, in general, software agents.

* The first and fourth authors are supported by the MIUR PRIN 2005-015491.

In [8] *Timed Concurrent Constraint Programming* (*tccp*), a timed extension of the pure formalism of *Concurrent Constraint Programming* (*ccp*) [19], is introduced. This extension is based on the hypothesis of *bounded asynchrony* (as introduced in [20]): computation takes a bounded period of time rather than being instantaneous as in the concurrent synchronous languages `ESTEREL` [1], `LUSTRE` [12], `SIGNAL` [15] and Statecharts [13]. Time itself is measured by a discrete global clock, i.e, the internal clock of the *tccp* process. In [8] they also introduced *timed reactive sequences* which describe at each moment in time the reaction of a *tccp* process to the input of the external environment. Formally, such a reaction is a pair of constraints $\langle c, d \rangle$, where c is the input given by the environment and d is the constraint produced by the process in response to c (due to the monotonicity of *ccp* computations, c includes always the input).

Soft constraints [2, 3] extend classical constraints to represent multiple consistency levels, and thus provide a way to express preferences, fuzziness, and uncertainty. The *ccp* framework has been extended to work with soft constraints [4], and the resulting framework is named *Soft Concurrent Constraint Programming* (*sccp*). With respect to *ccp*, in *sccp* the *tell* and *ask* agents are equipped with a preference (or consistency) threshold which is used to determine their success, failure, or suspension, as well as to prune the search; these preferences should preferably be satisfied but not necessarily (i.e. over-constrained problems).

In this paper we introduce a timed and soft extension of *ccp* that we call *Timed Soft Concurrent Constraint Programming* (*tsccp*), inheriting from both *tccp* and *sccp* at the same time. In *tccp*, action-prefixing is interpreted as the next-time operator and the parallel execution of agents follows the scheduling policy of maximal parallelism. Additionally, *tccp* includes a simple new primitive which allows to specify timing constraints. We adopt soft constraints (and the related *sccp*) instead of crisp ones, since we are sure that classic constraints can show evident limitations if applied to entities interactions, mainly because they do not appear to be very flexible when trying to represent real-life scenarios, where the knowledge is not completely available nor crisp. The introduced *Timed Soft Concurrent Constraint* (*tscc*) language, together with its semantics, results in a formal framework where it is possible to solve QoS related problems.

The agents use the centralized constraint store in order to ensure their community acts in a coherent manner, where "coherence" refers to how well a system of agents behaves as a unit. With *tccp*, the agent coordination is enriched with both timed and quantitative/qualitative aspects at the same time; this represents the most important expressivity improvement w.r.t. related works (see Sec. 8). One of the most straightforward applications is represented by the modelling of negotiation and management of resources, since both time and preference are naturally part of the problem. In Sec. 7 we show an example where we model an auction process, which can be seen as a particular instance of negotiation.

In Sec. 2 we sum up the most important background notions and frameworks from which *tsccp* derives, i.e. *tccp* and *sccp*. In Sec. 3 the *tscc* language is presented for the first time. Then, Sec. 4 and Sec. 5 respectively describe the operational and denotational semantics of the *tscc* agents. Section 6 outlines the

proof of the denotational model correctness with the aid of *connected reactive sequences*. At last, Sec. 7 shows an application example of the language and Sec. 8 concludes by discussing related work and indicating future research.

2 Background

2.1 Soft Concurrent Constraint System

A semiring is a tuple $\langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$ such that: *i)* A is a set and $\mathbf{0}, \mathbf{1} \in A$; *ii)* $+$ is commutative, associative and $\mathbf{0}$ is its unit element; *iii)* $\times$ is associative, distributes over $+$, $\mathbf{1}$ is its unit element and $\mathbf{0}$ is its absorbing element. A c-semiring is a semiring $\langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$ such that: $+$ is idempotent, $\mathbf{1}$ is its absorbing element and $\times$ is commutative. Let us consider the relation $\leq_S$ over A such that $a \leq_S b$ iff $a + b = b$. Then it is possible to prove that (see [3]): *i)* $\leq_S$ is a partial order; *ii)* $+$ and $\times$ are monotone on $\leq_S$; *iii)* $\mathbf{0}$ is its minimum and $\mathbf{1}$ its maximum; *iv)* $\langle A, \leq_S\rangle$ is a complete lattice and, for all $a, b \in A$, $a + b = lub(a, b)$ (where *lub* is the *least upper bound*). $\langle A, \leq_S\rangle$ is a complete distributive lattice and $\times$ its *glb* (*greatest lower bound*). Informally, the relation $\leq_S$ gives us a way to compare semiring values and constraints: when we have $a \leq_S b$, we will say that b *is better than* a. In the following, when the semiring will be clear from the context, $a \leq_S b$ will be often indicated by $a \leq b$.

A *soft constraint* [2, 3] may be seen as a constraint where each instantiation of its variables has an associated preference. Given a semiring $S = \langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$ and a set of variables V over a finite domain D, a soft constraint is a function which, given an assignment $\eta : V \to D$ of the variables, returns a value of the semiring. Using this notation $\mathcal{C} = \eta \to A$ is the set of all possible constraints that can be built starting from S, D and V.

Any function in $\mathcal{C}$ involves all the variables in V, but we impose that it depends on the assignment of only a finite subset of them. So, for instance, a binary constraint $c_{x,y}$ over variables x and y, is a function $c_{x,y} : V \to D \to A$, but it depends only on the assignment of variables $\{x, y\} \subseteq V$ (the *support* of the constraint, or *scope*). Note that $c\eta[v := d_1]$ means $c\eta'$ where η' is η modified with the assignment $v := d_1$. The partial order $\leq$ over A can be easily extended among constraints by defining $c_1 \sqsubseteq c_2 \iff c_1\eta \leq c_2\eta$.

Given the set $\mathcal{C}$, the combination function $\otimes : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ is defined as $(c_1 \otimes c_2)\eta = c_1\eta \times c_2\eta$ (see also [2–4]). Informally, performing the $\otimes$ between two constraints means building a new constraint whose support involves all the variables of the original ones, and which associates with each tuple of domain values for such variables a semiring element which is obtained by multiplying the elements associated by the original constraints to the appropriate sub-tuples.

Given a constraint $c \in \mathcal{C}$ and a variable $v \in V$, the *projection* [2–4] of c over $V - \{v\}$, written $c \Downarrow_{(V-\{v\})}$ is the constraint c' s.t. $c'\eta = \sum_{d \in D} c\eta[v := d]$. Informally, projecting means eliminating some variables from the support. This is done by associating with each tuple over the remaining variables a semiring element which is the sum of the elements associated by the original constraint to all the extensions of this tuple over the eliminated variables.

To treat the hiding operator of the language, a general notion of existential quantifier is introduced by using notions similar to those used in cylindric algebras. Consider a set of variables V with domain D and the corresponding soft constraint system $\mathcal{C}$. For each $x \in V$ the hiding function [2, 4] is the function $(\exists_x c)\eta = \sum_{d_i \in D} c\eta[x := d_i]$.

To model parameter passing, for each $x, y \in V$ a diagonal constraint [2, 4] is defined as $d_{xy} \in \mathcal{C}$ s.t., $d_{xy}\eta[x := a, y := b] = \mathbf{1}$ if $a = b$ and $d_{xy}\eta[x := a, y := b] = \mathbf{0}$ if $a \neq b$. Now it is possible to define a constraint systems *"a la Saraswat"* [4]. Consider the set $\mathcal{C}$ and the partial order $\sqsubseteq$. Then an entailment relation $\vdash \subseteq \wp(\mathcal{C}) \times \mathcal{C}$ is defined s.t. for each $C \in \wp(\mathcal{C})$ and $c \in \mathcal{C}$, we have $C \vdash c \iff \bigotimes C \sqsubseteq c$ (see also [2, 4]). Notice that in *sccp*, algebricity is not required, since the algebraic nature of the structure $\mathcal{C}$ strictly depends on the properties of the semiring [4].

If we consider a semiring $S = \langle \mathcal{A}, +, \times, \mathbf{0}, \mathbf{1} \rangle$, a domain of the variables D, a set of variables V, the corresponding structure $\mathcal{C}$, then $S_C = \langle \mathcal{C}, \otimes, \bar{\mathbf{0}}, \bar{\mathbf{1}}, \exists_x, d_{xy} \rangle$[5] is a cylindric constraint system [4].

2.2 Timed Concurrent Constraint Programming

When querying the store for some information which is not present (yet), a *(s)ccp* agent will simply suspend until the required information has arrived. In timed applications however often one cannot wait indefinitely for an event. Consider for example the case of a connection to a web service providing some on-line banking facility. In case the connection cannot be established, after a reasonable amount of time an appropriate time-out message has to be communicated to the user. A timed language should then allow us to specify that, in case a given time bound is exceeded (i.e. a *time-out* occurs), the wait is interrupted and an alternative action is taken.

In order to be able to specify this kind of timing constraints, in [20] and [8] the authors introduced a different timed extension of *ccp* (the differences between these two languages are explained in [8]). In particular, the timed ccp (*tccp*) language defined in [8] introduces a discrete global clock and assumes that *ask* and *tell* actions take one time-unit. Computation evolves in steps of one time-unit, so called clock-cycles, which are syntactically separated by action prefixing. Moreover *maximal parallelism* is assumed, that is at each moment every enabled agent of the system is activated (this implies that parallel processes are executed on different processors). Finally in *tccp* it is introduced a primitive construct of the form **now** c **then** A **else** B which can be interpreted as follows: if the constraint c is entailed by the store at the current time t then the above agent behaves as A at time t, otherwise it behaves as B at time t. By using the **now** construct one can express time-out, preemption and other timed programming idioms. For example, the agent **now** c **then** A **else** $ask(true) \rightarrow$ (**now** c **then** A **else** B) waits at most two time unit for the satisfaction of the guard c: If the

[5] $\bar{\mathbf{0}}$ and $\bar{\mathbf{1}}$ that represent respectively the constraints associating $\mathbf{0}$ and $\mathbf{1}$ to all the assignment of domain values.

guard is satisfied (in two time units) then the agent behaves as A, otherwise as B. By using an inductive definition it is easy to define in terms of the **now** the more general time-out agent $(\Sigma_{i=1}^{n}\mathbf{ask}(c_i) \longrightarrow A_i)$ $\mathbf{timeout}(m)B$ which allows to wait at most m time units for the satisfaction of one of the guards (see [8]).

3 Timed Soft Concurrent Constraint Programming

In this section we present the *tscc* language, which originates from both *tccp* and *sccp*. To obtain *tscc* we extend the *cc* language by introducing constructs to handle the cut level and constructs to handle temporal aspects. More precisely, we inherit from *sccp* the **tell** and **ask** constructs enriched by a threshold, which allows to specify when the agents have to succeed or to suspend. Moreover we derive from *tccp* the timing construct **now** c **then** A **else** B previously mentioned. However, differently from the case of *tccp*, the **now** operator here is modified by using thresholds, analogously to the case of **tell** and **ask**.

Definition 1 (*tscc* Language). *Given a soft constraint system $\langle S, D, V\rangle$, the corresponding structure $\mathcal{C}$, any semiring value a and any constraint $\phi \in \mathcal{C}$, the syntax of the tscc language is given by the following grammar:*

$$
\begin{array}{l}
P ::= F.A \\
F ::= p(x) :: A \\
A ::= \mathbf{success} \mid \mathbf{tell}(c) \rightarrow_{\Phi} A \mid \mathbf{tell}(c) \rightarrow^{a} A \mid E \mid A \parallel A \mid \exists x A \mid p(x) \mid \\
\qquad \Sigma_{i=1}^{n} E_i \mid \mathbf{now}_{\Phi}\ c\ \mathbf{then}\ A\ \mathbf{else}\ B \mid \mathbf{now}^{a}\ c\ \mathbf{then}\ A\ \mathbf{else}\ B \\
E ::= \mathbf{ask}(c) \rightarrow_{\Phi} A \mid \mathbf{ask}(c) \rightarrow^{a} A
\end{array}
$$

where, as usual, P is the class of processes, F is the class of sequences of procedure declarations (or clauses), A is the class of agents. The c is supposed to be a soft constraint *in $\mathcal{C}$. A* tsccp process *P is then an object of the form $F.A$, where F is a set of procedure declarations of the form $p(x) :: A$ and A is an agent.*

In the following, given an agent A, we denote by $Fv(A)$ the set of the free variables of A (namely, the variables which do not appear in the scope of the $\exists$ quantifier). As previously mentioned, differently from the original *cc* syntax in *tsccp* we have a semiring element a and constraint ϕ to be checked whenever an *ask* or *tell* operation is performed. Intuitively the level a (resp., ϕ) will be used as a cut level to prune computations that are not good enough. These levels, with an analogous meaning, are present also in the **now** c **then** A **else** B construct, differently from all the previous cc like languages. The remaining of the syntax is standard: Action prefixing is denoted by $\rightarrow$, Σ denotes guarded choice, $\parallel$ indicates parallel composition and a notion of locality is introduced by the agent $\exists x A$ which behaves like A with x considered local to A, thus hiding the information on x provided by the external environment. In the following we also assume guarded recursion, that is we assume that each procedure call is in the scope of either an **ask** or a **tell** construct.

4 An Operational Semantics for *tsccp* Agents

The operational model of *tscc* agents can be formally described by a transition system $T = (Conf, \longrightarrow)$ where we assume that each transition step takes exactly one time-unit. Configurations (in) *Conf* are pairs consisting of a process and a constraint in $\mathcal{C}$ representing the common *store*. The transition relation $\longrightarrow \subseteq Conf \times Conf$ is the least relation satisfying the rules **R1-R17** in Fig. 1 and characterizes the (temporal) evolution of the system. So, $\langle A, \gamma\rangle \longrightarrow \langle B, \delta\rangle$ means that if at time t we have the process A and the store γ then at time $t+1$ we have the process B and the store δ. Let us now briefly discuss the rules in Fig. 1.

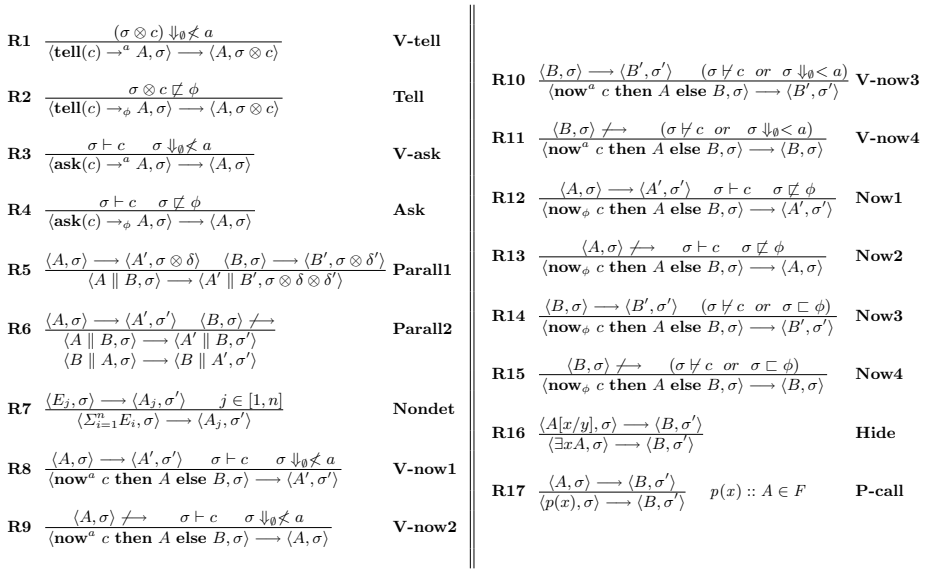

Fig. 1. The transition system for *tsccp*.

Valued-tell The valued-tell rule checks for the a-consistency of the *Soft Constraint Satisfaction Problem* [2] (SCSP) defined by the store $\sigma \otimes c$. A SCSP P is a-consistent if $blevel(P) = a$, where $blevel(P) = Sol(P) \Downarrow_\emptyset$, i.e. the *best level of consistency* of the problem P is a semiring value representing the least upper bound among the values yielded by the solutions. Rule **R1** can be applied only if the store $\sigma \otimes c$ is b-consistent with $b \not< a$[6]. In this case the agent evolves to the new agent A over the store $\sigma \otimes c$. Note that different choices of the *cut level* a could possibly lead to different computations. Finally note that the updated store $\sigma \otimes c$ will be visible only starting from the next time instant since each transition step involves exactly one time-unit.

[6] Notice that we use $b \not< a$ instead of $b \geq a$ because we can possibly deal with partial orders. The same happens also in other transition rules with $\not\sqsubset$ instead of $\sqsupseteq$.

Tell The tell action is a finer check of the store. In this case, a pointwise comparison between the store $\sigma \otimes c$ and the constraint ϕ is performed. The idea is to perform an overall check of the store and to continue the computation only if there is the possibility to compute a solution not worse than ϕ. As for the valued tell, the updated store will be visible from the next time instant.

Valued-ask The semantics of the valued-ask is extended in a way similar to what we have done for the valued-tell action. This means that, to apply the rule, we need to check if the store σ entails the constraint c and also if the store is "consistent enough" w.r.t. the threshold a set by the programmer.

Ask Similar to the *tell* rule, here a finer (pointwise) threshold ϕ is compared to the store σ. Notice that we need to check $\sigma \not\sqsubseteq \phi$ because previous tells could have a different threshold ϕ' and could not guarantee the consistency of the resulting store.

Nondeterminism According to rule **R7** the guarded choice operator gives rise to global non-determinism: the external environment can affect the choice since $\mathbf{ask}(c_j)$ is enabled at time t (and A_j is started at time $t+1$) if and only if the store σ entails c_j (and is compatible with the threshold), and σ can be modified by other agents.

Parallelism Rules **R5** and **R6** model the parallel composition operator in terms of *maximal parallelism*: the agent $A \parallel B$ executes in one time-unit all the initial enabled actions of A and B. Considering rule **R5**, notice that the ordering of the operands in $\sigma \otimes \delta \otimes \delta'$ is not relevant, since $\otimes$ is commutative and associative. Moreover, for the same two properties, if $\sigma \otimes \delta = \sigma \otimes \gamma$ and $\sigma \otimes \delta' = \sigma \otimes \gamma'$, we have that $\sigma \otimes \delta \otimes \delta' = \sigma \otimes \gamma \otimes \gamma'$. Therefore the resulting store $\sigma \otimes \delta \otimes \delta'$ is independent from the choice of the constraint δ such that $\langle A, \sigma \rangle \longrightarrow \langle A', \sigma' \rangle$ and $\sigma' = \sigma \otimes \delta$ (analogously for δ').

Hidden variables The agent $\exists x A$ behaves like A, with x considered *local* to A. This is obtained by substituting the variable x for a variable y which we assume to be new and not used by any other process (standard renaming techniques can be used to ensure this); here $A[x/y]$ denotes the process obtained from A by replacing the variable x for the variable y.

Procedure calls Rule **R17** treats the case of a procedure call when the actual parameter equals the formal parameter. We do not need more rules since, for the sake of simplicity, here and in the following we assume that the set F of procedure declarations is closed w.r.t. parameter names: that is, for every procedure call $p(y)$ appearing in a process $F.A$ we assume that if the original declaration for p in F is $p(x) :: A$ then F contains also the declaration $p(y) :: \exists x(\mathbf{tell}(d_{xy}) \parallel A)$[7]. Moreover, we assume that if $p(x) :: A \in F$ then $Fv(A) \subseteq x$.

Valued-Now The rules **R8-R11** show that the agent $\mathbf{now}^a\ c\ \mathbf{then}\ A\ \mathbf{else}\ B$ behaves as A if c is entailed by the store and the store is "consistent enough" w.r.t. the threshold a, and behaves as B otherwise. Note that, differently from the case of the ask here the evaluation of the guard is instantaneous:

[7] Here the (original) formal parameter is identified as a local alias of the actual parameter.

if $\langle A, \sigma\rangle$ ($\langle B, \sigma\rangle$) can make a transition at time t and the condition on the store and the cut level are satisfied then the agent **now** c **then** A **else** B can make the same transition at time t (and analogously for B). Moreover observe that, due to rules **R9** and **R11**, in any case the control is passed either to A (if the conditions are satisfied) or to B (if not), also if A and B cannot make any transition at the current time instant.

Now The rules **R12-R15** are similar to rules **R8-R11** described before, with the exception that here a finer (pointwise) threshold ϕ is compared to the store σ, analogously to what happens with the Tell and Ask agents.

Using the transition system described by (the rules in) Fig. 1 we can now define our notion of observables, which considers for each *tsccp* process $P = F.A$, the results of successful terminating computations that the agent A can perform.

Definition 2 (Observables). *Let $P = F.A$ be a* tsccp *process. We define*

$$\mathcal{O}_{io}(P) = \{\gamma \Downarrow_{Fv(A)} | \ \langle A, \bar{\mathbf{1}}\rangle \longrightarrow^* \langle \mathbf{Success}, \gamma\rangle\},$$

where **Success** is any agent which contains only occurrences of the agent **success** and of the operator $\|$.

5 The Denotational Model

In this section we define a denotational characterization of the operational semantics obtained by following the construction in [8] and using *timed reactive sequences* to represent *tsccp* computations. These sequences are similar to those used in the semantics of dataflow languages [14], imperative languages [7] and (timed) *ccp* [10, 8].

The denotational model associates with a process a set of timed reactive sequences of the form $\langle\sigma_1, \gamma_1\rangle \cdots \langle\sigma_n, \gamma_n\rangle\langle\sigma, \sigma\rangle$ where a pair of constraints $\langle\sigma_i, \gamma_i\rangle$ represents a reaction of the given process at time i: intuitively, the process transforms the global store from σ_i to γ_i or, in other words, σ_i is the assumption on the external environment while γ_i is the contribution of the process itself (which entails always the assumption). The last pair denotes a "stuttering step" in which the agent **Success** has been reached. Since the basic actions of *tsccp* are monotonic and we can also model a new input of the external environment by a corresponding tell operation, it is natural to assume that reactive sequences are monotonic. So in the following we will assume that each timed reactive sequence $\langle\sigma_1, \gamma_1\rangle \cdots \langle\sigma_{n-1}, \gamma_{n-1}\rangle\langle\sigma_n, \sigma_n\rangle$ satisfies the following condition: $\gamma_i \vdash \sigma_i$ and $\sigma_j \vdash \gamma_{j-1}$, for any $i \in [1, n-1]$ and $j \in [2, n]$.

The set of all reactive sequences is denoted by $\mathcal{S}$ and its typical elements by $s, s_1 \ldots$, while sets of reactive sequences are denoted by $S, S_1 \ldots$ and ε indicates the empty reactive sequence. Furthermore, $\cdot$ denotes the operator that concatenates sequences. In the following, $Process$ denotes the set of *tsccp* processes.

Formally the definition of the semantics is as follows.

Definition 3 (Processes Semantics). *The semantics $R \in Process \to \mathcal{P}(\mathcal{S})$ is defined as the least fixed-point of the operator $\Phi \in (Process \to \mathcal{P}(\mathcal{S})) \to Process \to \mathcal{P}(\mathcal{S})$ defined by*

$$\begin{aligned}\Phi(I)(F.A) = &\{\langle\sigma,\delta\rangle \cdot w \in \mathcal{S} \mid \langle A,\sigma\rangle \to \langle B,\delta\rangle \text{ and } w \in I(F.B)\}\\ &\cup\\ &\{\langle\sigma,\sigma\rangle \cdot w \in \mathcal{S} \mid \langle A,\sigma\rangle \not\to \text{ and either } A \neq \textbf{Success} \text{ and } w \in I(F.A)\\ &\qquad\qquad \text{or } A = \textbf{Success} \text{ and } w \in I(F.A) \cup \{\varepsilon\}\}.\end{aligned}$$

The ordering on $Process \to \mathcal{P}(\mathcal{S})$ is that of (point-wise extended) set-inclusion and since it is straightforward to check that Φ is continuous, standard results ensure that the least fixpoint exists (and it is equal to $\sqcup_{n\geq 0}\Phi^n(\bot)$).

Note that $R(F.A)$ is the union of the set of all successful reactive sequences which start with a reaction of P and the set of all successful reactive sequences which start with a stuttering step of P. In fact, when an agent is blocked, i.e. it cannot react to the input of the environment, a stuttering step is generated. After such a stuttering step the computation can either continue with the further evaluation of A (possibly generating more stuttering steps) or it can terminate, if A is the **Success** agent. Note also that, since the **Success** agent used in the transition system cannot make any move, an arbitrary (finite) sequence of stuttering steps is always appended to each reactive sequence.

5.1 Compositionality of the Denotational Semantics for *tsccp* Processes

In order to prove the compositionality of the denotational semantics we now introduce a semantics $[\![F.A]\!](e)$ which is compositional by definition and where, for technical reasons, we represent explicitly the environment e which associates a denotation to each procedure identifier. More precisely, assuming that *Pvar* denotes the set of procedure identifier, $Env = Pvar \to \mathcal{P}(\mathcal{S})$, with typical element e, is the set of *environments*. Given $e \in Env$, $p \in Pvar$ and $f \in \mathcal{P}(\mathcal{S})$, we denote by $e' = e\{f/p\}$ the new environment such that $e'(p) = f$ and $e'(p') = e(p')$ for each procedure identifier $p' \neq p$.

Given a process $F.A$, the denotational semantics $[\![F.A]\!] : Env \to \mathcal{P}(\mathcal{S})$ is defined by the equations in Fig. 2, where μ denotes the least fixpoint w.r.t. subset inclusion of elements of $\mathcal{P}(\mathcal{S})$. The semantic operators appearing in Fig. 2 are formally defined as follows. Intuitively they reflect, in terms of reactive sequences, the operational behavior of their syntactic counterparts[8].

We first need the following definition. Let σ, ϕ and c be constraints in $\mathcal{C}$ and let $a \in \mathcal{A}$. We say that

– $\sigma \tilde{\to}^a c$, if $(\sigma \vdash c$ and $\sigma \Downarrow_\emptyset \not\leq a)$ while $\sigma \tilde{\to}_\phi c$, if $(\sigma \vdash c$ and $\sigma \not\sqsubseteq \phi)$.

Definition 4 (Semantic operators). *Let S, S_i be sets of reactive sequences, c, c_i be constraints and let $\tilde{\to}_i$ be either of the form $\tilde{\to}^{a_i}$ or $\tilde{\to}_{\phi_i}$. Then we define the operators $\tilde{tell}$, $\tilde{\sum}$, $\tilde{\|}$, $\tilde{now}$ and $\tilde{\exists}x$ as follows:*

[8] In Fig. 2 the syntactic operator $\to_i$ is either of the form $\to^{a_i}$ or $\to_{\phi_i}$.

The (valued) tell operator

$$\tilde{tell}^a(c,S) = \{s \in \mathcal{S} \mid s = \langle \sigma, \sigma \otimes c\rangle \cdot s',\ \sigma \otimes c \Downarrow_\emptyset \not\prec a \text{ and } s' \in S\ \}.$$

$$\tilde{tell}_\phi(c,S) = \{s \in \mathcal{S} \mid s = \langle \sigma, \sigma \otimes c\rangle \cdot s',\ \sigma \otimes c \not\sqsubset \phi \text{ and } s' \in S\ \}.$$

The guarded choice

$$\tilde{\sum}_{i=1}^{n} c_i \tilde{\rightarrow}_i S_i = \{s \cdot s' \in \mathcal{S} \mid \begin{array}{l} s = \langle \sigma_1, \sigma_1\rangle \cdots \langle \sigma_m, \sigma_m\rangle, \sigma_j \not\tilde{\rightarrow}_i c_i \\ \text{for each } j \in [1, m\text{-}1], i \in [1,n], \\ \sigma_m \tilde{\rightarrow}_h c_h \text{ and } s' \in S_h \text{ for an } h \in [1,n]\ \} \end{array}$$

The parallel composition *Let* $\tilde{\|} \in \mathcal{S} \times \mathcal{S} \rightarrow \mathcal{S}$ *be the (commutative and associative) partial operator defined as follows:*

$$\langle \sigma_1, \sigma_1 \otimes \gamma_1\rangle \cdots \langle \sigma_n, \sigma_n \otimes \gamma_n\rangle \langle \sigma, \sigma\rangle\ \tilde{\|}\ \langle \sigma_1, \sigma_1 \otimes \delta_1\rangle \cdots \langle \sigma_n, \sigma_n \otimes \delta_n\rangle \langle \sigma, \sigma\rangle = \langle \sigma_1, \sigma_1 \otimes \gamma_1 \otimes \delta_1\rangle \cdots \langle \sigma_n, \sigma_n \otimes \gamma_n \otimes \delta_n\rangle \langle \sigma, \sigma\rangle.$$

We define $S_1 \tilde{\|} S_2$ *as the point-wise extension of the above operator to sets.*

The (valued) now operator

$$\tilde{now}^a(c, S_1, S_2) = \{s \in \mathcal{S} \mid s = \langle \sigma, \sigma'\rangle \cdot s' \text{ and either } \sigma \tilde{\rightarrow}^a c \text{ and } s \in S_1 \text{ or } \sigma \tilde{\rightarrow}^a c \text{ does not hold and } s \in S_2\ \}.$$

$$\tilde{now}_\phi(c, S_1, S_2) = \{s \in \mathcal{S} \mid s = \langle \sigma, \sigma'\rangle \cdot s' \text{ and either } \sigma \tilde{\rightarrow}_\phi c \text{ and } s \in S_1 \text{ or } \sigma \tilde{\rightarrow}_\phi c \text{ does not hold and } s \in S_2\ \}.$$

The hiding operator *The semantic hiding operator can be defined as follows:*

$$\tilde{\exists} x S = \{s \in \mathcal{S} \mid \text{there exists } s' \in S \text{ such that } s = s'[x/y] \text{ with } y \text{ new}\ \}$$

where $s'[x/y]$ *denotes the sequence obtained from* s' *by replacing the variable* x *for the variable* y *that we assume to be new*[9].

A few explanations are in order here. The semantic (valued) tell operator reflects in the obvious way the operational behavior of the syntactic (valued) tell. Concerning the semantic choice operator, a sequence in $\tilde{\sum}_{i=1}^{n} c_i \tilde{\rightarrow}_i S_i$ consists of an initial period of waiting for a store which satisfies one of the guards. During this waiting period only the environment is active by producing the constraints σ_j while the process itself generates the stuttering steps $\langle \sigma_j, \sigma_j\rangle$. When the store is strong enough to satisfy a guard, that is to entail a c_h and to satisfy the condition on the cut level the resulting sequence is obtained by adding $s' \in S_h$ to the initial waiting period. In the semantic parallel operator defined on sequences we require that the two arguments of the operator agree at each point of time

[9] To be more precise, we assume that each time that we consider a new applications of the operator $\tilde{\exists}$ we use a new, different y. As in the case of the operational semantics, this can be ensured by a suitable renaming mechanism.

E1 $[\![F.\mathbf{success}]\!](e) = \{\langle\sigma_1,\sigma_1\rangle\langle\sigma_2,\sigma_2\rangle\cdots\langle\sigma_n,\sigma_n\rangle \in \mathcal{S} \mid n \geq 1\}$

E2 $[\![F.\mathbf{tell}(c) \rightarrow^a A]\!](e) = \tilde{tell}^a(c, [\![F.A]\!](e))$

E3 $[\![F.\mathbf{tell}(c) \rightarrow_\phi A]\!](e) = \tilde{tell}_\phi(c, [\![F.A]\!](e))$

E4 $[\![F.\sum_{i=1}^n \mathbf{ask}(c_i) \rightarrow_i A_i]\!](e) = \tilde{\sum}_{i=1}^n c_i \tilde{\rightarrow}_i [\![F.A_i]\!](e)$

E5 $[\![F.\mathbf{now}^a\ c\ \mathbf{then}\ A\ \mathbf{else}\ B]\!](e) = n\tilde{o}w^a(c, [\![F.A]\!](e), [\![F.B]\!](e))$

E6 $[\![F.\mathbf{now}_\phi\ c\ \mathbf{then}\ A\ \mathbf{else}\ B]\!](e) = n\tilde{o}w_\phi(c, [\![F.A]\!](e), [\![F.B]\!](e))$

E7 $[\![F.A \parallel B]\!](e) = [\![F.A]\!](e)\ \tilde{\|}\ [\![G.B]\!](e)$

E8 $[\![F.\exists x A]\!](e) = \tilde{\exists} x [\![F.A]\!](e)$

E9 $[\![F.p(x)]\!](e) = \mu\Psi \qquad$ where $\Psi(f) = [\![F \setminus \{p\}.A]\!](e\{f/p\}), \quad p(x) :: A \in F$

Fig. 2. The semantics $[\![F.A]\!](e)$.

with respect to the contribution of the environment (the σ_i's) and that they have the same length (in all other cases the parallel composition is assumed being undefined).

If $F.A$ is a closed process, that is if all the procedure names occurring in A are defined in F, then $[\![F.A]\!](e)$ does not depend on e and will be indicated as $[\![F.A]\!]$. Environments in general allow us to define the semantics also of processes which are not closed. The following result shows the correspondence between the two semantics we have introduced and therefore the compositionality of $R(F.A)$.

Theorem 1 (Compositionality). *If $F.A$ is closed then $R(F.A) = [\![F.A]\!]$ holds.*

The proof of Theo. 1 is similar to the one proposed in [8] for the compositionality property of the *tccp* denotational semantics.

6 Correctness

The observables $\mathcal{O}_{io}(P)$ describing the input/output pairs of successful computations can be obtained from $R(P)$ by considering suitable sequences, namely those sequences which do not perform assumptions on the store. In fact, notice that some reactive sequences do not correspond to real computations: Clearly, when considering a real computation no further contribution from the environment is possible. This means that, at each step, the assumption on the current store must be equal to the store produced by the previous step. In other words, for any two consecutive steps $\langle\sigma_i,\sigma_i'\rangle\langle\sigma_{i+1},\sigma_{i+1}'\rangle$ we must have $\sigma_i' = \sigma_{i+1}$. So we are led to the following.

Definition 5 (Connected Sequences). *Let $s = \langle\sigma_1,\sigma_1'\rangle\langle\sigma_2,\sigma_2'\rangle\cdots\langle\sigma_n,\sigma_n\rangle$ be a reactive sequence. We say that s is connected if $\sigma_1 = \bar{\mathbf{1}}$ and $\sigma_i = \sigma_{i-1}'$ for each i, $2 \leq i \leq n$.*

According to the previous definition, a sequence is connected if all the information assumed on the store is produced by the process itself, apart from the initial input. To be defined as connected, a sequence must also have $\bar{\mathbf{1}}$ as the initial constraint. A connected sequence represents a *tsccp* computation, as it will be proved by the following theorem.

Theorem 2 (Correctness). *For any process $P = F.A$ we have*

$$\mathcal{O}_{io}(P) = \{\sigma_n \Downarrow_{Fv(A)} | \text{ there exists a connected sequence } s \in R(P) \text{ such that } s = \langle\sigma_1, \sigma_2\rangle\langle\sigma_2, \sigma_3\rangle \cdots \langle\sigma_n, \sigma_n\rangle\}.$$

The proof of Theo. 2 is similar to the one proposed in [8] for the correctness property of the *tccp* language.

7 Programming Idioms and an Auction Example

We can consider the primitives in Fig. 1 to derive the soft version of the programming idioms in [8], which are typical of reactive programming.

Delay. The delay constructs $\mathbf{tell}(c) \xrightarrow{t}_{\phi} A$ or $\mathbf{ask}(c) \xrightarrow{t}_{\phi} A$ are used to delay the execution of agent A after the execution of $\mathbf{tell}(c)$ or $\mathbf{ask}(c)$; t is the number of the time-units of delay. Therefore, in addiction to a constraint ϕ, in *tsccp* the transition arrow can have also a number of delay slots. This idiom can be defined by induction: the base case is $\xrightarrow{0}_{\phi} A \equiv \longrightarrow_{\phi} A$ and the inductive step is $\xrightarrow{n+1}_{\phi} A \equiv \longrightarrow_{\phi} \mathbf{tell}(\bar{\mathbf{1}}) \xrightarrow{n}_{\phi} A$. The valued version can be defined in an analogous way.

Timeout. The timed guarded choice agent $(\Sigma_{i=1}^{n} \mathbf{ask}(c_i) \longrightarrow_i A_i)$ $\mathbf{timeout}(m)$ B waits at most m time-units ($m \geq 0$) for the satisfaction of one of the guards; notice that all the ask actions have a "soft" transition arrow, i.e. $\longrightarrow_i$ is either of the form $\longrightarrow_{\phi_i}$ or $\longrightarrow^{a_i}$, as in Fig. 1. Before this time-out, the process behaves just like the guarded choice: as soon as there exist enabled guards, one of them (and the corresponding branch) is nondeterministically selected. After waiting for m time-units, if no guard is enabled, the timed choice agent behaves as B.

Watchdog. Watchdogs are used to interrupt the activity of a process on a signal from a specific event. The idiom $\mathbf{do}$ (A) $\mathbf{watching}(c)$ $\mathbf{else}$ (B) behaves as A, as long as c is not entailed by the store; when c is entailed, the process A is immediately aborted. The reaction is instantaneous, in the sense that A is aborted at the same time instant of the detection of the entailment of c.

Both *Timeout* and *Watchdogs* constructs can be assembled through the composition of several $\mathbf{now}_{\Phi}$ c $\mathbf{then}$ A $\mathbf{else}$ B or $\mathbf{now}^{a}$ c $\mathbf{then}$ A $\mathbf{else}$ B primitives, exactly as sketched in Section 2.2 and explained in detail in [8] (in the soft version of the timeout, the $\mathbf{else}$ $\mathbf{ask}(true)$ in Sec. 2.2 must be replaced with $\mathbf{else}$ $\mathbf{ask}(\bar{\mathbf{1}})$). For example, $\mathbf{do}$ ($\mathbf{tell}(c_1)$) $\mathbf{watching}(c_2)$ $\mathbf{else}$ B $\equiv$ $\mathbf{now}$ c_2 $\mathbf{then}$ B $\mathbf{else}$ $\mathbf{tell}(c_1)$, where the $\mathbf{now}$ can be valued or not. Clearly, in

tsccp all the constraints (e.g. c_1 and c_2) are soft. With this small set of idioms, we have now enough expressiveness to describe complex interactions.

In Fig. 3 we model the negotiation and the management of a generic service offered with a sort of auction: auctions, as other forms of negotiation, naturally need both timed and qualitative/quantitative means to describe the interactions among agents. The auctioneer (i.e. $AUCTIONEER$ in Fig. 3) begins by offering a service described with the soft constraint c_{A_1}. We suppose that the cost associated to the soft constraint is expressed in terms of computational capabilities needed to support the execution: $c_1 \sqsubseteq c_2$ means that the service described by c_1 needs more computational resources than c_2. By choosing the proper semiring, this load can be expressed as a percentage of the CPU use, or in terms of money, for example. We suppose that a constraint can be defined over three domains of QoS features: availability, reliability and execution time. For instance, c_{A_1} could be $availability > 95\% \wedge reliability > 99\% \wedge execution\ time < 3sec$. Clearly, providing a higher availability or reliability, and a lower execution time implies raising the computational resources, thus worsening the preference of the store.

$\underline{AUCTIONEER}$ ::
$INIT\_A \longrightarrow$
$\mathbf{tell}(c_{A_1}) \xrightarrow{t_{sell}} (\Sigma_{i=1}^{n} \mathbf{ask}(bidder_i = i) \longrightarrow^{a_A} \mathbf{tell}(winner = i) \longrightarrow CHECK)\ \mathbf{timeout}(wait_{auct})$
$\mathbf{tell}(c_{A_2}) \xrightarrow{t_{sell}} (\Sigma_{i=1}^{n} \mathbf{ask}(bidder_i = i) \longrightarrow^{a_A} \mathbf{tell}(winner = i) \longrightarrow CHECK)\ \mathbf{timeout}(wait_{auct})$
$\mathbf{tell}(c_{A_3}) \xrightarrow{t_{sell}} (\Sigma_{i=1}^{n} \mathbf{ask}(bidder_i = i) \longrightarrow^{a_A} \mathbf{tell}(winner = i) \longrightarrow CHECK)\ \mathbf{timeout}(wait_{auct})$
$\longrightarrow \mathbf{success}$

$\underline{CHECK}$ ::
$\mathbf{do}\ (\ (\mathbf{ask}(service = end) \longrightarrow \mathbf{success})\ \mathbf{timeout}(wait_{check})\ \mathbf{tell}(service = interrupt)\ )$
$\mathbf{watching}(c_{check})\ \mathbf{else}\ (\mathbf{tell}(service = interrupt) \longrightarrow STOP_c)$

$\underline{BIDDER_i}$ ::
$INIT\_B_i \longrightarrow$
$\mathbf{do}\ (\ TASK_i\ )\ \mathbf{watching}(c_{B_i})\ \mathbf{else}\ \mathbf{ask}(\bar{1}) \xrightarrow{t_{buy_i}} \mathbf{tell}(bidder_i = i) \longrightarrow$
$(\ (\mathbf{ask}(winner = i) \longrightarrow USE_i) + (\mathbf{ask}(winner \neq i) \longrightarrow \mathbf{success})\ )$

$\underline{USE_i}$ ::
$\mathbf{do}\ (\ USE\_SERVICE_i \longrightarrow \mathbf{tell}(service = end) \longrightarrow \mathbf{success}\ )$
$\mathbf{watching}(service = interrupt)\ \mathbf{else}\ (STOP_i)$

$\underline{AUCTION\&SUPERVISE}$ :: $AUCTIONEER \,||\, BIDER_1 \,||\, BIDDER_2 \,||\, \ldots \,||BIDDER_n$

Fig. 3. An "auction and management" example for a generic service

After the offer, the auctioneer gives time to the bidders (each of them described with a possibly different $BIDDER_i$ agent in Fig. 3) to make their offer, since the choice of the winner is delayed by t_{sell} time-units (as in many real-world auction schemes). A level a_A is used to effectively check that the global consistency of the store is enough good, i.e. the computational power would not be already consumed under the given threshold. After the winner is nondeterministically chosen among all the bidders asking for the service, the auctioneer becomes a supervisor of the used resource by executing the $CHECK$ agent. Otherwise, if no offer is received within $wait_{auct}$ time-units, a timeout interrupts the

wait and the auctioneer improves the offered service by adding a new constraint: for example, in **tell**(c_{A_2}), c_{A_2} could be equivalent to *execution time* $< 1sec$, thus reducing the latency of the service (from 3 to 1 seconds) and consequently raising, at the same time, its computational cost (i.e. $c_{A_2} \otimes \sigma \sqsubseteq \sigma$, we worsen the consistency level of the store). The same offer/wait process is repeated three times in Fig. 3. Each of the bidders in Fig. 3 is executing its own task (i.e. $TASK_i$), but as soon as the offered resource meets its demand of computational power (i.e. c_{B_i} is satisfied by the store: $\sigma \sqsubseteq c_{B_i}$), the bidder is interrupted and then asks to use the service. The time needed to react and make an offer is modeled with t_{buy_i}: fast bidders will have more chances to win the auction, if their request arrives before the choice of the auctioneer. If one bidder wins, then it becomes a user of the resource, by executing USE_i.

The USE_i agent uses the service (with the $USE\_SERVICE_i$ agent, left generic in Fig. 3), but it stops ($STOP_i$ agent, left generic in Fig. 3) as soon as the service is interrupted, i.e. as the store satisfies $service = interrupt$. On the other side, the $CHECK$ agent waits for the use termination, but it interrupts the user if the computation takes too long (more than $wait_{check}$ time-units), or if the user absorbs the computational capabilities beyond a given threshold, i.e. as soon as the c_{check} becomes implied by the store (i.e. $\sigma \sqsubseteq c_{check}$): in fact, $USE\_SERVICE_i$ could be allowed to ask for more power by "telling" some more constraints to the store. To interrupt the service use, the $CHECK$ agent performs a **tell**($service = interrupt$). All the $INIT$ agents, left generic in Fig. 3, can be used to initialize the computation.

In order to avoid a heavy notation in Fig. 3, we do not show the preference associated to constraints and the consistency check label on the transition arrows, when they are not significative for the example description.

Many other real-life automated tasks can be modeled with the *tscc* language, for example a quality-driven composition of web services: the agents that represent different web services can add to the store their functionalities (represented by soft constraints) with **tell** actions; the final store models their composition. The consistency level of the store sums up to a value the (for example) total cost of the single obtained service, or a value representing the consistency of the integrated functionalities: the reason is that when we compose the services offered by different providers, we could not be sure how much they are compatible. Then, a client wishing to use the composed service can perform an **ask** with threshold that prevents it from paying a high price or have an unreliable service. Softness is useful also to model incomplete service specifications that may evolve incrementally and, in general, non-functional aspects. Time sensitiveness is clearly needed too: all the most important orchestration/choreography languages of today (e.g. *BPEL4WS* and *WSCI*) support timeouts, the raising of events and delay activities [18].

8 Related and Future Work

We have introduced the *tscc* language in order to join together the expressive capabilities of soft constraints and timing mechanisms in a new programming framework. The agents modeled with this language are now able to deal with time and preference dependent decisions that can often be found during complex interactions: an example can be represented by entities that need to negotiate a satisfying QoS and manage generic resources. Mechanisms as timeout and interrupt can be very useful when waiting for pending conditions or when triggering some new necessary actions. All the *tsccp* rules have been formally described by a transition system and then also with a denotational characterization of the operational semantics obtained with the use of *timed reactive sequences*. The resulting semantics has been proved to be compositional and correct.

Other timed extension of concurrent constraint programming have been proposed in [16, 17, 20], however these languages, differently from *tsccp*, do not take into account quantitative aspects; therefore, this achievement represents a very important expressivity improvement w.r.t. related works. These have been considered by Di Pierro and Wiklicky who have extensively studied probabilistic *ccp* (see for example [11]). This language provides a construct for probabilistic choice which allows one to express randomness in a program, without assuming any additional structure on the underlying constraint system. This approach is therefore deeply different from ours. Recently stochastic *ccp* has been introduced in [6] to model biological systems. This language is obtained by adding a stochastic duration to the ask and tell primitives, thus differs from ours.

A first improvement of *tsccp* can be the inclusion of a *fail* agent in the syntax given in Definition 1. The transition system we have defined considers only successful computations. If this could be a reasonable choice in a don't know interpretation of the language it will lead to an insufficient analysis of the behavior in a pessimistic interpretation of the indeterminism. A second extension for this framework could be represented by considering *interleaving* (as in the classical *ccp*) instead of *maximal parallelism*, which is the scheduling policy followed in this paper when observing the parallel execution of agents. According to this policy, at each moment every enabled agent of the system is activated, while in the first paradigm an agent could not be assigned to a "free" processor.

Clearly, since we have dynamic process creation, a maximal parallelism approach has the disadvantage that in general it implies the existence of an unbound number of processes. On the other hand a naif interleaving semantic could be problematic form the time viewpoint, as in principle the time does not pass for enabled agent which are not scheduled. A possible solution, analogous to that one adopted in [9], could be to assume that the parallel operator is interpreted in terms of interleaving, as usual, however we must assume maximal parallelism for actions depending on time. In other words, time passes for all the parallel processes involved in a computation. To summarize, we could adopt maximal parallelism for time elapsing (i.e. for evaluating a (valued) **now** agent) and an interleaving model for basic computation steps (i.e. (valued) **ask** and (valued) **tell** actions).

At last, we would like to consider other time management strategies (as the one proposed in [21]) and to study how timing and non-monotonic constructs [5] can be integrated together.

References

1. G. Berry and G. Gonthier. The esterel synchronous programming language: design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, 1992.
2. S. Bistarelli. *Semirings for Soft Constraint Solving and Programming (LNCS)*. SpringerVerlag, 2004.
3. S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based constraint satisfaction and optimization. *J. ACM*, 44(2):201–236, 1997.
4. S. Bistarelli, U. Montanari, and F. Rossi. Soft concurrent constraint programming. *ACM Trans. Comput. Logic*, 7(3):563–589, 2006.
5. S. Bistarelli and F. Santini. A nonmonotonic soft constraint based language to model QoS negotiation. In *Doctoral Consortium of AAAI-08 (to appear)*. AAAI Press, 2008.
6. L. Bortolussi. Stochastic concurrent constraint programming. *Electr. Notes Theor. Comput. Sci.*, 164(3):65–80, 2006.
7. S. D. Brookes. Full abstraction for a shared variable parallel language. In *LICS*, pages 98–109. IEEE Computer Society, 1993.
8. F. S. de Boer, M. Gabbrielli, and M. C. Meo. A timed concurrent constraint language. *Inf. Comput.*, 161(1):45–83, 2000.
9. F. S. de Boer, M. Gabbrielli, and M. C. Meo. A timed linda language and its denotational semantics. *Fundam. Inf.*, 63(4):309–330, 2004.
10. F. S. de Boer and C. Palamidessi. A fully abstract model for concurrent constraint programming. In *TAPSOFT '91: Proceedings of CAAP '91*, volume 1, pages 296–319, New York, USA, 1991. Springer-Verlag New York, Inc.
11. A. Di Pierro and H. Wiklicky. Probabilistic concurrent constraint programming: Towards a fully abstract model. In *MFCS '98: Proceedings of Mathematical Foundations of Computer Science*, pages 446–455, London, UK, 1998. Springer-Verlag.
12. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.
13. D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.
14. B. Jonsson. A model and proof system for asynchronous networks. In *PODC '85: Proceedings ACM symposium on Principles of distributed computing*, pages 49–58, New York, USA, 1985. ACM Press.
15. P. le Guernic, M. le Borgne, T. Gautier, and C. le Maire. Programming real-time applications with signal. *Proceedings of the IEEE*, 79(9):1321–1336, 1991.
16. M. Nielsen and F. D. Valencia. Temporal concurrent constraint programming: Applications and behavior. In *Formal and Natural Computing - Essays Dedicated to Grzegorz Rozenberg*, pages 298–324, London, UK, 2002. Springer-Verlag.
17. C. Palamidessi and F. D. Valencia. A temporal concurrent constraint programming calculus. In *CP '01: Proceedings of Principles and Practice of Constraint Programming*, pages 302–316, London, UK, 2001. Springer-Verlag.
18. C. Peltz. Web services orchestration and choreography. *Computer*, 36(10):46–52, 2003.

19. V. Saraswat. Concurrent constraint programming languages, January 1989. PhD thesis.
20. V. Saraswat, R. Jagadeesan, and V. Gupta. Timed default concurrent constraint programming. *J. Symb. Comput.*, 22(5-6):475–520, 1996.
21. F. D. Valencia. Timed concurrent constraint programming: Decidability results and their application to LTL. In *ICLP*, volume 2916 of *LNCS*, pages 422–437. Springer, 2003.