TLP **15** (6): 743–782, 2015. © Cambridge University Press 2014
doi:10.1017/S1471068414000106 First published online 6 June 2014

Timed soft concurrent constraint programs: An interleaved and a parallel approach

STEFANO BISTARELLI
Dipartimento di Matematica e Informatica, Università di Perugia
Via Vanvitelli 1, 06123 Perugia, Italy
(*e-mail:* vista@dmi.unipg.it)

MAURIZIO GABBRIELLI
Dipartimento di Scienze dell'Informazione, Università di Bologna
Via Zamboni 33, 40126 Bologna, Italy
(*e-mail:* gabbri@cs.unibo.it)

MARIA CHIARA MEO
Dipartimento di Economia, Università "G. D'Annunzio"
Viale Pindaro 42, 65127 Pescara, Italy
(*e-mail:* cmeo@unich.it)

FRANCESCO SANTINI
Centrum Wiskunde & Informatica (CWI)
Science Park 123, 1098XG Amsterdam, The Netherlands
(*e-mail:* F.Santini@cwi.nl)

submitted 20 May 2012; revised 15 February 2014; accepted 18 February 2014

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 that 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 this 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. After presenting a semantics using maximal parallelism of actions, we also describe a version for their interleaving on a single processor (with maximal parallelism for time elapsing). Coordinating agents that need to take decisions on both preference values and time events may benefit from this language.

KEYWORDS: Soft Concurrent Constraint Programming, Timed Concurrent Constraint Programming, interleaving, parallelism

https://doi.org/10.1017/S1471068414000106 Published online by Cambridge University Press

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 timing and quantitative aspects related to 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 on these scores: Consider, for example, some generic communicating agents that need to take decisions on a (monetary) cost or a (fuzzy) preference for a shared resource. They both need to coordinate through time-dependent and preference-based decisions.

A practical example of such agents corresponds, for example, to software agents that need to negotiate some service-level agreement on a resource, or a service, with time-related side-conditions. For instance, a fitting example is given by auction schemes, where the seller/bidder agents need to agree on a preference for a given prize (e.g., a monetary cost). At the same time, the agents have to respect some timeout and alarm events respectively representing the absence and presence of bids for the prize (for instance). The language we present in this paper is well suited for this kind of interactions, as Section 5 shows with examples.

The *Timed Concurrent Constraint Programming* (*tccp*), a timed extension of pure formalism of *Concurrent Constraint Programming* (*ccp*) (Saraswat 1989), has been introduced in de Boer *et al.* (2000). The language is based on the hypothesis of *bounded asynchrony* (Saraswat *et al.* 1996): Computation takes a bounded period of time rather than being instantaneous, as in the concurrent synchronous languages `ESTEREL` (Berry and Gonthier 1992), `LUSTRE` (Halbwachs *et al.* 1991), `SIGNAL` (le Guernic *et al.* 1991) and Statecharts (Harel 1987). Time itself is measured by a discrete global clock, i.e., the internal clock of the *tccp* process. In de Boer *et al.* (2000), the authors also introduced *timed reactive sequences*, which describe the reaction of a *tccp* process to the input of external environment, at each moment in time. Formally, such a reaction is a pair of constraints $\langle c, d \rangle$, where c is the input and d is the constraint produced by the process in response to c.

Soft constraints (Bistarelli *et al.* 1997; Bistarelli 2004) 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 (Bistarelli *et al.* 2006), 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). We adopt soft constraints instead of crisp ones, since classic constraints show evident limitations when trying to represent real-life scenarios, where the knowledge is not completely available nor crisp.

https://doi.org/10.1017/S1471068414000106 Published online by Cambridge University Press

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 *tsccp*, we directly introduce a timed interpretation of the usual programming constructs of *sccp* by identifying a time-unit with the time needed for the execution of a basic *sccp* action (ask and tell), and by interpreting action prefixing as the next-time operator. An explicit timing primitive is also introduced in order to allow for the specification of timeouts. In the first place, the parallel operator of *tsccp* is first interpreted in terms of maximal parallelism, as in de Boer *et al.* (2000). Second, we also consider a different paradigm, where the parallel operator is interpreted in terms of interleaving, however, assuming maximal parallelism for actions depending on time. In other words, time passes for all the parallel processes involved in a computation. This approach, analogous to the one adopted in de Boer *et al.* (2004), is different from the one of de Boer *et al.* (2000) and Bistarelli *et al.* (2008) (where maximal parallelism was assumed for any kind of action), and it is also different from the one considered in Busi *et al.* (2000), where time does not elapse for timeout constructs. This can be accomplished by allowing all the time-only-dependent actions (τ-transitions) to concurrently run with at most one action manipulating the store (a ω-transition).

The paper extends the results in Bistarelli *et al.* (2008) by providing new semantics that allows maximal parallelism for time elapsing and an interleaving model for basic computation steps (see Section 7). This new language is called *tsccp with interleaving*, i.e., *tsccp-i*, to distinguish it from the version allowing maximal parallelism of all actions. According to the maximal parallelism policy (applied, for example, in the original works as Saraswat (1989) and Saraswat *et al.* (1994)), at each moment every enabled agent of the system is activated, while in the interleaving paradigm only one of the enabled agents is executed. This second paradigm is more realistic if we consider limited resources, since it does not imply the existence of an unbounded number of processors. However, in de Boer *et al.* (2000) it is shown that the notion of maximal parallelism of *tsccp* is more expressive than the notion of interleaving parallelism of other concurrent constraint languages. The presence of maximal parallelism can force the computation to discard some (non-enabled) branches which could became enabled later on (because of the information produced by parallel agents), while this is not possible when considering an interleaving model. Therefore, *tsccp* is sensitive to delays in adding constraints to the store, whereas this is not the case for *ccp* and *tsccp-i*.

The rest of the paper is organized as follows: In Section 2, we summarize the most important background notions and frameworks from which *tsccp* is derived, i.e. *tccp* and *sccp*. In Section 3, we present the *tsccp* language, and Section 4 describes the operational semantics of *tscc* agents. Section 5 better explains the programming idioms as *timeout* and *interrupt*, exemplifies the use of timed paradigms in the *tscc* language and shows an application example on modeling an auction interaction among several bidders and a single auctioneer. Section 6 describes the denotational semantics for *tsccp*, and proves the denotational model correctness with the aid of *connected reactive sequences*. Section 7 explains the semantics for interleaving with maximal parallelism of time-elapsing actions (i.e., the *tsccp-i* language), while

https://doi.org/10.1017/S1471068414000106 Published online by Cambridge University Press

Section 8 describes a timeline for the execution of three parallel agents in *tsccp-i*. Section 9 describes the denotational semantics of *tsccp-i* and proves the correctness of the denotational model. Section 10 reports the related work, and lastly Section 11 concludes by also indicating the future research.

2 Background

2.1 Soft constraints

A *soft constraint* (Bistarelli *et al.* 1997; Bistarelli 2004) may be seen as a constraint where each instantiation of its variables has an associated value from a partially ordered set that can be interpreted as a set of preference values. Combining constraints will then have to take into account such additional values, and thus the formalism has also to provide suitable operations for combination ($\times$) and comparison ($+$) of tuples of values and constraints. That is why this formalization is based on the concept of *c-semiring* (Bistarelli *et al.* 1997; Bistarelli 2004), just called semiring in the rest of the paper.

Semirings. 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 $\leqslant_S$ over A such that $a \leqslant_S b$ iff $a + b = b$. Then it is possible to prove that (see Bistarelli *et al.* 1997): (i) $\leqslant_S$ is a partial order; (ii) $+$ and $\times$ are monotone on $\leqslant_S$; (iii) $\mathbf{0}$ is its minimum and $\mathbf{1}$ its maximum; (iv) $\langle A, \leqslant_S\rangle$ is a complete lattice (a complete lattice is a partially ordered set in which all subsets have both a supremum and an infimum) and, for all $a, b \in A$, $a + b = lub(a, b)$ (where *lub* is the *least upper bound*).

Moreover, if $\times$ is idempotent, then: $+$ distributes over $\times$; $\langle A, \leqslant_S\rangle$ is a complete distributive lattice and $\times$ is its *greatest lower bound* (*glb*). Informally, the relation $\leqslant_S$ gives us a way to compare semiring values and constraints. In fact, when we have $a \leqslant_S b$, we will say that *b is better than a*. In the following, when the semiring will be clear from the context, $a \leqslant_S b$ will be often indicated by $a \leqslant b$.

Constraint system. Given a semiring $S = \langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$ and an ordered 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, $\mathscr{C} = \eta \to A$ is a set of all possible constraints that can be built starting from S, D and V.

Any function in $\mathscr{C}$ involves all the variables in V, but we impose that it depends on the assignment of their finite subset only. 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 $v := d_1$ (that is the operator [ ] has precedence over application). Note also that $c\eta$ is the application of a constraint function $c : (V \to D) \to A$ to a function $\eta : V \to D$; what we obtain is a semiring value $c\eta$.

https://doi.org/10.1017/S1471068414000106 Published online by Cambridge University Press

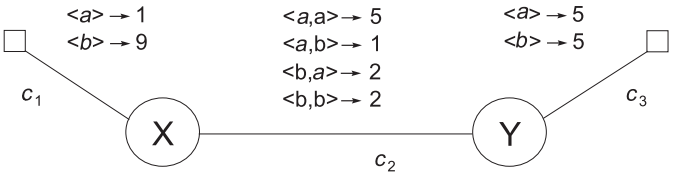


Fig. 1. An SCSP based on a weighted semiring.

The partial order $\leqslant_S$ over $\mathscr{C}$ can be easily extended among constraints by defining $c_1 \sqsubseteq c_2 \iff c_1\eta \leqslant c_2\eta$ for each possible η.

Combining and projecting soft constraints. Given the set $\mathscr{C}$, the combination function $\otimes : \mathscr{C} \times \mathscr{C} \to \mathscr{C}$ is defined as $(c_1 \otimes c_2)\eta = c_1\eta \times c_2\eta$ (see also Bistarelli *et al.* 1997, 2006; Bistarelli 2004). Informally, performing the $\otimes$ between two constraints means building a new constraint whose support involves all the variables of the original ones, and associates with each tuple of domain values for such variables a semiring element obtained by multiplying the elements associated by the original constraints to the appropriate sub-tuples.

Given a constraint $c \in \mathscr{C}$ and a variable $v \in V$, the *projection* (Bistarelli *et al.* 1997, 2006; Bistarelli 2004) of c over $V - \{v\}$, written as $c \Downarrow_{(V-\{v\})}$ 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.

We also define also a function $\bar{a}$ (Bistarelli 2004; Bistarelli *et al.* 2006) as the function that returns the semiring value a for all assignments η, that is, $\bar{a}\eta = a$. We will usually write $\bar{a}$ simply as a. An example of constants that will be useful later are $\bar{\mathbf{0}}$ and $\bar{\mathbf{1}}$ that represent respectively the constraints associating $\mathbf{0}$ and $\mathbf{1}$ to all the assignment of domain values.

Solutions. An SCSP (Bistarelli 2004) is defined as $P = \langle V, D, C, S\rangle$, where C is the set of constraints defined over variables in V (each with domain D), and whose preference is determined by semiring S. The *best level of consistency* notion is defined as $blevel(P) = Sol(P) \Downarrow_{\emptyset}$, where $Sol(P) = \bigotimes C$ (Bistarelli 2004). A problem P is α-consistent if $blevel(P) = \alpha$ (Bistarelli 2004). P is instead simply "consistent" iff there exists $\alpha >_S \mathbf{0}$ such that P is α-consistent. P is inconsistent if it is not consistent.

Example 1

Figure 1 shows a weighted SCSP as a graph: the weighted semiring is used, i.e., $\langle \mathbb{R}^+ \cup \{\infty\}, \min, \hat{+}, \infty, 0\rangle$ ($\hat{+}$ is the arithmetic plus operation). Variables and constraints are represented respectively by nodes and arcs (unary for c_1-c_3, and binary for c_2); $D = \{a, b\}$. The solution of the CSP in Figure 1 associates a semiring element to every domain value of variables X and Y by combining all the constraints together, i.e., $Sol(P) = \bigotimes C$. For instance, for the tuple $\langle a, a\rangle$ (that is, $X = Y = a$), we have to compute the sum of 1 (which is the value assigned to $X = a$ in constraint c_1), 5 ($\langle X = a, Y = a\rangle$ in c_2) and 5 ($Y = a$ in c_3): the value for this tuple is 11. The

https://doi.org/10.1017/S1471068414000106 Published online by Cambridge University Press

solution $X = a, Y = b$ is a 7-consistent solution, where 7 corresponds to the *blevel* of P, i.e., $Sol(P) \Downarrow_{\emptyset} = 7$.

2.2 Concurrent constraint programming over soft constraints

The basic idea underlying *ccp* (Saraswat 1989) is that computation progresses via monotonic accumulation of information in a constraint global store. Information is produced by the concurrent and asynchronous activity of several agents that can add (*tell*) a constraint to the store. Dually, agents can also check (*ask*) whether a constraint is entailed by the store, thus allowing synchronization among different agents. The *ccp* languages are defined parametrically w.r.t. a given *constraint system*. The notion of constraint system has been formalized in Saraswat and Rinard (1990) following Scott's treatment of information systems. Soft constraints over a semiring $S = \langle \mathscr{A}, +, \times, \mathbf{0}, \mathbf{1} \rangle$ and an ordered set of variables V (over a domain D) have been showed to form a constraint system *"à la Saraswat,"* thus leading to the definition of *sccp* (Bistarelli *et al.* 1997, 2006; Bistarelli 2004).

Consider the set $\mathscr{C}$ and the partial order $\sqsubseteq$. Then an entailment relation $\vdash \subseteq \wp(\mathscr{C}) \times \mathscr{C}$ is defined s.t. for each $C \in \wp(\mathscr{C})$ and $c \in \mathscr{C}$, we have $C \vdash c \iff \bigotimes C \sqsubseteq c$ (see also Bistarelli 2004; Bistarelli *et al.* 2006). Note that in this setting the notion of token (constraint) and set of tokens (set of constraints) closed under entailment is used indifferently. In fact, given a set of constraint functions C_1, its closure w.r.t. entailment is a set $\bar{C}_1$ that contains all the constraints greater than $\bigotimes C_1$. This set is univocally representable by the constraint function $\bigotimes C_1$. The definition of the entailment operator $\vdash$ on top of $\mathscr{C}$, and that of $\sqsubseteq$ relation, lead to the notion of *soft constraint system*. It is also important to note that in Saraswat and Rinard (1989) it is claimed that a constraint system is a *complete algebraic* lattice. In the *sccp* framework, algebraicity is not required (Bistarelli *et al.* 2006) instead, since the algebraic nature of the structure $\mathscr{C}$ strictly depends on the properties of the semiring.[1]

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 $\mathscr{C}$. For each $x \in V$, the hiding function (Bistarelli 2004; Bistarelli *et al.* 2006) is the function $(\exists_x c)\eta = \sum_{d_i \in D} c\eta[x := d_i]$. To make the hiding operator computationally tractable, it is required that the number of domain elements in D, having semiring values different from $\mathbf{0}$, is finite (Bistarelli *et al.* 2006). In this way, to compute the sum needed for $(\exists_x c)\eta$, we can consider just a finite number of elements (those different from $\mathbf{0}$), since $\mathbf{0}$ is the unit element of the sum. Note that by using the hiding function we can represent operator $\Downarrow$ defined in Section 2.1. In fact, for any constraint c and any variable $x \subseteq V$, $c \Downarrow_{V-x} = \exists_x c$ (Bistarelli *et al.* 2006).

[1] Note that we do not aim at computing the closure of the entailment relation, but only to use the entailment relation to establish whether a constraint is entailed by the current store, and this can be established even if the lattice is not algebraic (that is, even if the times operator is not idempotent).

https://doi.org/10.1017/S1471068414000106 Published online by Cambridge University Press

To model parameter passing also diagonal elements have to be defined. Consider a set of variables V and the corresponding soft constraint system. Then, for each $x, y \in V$, a diagonal constraint is defined as $d_{xy} \in \mathscr{C}$ 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 $a \neq b$ (Bistarelli 2004; Bistarelli *et al.* 2006).

Theorem 1 (*Cylindric constraint system* (Bistarelli *et al.* 2006))
Consider a semiring $S = \langle \mathscr{A}, +, \times, \mathbf{0}, \mathbf{1} \rangle$, a domain of variables D, an ordered set of variables V, and the corresponding structure $\mathscr{C}$. Then $S_C = \langle \mathscr{C}, \otimes, \mathbf{0}, \mathbf{1}, \exists_x, d_{xy} \rangle$ is a cylindric constraint system.

2.3 Timed concurrent constraint programming

A timed extension of *ccp*, called *tccp*, has been introduced in de Boer *et al.* (2000). Similarly to other existing timed extensions of *ccp* defined in Saraswat *et al.* (1996), *tccp* is a language for reactive programming designed around the hypothesis of *bounded asynchrony* (as introduced in Saraswat *et al.* (1996): computation takes a bounded period of time rather than being instantaneous).

When querying the store for some information that is not present (yet), a *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 online banking facility. In case the connection cannot be established, after a reasonable amount of time an appropriate timeout 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 *timeout* occurs), the wait is interrupted and an alternative action is taken. Moreover, in some cases it is also necessary to have a preemption mechanism that allows one to abort an active process A and to start a process B when a specific (abnormal) event occurs.

In order to be able to specify these timing constraints, *tccp* introduces a discrete global clock and assumes that *ask* and *tell* actions take one time-unit. Computation evolves in steps of one time-unit, the so-called clock-cycles. Action prefixing is the syntactic marker that distinguishes a time instant from the next one and it is assumed that parallel processes are executed on different processors, which implies that, at each moment, every enabled agent of the system is activated. This assumption gives rise to what is called *maximal parallelism.* The time in between two successive moments of the global clock intuitively corresponds to the response time of the underlying constraint system. Thus, all parallel agents are synchronized by the response time of the underlying constraint system. Since the store is monotonically increasing and one can have dynamic process creation, clearly the previous assumptions imply that the constraint solver takes a constant time (no matter how big the store is), and that there is an unbounded number of processors. However, one can impose suitable restriction on programs, thus ensuring that the (significant part of the) store and the number of processes do not exceed a fixed bound; these restrictions would still allow significant forms of recursion with parameters.

https://doi.org/10.1017/S1471068414000106 Published online by Cambridge University Press

Furthermore, a timing construct of the form **now** c **then** A **else** B is introduced in *tccp*, whose semantics is 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. This basic construct allows to derive such timing mechanisms as timeout and preemption (Saraswat *et al.* 1996; de Boer *et al.* 2000). The instantaneous reaction can be obtained by evaluating **now**c in parallel with A and B within the same time-unit. At the end of this time-unit, the store will be updated by using either the constraint produced by A, or the one produced by B, depending on the result of the evaluation of **now**c. Clearly, since A and B could contain nested **now then else** agents, a limit for the number of these nested agents should be fixed. Note that, for recursive programs, such a limit is ensured by the presence of the procedure-call, since we assume that the evaluation of such calls takes one time-unit.

3 Timed soft concurrent constraint programming

In this section, we present the *tsccp* language, which originates from both *tccp* and *sccp*. To obtain this aim, we extend the syntax of the *cc* language with the timing construct **now**c **then**A **else**B (inherited from *tccp*), and also in order to directly handle the cut level as in *sccp*. This means that the syntax and semantics of **tell**, **ask** and **now** agents have to be enriched with a threshold that is used to check when the agents may succeed, or suspend.

Definition 1 (*tsccp language*)
Given a soft constraint system $\langle S, D, V \rangle$, the corresponding structure $\mathscr{C}$, any semiring value a, *soft constraints* ϕ, $c \in \mathscr{C}$ and any tuple of variables x, the syntax of the *tsccp* language is given by the following grammar:

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

where, as usual, P is the class of processes, F is the class of sequences of procedure declarations (or clauses) and A is the class of agents. In a *tsccp process* $P = F.A$, A is the initial agent, to be executed in the context of the set of declarations F. Agent **success** represents a successful termination, so it may not make any further transition.

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). Besides the use of soft constraints (see Section 2.2) instead of crisp ones, there are two fundamental differences between *tsccp* and *ccp*. The first main difference w.r.t. the original *cc* syntax is the presence of a semiring element a and a constraint ϕ to be checked whenever an *ask* or *tell* operation is performed. More precisely, the level a (respectively, ϕ) will be used as a cut level to prune computations that are not good enough. The second main difference with respect to *ccp* (but, this time,

https://doi.org/10.1017/S1471068414000106 Published online by Cambridge University Press

also with respect to *sccp*) is instead the presence of the **now**c **then** A **else** B construct introduced in Section 2.3. Even for this construct, the level a (or ϕ) is used as a cut level to prune computations.

Action prefixing is denoted by $\rightarrow$, non-determinism is introduced via the guarded choice construct $\Sigma_{i=1}^{n} E_i$, parallel composition is denoted by $\|$, and a notion of locality is introduced by agent $\exists xA$, which behaves like A with x considered local to A, thus hiding the information on x provided by the external environment.

In the next section we formally describe the operational semantics of *tsccp*. In order to simplify the notation, in the following we will usually write a *tsccp process* $P = F.A$ simply as the corresponding agent A.

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 of a constraint in $\mathscr{C}$, representing the common *store* shared by all the agents. The transition relation $\longrightarrow \subseteq Conf \times Conf$ is the least relation satisfying the rules **R1**–**R17** in Figure 2, and it characterizes the (temporal) evolution of the system. So $\langle A, \sigma \rangle \longrightarrow \langle B, \delta \rangle$ means that if at time t we have the process A and the store σ, then at time $t+1$ we have the process B and the store δ.

Let us now briefly discuss the rules in Figure 2. Here is a brief description of transition rules:

Valued-tell. The valued-tell rule checks for the a-consistency of the *Soft Constraint Satisfaction Problem* (SCSP) (Bistarelli 2004) defined by the store $\sigma \otimes c$. An SCSP P is a-consistent if $blevel(P) = a$, where $blevel(P) = Sol(P) \Downarrow_{\emptyset}$, i.e., the *best level of consistency* of 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 $b \not< a$.[2] In this case the agent evolves to a 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.

Tell. The tell action is a finer check of the store. In this case (see rule **R2**), 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. Note that this notion of tell could be also applied to the classical *cc* framework: The tell operation would succeed when the set of tuples satisfying constraint ϕ is not a superset of the set of tuples allowed by $\sigma \cap c$.[3] As for the valued tell, the updated

[2] Note that we use $b \not< a$ instead of $b \geqslant a$ because we can possibly deal with partial orders. The same also holds for $\not\sqsubset$ instead of $\sqsupseteq$.

[3] Note that the $\otimes$ operator in the crisp case reduces to set intersection.

https://doi.org/10.1017/S1471068414000106 Published online by Cambridge University Press

R1 $\dfrac{(\sigma \otimes c) \Downarrow_\emptyset \not< a}{\langle \mathbf{tell}(c) \rightarrow^a A, \sigma\rangle \longrightarrow \langle A, \sigma \otimes c\rangle}$ **V-tell**

R2 $\dfrac{\sigma \otimes c \not\sqsubset \phi}{\langle \mathbf{tell}(c) \rightarrow_\phi A, \sigma\rangle \longrightarrow \langle A, \sigma \otimes c\rangle}$ **Tell**

R3 $\dfrac{\sigma \vdash c \qquad \sigma \Downarrow_\emptyset \not< a}{\langle \mathbf{ask}(c) \rightarrow^a A, \sigma\rangle \longrightarrow \langle A, \sigma\rangle}$ **V-ask**

R4 $\dfrac{\sigma \vdash c \qquad \sigma \not\sqsubset \phi}{\langle \mathbf{ask}(c) \rightarrow_\phi A, \sigma\rangle \longrightarrow \langle A, \sigma\rangle}$ **Ask**

R5 $\dfrac{\langle A, \sigma\rangle \longrightarrow \langle A', \sigma \otimes \delta\rangle \qquad \langle B, \sigma\rangle \longrightarrow \langle B', \sigma \otimes \delta'\rangle}{\langle A \parallel B, \sigma\rangle \longrightarrow \langle A' \parallel B', \sigma \otimes \delta \otimes \delta'\rangle}$ **Parall1**

R6 $\dfrac{\langle A, \sigma\rangle \longrightarrow \langle A', \sigma'\rangle \qquad \langle B, \sigma\rangle \not\longrightarrow}{\begin{array}{c}\langle A \parallel B, \sigma\rangle \longrightarrow \langle A' \parallel B, \sigma'\rangle \\ \langle B \parallel A, \sigma\rangle \longrightarrow \langle B \parallel A', \sigma'\rangle\end{array}}$ **Parall2**

R7 $\dfrac{\langle E_j, \sigma\rangle \longrightarrow \langle A_j, \sigma'\rangle \qquad j \in [1, n]}{\langle \Sigma_{i=1}^n E_i, \sigma\rangle \longrightarrow \langle A_j, \sigma'\rangle}$ **Nondet**

R8 $\dfrac{\langle A, \sigma\rangle \longrightarrow \langle A', \sigma'\rangle \qquad \sigma \vdash c \qquad \sigma \Downarrow_\emptyset \not< a}{\langle \mathbf{now}^a\ c\ \mathbf{then}\ A\ \mathbf{else}\ B, \sigma\rangle \longrightarrow \langle A', \sigma'\rangle}$ **V-now1**

R9 $\dfrac{\langle A, \sigma\rangle \not\longrightarrow \qquad \sigma \vdash c \qquad \sigma \Downarrow_\emptyset \not< a}{\langle \mathbf{now}^a\ c\ \mathbf{then}\ A\ \mathbf{else}\ B, \sigma\rangle \longrightarrow \langle A, \sigma\rangle}$ **V-now2**

R10 $\dfrac{\langle B, \sigma\rangle \longrightarrow \langle B', \sigma'\rangle \qquad \sigma \nvdash c \qquad \sigma \Downarrow_\emptyset \not< a}{\langle \mathbf{now}^a\ c\ \mathbf{then}\ A\ \mathbf{else}\ B, \sigma\rangle \longrightarrow \langle B', \sigma'\rangle}$ **V-now3**

R11 $\dfrac{\langle B, \sigma\rangle \not\longrightarrow \qquad \sigma \nvdash c \qquad \sigma \Downarrow_\emptyset \not< a}{\langle \mathbf{now}^a\ c\ \mathbf{then}\ A\ \mathbf{else}\ B, \sigma\rangle \longrightarrow \langle B, \sigma\rangle}$ **V-now4**

R12 $\dfrac{\langle A, \sigma\rangle \longrightarrow \langle A', \sigma'\rangle \qquad \sigma \vdash c \qquad \sigma \not\sqsubset \phi}{\langle \mathbf{now}_\phi\ c\ \mathbf{then}\ A\ \mathbf{else}\ B, \sigma\rangle \longrightarrow \langle A', \sigma'\rangle}$ **Now1**

R13 $\dfrac{\langle A, \sigma\rangle \not\longrightarrow \qquad \sigma \vdash c \qquad \sigma \not\sqsubset \phi}{\langle \mathbf{now}_\phi\ c\ \mathbf{then}\ A\ \mathbf{else}\ B, \sigma\rangle \longrightarrow \langle A, \sigma\rangle}$ **Now2**

R14 $\dfrac{\langle B, \sigma\rangle \longrightarrow \langle B', \sigma'\rangle \qquad \sigma \nvdash c \qquad \sigma \not\sqsubset \phi}{\langle \mathbf{now}_\phi\ c\ \mathbf{then}\ A\ \mathbf{else}\ B, \sigma\rangle \longrightarrow \langle B', \sigma'\rangle}$ **Now3**

R15 $\dfrac{\langle B, \sigma\rangle \not\longrightarrow \qquad \sigma \nvdash c \qquad \sigma \not\sqsubset \phi}{\langle \mathbf{now}_\phi\ c\ \mathbf{then}\ A\ \mathbf{else}\ B, \sigma\rangle \longrightarrow \langle B, \sigma\rangle}$ **Now4**

R16 $\dfrac{\langle A[x/y], \sigma\rangle \longrightarrow \langle B, \sigma'\rangle}{\langle \exists x A, \sigma\rangle \longrightarrow \langle B, \sigma'\rangle}$ **Hide**

R17 $\langle p(x), \sigma\rangle \longrightarrow \langle A, \sigma\rangle \qquad p(x) :: A \in F$ **P-call**

Fig. 2. The transition system for *tsccp*.

https://doi.org/10.1017/S1471068414000106 Published online by Cambridge University Press

store $\sigma \otimes c$ will be visible only since the next time instant. In the following, let us use **tell**$(c) \rightarrow A$ and **tell**(c) as a shorthand for **tell**$(c) \rightarrow_{\bar{0}} A$ and **tell**$(c) \rightarrow_{\bar{0}}$ **success** respectively.

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 **R3**, we need to check if the store σ entails the constraint c, and also if σ is "consistent enough" w.r.t. the threshold a set by the programmer.

Ask. In rule **R4**, we check if the store σ entails the constraint c, but, similar to rule **R2**, we also compare a finer (pointwise) threshold ϕ to the store σ. As for the tell action, let us use **ask**$(c) \rightarrow A$ as a shorthand for **ask**$(c) \rightarrow_{\bar{0}} A$.

Parallelism. Rules **R5** and **R6** model the parallel composition operator in terms of *maximal parallelism*: Agent $A \parallel B$ executes in one time-unit all the initial enabled actions of A and B. Considering rule **R5** (where *maximal parallelism* is accomplished in practice), note that the ordering of 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 of the choice of constraint δ such that $\langle A, \sigma \rangle \longrightarrow \langle A', \sigma' \rangle$ and $\sigma' = \sigma \otimes \delta$ (analogously for δ').

Nondeterminism. According to rule **R7**, the guarded choice operator gives rise to global non-determinism: The external environment can affect the choice, since **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 if it is compatible with the threshold too), and σ can be modified by other agents.

Valued-now and Now. Rules **R8**–**R11** show that the agent **now**^{a} c **then** A **else** B behaves as A or B depending on the fact that c is or is not entailed by the store, provided that the current store σ is compatible with the threshold. Differently from the case of the ask, here the evaluation of the guard is instantaneous: If current store σ is compatible with the threshold a, $\langle A, \sigma \rangle$ ($\langle B, \sigma \rangle$) can make a transition at time t and c is (is not) entailed by the store σ, then the agent **now**^{a} c **then** A **else** B can make the same transition at time t. Moreover, observe that in any case the control is passed either to A (if c is entailed by the current store σ and σ is compatible with the threshold) or to B (in case σ does not entail c and σ is compatible with the threshold). Analogous for the not-valued version, i.e., **now**$_\phi$ c **then** A **else** B (see rules **R12**–**R15**). Finally, we use **now** c **then** A **else** B as a shorthand for the agent **now**$_{\bar{0}}$ c **then** A **else** B.

Hiding variables. Agent $\exists x A$ behaves like A, with x considered *local* to A, as shown by rule **R16**. This is obtained by substituting variable x for variable y, which we assume to be new and not used by any other process. Standard renaming techniques can be used to ensure this; in rule **R16**, $A[x/y]$ denotes the process obtained from A by replacing variable x for 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

https://doi.org/10.1017/S1471068414000106 Published online by Cambridge University Press

$p(x) :: A$, then F also contains the declaration $p(y) :: \exists x(\mathbf{tell}(d_{xy}) \parallel A)$.[4] Moreover, we assume that if $p(x) :: A \in F$, then $Fv(A) \subseteq x$.

Using the transition system described in (the rules in) Figure 2, we can now define our notion of observables, which considers the results of successful terminating computations that agent A can perform for each *tsccp* process $P = F.A$.
Here and in the following, given a transition relation $\longrightarrow$, we denote by $\longrightarrow^*$ its reflexive and transitive closure.

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

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

where **Success** is any agent containing only occurrences of agent **success** and operator $\parallel$.

5 Programming idioms and examples

We can consider the primitives in Definition 1 to derive the soft version of the programming idioms in de Boer *et al.* (2000), which are typical of reactive programming.

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

Timeout. The timed guarded choice agent $\Sigma_{i=1}^{n} \mathbf{ask}(c_i) \rightarrow_i A_i \, \mathbf{timeout}(m) \, B$ waits at most m time-units ($m \geqslant 0$) for the satisfaction of one of the guards; note that all the ask actions have a soft transition arrow, i.e., $\rightarrow_i$ is either of the form $\rightarrow_{\phi_i}$ or $\rightarrow^{a_i}$, as in Figure 2. Before this timeout, the process behaves just like the guarded choice: as soon as there exist enabled guards, one of these (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. Timeout constructs can be assembled through the composition of several $\mathbf{now}_{\phi}\ c\ \mathbf{then}\ A\ \mathbf{else}\ B$ primitives (or their valued version), as explained in de Boer *et al.* (2000) for the (crisp) *tccp* language. The timeout can be defined inductively as follows: Let us denote by A the agent $\Sigma_{i=1}^{n} \mathbf{ask}(c_i) \rightarrow_i A_i$. In the base case, that is $m = 0$, we define $\Sigma_{i=1}^{n} \mathbf{ask}(c_i) \rightarrow_i A_i \, \mathbf{timeout}(0) \, B$ as the agent:

[4] Here the (original) formal parameter is identified as a local alias of the actual parameter. Alternatively, we could have introduced a new rule treating explicitly this case, as it was in the original *ccp* papers.

https://doi.org/10.1017/S1471068414000106 Published online by Cambridge University Press

$$\begin{array}{l}\mathbf{now}_1\ c \mathbf{\ then}\ A \\ \quad \mathbf{else}\ (\ \mathbf{now}_2\ c \mathbf{\ then}\ A \\ \qquad\qquad \mathbf{else}\ (\ldots(\ \mathbf{now}_n\ c_n\ \mathbf{then}\ A\ \mathbf{else}\ \mathbf{ask}(\bar{1}) \rightarrow\ B)\ldots))\end{array}$$

where for $i = 1,\ldots,n$, either $\mathbf{now}_i = \mathbf{now}_{\phi_i}$ if $\rightarrow_i$ is of the form $\rightarrow_{\phi_i}$ or $\mathbf{now}_i = \mathbf{now}^{a_i}$ if $\rightarrow_i$ is of the form $\rightarrow^{a_i}$. Because of the operational semantics explained in rules **R8–R11** (see Figure 2), if a guard c_i is true, then the agent $\Sigma_{i=1}^{n}\mathbf{ask}(c_i) \rightarrow_i A_i$ is evaluated in the same time slot. Otherwise, if no guard c_i is true, agent B is evaluated in the next time slot. Then, by inductively reasoning on the number of time-units m, we can define $\Sigma_{i=1}^{n}\mathbf{ask}(c_i) \rightarrow_i A_i\,\mathbf{timeout}(m)\,B$ as

$$\Sigma_{i=1}^{n}\mathbf{ask}(c_i) \rightarrow_i A_i\,\mathbf{timeout}(0)\,(\Sigma_{i=1}^{n}\mathbf{ask}(c_i) \rightarrow_i A_i\,\mathbf{timeout}(m-1)\,B).$$

Watchdog. Watchdogs are used to interrupt the activity of a process on a signal from a specific event. The idiom **do** A **watching**$_\phi$ c behaves as A as long as c is not entailed by the store and the current store is compatible with the threshold; when c is entailed and the current store is compatible with the threshold, 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. However, according to the computational model, if c is detected at time t, then c has to be produced at time t' with $t' < t$. Thus, we have a form of weak preemption. As well as timeouts, also watchdog agents can be defined in terms of other basic constructs of the language (see Figure 3).

In the following we assume that there exists an (injective) renaming function ρ, which, given a procedure name p, returns a new name $\rho(p)$ that is not used elsewhere in the program. Moreover, let us use $\mathbf{now}_\phi\ c$ **else** B as a shorthand for $\mathbf{now}_\phi\ c$ **then success else** B, where we assume that, for any procedure p declared as $p(x) :: A$, a declaration $\rho(p)(x) ::$ **do** $\rho(A)$ **watching**$_\phi$ c is added, where $\rho(A)$ denotes the agent obtained from A by replacing in it each occurrence of any procedure q by $\rho(q)$. The assumption in the case of $\exists xA$ agent is needed for correctness. In practical cases, it can be satisfied by suitably renaming the variables associated with signals. In the following, $\rightarrow'$ is either of the form $\rightarrow_\psi$ or $\rightarrow^a$. Analogously for $\mathbf{now}'$.

The translation in Figure 3 can be easily extended to the case of the agent **do** A **watching**$_\phi$ c **else** B, which behaves as the previous watchdog and also activates the process B when A is aborted (i.e., when c is entailed and the current state is compatible with the threshold). In the following we will also use this form of watchdog.

The assumption on the instantaneous evaluation of $\mathbf{now}_\phi\ c$ is essential in order to obtain a preemption mechanism that can be expressed in terms of the $\mathbf{now}_\phi$ **then else** primitive. In fact, if the evaluation of $\mathbf{now}_\phi\ c$ took one time-unit, then this unit delay would change the compositional

https://doi.org/10.1017/S1471068414000106 Published online by Cambridge University Press

$$\begin{array}{ll}
\textbf{do success watching}_{\phi}\ c & \Longrightarrow \textbf{success} \\
\textbf{do tell}(d) \rightarrow' A\ \textbf{watching}_{\phi}\ c & \Longrightarrow \textbf{now}_{\phi}\ c\ \textbf{else tell}(d) \rightarrow' \textbf{do}\ A\ \textbf{watching}_{\phi}\ c \\
\textbf{do}\ \Sigma_{i=1}^{n}\textbf{ask}(c_i) \rightarrow_i A_i\ \textbf{watching}_{\phi}\ c & \Longrightarrow \textbf{now}_{\phi}\ c\ \textbf{else}\ \Sigma_{i=1}^{n}\textbf{ask}(c_i) \rightarrow_i \textbf{do}\ A_i\ \textbf{watching}_{\phi}\ c \\
\textbf{do (now}'\ d\ \textbf{then}\ A\ \textbf{else}\ B)\ \textbf{watching}_{\phi}\ c & \Longrightarrow \textbf{now}'\ d\ \textbf{then do}\ A\ \textbf{watching}_{\phi}\ c \\
 & \qquad\qquad \textbf{else do}\ B\ \textbf{watching}_{\phi}\ c \\
\textbf{do}\ A \parallel B\ \textbf{watching}_{\phi}\ c & \Longrightarrow \textbf{do}\ A\ \textbf{watching}_{\phi}\ c \parallel \textbf{do}\ B\ \textbf{watching}_{\phi}\ c \\
\textbf{do}\ \exists x A\ \textbf{watching}_{\phi}\ c & \Longrightarrow \exists x\ \textbf{do}\ A\ \textbf{watching}_{\phi}\ c, \text{ assuming } \exists_x c = c \\
\textbf{do}\ p(x)\ \textbf{watching}_{\phi}\ c & \Longrightarrow \textbf{now}_{\phi}\ c\ \textbf{else}\ \rho(p)(x)\ \textbf{watching}_{\phi}\ c
\end{array}$$

Fig. 3. Examples of watchdog constructs.

$$c_1 : (\{x\} \rightarrow N) \rightarrow R^+ \quad \text{s.t. } c_1(x) = x + 3 \qquad c_2 : (\{x\} \rightarrow N) \rightarrow R^+ \quad \text{s.t. } c_2(x) = x + 5$$
$$c_3 : (\{x\} \rightarrow N) \rightarrow R^+ \quad \text{s.t. } c_3(x) = 2x + 8$$

Fig. 4. Three (weighted) soft constraints: $c_3 = c_1 \otimes c_2$, $c_2 \vdash c_1$, $c_3 \vdash c_1$ and $c_3 \vdash c_2$.

behavior of the agent controlled by the watchdog. Consider, for example, agent $A = \textbf{tell}(a) \rightarrow \textbf{tell}(b)$, which takes two time-units to complete its computation. Agent $A^t = \textbf{now}\ c\ \textbf{else tell}(a) \rightarrow \textbf{now}\ c\ \textbf{else tell}(b)$ (resulting from the translation of $\textbf{do}\ A\ \textbf{watching}_{\bar{0}}\ c$) compositionally behaves as A unless a c signal is detected and the current state is compatible with the threshold, in which case the evaluation of A is interrupted. On the other hand, if the evaluation of **now** c took one time-unit, then A^t would take four time-units and would not behave anymore as A when c is not present. In fact, in this case, agent $A \parallel B$ would produce d while $A^t \parallel B$ would not, where B is the agent $\textbf{ask}(\bar{1}) \rightarrow \textbf{now}\ a\ \textbf{then tell}(d)\ \textbf{else success}$. The valued version of watchdogs can be defined in an analogous way.

With this small set of idioms, we have now enough expressiveness to describe complex interactions. For the following examples on the new programming idioms, we consider the *Weighted* semiring $\langle \mathbb{R}^+ \cup \{+\infty\}, min, +, +\infty, 0\rangle$ (Bistarelli 2004; Bistarelli *et al.* 1997) and the (weighted) soft constraints in Figure 4. We first provide simple program examples to explain as more details as possible on how a computation of *tsccp* agents proceeds. In Section 5.1 we show a more complex example describing classical actions during a negotiation process; the aim of that example is instead to show the expressivity of the *tsccp* language without analyzing its execution in detail.

Example 2 (*Delay*)
As a first very simple example, suppose to have two agents A_1, A_2 of the form: $A_1 :: \textbf{tell}(\bar{1}) \xrightarrow{2}{}^{+\infty} \textbf{tell}(c_2) \rightarrow^{+\infty} \textbf{success}$ and $A_2 :: \textbf{tell}(\bar{1}) \xrightarrow{1}{}^{+\infty} \textbf{ask}(c_1) \rightarrow^{9} \textbf{success}$; their concurrent evaluation in the $\bar{1} \equiv \bar{0}$ empty store is:

$$\langle (\textbf{tell}(\bar{0}) \xrightarrow{2}{}^{+\infty} \textbf{tell}(c_2) \rightarrow^{+\infty} \textbf{success}) \parallel (\textbf{tell}(\bar{0}) \xrightarrow{1}{}^{+\infty} \textbf{ask}(c_1) \rightarrow^{9} \textbf{success}), \bar{0}\rangle.$$

https://doi.org/10.1017/S1471068414000106 Published online by Cambridge University Press

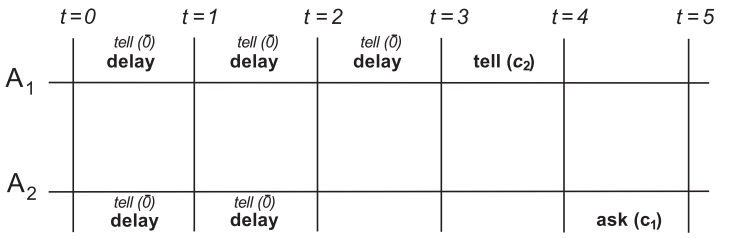


Fig. 5. The timeline of the execution of $A_1 \parallel A_2$ parallel agent in Example 2.

The timeline for this parallel execution is described in Figure 5. For the evaluation of **tell** and **ask** we respectively consider the rules **R1** and **R3** in Figure 2, since both transitions are a-valued. However, both these two actions are delayed: three time-units for the $\mathbf{tell}(c_2)$ of A_1 (including the first $\mathbf{tell}(\bar{0})$), and two time-units for the $\mathbf{ask}(c_1)$ of A_2 (including the first $\mathbf{tell}(\bar{0})$). As explained before, this can be obtained by adding $\bar{\mathbf{1}}$ to the store with a **tell** action respectively three, and two times. Therefore, the parallel agent $A_1 \parallel A_2$ corresponds to:

$$(\mathbf{tell}(\bar{0}) \rightarrow^{+\infty} \mathbf{tell}(\bar{0}) \rightarrow^{+\infty} \mathbf{tell}(\bar{0}) \rightarrow^{+\infty} \mathbf{tell}(c_2) \rightarrow^{+\infty} \mathbf{success}) \parallel$$

$$(\mathbf{tell}(\bar{0}) \rightarrow^{+\infty} \mathbf{tell}(\bar{0}) \rightarrow^{+\infty} \mathbf{ask}(c_1) \rightarrow^{9} \mathbf{success}).$$

This agent is interpreted by using **R5** and **R6** in Figure 2 in terms of maximal parallelism, i.e., all the actions are executed in parallel. The first two **tell** of A_1 and A_2 can be simultaneously executed by using rule **R1**: the precondition $(\bar{0} \otimes \bar{0}) \Downarrow_{\emptyset} = 0 \not< 9$ of the rule is then satisfied. The store does not change since $\bar{0} \otimes \bar{0} = \bar{0}$. At this point, the **ask** action of A_2 is not enabled because $\bar{0} \not\vdash c_1$, that is, the precondition $\sigma \vdash c_1$ of **R3** is not satisfied. Therefore, the processor can only be allocated to A_1 and, since $(\bar{0} \otimes \bar{0}) \Downarrow_{\emptyset} = 0 \not< +\infty$ is true (i.e., the precondition of **R1** is satisfied), at $t = 3$ the computation is in the state:

$$\langle \mathbf{tell}(c_2) \rightarrow^{+\infty} \mathbf{success} \parallel \mathbf{ask}(c_1) \rightarrow^{9} \mathbf{success}, \bar{0} \rangle.$$

Now the **tell** can be executed because $(\bar{0} \otimes c_2) \Downarrow_{\emptyset} = 5 \not< +\infty$: therefore, the store becomes equal to $\bar{0} \otimes c_2 = c_2$:

$$\langle \mathbf{success} \parallel \mathbf{ask}(c_1) \rightarrow^{9} \mathbf{success}, c_2 \rangle.$$

At $t = 5$ (see Figure 5) we can successfully terminate the program: in the store $\sigma = c_2$ the **ask** is finally enabled at $t = 4$, according to the two preconditions of rule **R3**, i.e., $c_2 \vdash c_1$ and $c_2 \Downarrow_{\emptyset} = 5 \not< 9$: therefore, we have $A_1 \parallel A_2 :: \langle \mathbf{success} \parallel \mathbf{success}, c_2 \rangle$.

Example 3 (*Timeout*)
In this second example, we evaluate a timeout construct. Suppose we have two agents A_1 and A_2 of the form:

$$A_1 :: ((\mathbf{ask}(c_1) \rightarrow^{+\infty} \mathbf{success}) + (\mathbf{ask}(c_2) \rightarrow^{+\infty} \mathbf{success}))\ \mathbf{timeout}(1)$$

$$\mathbf{ask}(c_1) \rightarrow^{+\infty} \mathbf{success}$$

and

$$A_2 :: \mathbf{tell}(\bar{0}) \xrightarrow{2} {}^{+\infty} \mathbf{tell}(c_3) \rightarrow^{+\infty} \mathbf{success}$$

https://doi.org/10.1017/S1471068414000106 Published online by Cambridge University Press

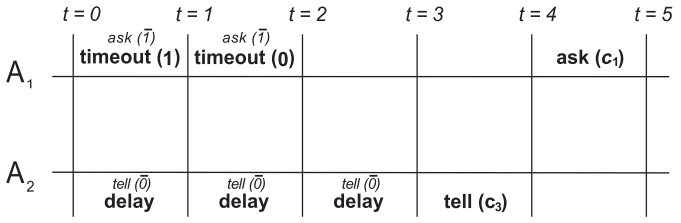


Fig. 6. The timeline of the execution of $A_1 \parallel A_2$ agent in Example 3.

The description of agent A_1 is a shortcut for the following agent, as previously explained in the definition of the timeout:

$$\mathbf{now}^{+\infty}\ c_1\ \mathbf{then}\ B\ \mathbf{else}\ (\mathbf{now}^{+\infty}\ c_2\ \mathbf{then}\ B\ \mathbf{else}$$

$$(\mathbf{ask}(\bar{1}) \rightarrow\ \mathbf{now}^{+\infty}\ c_1\ \mathbf{then}\ B\ \mathbf{else}$$

$$(\mathbf{now}^{+\infty}\ c_2\ \mathbf{then}\ B\ \mathbf{else}\ (\mathbf{ask}(\bar{1}) \rightarrow \mathbf{ask}(c_1) \rightarrow^{+\infty} \mathbf{success}))))).$$

where $B :: (\mathbf{ask}(c_1) \rightarrow^{+\infty} \mathbf{success} + \mathbf{ask}(c_2) \rightarrow^{+\infty} \mathbf{success})$. Their concurrent evaluation in the $\bar{\mathbf{1}} \equiv \bar{0}$ empty store is:

$$\langle (B\ \mathbf{timeout}(1)\ \mathbf{ask}(c_1) \rightarrow^{+\infty} \mathbf{success} \parallel$$
$$\mathbf{tell}(\bar{0}) \xrightarrow{2} {}^{+\infty} \mathbf{tell}(c_3) \rightarrow^{+\infty} \mathbf{success}), \bar{0}\rangle.$$

The timeline for this parallel execution is given in Figure 6. At $t = 0$ the store is empty (i.e., $\sigma = \bar{0}$), thus both constraints c_1 and c_2 asked by the nondeterministic choice agent A_1 are not entailed. In A_2, the **tell** of c_3, which would entail both c_1 and c_2, is delayed by three time-units: in the first three time-units, $\mathbf{tell}(\bar{0}) \rightarrow^{+\infty}$ is executed according to the delay construct, as shown in Example 2. At $t = 2$, the timeout is triggered in A_1, since, according to **R1**, **R6** and **R9** (see Figure 2), the time elapsing in the timeout construct can be executed together with the delay-**tell** actions of A_2. After the timeout triggering, agent A_1 is however blocked, since c_1 is not entailed by the current empty store, and the precondition of the **ask** (rule **R3**) is not satisfied. A_2 can execute the last delay-**tell**, and then perform the $\mathbf{tell}(c_3)$ operation at $t = 3$; the store becomes $\sigma = \bar{0} \otimes c_3 = c_3$. This finally unblocks A_1 at $t = 4$, since, according to the precondition of rule **R3**, $\sigma \sqsubseteq c_1$ (i.e., $c_3 \sqsubseteq c_1$). Finally, at $t = 5$ we have $\langle \mathbf{success} \parallel \mathbf{success}, c_3\rangle$.

Example 4 (*Watchdog*)
In this example, let

$$A_1 :: \mathbf{do}\ (\mathbf{tell}(c_1) \rightarrow^{+\infty} \mathbf{ask}(c_3) \rightarrow^{+\infty} \mathbf{success})\ \mathbf{watching}^{+\infty}(c_2)\ \mathbf{else}$$

$$(\ \mathbf{tell}(c_3) \rightarrow^{+\infty} \mathbf{success}\ )$$

and

$$A_2 :: \mathbf{tell}(c_2) \rightarrow^{+\infty} \mathbf{success}.$$

We evaluate the following watchguard construct with two agents A_1 and A_2 in parallel:

$$\langle (\mathbf{do}\ (\mathbf{tell}(c_1) \rightarrow^{+\infty} \mathbf{ask}(c_3) \rightarrow^{+\infty} \mathbf{success})\ \mathbf{watching}^{+\infty}(c_2)\ \mathbf{else}$$

https://doi.org/10.1017/S1471068414000106 Published online by Cambridge University Press

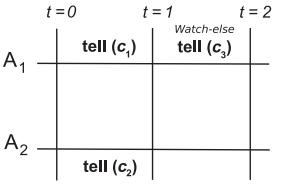


Fig. 7. The timeline of the execution of $A_1 \parallel A_2$ parallel agent in Example 4.

$$(\,\mathbf{tell}(c_3) \rightarrow^{+\infty} \mathbf{success}\,) \parallel \mathbf{tell}(c_2) \rightarrow^{+\infty} \mathbf{success}), \bar{0}\rangle.$$

According to Figure 3, agent A_1 is translated in the following way, where the agent B is a shorthand for the "else" branch of the watchdog, that is $\mathbf{tell}(c_3) \rightarrow^{+\infty} \mathbf{success}$:

$$\mathbf{now}^{+\infty}\ c_2\ \mathbf{then}\ B\ \mathbf{else}\ (\mathbf{tell}(c_1) \rightarrow^{+\infty} \mathbf{now}^{+\infty}\ c_2\ \mathbf{then}\ B\ \mathbf{else}$$

$$(\mathbf{ask}(c_3) \rightarrow^{+\infty}\ \mathbf{now}^{+\infty}\ c_2\ \mathbf{then}\ B\ \mathbf{else}\ \mathbf{success})).$$

The execution timeline for this parallel agent is shown in Figure 7. In the first time-unit we have $\sigma = \bar{0} \not\sqsupseteq c_2$, i.e., the store does not imply the guard of the $\mathbf{now}^{+\infty}$, and therefore the interruption of the watchguard in A_1 is not triggered yet. Thus, in the first time-unit, both $\mathbf{tell}(c_1) \rightarrow^{+\infty}$ of agent A_1 and $\mathbf{tell}(c_2) \rightarrow^{+\infty}$ of agent A_2 are executed. At time $t = 1$, the interruption of the watchguard is immediately activated (i.e., $\mathbf{now}^{+\infty}c_2$), since the store is now equal to $c_1 \otimes c_2 = c_3$ and $c_3 \vdash c_2$ (rule **R8** in Figure 2). Therefore, $\mathbf{tell}(c_3) \rightarrow^{+\infty}$ of agent B in A_1 is executed, while A_2 already corresponds to the **success** agent).

5.1 An auction example

In Figure 8 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 quantitative means to describe interactions among agents. We reckon that an auction provides one of the most suitable examples to show the expressivity of the *tsccp* language, since both time and preference (for a service or an object) are considered. In the following description we consider a *buyout* auction (Gallien and Gupta 2007), where the auctioneer improves the service and the related consumed resources (or, alternatively, its money price) bid after bid. When one (or more) of the bidders agrees with the offer, it bids for it and the auction is immediately declared as over.

The auctioneer (i.e., *AUCTIONEER* in Figure 8) begins by offering a service described with the soft constraint c_{A_1}. We suppose that the cost associated with the soft constraint is expressed in terms of computational capabilities needed to support the execution of the service: e.g., $c_i \sqsubseteq c_j$ means that the service described by c_i needs more computational resources than c_j. 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 left this preference generic in the example, since we focus on the interaction among agents.

https://doi.org/10.1017/S1471068414000106 Published online by Cambridge University Press

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

$\underline{CHECK}$ ::
$\mathbf{do}\ (\ (\mathbf{ask}(service = end) \longrightarrow \mathbf{success})\ \mathbf{timeout}(w_C)\ \mathbf{tell}(service = interrupt)\ )$
$\mathbf{watching}_{\phi_{Check}}(c_{check})\ \mathbf{else}\ (\mathbf{tell}(service = interrupt) \longrightarrow STOP_C)$

$\underline{BIDDER_i}$ ::
$INIT_B_i \longrightarrow$
$\mathbf{do}\ (\ TASK_i\ )\ \mathbf{watching}_{\phi_{Bidder}}(c_{B_i})\ \mathbf{else}\ \mathbf{ask}(\bar{1}) \xrightarrow{t_{buy_i}} \mathbf{tell}(bidder_i = i) \longrightarrow$
$(\ (\mathbf{ask}(winner = i) \longrightarrow USER_i) + (\mathbf{ask}(winner \neq i) \longrightarrow \mathbf{success})\ )$

$\underline{USER_i}$ ::
$\mathbf{do}\ (\ USE_SERVICE_i \longrightarrow \mathbf{tell}(service = end) \longrightarrow \mathbf{success}\ )$
$\mathbf{watching}_{\phi_{User}}(service = interrupt)\ \mathbf{else}\ (STOP_i)$

$\underline{AUCTION\&MONITOR} :: AUCTIONEER \parallel BIDER_1 \parallel BIDDER_2 \parallel \ldots \parallel BIDDER_n$

Fig. 8. An "auction and management" example for a generic service.

We suppose that a constraint can be defined over three domains of QoS features: availability, reliability and execution time. For instance, c_{A_1} is defined as $availability > 95\% \wedge reliability > 99\% \wedge execution\ time < 3sec$. Clearly, providing a higher availability or reliability, and a lower execution time implies raising the computational resources to support this improvement, thus worsening the preference of the store.

After the offer, the auctioneer gives time to the bidders (each of them described with a possibly different agent $BIDDER_i$ in Figure 8) 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 agent $CHECK$. Otherwise, if no offer is received within w_A time-units, a timeout interrupts the wait and the auctioneer improves the offered service by adding a new constraint: for example, in $\mathbf{tell}(c_{A_2})$, c_{A_2} could be equivalent to $execution\ time < 1sec$, thus reducing the latency of the service (from 3 to 1 sec) and consequently raising, at the same time, its computational cost (i.e., $\sigma = c_{A_1} \otimes c_{A_2} \sqsubseteq c_{A_1}$ means that we worsen the consistency level of the store). The same offer/wait process is repeated three times in Figure 8.

Each of the bidders in Figure 8 executes its own task (i.e., $TASK_i$, left generic since not in the scope of the example), but as soon as the offered resource meets its demand (i.e., c_{B_i} is satisfied by the store: $\sigma \sqsubseteq c_{B_i}$), the bidder is interrupted and then

https://doi.org/10.1017/S1471068414000106 Published online by Cambridge University Press

$\underline{AUCTIONEER'}$ ::
$INIT_A \longrightarrow$
$\mathbf{tell}(c_{A_1}) \xrightarrow{t_{sell}} (\Sigma_{i=1}^{n} \mathbf{ask}(bidder_i = i) \rightarrow^{a_A} CHOOSE)\ \mathbf{timeout}(w_A)$
$(\mathbf{tell}(c_{A_2}) \xrightarrow{t_{sell}} (\Sigma_{i=1}^{n} \mathbf{ask}(bidder_i = i) \rightarrow^{a_A} CHOOSE)\ \mathbf{timeout}(w_A)$
$(\mathbf{tell}(c_{A_3}) \xrightarrow{t_{sell}} (\Sigma_{i=1}^{n} \mathbf{ask}(bidder_i = i) \rightarrow^{a_A} CHOOSE)\ \mathbf{timeout}(w_A)\ \mathbf{success}))$

$\underline{CHOOSE}$ ::
$\mathbf{now}\ (bidder_n = n)\ \mathbf{then}\ \mathbf{tell}(winner = n) \rightarrow CHECK$
$\mathbf{else}\ (\ \mathbf{now}\ (bidder_{n-1} = n-1)\ \mathbf{then}\ \mathbf{tell}(winner = n-1) \rightarrow CHECK$
$\mathbf{else}\ (\ldots(\ \mathbf{now}\ (bidder_2 = 2)\ \mathbf{then}\ \mathbf{tell}(winner = 2) \rightarrow CHECK$
$\mathbf{else}\ \mathbf{tell}(winner = 1) \rightarrow CHECK)\ldots)).$

$\underline{newAUCTION\&MONITOR}$ :: $AUCTIONEER' \parallel BIDER_1 \parallel BIDDER_2 \parallel \ldots \parallel BIDDER_n$

Fig. 9. A new "auction and management" example for a generic service.

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 of the bidders wins, then it becomes a user of the resource by executing $USER_i$.

Agent $USER_i$ uses the service (through the agent $USE_SERVICE_i$, left generic in Figure 8), but it stops (using agent $STOP_i$, left generic in Figure 8) as soon as the service is interrupted, i.e., as the store satisfies $service = interrupt$. On the other side, agent $CHECK$ waits for the use termination, but it interrupts the user if the computation takes too long (more than w_C time-units), or if the user absorbs 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, agent $CHECK$ performs a $\mathbf{tell}(service = interrupt)$. All $INIT$ agents, left generic in Figure 8, can be used to initialize the computation. In order to avoid a heavy notation in Figure 8, we do not show the preference associated with constraints and the consistency check label on the transition arrows, when they are of no significative for the example description. Also, the ϕ_{Check}, ϕ_{Bidder} and ϕ_{User} thresholds of watchguard constructs are not detailed.

Finally, in Fig. 9 we model a more refined behavior of the auctioneer, which accepts the bidding with the highest value, where $\underline{CHECK}$, $\underline{BIDDER_i}$ and $\underline{USER_i}$ are defined as in Figure 8.

Many other real-life automated tasks can be modeled with the *tsccp* 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 represents (for example) the total monetary cost of the obtained service, or a value representing the consistency of integrated functionalities. The reason is that when we compose the services offered by different providers, we cannot be sure of how much they are compatible. A client wishing to use the composed service can perform an **ask** with a threshold such that it prevents the client from paying a high price, or having an unreliable service. Softness is also

https://doi.org/10.1017/S1471068414000106 Published online by Cambridge University Press

useful to model incomplete service specifications that may evolve incrementally and, in general, non-functional aspects.

6 The denotational model

In this section, we define a denotational characterization of the operational semantics obtained by following the construction in de Boer *et al.* (2000), and by using *timed reactive sequences* to represent *tsccp* computations. These sequences are similar to those used in the semantics of dataflow languages (Jonsson 1985), imperative languages (Brookes 1993) and (timed) *ccp* (de Boer and Palamidessi 1991; de Boer *et al.* 2000).

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 a 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 always entails 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 external environment by a corresponding tell operation, it is natural to assume that reactive sequences are monotonic. Thus, in the following we 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 conditions $\gamma_i \vdash \sigma_i$ 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 $\mathscr{S}$, 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, symbol $\cdot$ denotes the operator that concatenates sequences. In the following, *Process* denotes a set of *tsccp* processes.

Operationally, the reactive sequences of an agent are generated as follows.

Definition 3 (*Processes semantics*)

We define the semantics $\mathscr{R} \in \mathit{Process} \to \mathscr{P}(\mathscr{S})$ by

$$\begin{aligned}\mathscr{R}(F.A) = &\{\langle\sigma,\sigma'\rangle\cdot w \in \mathscr{S} \mid \langle A,\sigma\rangle \longrightarrow \langle B,\sigma'\rangle \text{ and } w \in \mathscr{R}(F.B)\}\\ &\cup\\ &\{\langle\sigma,\sigma\rangle\cdot w \in \mathscr{S} \mid \langle A,\sigma\rangle \not\longrightarrow \text{ and}\\ &\qquad\qquad \text{either } A \neq \textbf{Success} \text{ and } w \in \mathscr{R}(F.A)\\ &\qquad\qquad \text{or } A = \textbf{Success} \text{ and } w \in \mathscr{R}(F.A)\cup\{\varepsilon\}\}.\end{aligned}$$

Formally, $\mathscr{R}$ is defined as the least fixed-point of the operator $\Phi \in (\mathit{Process} \to \mathscr{P}(\mathscr{S})) \to \mathit{Process} \to \mathscr{P}(\mathscr{S})$ defined by

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

https://doi.org/10.1017/S1471068414000106 Published online by Cambridge University Press

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

Note that $\mathscr{R}(F.A)$ is the union of the set of all successful reactive sequences that start with a reaction of A, and the set of all successful reactive sequences that start with a stuttering step of A. 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.

6.1 Correctness

The observables $\mathscr{O}_{io}^{mp}(P)$ describing the input/output pairs of successful computations can be obtained from $\mathscr{R}(P)$ by considering suitable sequences, namely those sequences which do not perform assumptions on the store. In fact, note 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 $\sigma_i' = \sigma_{i+1}$. Thus, we are led to the following.

Definition 4 (*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 $\sigma_1 = \mathbf{1}$ and $\sigma_i = \sigma_{i-1}'$ for each i, $2 \leqslant i \leqslant n$.

According to previous definition, a sequence is connected if all the information assumed on the store is produced by the process itself. To be defined as connected, a sequence must also have $\mathbf{1}$ as an initial constraint. A connected sequence $s = \langle\mathbf{1},\sigma_1\rangle\langle\sigma_1,\sigma_2\rangle\cdots\langle\sigma_n,\sigma_n\rangle$ represents a *tsccp* computation of process $F.A$, where $\mathbf{1}$ is the input constraint and $\sigma_n \Downarrow_{Fv(A)}$ is the result. From the above discussion we can derive the following property.

Proposition 1 (*Correctness*)
For any process $P = F.A$ we have

$$\mathscr{O}_{io}^{mp}(P) = \{\sigma_n \Downarrow_{Fv(A)} | \text{ there exists a connected sequence } s \in \mathscr{R}(P) \text{ such that } s = \langle\mathbf{1},\sigma_1\rangle\langle\sigma_1,\sigma_2\rangle\cdots\langle\sigma_n,\sigma_n\rangle\}.$$

Proof
From the close correspondence between the rules of the transition system and the definition of denotational semantics, we have $s \in \mathscr{R}(P)$ if and only if $s = \langle\sigma_1,\sigma_1'\rangle\langle\sigma_2,\sigma_2'\rangle\cdots\langle\sigma_n,\sigma_n\rangle$, $A_1 = A$, $A_n =$ **Success** and for $i \in [1, n-1]$,

- either $\langle A_i,\sigma_i\rangle \longrightarrow \langle A_{i+1},\sigma_i'\rangle$
- or $\langle A_i,\sigma_i\rangle \not\longrightarrow$, $A_{i+1} = A_i$ and $\sigma_i' = \sigma_i$.

https://doi.org/10.1017/S1471068414000106 Published online by Cambridge University Press

Then there exists a connected sequence $s \in \mathscr{R}(P)$ if and only if $s = \langle \sigma_1, \sigma_2 \rangle \langle \sigma_2, \sigma_3 \rangle \cdots \langle \sigma_n, \sigma_n \rangle$, $A_1 = A$, $\sigma_1 = \mathbf{1}$, $A_n =$ **Success** and for $i \in [1, n-1]$, $\langle A_i, \sigma_i \rangle \longrightarrow \langle A_{i+1}, \sigma_{i+1} \rangle$. Therefore, the proof follows by definition of $\mathscr{O}_{io}^{mp}(P)$. □

6.2 *Compositionality of denotational semantics for* **tsccp** *processes*

In order to prove the compositionality of denotational semantics, we now introduce a semantics $[\![F.A]\!](e)$, which is compositional by definition and where, for technical reasons, we explicitly represent the environment e that associates a denotation to each procedure identifier. More precisely, assuming that *Pvar* denotes the set of procedure identifiers, $Env = Pvar \rightarrow \mathscr{P}(\mathscr{S})$, with typical element e, is the set of *environments*. Given $e \in Env$, $p \in Pvar$ and $f \in \mathscr{P}(\mathscr{S})$, 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 $[\![F.A]\!] : Env \rightarrow \mathscr{P}(\mathscr{S})$ is defined by the equations given in Figure 10, where μ denotes the least fix-point with respect to the subset inclusion of elements of $\mathscr{P}(\mathscr{S})$. The semantic operators appearing in Figure 10 are formally defined as follows; intuitively they reflect the operational behavior of their syntactic counterparts in terms of reactive sequences.[5] We first need the following definition.

Definition 5
Let σ, ϕ and c be constraints in $\mathscr{C}$ and let $a \in \mathscr{A}$. We say that

- $\sigma \succ^a c$, if $(\sigma \vdash c$ and $\sigma \Downarrow_\emptyset \not\succ a)$ while $\sigma \succ_\phi c$, if $(\sigma \vdash c$ and $\sigma \not\sqsubset \phi)$.

Definition 6 (*Semantic operators*)
Let S, S_i be sets of reactive sequences, c, c_i be constraints and let $\succ_i$ be either of the form $\succ^{a_i}$ or $\succ_{\phi_i}$. Then we define operators $\tilde{tell}$, $\tilde{\sum}$, $\tilde{\|}$, $\tilde{now}$ and $\tilde{\exists} x$ as follows.
The (valued) tell operator

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

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

The guarded choice

$$\tilde{\sum}_{i=1}^{n} c_i \succ_i S_i = \{ s \cdot s' \in \mathscr{S} \mid s = \langle \sigma_1, \sigma_1 \rangle \cdots \langle \sigma_m, \sigma_m \rangle, \sigma_j \not\succ_i c_i \text{ for each } j \in [1, m\text{-}1], i \in [1, n], \sigma_m \succ_h c_h \text{ and } s' \in S_h \text{ for an } h \in [1, n] \}.$$

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

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

[5] In Figure 10, the syntactic operator $\rightarrow_i$ is either of the form $\rightarrow^{a_i}$ or $\rightarrow_{\phi_i}$.

https://doi.org/10.1017/S1471068414000106 Published online by Cambridge University Press

We define $S_1 \tilde{\|} S_2$ as the pointwise extension of the above operator to sets.

The (valued) now operator

$$n\tilde{o}w^a(c, S_1, S_2) = \{s \in \mathscr{S} \mid s = \langle \sigma, \sigma' \rangle \cdot s',\ \sigma \Downarrow_\emptyset \not< a \text{ and}$$
$$\text{either } \sigma \vdash c \text{ and } s \in S_1$$
$$\text{or } \sigma \nvdash c \text{ and } s \in S_2\ \}.$$

$$n\tilde{o}w_\phi(c, S_1, S_2) = \{s \in \mathscr{S} \mid s = \langle \sigma, \sigma' \rangle \cdot s', \sigma \not\sqsubset \phi \text{ and}$$
$$\text{either } \sigma \vdash c \text{ and } s \in S_1$$
$$\text{or } \sigma \nvdash c \text{ and } s \in S_2\ \}.$$

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

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

where $s'[x/y]$ denotes the sequence obtained from s' by replacing variable x for variable y, which we assume to be new.[6]

Obviously, the semantic (valued) tell operator reflects the operational behavior of the syntactic (valued) tell. Concerning the semantic choice operator, a sequence in $\tilde{\sum}_{i=1}^{n} c_i \succ_i S_i$ consists of an initial period of waiting for a store that 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 with respect to the contribution of the environment (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 it will be indicated as $[\![F.A]\!]$. Environments in general allow us to define the semantics also of the processes that are not closed. The following result shows the correspondence between two semantics that we have introduced, and therefore it proves the compositionality of $\mathscr{R}(F.A)$. From the above discussion we can derive the following property.

Proposition 2 (*Compositionality*)
If $F.A$ is closed, then $\mathscr{R}(F.A) = [\![F.A]\!]$ holds.

Proof
We prove by induction on the complexity of agent A that

$$[\![F.A]\!] = \{s \mid s = \langle \sigma_1, \sigma_1' \rangle \langle \sigma_2, \sigma_2' \rangle \cdots \langle \sigma_n, \sigma_n \rangle,$$
$$A_1 = A, A_n = \textbf{Success} \text{ and for } i \in [1, n-1],$$
$$\text{either } \langle A_i, \sigma_i \rangle \longrightarrow \langle A_{i+1}, \sigma_i' \rangle$$
$$\text{or } \langle A_i, \sigma_i \rangle \not\longrightarrow, A_{i+1} = A_i,\ \sigma_i' = \sigma_i\}.$$

Then the proof follows by the definition of $\mathscr{R}(P)$.

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

https://doi.org/10.1017/S1471068414000106 Published online by Cambridge University Press

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

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

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$[\![F.p(x)]\!](e) = \mu\Psi$ where $\Psi(f) = [\![F \setminus \{p\}.ask(\bar{\mathbf{1}}) \rightarrow A]\!](e\{f/p\}),\ \ p(x) :: A \in F$

Fig. 10. The semantics $[\![F.A]\!](e)$.

When P is not of the form $F.B \parallel C$, the thesis follows immediately from the close correspondence between the rules of the transition system and the definition of denotational semantics.

Assume now that P is of the form $F.B \parallel C$. By definition of denotational semantics, $s \in [\![F.A]\!]$ if and only if $s = \langle\sigma_1,\sigma'_1\rangle\langle\sigma_2,\sigma'_2\rangle\cdots\langle\sigma_n,\sigma_n\rangle$ and there exist $s' \in [\![F.B]\!]$ and $s'' \in [\![F.C]\!]$,

$$\begin{aligned} s' &= \langle\sigma_1,\sigma_1\otimes\gamma_1\rangle\langle\sigma_2,\sigma_2\otimes\gamma_2\rangle\cdots\langle\sigma_n,\sigma_n\rangle \\ s'' &= \langle\sigma_1,\sigma_1\otimes\delta_1\rangle\langle\sigma_2,\sigma_2\otimes\delta_2\rangle\cdots\langle\sigma_n,\sigma_n\rangle \end{aligned}$$

such that for each $i \in [1, n-1]$, $\sigma'_i = \sigma_i \otimes \gamma_i \otimes \delta_i$. By inductive hypothesis, $s' \in [\![F.B]\!]$ and $s'' \in [\![F.C]\!]$ if and only if for $i \in [1, n-1]$,

$$\begin{array}{l} \text{either } \langle B_i,\sigma_i\rangle \longrightarrow \langle B_{i+1},\sigma_i\otimes\gamma_i\rangle, \\ \quad \text{or } \langle B_i,\sigma_i\rangle \not\longrightarrow,\ B_{i+1} = B_i,\ \sigma_i\otimes\gamma_i = \sigma_i \text{ and} \\ \text{either } \langle C_i,\sigma_i\rangle \longrightarrow \langle C_{i+1},\sigma_i\otimes\delta_i\rangle, \\ \quad \text{or } \langle C_i,\sigma_i\rangle \not\longrightarrow,\ C_{i+1} = C_i,\ \sigma_i\otimes\delta_i = \sigma_i. \end{array} \tag{1}$$

$B_1 = B$, $B_n = \mathbf{Success}$, $C_1 = C$ and $C_n = \mathbf{Success}$. Therefore, by Rule **R8** and previous observations, we have that (1) holds if and only if $B_1 \parallel C_1 = B \parallel C$, $B_n \parallel C_n = \mathbf{Success}$ and for $i \in [1, n-1]$,

$$\begin{array}{l} \text{either } \langle B_i \parallel C_i,\sigma_i\rangle \longrightarrow \langle B_{i+1} \parallel C_{i+1},\sigma'_i\rangle \\ \text{or } \langle B_i \parallel C_i,\sigma_i\rangle \not\longrightarrow,\ A_{i+1} \parallel B_{i+1} = A_i \parallel B_i,\ \sigma'_i = \sigma_i \end{array}$$

and then the thesis. □

https://doi.org/10.1017/S1471068414000106 Published online by Cambridge University Press

7 An interleaving approach for non-time-elapsing actions

In this section, we show a different version of the *tsccp* language: While in *tsccp* the parallel operator is modeled in terms of *maximal parallelism*, the same operator can be treated also in terms of interleaving. According to *maximal parallelism*, at each moment every enabled agent of the system is activated, while in the second 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 naive interleaving semantic could be problematic from the viewpoint of time, as in principle the time does not pass for enabled agent which are not scheduled. For the semantics in this section, we follow a solution analogous to the one adopted in de Boer *et al.* (2004): We 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, in this section, we adopt maximal parallelism for time elapsing (i.e., for timeout constructs) and an interleaving model for basic computation steps (i.e., (valued) **ask** and (valued) **tell** actions).

To distinguish this new approach, we named the resulting language as *tsccp-i*, i.e., *tsccp* with interleaving. Timeouts are modeled in *tsccp-i* by the construct $\mathbf{askp}_t(c)?_\phi A{:}B$ which replaces the $\mathbf{now}_\phi\ c$ **then** A **else** B construct of *tsccp* and directly has time t as one of its parameters, differently from the $\mathbf{now}_\phi$ agent. The $\mathbf{askp}_t$ agent can be interpreted as follows: One is allowed to wait t time-units for the entailment of constraint c by the store and the subsequent evaluation of process A; if this time limit is exceeded, then the process B is evaluated. Analogously for the construct $\mathbf{askp}_t(c)?^a A{:}B$.

Definition 7 (*tsccp-i*)
Given a soft constraint system $\langle S, D, V\rangle$, the corresponding structure $\mathscr{C}$, any semiring value a, *soft constraints* $\phi, c \in \mathscr{C}$ and any tuple of variables x, the syntax of the *tsccp-i* language is given by the following grammar:

$$
\begin{aligned}
P &::= F.A \\
F &::= p(x) :: A \mid F.F \\
A &::= \mathbf{success} \mid \mathbf{tell}(c) \rightarrow_\phi A \mid \mathbf{tell}(c) \rightarrow^a A \mid E \mid A \parallel A \mid \exists x A \mid p(x) \mid \\
&\quad\ \Sigma_{i=1}^n E_i \mid \mathbf{askp}_t(c)?_\phi A{:}A \mid \mathbf{askp}_t(c)?^a A{:}A \\
E &::= \mathbf{ask}(c) \rightarrow_\phi A \mid \mathbf{ask}(c) \rightarrow^a A
\end{aligned}
$$

where, as in Definition 1, P is the class of processes, F is the class of sequences of procedure declarations (or clauses) and A is the class of agents. As before, in a *tsccp-i process* $P = F.A$, A is the initial agent, to be executed in the context of the set of declarations F.

Analogous to *tsccp* processes, in order to simplify the notation, in the following we will usually write a *tsccp-i process* $P = F.A$ simply as the corresponding agent A.

The operational model of *tsccp-i* processes can be formally described by a labeled transition system $T = (Conf, Label, \longmapsto)$, where we assume that each transition

https://doi.org/10.1017/S1471068414000106 Published online by Cambridge University Press

step exactly takes one time-unit. Configurations (in) *Conf* are pairs comprising a process and a constraint in $\mathscr{C}$ representing the common *store*. $\mathscr{L} = \{\tau, \omega\}$ is a set of labels. We use labels to distinguish "real" computational steps performed by processes which have the control (label ω) from the transitions that model only the passing of time (label τ). So ω-actions are those performed by processes that modify the store (**tell**), perform a check on the store (**ask**, $\mathbf{askp}_t$), correspond to exceeding a timeout ($\mathbf{askp}_0$) or perform a choice ($\Sigma_{i=1}^{n} E_i$). On the other hand, τ-actions are those performed by timeout processes ($\mathbf{askp}_t$) in case they have not the control. In Figure 11 we show the semantics of all *tsccp-i* actions, but in the following we describe only the actions whose semantics is different from the one presented in Figure 2 (i.e., for *tsccp*), that is, we describe in detail the parallelism and the $\mathbf{askp}_t$ agent. The semantics of other actions of *tsccp-i* is the same as for *tsccp*, except for the fact that their transition is labeled with ω.

Parallelism Rules **Q5** and **Q6** in Figure 11 model the parallel composition operator in terms of *interleaving*, since only one basic ω-action is allowed for each transition (i.e., for each unit of time). This means that the access to the shared store is granted to one process at a time. However, time passes for all the processes appearing in the $\|$ context at the external level, as shown by rule **Q5**, since τ-actions are allowed together with a ω-action. On the other hand, a parallel component is allowed to proceed in isolation if (and only if) the other parallel component cannot perform a τ-action (rule **Q6**). To summarize, we adopt maximal parallelism for time elapsing (i.e., τ-actions) and an interleaving model for basic computation steps (i.e., ω-actions).

We have adopted this approach because it seems more adequate to the nature of timeout operators not to interrupt the elapsing of time once the evaluation of a timeout has started. Clearly, one could start the elapsing of time when the timeout process is scheduled, rather than when it appears in the top-level current parallel context. This modification could easily be obtained by adding a syntactic construct to differentiate active timeouts from inactive ones, and by accordingly changing the transition system. One could also easily modify the semantics (both operational and denotational) to consider a more liberal assumption that allows multiple ask actions in parallel.

Valued-Askp$_t$ The rules **Q10**–**Q14** in Figure 11 show that the timeout process $\mathbf{askp}_t(c)?^a A{:}B$ behaves as A if c is entailed by the store and the store is "consistent enough" with respect to the threshold a in the next t time-units: If $t > 0$ and the condition on the store and the cut level are satisfied, then the agent A is evaluated (rule **Q10**). If $t > 0$ and the condition on the cut level is not satisfied, then the agent B is evaluated (rule **Q11**). Finally, if $t > 0$, the condition on the cut level is satisfied, but the condition on the store is not satisfied, then the control is repeated at the next time instant and the value of the counter t is decreased (axiom **Q12**); note that in this case we use label ω, since a check on the store has been performed. As shown by axiom **Q13**, the counter can be decreased also by performing a τ-action: intuitively, this rule is used to model the situation in which, even though the evaluation of the timeout started already, another (parallel)

https://doi.org/10.1017/S1471068414000106 Published online by Cambridge University Press

Q1 $\dfrac{(\sigma \otimes c) \Downarrow_\emptyset \not< a}{\langle \mathbf{tell}(c) \rightarrow^a A, \sigma\rangle \overset{\omega}{\longmapsto} \langle A, \sigma \otimes c\rangle}$ **V-tell**

Q2 $\dfrac{\sigma \otimes c \not\sqsubset \phi}{\langle \mathbf{tell}(c) \rightarrow_\phi A, \sigma\rangle \overset{\omega}{\longmapsto} \langle A, \sigma \otimes c\rangle}$ **Tell**

Q3 $\dfrac{\sigma \vdash c \quad \sigma \Downarrow_\emptyset \not< a}{\langle \mathbf{ask}(c) \rightarrow^a A, \sigma\rangle \overset{\omega}{\longmapsto} \langle A, \sigma\rangle}$ **V-ask**

Q4 $\dfrac{\sigma \vdash c \quad \sigma \not\sqsubset \phi}{\langle \mathbf{ask}(c) \rightarrow_\phi A, \sigma\rangle \overset{\omega}{\longmapsto} \langle A, \sigma\rangle}$ **Ask**

Q5 $\dfrac{\langle A, \sigma\rangle \overset{\xi}{\longmapsto} \langle A', \sigma'\rangle \quad \langle B, \sigma\rangle \overset{\tau}{\longmapsto} \langle B', \sigma\rangle \quad \xi \in \{\tau, \omega\}}{\begin{array}{c}\langle A \parallel B, \sigma\rangle \overset{\xi}{\longmapsto} \langle A' \parallel B', \sigma'\rangle \\ \langle B \parallel A, \sigma\rangle \overset{\xi}{\longmapsto} \langle B' \parallel A', \sigma'\rangle\end{array}}$ **Parall1**

Q6 $\dfrac{\langle A, \sigma\rangle \overset{\xi}{\longmapsto} \langle A', \sigma'\rangle \quad \langle B, \sigma\rangle \overset{\tau}{\not\longmapsto} \quad \xi \in \{\tau, \omega\}}{\begin{array}{c}\langle A \parallel B, \sigma\rangle \overset{\xi}{\longmapsto} \langle A' \parallel B, \sigma'\rangle \\ \langle B \parallel A, \sigma\rangle \overset{\xi}{\longmapsto} \langle B \parallel A', \sigma'\rangle\end{array}}$ **Parall2**

Q7 $\dfrac{\langle E_j, \sigma\rangle \overset{\omega}{\longmapsto} \langle A_j, \sigma'\rangle \quad j \in [1, n]}{\langle \Sigma_{i=1}^n E_i, \sigma\rangle \overset{\omega}{\longmapsto} \langle A_j, \sigma'\rangle}$ **Nondet**

Q8 $\langle p(x), \sigma\rangle \overset{\omega}{\longmapsto} \langle A, \sigma\rangle \quad p(x) :: A \in F$ **P-call**

Q9 $\dfrac{\langle A[x/y], \sigma\rangle \overset{\xi}{\longmapsto} \langle B, \sigma'\rangle \quad \xi \in \{\tau, \omega\}}{\langle \exists x A, \sigma\rangle \overset{\xi}{\longmapsto} \langle B, \sigma'\rangle}$ **Hide**

Q10 $\dfrac{\sigma \vdash c \quad \sigma \Downarrow_\emptyset \not< a \quad t > 0}{\langle \mathbf{askp}_t(c)?^a A : B, \sigma\rangle \overset{\omega}{\longmapsto} \langle A, \sigma\rangle}$ **V-askp1**

Q11 $\dfrac{\sigma \Downarrow_\emptyset < a \quad t > 0}{\langle \mathbf{askp}_t(c)?^a A : B, \sigma\rangle \overset{\omega}{\longmapsto} \langle B, \sigma\rangle}$ **V-askp2**

Q12 $\dfrac{\sigma \not\vdash c \quad \sigma \Downarrow_\emptyset \not< a \quad t > 0}{\langle \mathbf{askp}_t(c)?^a A : B, \sigma\rangle \overset{\omega}{\longmapsto} \langle \mathbf{askp}_{t-1}(c)?^a A : B, \sigma\rangle}$ **V-askp3**

Q13 $\langle \mathbf{askp}_t(c)?^a A : B, \sigma\rangle \overset{\tau}{\longmapsto} \langle \mathbf{askp}_{t-1}(c)?^a A : B, \sigma\rangle \quad t > 0$ **V-askp4**

Q14 $\langle \mathbf{askp}_0(c)?^a A : B, \sigma\rangle \overset{\omega}{\longmapsto} \langle B, \sigma\rangle$ **V-askp5**

Q15 $\dfrac{\sigma \vdash c \quad \sigma \not\sqsubset \phi \quad t > 0}{\langle \mathbf{askp}_t(c)?_\phi A : B, \sigma\rangle \overset{\omega}{\longmapsto} \langle A, \sigma\rangle}$ **Askp1**

Q16 $\dfrac{\sigma \sqsubset \phi \quad t > 0}{\langle \mathbf{askp}_t(c)?_\phi A : B, \sigma\rangle \overset{\omega}{\longmapsto} \langle B, \sigma\rangle}$ **Askp2**

Q17 $\dfrac{\sigma \not\vdash c \quad \sigma \not\sqsubset \phi \quad t > 0}{\langle \mathbf{askp}_t(c)?_\phi A : B, \sigma\rangle \overset{\omega}{\longmapsto} \langle \mathbf{askp}_{t-1}(c)?_\phi A : B, \sigma\rangle}$ **Askp3**

Q18 $\langle \mathbf{askp}_t(c)?_\phi A : B, \sigma\rangle \overset{\tau}{\longmapsto} \langle \mathbf{askp}_{t-1}(c)?_\phi A : B, \sigma\rangle \quad t > 0$ **Askp4**

Q19 $\langle \mathbf{askp}_0(c)?_\phi A : B, \sigma\rangle \overset{\omega}{\longmapsto} \langle B, \sigma\rangle$ **Askp5**

Fig. 11. The transition system for *tsccp-i.*

https://doi.org/10.1017/S1471068414000106 Published online by Cambridge University Press

process has the control. In this case, analogous to the approach in de Boer *et al.* (2004) and differently from the approach in Busi *et al.* (2000), time continues to elapse (via τ-actions) also for the timeout process (see also rules **Q5** and **Q6** of parallel operator). Axiom **Q14** shows that if the timeout is exceeded, i.e., the counter t has reached the value of 0, then the process $\mathbf{askp}_t(c)?^a A{:}B$ behaves as B.

Askp$_t$ Rules **Q15**–**Q19** in Figure 11 are similar to rules **Q10**–**Q14** described before, with the exception that here a finer (pointwise) threshold ϕ is compared with store σ, analogous to what happens with **tell** and **ask** agents.

In the following we provide the definition for the observables of the language, which are clearly based only on ω-actions.

Definition 8 (*Observables for tsccp-i*)
Let $P = F.A$ be a *tsccp-i* process. We define

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

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

8 An execution timeline for a *tsccp-i* parallel agent

In this section, we show a timeline for the execution of three *tsccp-i* agents in parallel. We consider the three soft constraints shown in Figure 4 and the *Weighted* semiring $\langle \mathbb{R}^+ \cup \{+\infty\}, min, +, +\infty, 0\rangle$ (Bistarelli 2004; Bistarelli *et al.* 1997). Our parallel agent is defined by:

$$A_1 :: \mathbf{askp}_5(c_3)?^{+\infty}(\mathbf{tell}(c_1) \rightarrow^{+\infty} \mathbf{success}){:}(\mathbf{success})$$
$$A_2 :: \mathbf{tell}(c_1) \rightarrow^{+\infty} \mathbf{success}$$
$$A_3 :: \mathbf{tell}(c_2) \rightarrow^{+\infty} \mathbf{success}.$$

Their concurrent evaluation in $\bar{0}$ empty store is shown in Figure 12. At $t = 0$ and $t = 1$, agent A_1 can make a τ-transition (rule **Q13** in Figure 11), waiting for the elapsing of 1 time-unit. This can be done in parallel with a single other ω-action: therefore, the $\mathbf{tell}(c_1)$ of agent A_2, and the $\mathbf{tell}(c_2)$ of agent A_3 cannot run in parallel at the same time, since both these are ω-actions. In the execution shown in Figure 12, A_2 is executed before A_3 (also the opposite is possible, depending on the scheduling), leading to the store $\sigma = c_1 \otimes c_2 = c_3$. At $t = 2$, the guard of $\mathbf{askp}_5$ in agent A_1 is enabled, since $\sigma \vdash c_3$, and therefore rule **Q10** in Figure 11 is executed. Finally, at $t = 3$ the $\mathbf{tell}(c_1)$ action of agent A_1 is executed as the last action, and at $t = 4$ we have $\langle \mathbf{success} \parallel \mathbf{success} \parallel \mathbf{success}, c_1 \otimes c_2 \otimes c_1\rangle$.

9 Denotational semantics for *tsccp-i*

In this section, we define a denotational characterization of operational semantics for *tsccp-i*. Different from the denotational semantics for the maximal parallelism version

https://doi.org/10.1017/S1471068414000106 Published online by Cambridge University Press

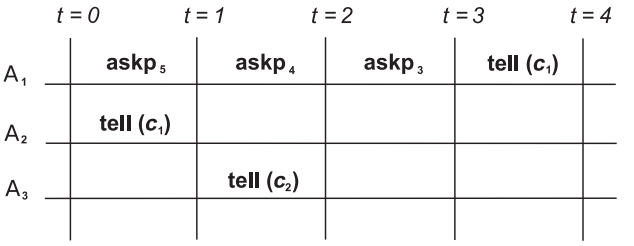


Fig. 12. Timeline for the execution of $A_1 \parallel A_2 \parallel A_3$.

presented in Section. 6.2, here for computational states we consider triples rather than pairs, as ω-actions have to be distinguished from τ-actions. This difference leads to a different technical development.

Our denotational model for *tsccp-i* associates with a process a set of timed reactive sequences of the form $\langle \sigma_1, \gamma_1, \xi_1 \rangle \cdots \langle \sigma_n, \gamma_n, \xi_n \rangle \langle \sigma, \sigma, \omega \rangle$. Any triple $\langle \sigma_i, \gamma_i, \xi_i \rangle$ represents a reaction (a computation step) of the given process at time i: intuitively, the process transforms the global store from σ_i to γ_i by performing a transition step labeled by ξ_i or, in other words, σ_i is the assumption on the external environment, ξ_i is the label of the performed step, while γ_i is the contribution of the process itself (which always entails the assumption). The last pair denotes a "stuttering step," in which the agent **Success** has been reached. In the following we will assume that each timed reactive sequence $\langle \sigma_1, \gamma_1, \xi_1 \rangle \cdots \langle \sigma_{n-1}, \gamma_{n-1}, \xi_{n-1} \rangle \langle \sigma_n, \sigma_n, \omega \rangle$ 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 basic idea underlying the denotational model then is that, differently from the operational semantics, inactive processes can always make a τ-step, where an inactive process is either suspended (due to the absence of the required constraint in the store) or it is a non-scheduled component of a parallel construct. These additional τ-steps, which represent time-elapsing and are needed to obtain a compositional model in a simple way, are then added to denotations as triples of the form $\langle \sigma, \sigma, \tau \rangle$. For example, the denotation of the process **tell**$(c) \rightarrow^a$ **success** contains all the reactive sequences that have, as first element, a triple $\langle \sigma, \sigma \otimes c, \omega \rangle$ for any possible initial store σ with $(\sigma \otimes c) \Downarrow_\emptyset \not< a$, as these represent the action of adding constraint c to the current store. However, such a denotation contains also sequences where the triple $\langle \sigma, \sigma \otimes c, \omega \rangle$ (still with $(\sigma \otimes c) \Downarrow_\emptyset \not< a$) is preceded by a finite sequence of triples of the form $\langle \sigma_1, \sigma_1, \tau \rangle \langle \sigma_2, \sigma_2, \tau \rangle \ldots \langle \sigma_n, \sigma_n, \tau \rangle$. Such a sequence represents time-elapsing while the process is inactive because some other parallel process is scheduled.

The set of all reactive sequences for *tsccp-i* process is denoted by $\mathscr{S}_i$, 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. Operator $\cdot$ denotes the operator that concatenates these sequences.

9.1 *Compositionality of denotational semantics for* tsccp-i *processes*

As in Section 6.2 for the *tsccp* version, we now introduce a denotational semantics $\mathscr{D}(F.A)(e)$, which is compositional by definition and where, for technical reasons, we represent explicitly the environment e which associates a denotation to each

https://doi.org/10.1017/S1471068414000106 Published online by Cambridge University Press

procedure identifier. More precisely, assuming that $Pvar$ denotes the set of procedure identifier, $Env_i = Pvar \to \mathscr{P}(\mathscr{S}_i)$, with typical element e, is the set of *environments*. Analogous to Section 6.2, given $e \in Env_i$, $p \in Pvar$ and $f \in \mathscr{P}(\mathscr{S}_i)$, we denote by $e' = e\{f/p\}$ the new environment such that $e'(p) = f$ and $e'(p') = e(p')$ for each procedure identifier $p' \neq p$.

Before defining formally the denotational semantics, we need to define operators $\bar{tell}$, $\bar{\sum}$, $\bar{\|}$, $\bar{ask}p$ and $\bar{\exists}x$, analogous to those given in Section 6.2 for maximal parallelism language.

Definition 9 (*Semantic operators for tsccp-i*)
Let S, S_i be sets of reactive sequences, and c, c_i be constraints. Moreover, let $\succ_i$ be either of the form $\succ^{a_i}$ or $\succ_{\phi_i}$, defined as in Definition 5. Then we define operators $\bar{tell}$, $\bar{\sum}$, $\bar{\|}$, $\bar{ask}p$ and $\bar{\exists}x$ as follows.
The (valued) tell operator $\bar{tell}^a : \mathscr{C} \times \wp(\mathscr{S}_i) \to \wp(\mathscr{S}_i)$ $(\bar{tell}_\phi : \mathscr{C} \times \wp(\mathscr{S}_i) \to \wp(\mathscr{S}_i))$ is the least function (w.r.t. the ordering induced by $\subseteq$) that satisfies the following equation:

$$\bar{tell}^a(c,S) = \{s \in \mathscr{S}_i \mid s = \langle \sigma, \sigma \otimes c, \omega\rangle \cdot s',\ \sigma \otimes c \Downarrow_\emptyset \not\succ a \text{ and } s' \in S\ \} \cup \{s \in \mathscr{S}_i \mid s = \langle \sigma, \sigma, \tau\rangle \cdot s' \text{ and } s' \in \bar{tell}^a(c,S)\ \}.$$

$$\bar{tell}_\phi(c,S) = \{s \in \mathscr{S}_i \mid s = \langle \sigma, \sigma \otimes c, \omega\rangle \cdot s',\ \sigma \otimes c \not\sqsubset \phi \text{ and } s' \in S\ \} \cup \{s \in \mathscr{S}_i \mid s = \langle \sigma, \sigma, \tau\rangle \cdot s' \text{ and } s' \in \bar{tell}_\phi(c,S)\ \}.$$

The guarded choice The semantic choice operator $\bar{\sum}_{i=1}^n : (\mathscr{C} \times \wp(\mathscr{S}_i)) \times \cdots \times (\mathscr{C} \times \wp(\mathscr{S}_i)) \to \wp(\mathscr{S}_i)$ is the least function that satisfies the following equation:

$$\bar{\sum}_{i=1}^n c_i \succ_i S_i = \{s \in \mathscr{S}_i \mid s = \langle \sigma, \sigma, \omega\rangle \cdot s',\ \sigma \succ_h c_h \text{ and } s' \in S_h \text{ for an } h \in [1,n]\ \} \cup \{s \in \mathscr{S}_i \mid s = \langle \sigma, \sigma, \tau\rangle \cdot s' \text{ and } s' \in \bar{\sum}_{i=1}^n c_i \succ_i S_i\}.$$

Parallel Composition. Let $\bar{\|} \in \mathscr{S}_i \times \mathscr{S}_i \to \mathscr{S}_i$ be the (commutative and associative) partial operator defined by induction on the length of the sequences as follows:

$$\langle \sigma, \sigma, \omega\rangle \bar{\|} \langle \sigma, \sigma, \omega\rangle = \langle \sigma, \sigma, \omega\rangle$$
$$\langle \sigma, \sigma', x\rangle \cdot s \bar{\|} \langle \sigma, \sigma, \tau\rangle \cdot s' = \langle \sigma, \sigma, \tau\rangle \cdot s' \bar{\|} \langle \sigma, \sigma', x\rangle \cdot s = \langle \sigma, \sigma', x\rangle \cdot (s \bar{\|} s'),$$

where $x \in \{\omega, \tau\}$.
We define operator $S_1 \bar{\|} S_2$ on sets as the image of $\mathscr{S}_i \times \mathscr{S}_i$ under the above operator.
The (valued) askp operator $\bar{ask}p(t)^a : \mathscr{C} \times \wp(\mathscr{S}_i) \times \wp(\mathscr{S}_i) \to \wp(\mathscr{S}_i)$ $(\bar{ask}p(t)_\phi : \mathscr{C} \times \wp(\mathscr{S}_i) \times \wp(\mathscr{S}_i) \to \wp(\mathscr{S}_i))$, with $t > 0$, is defined as:

$$\bar{ask}p(t)^a(c,S_1,S_2) = \{s \in \mathscr{S}_i \mid s = \langle \sigma, \sigma, \omega\rangle \cdot s' \text{ and either } \sigma \succ^a c \text{ and } s \in S_1 \text{ or } \sigma \Downarrow_\emptyset < a \text{ and } s \in S_2\} \cup \{s \in \mathscr{S}_i \mid s = \langle \sigma, \sigma, x\rangle \cdot s',\ s' \in \bar{ask}p(t-1)^a(c,S_1,S_2) \text{ and either } x = \tau \text{ or } x = \omega,\ \sigma \not\vdash c \text{ and } \sigma \Downarrow_\emptyset \not\succ a\ \}.$$

https://doi.org/10.1017/S1471068414000106 Published online by Cambridge University Press

$$\begin{aligned} a\bar{s}\bar{k}p(t)_\phi(c,S_1,S_2) = \{s \in \mathscr{S}_i \mid s = \langle \sigma, \sigma', \omega\rangle \cdot s' \text{ and} \\ \text{either } \sigma \succ_\phi c \text{ and } s \in S_1 \\ \text{or } \sigma \sqsubset \phi \text{ and } s \in S_2\} \cup \\ \{s \in \mathscr{S}_i \mid s = \langle \sigma, \sigma, x\rangle \cdot s',\ s' \in a\bar{s}\bar{k}p(t-1)_\phi(c,S_1,S_2) \\ \text{and either } x = \tau \\ \text{or } x = \omega,\ \sigma \nvdash c \text{ and } \sigma \not\sqsubset \phi\ \}. \end{aligned}$$

The (valued) askp operator $a\bar{s}\bar{k}p(0)^a : \mathscr{C} \times \wp(\mathscr{S}_i) \times \wp(\mathscr{S}_i) \to \wp(\mathscr{S}_i)$ $(a\bar{s}\bar{k}p(0)_\phi : \mathscr{C} \times \wp(\mathscr{S}_i) \times \wp(\mathscr{S}_i) \to \wp(\mathscr{S}_i))$ is the least function that satisfies the following equation:

$$\begin{aligned} a\bar{s}\bar{k}p(0)^a(c,S_1,S_2) = \{s \in \mathscr{S}_i \mid \text{either } s = \langle \sigma, \sigma, \omega\rangle \cdot s' \text{ and } s' \in S_2 \\ \text{or } s = \langle \sigma, \sigma, \tau\rangle \cdot s' \\ \text{and } s' \in a\bar{s}\bar{k}p(0)^a(c,S_1,S_2)\ \}. \end{aligned}$$

$$\begin{aligned} a\bar{s}\bar{k}p(0)_\phi(c,S_1,S_2) = \{s \in \mathscr{S}_i \mid \text{either } s = \langle \sigma, \sigma, \omega\rangle \cdot s' \text{ and } s' \in S_2 \\ \text{or } s = \langle \sigma, \sigma, \tau\rangle \cdot s' \\ \text{and } s' \in a\bar{s}\bar{k}p(0)_\phi(c,S_1,S_2)\ \}. \end{aligned}$$

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

$$\bar{\exists} xS = \{s \in \mathscr{S}_i \mid \text{there exists } s' \in S \text{ such that } s = s'[x/y] \text{ with } y \text{ new}\ \}$$

where $s'[x/y]$ denotes the sequence obtained from s' by replacing variable x for variable y that we assume to be new.[7]

It is immediate to see that the previous semantic operators are well defined, that is, the least function that satisfies the equations actually exists and can be obtained by a standard fix-point construction. The $t\bar{e}ll$, $\bar{\sum}$, $\bar{\|}$, $a\bar{s}\bar{k}p$ and $\bar{\exists}x$ operators have the expected definition, including the mentioned addition of τ-steps.

In the semantic parallel operator (acting on sequences) we require that at each point of time at most one ω-action is present and the two arguments of the operator agree with respect to the contribution of the environment (the first component of the triple). We also require that the two arguments have the same length (in all other cases the parallel composition is assumed being undefined): this is necessary to reflect the passage of time since the ith element of any sequence corresponds to the given processes action on the ith time step. Even though we merge pointwise sequences of the same length, this models an interleaving approach for ω-actions because of the previously mentioned addition of τ-steps to denotations. Concerning the semantic choice operator, a sequence in $\bar{\sum}_{i=1}^{n} c_i \succ_i S_i$ consists of an initial period of waiting for a store that satisfies one of the guards. During this waiting period, only the environment is active by producing constraint σ, while the process itself generates the stuttering steps $\langle \sigma, \sigma, \tau\rangle$. When the store is strong enough to satisfy a

[7] As before, we assume that each time that we consider a new applications of operator $\bar{\exists}$, we use a new, different y.

https://doi.org/10.1017/S1471068414000106 Published online by Cambridge University Press

$$\mathbf{F1}\mathscr{D}(F.\mathbf{success})(e) = \{\langle\sigma_1,\sigma_1,\tau\rangle\langle\sigma_2,\sigma_2,\tau\rangle\cdots\langle\sigma_n,\sigma_n,\omega\rangle \in \mathscr{S}_i \mid n \geqslant 1\}$$

$$\mathbf{F2}\mathscr{D}(F.\mathbf{tell}(c) \rightarrow^a A)(e) = t\bar{e}ll^a(c, \mathscr{D}(F.A)(e))$$

$$\mathbf{F3}\mathscr{D}(F.\mathbf{tell}(c) \rightarrow_\phi A)(e) = t\bar{e}ll_\phi(c, \mathscr{D}(F.A)(e))$$

$$\mathbf{F4}\mathscr{D}(F.\textstyle\sum_{i=j}^{n}\mathbf{ask}(c_i) \rightarrow_i A_i)(e) = \tilde{\sum}_{i=1}^{n} c_i \succ_i \mathscr{D}(F.A_i)(e)$$

$$\mathbf{F5}\mathscr{D}(F.\mathbf{askp}_t(c)?^a A{:}B)(e) = a\bar{s}kp(t)^a(c, \mathscr{D}(F.A)(e), \mathscr{D}(F.B)(e))$$

$$\mathbf{F6}\mathscr{D}(F.\mathbf{askp}_t(c)?_\phi A{:}B)(e) = a\bar{s}kp(t)_\phi(c, \mathscr{D}(F.A)(e), \mathscr{D}(F.B)(e))$$

$$\mathbf{F7}\mathscr{D}(F.A \parallel B)(e) = \mathscr{D}(F.A)(e) \;\bar{\|}\; \mathscr{D}(F.B)(e)$$

$$\mathbf{F8}\mathscr{D}(F.\exists x A)(e) = \bar{\exists} x \mathscr{D}(F.A)(e)$$

$$\mathbf{F9}\mathscr{D}(F.p(x))(e) = \mu\Psi^i \text{ where } \Psi^i(f) = \mathscr{D}(F \setminus \{p\}.ask(\bar{\mathbf{1}}) \rightarrow A)(e\{f/p\}),\ p(x) :: A \in F$$

Fig. 13. Semantics $D(F.A)$(e) for *tsccp-i*.

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.

We can define the denotational semantics $\mathscr{D}$ as follows. Here $Process_i$ denotes the set of *tsccp-i* processes.

Definition 10 (*Processes semantics*)
We define that the semantics $\mathscr{D} \in Process_i \rightarrow \mathscr{P}(\mathscr{S}_i)$ is the least function with respect to the ordering induced by the set-inclusion, which satisfies the equations given in Figure 13.

Also, $\mathscr{D}$ is well defined and can be obtained by a fix-point construction. To see this, let us define an interpretation as a mapping $I : Process_i \rightarrow \wp(\mathscr{S}_i)$. Then let us denote by $\mathscr{I}$ the cpo of all interpretations (with the ordering induced by $\subseteq$). To the equations given in Figure 13, we can then associate a monotonic (and continuous) mapping $\mathscr{F} : \mathscr{I} \rightarrow \mathscr{I}$ defined by the equations of Figure 13, provided that we replace the symbol $\mathscr{D}$ for $\mathscr{F}(I)$, delete the environment e and replace equation **F9** for the following one: $\mathscr{F}(I)(F.p(x)) = I(F.ask(\bar{\mathbf{1}}) \rightarrow A)$.

Then, one can easily prove that a function satisfies the equations given in Figure 13 iff it is a fix-point of function $\mathscr{F}$. Since this function is continuous (on a cpo), well-known results ensure us that its least fix-point exists and it equals $\mathscr{F}^\omega$, where the powers are defined as follows: $\mathscr{F}^0 = I_0$ (this is the least interpretation that maps any process to an empty set); $\mathscr{F}^n = \mathscr{F}(\mathscr{F}^{n-1})$ and $\mathscr{F}^\omega = lub\{\mathscr{F}^n | n \geqslant 0\}$ (where lub is the least upper bound on cpo $\mathscr{I}$).

9.2 *Correctness of denotational semantics for* **tsccp-i** *processes*

As for the correctness of the denotational semantics presented in Section 6.1, at each step, the assumption on the current store must be equal to the store produced by the

https://doi.org/10.1017/S1471068414000106 Published online by Cambridge University Press

Q0' $\langle \text{success}, \sigma \rangle \stackrel{\tau}{\Rightarrow} \langle \text{success}, \sigma \rangle$ **success**

Q1' $\langle \mathbf{tell}(c) \rightarrow^{a} A, \sigma \rangle \stackrel{\tau}{\Rightarrow} \langle \mathbf{tell}(c) \rightarrow^{a} A, \sigma \rangle$ **V-Telll**

Q2' $\langle \mathbf{tell}(c) \rightarrow_{\phi} A, \sigma \rangle \stackrel{\tau}{\Rightarrow} \langle \mathbf{tell}(c) \rightarrow_{\phi} A, \sigma \rangle$ **Tell**

Q3' $\langle \mathbf{ask}(c) \rightarrow^{a} A, \sigma \rangle \stackrel{\tau}{\Rightarrow} \langle \mathbf{ask}(c) \rightarrow^{a} A, \sigma \rangle$ **V-ask**

Q4' $\langle \mathbf{ask}(c) \rightarrow_{\phi} A, \sigma \rangle \stackrel{\tau}{\Rightarrow} \langle \mathbf{ask}(c) \rightarrow_{\phi} A, \sigma \rangle$ **Ask**

Q7' $\langle \Sigma_{i=1}^{n} E_i, \sigma \rangle \stackrel{\tau}{\Rightarrow} \langle \Sigma_{i=1}^{n} E_i, \sigma \rangle$ **Nondet**

Q8' $\langle p(x), \sigma \rangle \stackrel{\tau}{\Rightarrow} \langle A, \sigma \rangle \quad p(x) :: A \in F$ **P-call**

Q14' $\langle \mathbf{askp}_0(c)?^{a}A{:}B, \sigma \rangle \stackrel{\tau}{\Rightarrow} \langle \mathbf{askp}_0(c)?^{a}A{:}B, \sigma \rangle$ **V-askp5**

Q19' $\langle \mathbf{askp}_0(c)?_{\phi}A{:}B, \sigma \rangle \stackrel{\tau}{\Rightarrow} \langle \mathbf{askp}_0(c)?_{\phi}A{:}B, \sigma \rangle$ **Askp5**

Fig. 14. The τ-rules for *tsccp-i*.

previous step. In other words, for any two consecutive steps $\langle \sigma_i, \sigma'_i, x_i \rangle \langle \sigma_{i+1}, \sigma'_{i+1}, x_{i+1} \rangle$ we must have $\sigma'_i = \sigma_{i+1}$. Furthermore, triples containing τ-actions do not correspond to observable computational steps, as these involve ω-actions only.

Definition 11 (*Connected sequences in tsccp-i*)
Let $s = \langle \sigma_1, \sigma'_1, x_1 \rangle \langle \sigma_2, \sigma'_2, x_2 \rangle \cdots \langle \sigma_n, \sigma_n, \omega \rangle$ be a reactive sequence. We say that s is connected if $\sigma_1 = \bar{\mathbf{1}}$, $\sigma_i = \sigma'_{i-1}$ and $x_j = \omega$ for each i, j, $2 \leqslant i \leqslant n$ and $1 \leqslant j \leqslant n-1$.

According to the previous definition, a sequence is connected if all the information assumed on the tuple space is produced by the process itself and only ω-actions are involved. To be defined as connected, a sequence must also have $\bar{\mathbf{1}}$ as an initial constraint. A connected sequence represents a *tsccp-i* computation, as it will be proved in the remaining of this section.

In order to prove the correctness of denotational semantics, we use a modified transition system T', where inactive (either suspended or not scheduled) processes can perform τ-actions. When considering our notions of observables, we can prove that such a modified transition system is equivalent to the previous one and agrees with the denotational model.

A new transition system T' is obtained from the one in Figure 11 by deleting rule **Q6** and adding rules **Q0'**, **Q1'**, **Q2'**, **Q3'**, **Q4'**, **Q7'**, **Q8'**, **Q14'** and **Q19'**, contained in Figure 14. We denote by $\Rightarrow$ the relation defined by T'.

The observables induced by the transition system T' are formally defined as follows.

Definition 12
Let $P = F.A$ be a *tsccp-i* process. We define

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

https://doi.org/10.1017/S1471068414000106 Published online by Cambridge University Press

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

Lemma 3 shows that the modified transition system agrees with the original one when considering our notion of observables.

We first need some definitions and technical lemmata. In the following, given two agents A and B, we say that $A \simeq B$ if and only if B is obtained from A by replacing an agent of the form $\exists x A_1$ in A with $A_1[x/y]$, where y is new in A. $\approx$ denotes the reflexive and transitive closure of $\simeq$. The following lemmata hold.

Lemma 1
Let $F.A$ and $F.B$ be *tsccp-i* processes such that $A \approx B$. Then for each store σ and $x \in \{\omega, \tau\}$

$$\langle F.A, \sigma\rangle \stackrel{x}{\longmapsto} \langle F.C, \sigma'\rangle \text{ if and only if } \langle F.B, \sigma\rangle \stackrel{x}{\longmapsto} \langle F.C, \sigma'\rangle.$$

Proof
The proof is immediate by using rule **Q9**, and by a straightforward inductive argument. □

From the above Lemma we derive the following corollary.

Corollary 1
Let $F.A$ and $F.B$ be *tsccp-i* processes such that $A \approx B$. Then for each store σ, $\langle F.A, \sigma\rangle \stackrel{\omega}{\longmapsto}{}^{*} \langle \mathbf{Success}, \gamma\rangle$ if and only if $\langle F.B, \sigma\rangle \stackrel{\omega}{\longmapsto}{}^{*} \langle \mathbf{Success}, \gamma\rangle$.

Lemma 2
Let $P = F.A$ be a *tsccp-i* process. Then for each store σ,

1. $\langle F.A, \sigma\rangle \stackrel{\tau}{\Rightarrow} \langle F.B, \sigma'\rangle$ if and only if $\sigma = \sigma'$ and
   - either $\langle F.A, \sigma\rangle \stackrel{\tau}{\longmapsto} \langle F.C, \sigma\rangle$ and $C \approx B$
   - or $\langle F.A, \sigma\rangle \stackrel{\tau}{\not\longmapsto}$ and $B \approx A$.
2. $\langle F.A, \sigma\rangle \stackrel{\omega}{\Rightarrow} \langle F.B, \sigma'\rangle$ if and only if $\langle F.A, \sigma\rangle \stackrel{\omega}{\longmapsto} \langle F.C, \sigma'\rangle$ and $C \approx B$.

Proof
1. The proof is by induction on the complexity of agent A.
   - A is of the form **success**, $\mathbf{tell}(c) \rightarrow^{a} A$, $\mathbf{tell}(c) \rightarrow_{\phi} A$, $\mathbf{ask}(c) \rightarrow^{a} A$, $\mathbf{ask}(c) \rightarrow_{\phi} A$, $\Sigma_{i=1}^{n} E_i$, $p(x)$, $\mathbf{askp}_0(c)?^{a}A{:}B$ and $\mathbf{askp}_0(c)?_{\phi}A{:}B$.
     The proof is immediate by observing that by the rules given in Figure 11, $\langle A, \sigma\rangle \stackrel{\tau}{\not\longmapsto}$ and by the rules given in Figure 14, $\langle A, \sigma\rangle \stackrel{\tau}{\Rightarrow} \langle B, \sigma'\rangle$ if and only if $\langle A, \sigma\rangle = \langle B, \sigma'\rangle$.
   - A is of the form $\mathbf{askp}_t(c)?^{a}A_1{:}A_2$ ($\mathbf{askp}_t(c)?_{\phi}A_1{:}A_2$), with $t > 0$.
     The proof is immediate since both transition systems use rule **Q13** and (**Q18**) of Figure 11.
   - If A is of the form $A_1 \parallel A_2$.
     In this case, by definition of the transition system T' and by using rule **Q5** of Figure 11, for each store σ,

$$\langle A_1 \parallel A_2, \sigma\rangle \stackrel{\tau}{\Rightarrow} \langle B_1 \parallel B_2, \sigma'\rangle \text{ if and only if}$$
$$\langle A_1, \sigma\rangle \stackrel{\tau}{\Rightarrow} \langle B_1, \sigma'\rangle \text{ and } \langle A_2, \sigma\rangle \stackrel{\tau}{\Rightarrow} \langle B_2, \sigma\rangle$$

https://doi.org/10.1017/S1471068414000106 Published online by Cambridge University Press

(the symmetric case is analogous and hence it is omitted).
By inductive hypothesis this holds if and only if $\sigma' = \sigma$ and for $i = 1, 2$

— either $\langle A_i, \sigma\rangle \overset{\tau}{\longmapsto} \langle C_i, \sigma\rangle$ and $C_i \approx B_i$, or

— $\langle A_i, \sigma\rangle \overset{\tau}{\not\longmapsto}$ and $B_i \approx A_i$.

If there exists $i \in [1, 2]$ such that $\langle A_i, \sigma\rangle \overset{\tau}{\longmapsto} \langle C_i, \sigma\rangle$ then the thesis follows by using either rule **Q5** or **Q6**.
Otherwise, $\langle A_1 \parallel A_2, \sigma\rangle \overset{\tau}{\not\longmapsto}$. Then the thesis follows, since by the previous results $B_1 \parallel B_2 \approx A_1 \parallel A_2$.

- A is of the form $\exists x A_1$. By rule **Q9** of Figure 11 for each store σ,

$$\langle \exists x A_1, \sigma\rangle \overset{\tau}{\Rightarrow} \langle B, \sigma'\rangle \text{ if and only if } \langle A_1[x/y], \sigma\rangle \overset{\tau}{\Rightarrow} \langle B, \sigma'\rangle$$

By inductive hypothesis, this holds if and only if $\sigma' = \sigma$

— either $\langle A_1[x/y], \sigma\rangle \overset{\tau}{\longmapsto} \langle C, \sigma\rangle$ and $C \approx B$, or

— $\langle A_1[x/y], \sigma\rangle \overset{\tau}{\not\longmapsto}$ and $B \approx A_1[x/y]$.

Therefore, by using rule **Q9** of Figure 11 and since $\exists x A_1 \approx A_1[x/y]$, we have

— either $\langle \exists x A_1, \sigma\rangle \overset{\tau}{\longmapsto} \langle C, \sigma\rangle$ and $C \approx B$, or

— $\langle \exists x A_1, \sigma\rangle \overset{\tau}{\not\longmapsto}$ and $B \approx \exists x A_1$ and then the thesis.

2. The proof is analogous to the previous one and hence it is omitted.

□

Lemma 3
Let $P = F.A$ be a *tsccp-i* process. Then $\mathcal{O}^{i'}_{io}(P) = \mathcal{O}^{i}_{io}(P)$.

Proof
We prove that there exists a computation $\langle A, \sigma\rangle \overset{\omega}{\Rightarrow}{}^{*} \langle \mathbf{Success}, \gamma\rangle$ if and only if there exists a computation $\langle A, \sigma\rangle \overset{\omega}{\longmapsto}{}^{*} \langle \mathbf{Success}, \gamma\rangle$. Then the thesis is followed by the definition of $\mathcal{O}^{i}_{io}(P)$ and $\mathcal{O}^{i'}_{io}(P)$. The proof is by induction on the length of computation $\langle A, \sigma\rangle \overset{\omega}{\Rightarrow}{}^{*} \langle \mathbf{Success}, \gamma\rangle$.
$n = 1$) In this case $A = \mathbf{Success}$ and then the thesis.
$n > 1$) In this case

$\langle A, \sigma\rangle \overset{\omega}{\Rightarrow}{}^{*} \langle \mathbf{Success}, \gamma\rangle$ iff
(by definition)
$\langle A, \sigma\rangle \overset{\omega}{\Rightarrow} \langle A_1, \sigma_1\rangle$ and $\langle A_1, \sigma_1\rangle \overset{\omega}{\Rightarrow}{}^{*} \langle \mathbf{Success}, \gamma\rangle$ iff
(by inductive hypothesis)
$\langle A, \sigma\rangle \overset{\omega}{\Rightarrow} \langle A_1, \sigma_1\rangle$ and $\langle A_1, \sigma_1\rangle \overset{\omega}{\longmapsto}{}^{*} \langle \mathbf{Success}, \gamma\rangle$ iff
(by Point 2 of Lemma 2)
$\langle A, \sigma\rangle \overset{\omega}{\longmapsto} \langle A_2, \sigma_1\rangle$, $A_2 \approx A_1$ and $\langle A_1, \sigma_1\rangle \overset{\omega}{\longmapsto}{}^{*} \langle \mathbf{Success}, \gamma\rangle$ iff
(by Corollary 1)
$\langle A, \sigma\rangle \overset{\omega}{\longmapsto} \langle A_2, \sigma_1\rangle$ and $\langle A_2, \sigma_1\rangle \overset{\omega}{\longmapsto}{}^{*} \langle \mathbf{Success}, \gamma\rangle$ iff
(by definition)
$\langle A, \sigma\rangle \overset{\omega}{\longmapsto}{}^{*} \langle \mathbf{Success}, \gamma\rangle$.

□

https://doi.org/10.1017/S1471068414000106 Published online by Cambridge University Press

We can now easily prove that, given our definition of $\mathscr{D}$, the modified transition system T' agrees with the denotational model.

Theorem 2
For any *tsccp-i* process $P = F.A$ we have

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

Proof
We prove by induction on the complexity of agent A that

$$\mathscr{D}(P) = \{s \mid \; s = \langle \sigma_1, \sigma'_1, x_1\rangle\langle \sigma_2, \sigma'_2, x_2\rangle \cdots \langle \sigma_n, \sigma_n, \omega\rangle,\ A_1 = A, \text{ for } i \in [1, n-1],\ \langle A_i, \sigma_i\rangle \stackrel{x_i}{\Rightarrow} \langle A_{i+1}, \sigma'_i\rangle \text{ and } A_n = \textbf{Success}\}.$$

Then the proof follows by the definition of $\mathscr{O}^{i'}_{io}(P)$.

When *tsccp-i* P is not of the form $F.B \parallel C$, the thesis follows immediately from the close correspondence between the rules of transition system and the definition of denotational semantics.

Assume now that P is of the form $F.B \parallel C$. By definition of denotational semantics, $s \in \mathscr{D}(P)$ if and only if $s = \langle \sigma_1, \sigma'_1, x_1\rangle\langle \sigma_2, \sigma'_2, x_2\rangle \cdots \langle \sigma_n, \sigma_n, \omega\rangle$ and there exist $s' \in \mathscr{D}(F.B)$ and $s'' \in \mathscr{D}(F.C)$,

$$s' = \langle \sigma_1, \kappa'_1, x'_1\rangle\langle \sigma_2, \kappa'_2, x'_2\rangle \cdots \langle \sigma_n, \sigma_n, \omega\rangle \quad \text{and}$$
$$s'' = \langle \sigma_1, \kappa''_1, x''_1\rangle\langle \sigma_2, \kappa''_2, x''_2\rangle \cdots \langle \sigma_n, \sigma_n, \omega\rangle,$$

such that for each $i \in [1, n-1]$,

$$\begin{aligned} &x_i = \tau \text{ if and only if } \; x'_i = x''_i = \tau \text{ and in this case } \sigma'_i = \kappa'_i = \kappa''_i = \sigma_i, \\ &x_i = \omega \text{ if and only if either } x'_i = \omega,\ x''_i = \tau,\ \kappa'_i = \sigma'_i \text{ and } \kappa''_i = \sigma_i \\ &\qquad\qquad \text{or } x'_i = \tau,\ x''_i = \omega,\ \kappa''_i = \sigma'_i \text{ and } \kappa'_i = \sigma_i. \end{aligned} \tag{2}$$

By inductive hypothesis, $s' \in \mathscr{D}(F.B)$ and $s'' \in \mathscr{D}(F.C)$ if and only if

$$\begin{aligned} &\langle B_i, \sigma_i\rangle \stackrel{x'_i}{\Rightarrow} \langle B_{i+1}, \kappa'_i\rangle \text{ for } i \in [1, n-1],\ B_1 = B \text{ and } B_n = \textbf{Success}, \\ &\langle C_i, \sigma_i\rangle \stackrel{x''_i}{\Rightarrow} \langle C_{i+1}, \kappa''_i\rangle \text{ for } i \in [1, n-1],\ C_1 = C \text{ and } C_n = \textbf{Success}. \end{aligned} \tag{3}$$

Therefore, by rule **R8** and by (2), we have that (3) holds if and only if

$$\langle B_i \parallel C_i, \sigma_i\rangle \stackrel{x_i}{\Rightarrow} \langle B_{i+1} \parallel C_{i+1}, \sigma_{i+1}\rangle \text{ for } i \in [1, n-1],$$
$$B_1 \parallel C_1 = B \parallel C \text{ and } B_n \parallel C_n = \textbf{Success}$$

and then the thesis. □

Thus, we obtain the following correctness result whose proof is immediate from the previous theorems.

Corollary 2 (*Correctness of tsccp-i*)
For any *tsccp-i* process $P = F.A$ we have

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

https://doi.org/10.1017/S1471068414000106 Published online by Cambridge University Press

10 Related works

By comparing this work with other timed languages using crisp constraints (instead of soft ones as in this paper), such as Saraswat *et al.* (1994, 1996), there are three main differences we can find out.

First, the computational model of both *tcc* (Saraswat *et al.* 1994) and *default tcc* (Saraswat *et al.* 1996) languages is inspired by that of synchronous languages: each time interval is identified with the time needed for a *ccp* process to terminate a computation. Clearly, in order to ensure that the next time instant is reached, the (default) *ccp* program has to be always terminating; thus, it is assumed that it does not contain recursion. On the other hand, we directly introduce a timed interpretation of the usual programming constructs of *ccp* by considering the primitive *ccp* constructs **ask** and **tell** as the elementary actions whose evaluation takes one time-unit. Therefore, in our model, each time interval is identified with the time needed for the underlying constraint system to accumulate the tells and to answer the queries (asks) issued at each computation step by the processes of the system. For the definition of our *tsccp* agents we do not need any restriction on recursion to ensure that the next time instant is reached, since at each moment there are only a finite number of parallel agents, and the next moment in time occurs as soon as the underlying constraint system has responded to the initial actions of all the current agents of the system.

The second difference relies in the transfer of information across time boundaries. In Saraswat *et al.* (1994, 1996), the programmer has to explicitly transfer the (positive) information from a time instant to the next one by using special primitives that allow one to control the temporal evolution of the system. In fact, at the end of a time interval all the constraints accumulated and all the processes suspended are discarded, unless they are arguments to a specific primitive. On the contrary, no explicit transfer is needed in *tsccp*, since the computational model is based on the monotonic evolution of store which is usual in *ccp*.

The third relevant difference is in Saraswat *et al.* (1994, 1996), where the authors have presented deterministic languages whereas our language allows nondeterminism. These three differences also hold between Saraswat *et al.* (1994, 1996) and the original crisp version of the language, i.e., *tccp* (de Boer *et al.* 2000).

In Olarte *et al.* (2007), the authors generalize the model given in Saraswat *et al.* (1994) to extend it with temporary parametric **ask** operations. Intuitively, these operations behave as persistent parametric asks during a time-interval, but may disappear afterwards. The presented extension goes in the direction of better modeling *mobile systems* with the use of private channels between the agents. However, also the agents in Olarte *et al.* (2007) show a deterministic behavior, instead of our nondeterministic choice.

Other timed extensions of concurrent constraint programming have been proposed in Palamidessi and Valencia (2001) and Nielsen and Valencia (2002); however, these languages, different from *tsccp*, do not take into account quantitative aspects; therefore, this achievement represents a very important expressivity improvement with respect to related works. These have been considered by Di Pierro and Wiklicky (1998), who have extensively studied probabilistic *ccp*. This language provides a

https://doi.org/10.1017/S1471068414000106 Published online by Cambridge University Press

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. More recently, stochastic *ccp* has been introduced in Bortolussi (2006) to model biological systems. This language is obtained by adding a stochastic duration to ask and tell primitives, thus it differs from our solutions.

In literature we can find other proposals that are related to tuple-based kernel-languages instead of a constraint store, such as the *Kernel Language for Agents Interaction and Mobility (* KLAIM) (de Nicola *et al.* 1998) or the *Software Component Ensemble Language (* SCEL) (De Nicola *et al.* 2011), for instance. These languages are designed to study different properties of systems, such as mobility and autonomicity of modeled agents. Their basic specifications do not encompass time-based primitives, while mobility features are not present in any of the constraint-based languages reported in this section. The purpose of our language is to model systems where a level of preference and time-sensitive primitives (as a timeout) are required: a good example is represented by agents participating to an auction, as given in Section 5.1.

In general, since semiring-based soft constraints allow one to express several quantitative features, our proposal provides a framework that can be instantiated to obtain a variety of specific extensions of *ccp*.

11 Conclusions and the future work

We have presented *tsccp* and *tsccp-i* in order to join together the expressive capabilities of soft constraints and timing mechanisms in a new programming framework. The agents modeled with these languages are able to deal with time- and preference-dependent decisions that are often found during complex interactions. An application scenario can be represented by different entities that need to negotiate generic resources or services, as, for instance, during an auction process. Mechanisms as timeout and interrupt may model the wait for pending conditions or the triggering of some new events. All the *tsccp* and *tsccp-i* rules have been formally described by a transition system and then also by a denotational characterization of operational semantics obtained with the use of *timed reactive sequences*. The resulting semantics has been proved to be compositional and correct.

About future work, the first improvement of the presented languages could be the inclusion of a *fail* agent in the syntax given in Definitions 1 and 7, and a semantics for the transition rules that lead to a failed computation, in case the guard on the transition rule cannot be enforced due to the preference of store. In fact, the transition systems that we have defined consider successful computations only. If this could be a reasonable choice in a don't know interpretation of a language, it will lead to an insufficient analysis of behavior in a pessimistic interpretation of indeterminism.

At last, we would like to consider other time management strategies (as the one proposed in Valencia 2003), and to study how timing and non-monotonic constructs (Bistarelli and Santini 2011) can be integrated together.

https://doi.org/10.1017/S1471068414000106 Published online by Cambridge University Press

References

Berry, G. and Gonthier, G. 1992. The ESTEREL synchronous programming language: Design, semantics, implementation. *Science of Computer Programming 19,* 2, 87–152.

Bistarelli, S. 2004. *Semirings for Soft Constraint Solving and Programming (LNCS)*. Springer-Verlag, London.

Bistarelli, S., Gabbrielli, M., Meo, M. C. and Santini, F. 2008. Timed soft concurrent constraint programs. In *Proceedings of the 10th International Conference on Coordination Models and Languages, COORDINATION*. Lecture Notes in Computer Science, vol. 5052. Springer, London, 50–66.

Bistarelli, S., Montanari, U. and Rossi, F. 1997. Semiring-based constraint satisfaction and optimization. *Journal of the ACM 44,* 2, 201–236.

Bistarelli, S., Montanari, U. and Rossi, F. 2006. Soft concurrent constraint programming. *ACM Transactions on Computer Logic 7,* 3, 563–589.

Bistarelli, S. and Santini, F. 2011. A non-monotonic soft concurrent constraint language to model the negotiation process. *Fundamenta Informaticae 111,* 3, 257–279.

Bortolussi, L. 2006. Stochastic concurrent constraint programming. *Electronic Notes in Theoretical Computer Science 164,* 3, 65–80.

Brookes, S. D. 1993. Full abstraction for a shared variable parallel language. In *Proceedings of 8th Annual IEEE Symposium on Logic in Computer Science,* (LICS '93) . IEEE Computer Society, Los Alamitos, CA, 98–109.

Busi, N., Gorrieri, R. and Zavattaro, G. 2000. Process calculi for coordination: From Linda to JavaSpaces. In *Proceedings of AMAST*, LNCS, vol. 1816. Springer-Verlag, London, 198–212.

de Boer, F. S., Gabbrielli, M. and Meo, M. C. 2000. A timed concurrent constraint language. *Information and Computation 161,* 1, 45–83.

de Boer, F. S., Gabbrielli, M. and Meo, M. C. 2004. A timed Linda language and its denotational semantics. *Fundamenta Informaticae 63,* 4, 309–330.

de Boer, F. S. and Palamidessi, C. 1991. A fully abstract model for concurrent constraint programming. In *TAPSOFT '91: Proceedings of CAAP '91*, vol. 1. Springer-Verlag, New York, NY, 296–319.

De Nicola, R., Ferrari, G. L., Loreti, M. and Pugliese, R. 2011. A language-based approach to autonomic computing. In *10th International Symposium on Formal Methods for Components and Objects (FMCO)*, B. Beckert, F. Damiani, F. S. de Boer and M. M. Bonsangue, Eds. Lecture Notes in Computer Science, vol. 7542. Springer, New York, NY, 25–48.

de Nicola, R., Ferrari, G. L., and Pugliese, R. 1998. KLAIM: A Kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering 24,* 5, 315–330.

Di Pierro, A. and Wiklicky, H. 1998. Probabilistic concurrent constraint programming: Towards a fully abstract model. In *Proceedings of Mathematical Foundations of Computer Science (MFCS '98)*. Springer-Verlag, London, 446–455.

Gallien, J. and Gupta, S. 2007. Temporary and permanent buy out prices in online auctions. *Management Science 53,* 5, 814–833.

Halbwachs, N., Caspi, P., Raymond, P. and Pilaud, D. 1991. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE 79,* 9, 1305–1320.

Harel, D. 1987. Statecharts: A visual formalism for complex systems. *Science of Computer Programming 8,* 3, 231–274.

https://doi.org/10.1017/S1471068414000106 Published online by Cambridge University Press

Jonsson, B. 1985. A model and proof system for asynchronous networks. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC '85)*. ACM Press, New York, NY, 49–58.

le Guernic, P., le Borgne, M., Gautier, T. and le Maire, C. 1991. Programming real-time applications with signal. *Proceedings of the IEEE 79,* 9, 1321–1336.

Nielsen, M. and Valencia, F. D. 2002. Temporal concurrent constraint programming: Applications and behavior. In *Formal and Natural Computing – Essays Dedicated to Grzegorz Rozenberg*, W. Brauer, H. Ehrig, J. Karhumäki and A. Salomaa, Eds. Springer-Verlag, London, 298–324.

Olarte, C., Palamidessi, C. and Valencia, F. 2007. Universal timed concurrent constraint programming. In *Proceedings of the 23rd International Conference on Logic Programming (ICLP)*. LNCS, vol. 4670. Springer, London, 464–465.

Palamidessi, C. and Valencia, F. D. 2001. A temporal concurrent constraint programming calculus. In *Proceedings of Principles and Practice of Constraint Programming (CP '01)*. Springer-Verlag, London, 302–316.

Saraswat, V. 1989. *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie-Mellon University, Pittsburgh, PA.

Saraswat, V., Jagadeesan, R. and Gupta, V. 1996. Timed default concurrent constraint programming. *Journal of Symbolic Computation 22,* 5–6, 475–520.

Saraswat, V. and Rinard, M. 1990. Concurrent constraint programming. In *Proceedings of Principles of Programming Languages (POPL '90)*. ACM Press, New York, NY, 232–245.

Saraswat, V. A., Jagadeesan, R. and Gupta, V. 1994. Foundations of timed concurrent constraint programming. In *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science (LICS '94)*. IEEE Computer Society, Los Alamitos, CA, 71–80.

Valencia, F. D. 2003. Timed concurrent constraint programming: Decidability results and their application to LTL. In *19th International Conference on Logic Programming (ICLP '03)*. LNCS, vol. 2916. Springer, London, 422–437.

https://doi.org/10.1017/S1471068414000106 Published online by Cambridge University Press