A Timed Concurrent Constraint Language

F. S. de Boer  
*Department of Computer Science, Universiteit Utrecht, Padualaan 14, De Uithof, 3584 CH Utrecht, The Netherlands*  
E-mail: frankb@cs.uu.nl

M. Gabbielli  
*Dipartimento di Matematica e Informatica, Università di Udine, Via delle Scienze 206, 33100 Udine, Italy*  
E-mail: gabbri@dimi.uniud.it

and

M. C. Meo  
*Dipartimento di Matematica Pura ed Applicata, Università di L’Aquila, Via Vetoio, Loc. Coppito, 67010 L’Aquila, Italy*  
E-mail: meo@univaq.it

We study a timed concurrent constraint language, called *tccp*, which is obtained by a natural timed interpretation of the usual *ccp* constructs: action-prefixing is interpreted as the next-time operator and the parallel execution of agents follows the scheduling policy of maximal parallelism. Additionally, *tccp* includes a simple primitive which allows one to specify timing constraints. We define the operational semantics of *tccp* by means of a transition system and we define a denotational model which is fully abstract with respect to the usual notion of observables (that is, the results of terminating computations). Moreover, we study the semantics and expressive power of the notion of maximal parallelism underlying the computational model of *tccp*: We define a fully abstract semantics for a sublanguage of *tccp*, called *ccpm*, which essentially is concurrent constraint programming, provided that we interpret the parallel operator in terms of maximal parallelism rather than of interleaving. We show that *tccp* is strictly more expressive than *ccpm* which, in its turn, is strictly more expressive than *ccp*.

1. INTRODUCTION

Time critical aspects are essential to an increasingly large number of applications, including the representation of time-dependent data, modeling of reactive and real-time systems, and the specification and verification of distributed, concurrent systems.
The concept of time is particularly important in reactive systems [25, 22]: These are systems which react continuously with their environment and which often require a programmer to specify timing constraints such as, for example, that a certain input is required within a certain bounded period of time. Reactive systems include real-time systems (e.g., process controllers, signal processing systems) which are subject to hard timing constraints. Many different formalisms have been specifically developed to deal with reactive systems and they can be roughly classified according to the following three categories.

Timed process algebras [1, 2, 14, 20, 32] have been obtained from classic process algebras (like CCS, CSP, and ACP) by adding the notion time and by including several timing operators. The resulting formalisms, differently from their untimed ancestors, can be used to specify and verify reactive (and real-time) systems, since they allow one to model such notions as time-outs, exceptions, priorities, and interrupts.

The second category includes a variety of formalisms based on (temporal) logic which have been mainly devoted to the verification of reactive systems. Recently executable temporal logics (ETL) have been proposed as powerful tools which combine the logical perspective with an operational model, often tailored to some intended application. These formalisms have already been used for applications in several different areas including hardware simulation, temporal databases, and temporal planning. We refer to [18, 19] for an introduction to ETL and for a specific bibliography on this subject ([19] includes also a short description of a few important existing ETL systems such as Chronolog, F-LIMETTE, Concurrent METATEM, and Tempura).

The third category comprehends those languages which have been specifically designed for programming reactive systems. Traditionally these systems were programmed mainly by using deterministic automata. Since large automata are difficult to design, maintain, and modify, several high-level languages for reactive programming have been defined in the past few years. Particularly important in this context are the concurrent synchronous languages ESTEREL [5], LUSTRE [23], SIGNAL [29], and Statecharts [24], which have already been used in many industrial applications. These languages are based on the instantaneous reaction (or perfect synchrony) hypothesis: A program is activated by some input signals and reacts instantly by producing the required output. So computation is performed in no time, unless a statement which explicitly consumes time is present. Communication is done by instantaneous broadcasting to all the processes of the system and the presence or absence of a signal can be detected at any instant. The perfect synchrony assumption can be realized in practice by compiling programs (which satisfy some requirements) into finite state automata whose single step execution time is bounded. A direct compilation of pure ESTEREL programs in hardware has also been defined.

Inspired by these formalisms a different approach to specify and program reactive systems has recently emerged in the context of constraint programming. This is a promising programming paradigm in which the idea of generating and satisfying constraints is central to the computing process. Constraint programming has been well blended with logic programming (see [27] for an overview) and with concurrency: Concurrent constraint programming (ccp) [36, 38, 39] has been proposed
as a general concurrent computational model and Oz [43] has been developed as a concurrent, high-level language which combines object oriented features with symbolic computation and constraints. The abstraction from the flow of control inherent to these declarative languages facilitates the transition from specifications to programs and simplifies the semantic issues. Consequently in these languages the additional complexity induced by timing constraints of various sorts can be singled out more clearly.

In this paper we study a timed extension of ccp that we call tccp. Similarly to the other existing timed extensions of ccp [40, 41], tccp is a language for reactive programming designed around the hypothesis of bounded asynchrony [40]: Computation takes a bounded period of time rather than being instantaneous. However, our proposal differs from those in [40, 41] for three main reasons which are discussed in Section 6. Notably, while the computational model of both the languages tcc (timed concurrent constraint programming) [40] and default tcc [41] is inspired by that one of synchronous languages our proposal follows the guidelines of the timed process algebras approach. Therefore, while tcc and default tcc are deterministic languages our language allows for nondeterminism. As advocated by the designers of ESTEREL [5], deterministic (concurrent) languages should be used for programming kernels of real-time systems, since deterministic systems are simpler to specify, debug, and analyze than nondeterministic ones. However, nondeterminism arises when considering larger reactive systems involving several processes running on different processors and communicating via asynchronous links. These (timed) systems can then be naturally specified and programmed by using a nondeterministic language. Furthermore, even though a system is ultimately implemented by using deterministic constructs, often using nondeterminism allows one to abstract away uninteresting details, thus simplifying the task of the programmer.1 As a matter of fact, all the existing timed process algebras [1, 2, 14, 20, 32] and almost all the variants of Statecharts [3] admit nondeterminism.

We describe semantically our timed extension of ccp both operationally, in terms of a transition system, and denotationally, by defining a fix-point semantics which is fully abstract w.r.t. the input-output notion of observables. The denotational semantics is based on sequences of pairs of constraints, so-called reactive sequences, similar to those used in the context of dataflow languages [28] of (standard) ccp [12] and of imperative languages [10, 15]. However reactive sequences are now provided with a different interpretation which accounts for the timing aspects: Intuitively, each pair \((c_i, d_i)\) represents a computation step performed by the agent \(A\) which, at time \(i\), assuming \(c_i\) as input constraint, produces the constraint \(d_i\). The parallel operator in tccp is interpreted in terms of maximal parallelism; i.e., at each moment every enabled agent of the system is activated. This interpretation, which is common to many timed process algebras, is natural when considering a timed language and it is different from the one of standard ccp, where parallelism is interpreted in terms of interleaving. Maximal parallelism, together with the presence of an explicit timing primitive, introduces new issues when considering the problem of

1 For example, in the context of finite state automata, it is well known nondeterminism can be replaced by determinism at the cost of an (worst case) exponential increase of the number of states.
full abstraction. In fact, the proof of our full abstraction result differs substantially from the corresponding one for ccp. One of the main differences lies in the need for further assumptions on the constraint system. In Section 4 we discuss these assumptions and show that, under some reasonable conditions, they are necessary in order to obtain in general a fully abstract semantics based on timed reactive sequences. We focus on the specific characteristics of maximal parallelism by defining a fully abstract semantics for the (sub)language ccpm obtained from tcep by removing the explicit timing primitive.

The differences appearing at a semantic level among tcep, ccpm, and ccp are further investigated by formally comparing the expressive power of these languages. Intuitively, a (programming) language $L$ is more expressive than a language $L'$ if each program written in $L$ can be translated into an $L'$ program in such a way that the intended observable behavior of the original program is preserved. This notion has been formalized under the name of embedding as follows [37, 13]. Consider two languages $L$ and $L'$ and let $P_L$ and $P_{L'}$ denote the set of the programs which can be written in $L$ and in $L'$, respectively. Assume that the meaning of programs is given by two functions (observables) $\mathcal{O}: P_L \rightarrow \text{Obs}$ and $\mathcal{O}': P_{L'} \rightarrow \text{Obs}'$ which associate to each program the set of its observable properties (thus Obs and Obs' are assumed being some suitable power sets). Then we say that $L$ is more expressive than $L'$, or equivalently that $L'$ can be embedded into $L$, if there exist a mapping $\mathcal{E}: P_{L'} \rightarrow P_L$ (compiler) and a mapping $\mathcal{D}: \text{Obs} \rightarrow \text{Obs}'$ (decoder) such that, for each program $P'$ in $P_{L'}$, the equality $\mathcal{D}(\mathcal{O}(\mathcal{E}(P'))) = \mathcal{O}'(P')$ holds; i.e., given a program $P'$ in $L'$, its observables can be obtained by decoding the observables of the program $\mathcal{E}(P')$ resulting from the translation of $P'$ into $L$. Clearly, as discussed in [13], in order to use the notion of embedding as a tool for language comparison some further restrictions should be imposed on the decoder and on the compiler. Otherwise the previous equation would be satisfied by any Turing complete language, provided that we choose a powerful enough $\mathcal{E}$ for the target language. Usually these conditions indicate how easy the translation process is and how reasonable the decoder is. The notion of embedding in general depends on the notion of observables, which should be expressive enough.2 We consider a quite general class of observables which covers all the properties derivable from finite computations.

We show that, due to the presence of the explicit timing primitive, tcep cannot be embedded in ccpm nor in ccp, while ccpm cannot be embedded into ccp, since maximal parallelism augments the expressive power of the (global) choice operator. Differently from [13], we obtain these separation results without taking into account termination modes and by using an (abstract) operational semantics rather than a denotational one.

The rest of the paper is organized as follows. In the next section we introduce tcep, our timed extension of ccp, and we define its operational semantics. Section 3 describes some derived constructs and two programming examples. Section 4 is devoted to the definition of the denotational semantics and to the full abstraction

---

2 Considering, for example, a trivial $\mathcal{E}$ which associates the same element to any program, clearly we could embed a language into any other one.
In this section we first introduce the \textit{tccp} language and provide its basic operational intuitions. Then we define formally the operational semantics of \textit{tccp} using a transition system.

As in [40, 41] the starting point is \textit{ccp}, so we introduce first some basic notions related to this programming paradigm (we refer to [38, 39] for more details). The \textit{ccp} languages are defined parametrically w.r.t. to a given cylindric constraint system. The notion of a cylindric constraint system has been formalized in [38] following Scott’s treatment of information systems [42] and using ideas from cylindric algebras [26] in order to define the hiding operator of the language in terms of a general notion of an existential quantifier. Here we only consider the resulting structure.

**Definition 2.1.** Let \( \langle \mathcal{C}, \leq, \sqcup, \text{true}, \text{false} \rangle \) be a complete algebraic lattice where \( \sqcup \) is the lub operation, and \text{true}, \text{false} are the least and the greatest elements of \( \mathcal{C} \), respectively. Assume a (denumerable) set of variables \( \text{Var} \) with typical elements \( x, y, z \ldots \) is given. Moreover, assume that for each \( x \in \text{Var} \) a function \( \_x : \mathcal{C} \to \mathcal{C} \) is defined such that, for any \( c, d \in \mathcal{C} \), the following axioms hold:

\begin{enumerate}
  \item \( c \models \exists_x(c) \),
  \item if \( c \models d \) then \( \exists_x(c) \models \exists_x(d) \),
  \item \( \exists_x(c \sqcup \exists_x(d)) = \exists_x(c) \sqcup \exists_x(d) \),
  \item \( \exists_x(\exists_x(c)) = \exists_x(\exists_x(c)) \).
\end{enumerate}

Then \( \mathcal{C} = \langle \mathcal{C}, \leq, \sqcup, \text{true}, \text{false}, \text{Var}, \exists \rangle \) is a cylindric constraint system.

Following the standard terminology and notation, instead of \( \leq \) we will refer to its inverse relation, denoted by \( \models \) and called entailment. Formally, \( \forall c, d \in \mathcal{C} \), \( c \models d \iff d \leq c \). Moreover, in the following we will identify a system \( \mathcal{C} \) with its underlying set of constraints \( \mathcal{C} \). Finally, in order to model parameter passing, diagonal elements [26] are added to the primitive constraints: We assume that, for \( x, y \) ranging in \( \text{Var}, D \) contains the constraints \( d_{xy} \) which satisfy the following axioms.

\begin{enumerate}
  \item \( \text{true} \models d_{xx} \),
  \item if \( z \neq x, y \) then \( d_{xy} = \exists_x(d_{xx} \sqcup d_{yy}) \),
  \item if \( x \neq y \) then \( d_{xy} \sqcup \exists_x(c \sqcup d_{xy}) \models c \).
\end{enumerate}

Note that if \( \mathcal{C} \) models the equality theory, then the elements \( d_{xy} \) can be thought of as the formulas \( x = y \). In the following \( \exists_x(c) \) is denoted by \( \exists_x c \) with the convention that, in case of ambiguity, the scope of \( \exists_x \) is limited to the first constraint subexpression. (So, for instance, \( \exists_x c \sqcup d \) stands for \( \exists_x(c \sqcup d) \).)
The basic idea underlying ccp is that computation progresses via monotonic accumulation of information in a global store. Information is produced by the concurrent and asynchronous activity of several agents which can add (tell) a constraint to the store. More precisely, given a store $d$, the agent $\text{tell}(c)$ updates the store to $c \cup d$. Dually, agents can also check (ask) whether a constraint is entailed by the store, thus allowing synchronization among different agents. So the action $\text{ask}(c)$ represents a guard, i.e., a test on the current store $d$, whose execution does not modify $d$: if $d \models c$ then $\text{ask}(c)$ is enabled (or satisfied) in $d$, otherwise $\text{ask}(c)$ is suspended. Nondeterminism arises by introducing a guarded choice operator: The agent $\sum_{i=1}^n \text{ask}(c_i) \rightarrow A_i$ nondeterministically selects one $\text{ask}(c_i)$ which is enabled in the current store and then behaves like $A_i$. If no guard is enabled, then this agent suspends, waiting for other (parallel) agents to add information to the store. Deterministic ccp is obtained by imposing the restriction $n = 1$ in the above construct.

The $\|$ operator allows one to express parallel composition of two agents $A \parallel B$ and it is usually described in terms of interleaving. Finally a notion of locality is obtained by introducing the agent $xA$ which behaves like $A$, with $x$ considered local to $A$.

When querying the store for some information which is not present (yet) a ccp agent will simply suspend until the required information has arrived. In many applications involving time, however, often one cannot wait indefinitely for an event. Consider, for example, the case of a bank teller machine. Once a card is accepted and its identification number has been checked, the machine asks the authorization of the bank to release the requested money. If the authorization does not arrive within a reasonable amount of time, then the card should be given back to the customer. In order to model such a situation then the language should allow us to specify that, in case a given time bound is exceeded (i.e., a time-out occurs), the wait is interrupted and an alternative action is taken. Moreover, in some cases it is also necessary to abort an active process $A$ and to start a process $B$ when a specific event occurs (this is usually called preemption of $A$). For example, according to a typical pattern, $A$ is the process controlling the normal activity of some physical device, the event indicates some abnormal situation, and $B$ is the exception handler.

In order to enrich ccp agents with such timing mechanisms, we introduce a discrete global clock and assume that ask and tell actions take one time-unit. Computation evolves in steps of one time-unit, so called clock-cycles. We consider action prefixing as the syntactic marker which distinguishes a time instant from the next one. So $\text{tell}(c)$ has now to be regarded as the agent which updates the current store by adding $c$ and then, at the next time instant, stops. Analogously, if $c$ is entailed by the current store then the agent $\text{ask}(c) \rightarrow A$ behaves like $A$ at the next time instant. If $c$ is not entailed at time $t$ then the agent is suspended, i.e., at time $t+1$ it is checked again whether the store entails $c$.

Note that if a $\text{tell}(c)$ action is performed at time $t$ then the updated store will be visible only from time $t+1$ onward, since a $\text{tell}$ takes one time-unit to be completed. Thus, for example, the agent $A : (\text{ask}(c) \rightarrow \text{stop}) \parallel \text{tell}(c)$ evaluated in the empty store will take two time-units to successfully terminate.

---

3 The extension to the nondeterministic case is immediate.
Furthermore, we make the assumption 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 and, for example, implies that previous agent $A$ evaluated in the store $c$ terminates in one time-unit. The time between two successive moments of the global clock intuitively corresponds to the response time of the underlying constraint system. Thus, essentially in our model all parallel agents are synchronized by such a response time. Since the store is monotonically increasing and one can have dynamic process creation, clearly the previous assumptions in principle imply that the constraint solver takes a constant time (no matter how big the store is) and that there is an unbound number of processors. In practice, however, one can impose suitable restrictions 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).

So far we have only described a timed interpretation of the usual $ccp$ combinators. We still have to introduce the notions of time-out and preemption which, as previously mentioned, are essential to many applications. There exist some time critical applications (see [41, 4]) in which strong preemption is required: The abort of a process and the execution of the new one must happen at the same time of the detection of the event. However, often weak preemption is sufficient; i.e., it is acceptable having a unit delay between the detection of the event and the consequent action. We will then consider here a form of weak preemption: The abort of a process and the start of the new one happen at the same time of the detection of the event, while the result of the execution of the new process will be visible only in the next time instant. This choice allows us to obtain a programming paradigm useful for many applications, while maintaining a simple semantic model.

In general, as pointed out in [40], the essence of the time-out and the preemption mechanisms is in the ability to detect the absence of an event, as well as its presence. Such a detection can interrupt a process and trigger some alternative actions. Since events in $ccp$ can be expressed by the presence (more precisely, entailment) of a constraint in the store, we are led to the following timing construct

$$\text{now } c \text{ then } A \text{ else } B$$

which is similar to the analogous construct in [40]. However, while the now construct in [40] allows one to specify the behavior at the next time instant, we interpret the above construct in terms of instantaneous reaction as follows: If $c$ is entailed by the store then the above agent behaves as $A$ at the current time instant; otherwise it behaves as $B$ (at the current time instant). As we will discuss in Section 3, assuming this instantaneous reaction we can express such timing constructs as time-out and preemption in terms of the now then else operator. In practice, this instantaneous reaction can be obtained by evaluating now $c$ in parallel with $A$ and $B$ within one time-unit. At the end of the time-unit the store will be updated by using either the constraint produced by $A$ or that 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. Notice that for recursive programs such a limit is ensured by the presence of the procedure call, since we assume that the evaluation of such a call takes one time-unit.

Thus, we end up with the following syntax.

**Definition 2.2 (tccp Language).** Assuming a given cylindric constraint system $C$ the syntax of agents is given by the following grammar,

\[
A ::= \text{stop} \mid \text{tell}(c) \mid \sum_{i=1}^{n} \text{ask}(c_i) \rightarrow A_i \mid \text{now } c \text{ then } A \text{ else } B \mid A \& B \mid \exists x\ A \mid p(x),
\]

where the $c, c_i$ are supposed to be finite constraints (i.e., algebraic elements) in $C$.

A *tccp process* $P$ is then an object of the form $D.A$, where $D$ is a set of procedure declarations of the form $p(x) :- A$ and $A$ is an agent.

In order to simplify the notation, in the following we will omit the $\sum_{i=1}^{n}$ whenever $n = 1$ and we will use $\text{tell}(c) \rightarrow A$ as a shorthand for $\text{tell}(c) \& (\text{ask}(\text{true}) \rightarrow A)$.

### 2.1. Operational Semantics

The operational model of *tccp* can be formally described by a transition system $T = (\text{Conf}, \rightarrow)$ where we assume that each transition step takes exactly one time-unit. Configurations (in) $\text{Conf}$ are pairs consisting of an agent and a constraint in $C$ representing the common store. The transition relation $\rightarrow \subseteq \text{Conf} \times \text{Conf}$ is the least relation satisfying the rules $\text{R1}-\text{R10}$ in Table 1 and characterizes the (temporal) evolution of the system. So, $\langle A, c \rangle \rightarrow \langle B, d \rangle$ means that if at time $t$ we have the agent $A$ and the store $c$ then at time $t+1$ we have the agent $B$ and the store $d$.

Let us now briefly discuss the rules in Table 1.

The agent *stop* represents successful termination, so it cannot make any transition. Rule *R1* shows that we are considering here the so-called eventual tell: The agent $\text{tell}(c)$ adds $c$ to the store $d$ without checking for consistency of $c \cup d$ and then stops at the next time instant. Note that the updated store $c \cup d$ will be visible only starting from the next time instant. This means that the evaluation of a tell action takes one time-unit, since each transition step involves exactly one time-unit.

According to rule *R2* the guarded choice operator gives rise to global nondeterminism: The external environment can affect the choice since $\text{ask}(c_i)$ is enabled at time $t$ (and $A_i$ is started at time $t+1$) iff the store $d$ entails $c_i$ and $d$ can be modified by other agents. As it results from the transition rule, also the evaluation of an ask action takes one time-unit.

The rules *R3*-R6 show that the agent *now c then A else B* behaves as A or B depending on the fact that $c$ is or is not entailed by the store. Note that here, differently from the case of the ask, the evaluation of the guard is instantaneous: If $\langle A, d \rangle \rightarrow \langle B, d \rangle$ can make a transition at time $t$ and $c$ is (not) entailed by the store $d$, then the agent *now c then A else B* can make the same transition at time $t$. As previously mentioned, this assumption on the instantaneous evaluation is needed to express the preemption mechanism in terms of the *now then else* construct.
### TABLE 1
The Transition System for tccp

<table>
<thead>
<tr>
<th>Rule</th>
<th>Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>(\langle \text{tell } c, d \rangle \rightarrow \langle \text{stop}, c \cup d \rangle)</td>
</tr>
<tr>
<td>R2</td>
<td>(\sum_{i=1}^{n} \langle \text{ask}(c_i) \rightarrow A_i, d \rangle \rightarrow \langle A_i, d \rangle) for (j \in {1, n}) and (d \vdash c_j)</td>
</tr>
</tbody>
</table>
| R3    | \(\langle A, d \rangle \rightarrow \langle A', d' \rangle\) for 
\(\langle \text{now } c \text{ then } A \text{ else } B, d \rangle \rightarrow \langle A', d' \rangle\) with \(d \vdash c\) |
| R4    | \(\langle A, d \rangle \rightarrow \langle B, d \rangle\) for 
\(\langle \text{now } c \text{ then } A \text{ else } B, d \rangle \rightarrow \langle B, d' \rangle\) with \(d \vdash c\) |
| R5    | \(\langle B, d \rangle \rightarrow \langle B', d' \rangle\) for 
\(\langle \text{now } c \text{ then } A \text{ else } B, d \rangle \rightarrow \langle B', d' \rangle\) with \(d \vdash c\) |
| R6    | \(\langle B, d \rangle \rightarrow \langle B', d' \rangle\) for 
\(\langle \text{now } c \text{ then } A \text{ else } B, d \rangle \rightarrow \langle B', d' \rangle\) with \(d \vdash c\) |
| R7    | \(\langle A, c \rangle \rightarrow \langle A', c' \rangle\) for 
\(\langle A \parallel B, c \rangle \rightarrow \langle A' \parallel B', c' \cup d' \rangle\) |
| R8    | \(\langle A, c \rangle \rightarrow \langle A', c' \rangle\) for 
\(\langle A \parallel B, c \rangle \rightarrow \langle A' \parallel B', c' \rangle\) |
| R9    | \(\langle A, d \parallel \exists x c \rangle \rightarrow \langle A, d' \parallel \exists x c' \rangle\) |
| R10   | \(\langle p(x), c \rangle \rightarrow \langle A, c \rangle\) for 
\(p(x) :- A \in D\) |

Rules R7 and R8 model the parallel composition operator in terms of maximal parallelism: The agent \(A \parallel B\) executes in one time-unit all the initial enabled actions of \(A\) and \(B\).

As specified by rule R9, the agent \(\exists x A\) behaves like \(A\), with \(x\) considered local to \(A\); i.e., the information on \(x\) provided by the external environment is hid from \(A\) and, conversely, the information on \(x\) produced locally by \(A\) is hid from the external world.

To describe locality in rule R9 the syntax has been extended by an agent \(\exists x A\) where \(d\) is a local store of \(A\) containing information on \(x\) which is hidden in the external store. Initially the local store is empty, i.e., \(\exists x A = \exists x_{\text{true}} A\).

Rule R10 treats the case of a procedure call when the actual parameter equals the formal parameter: in this case a simple body replacement suffices. We do not need more rules since, for the sake of simplicity, here and in the following we assume that the set \(D\) of procedure declarations is closed w.r.t. parameter names: That is, for every procedure call \(p(y)\) appearing in a process \(D.A\) we assume that if the original declaration for \(p\) in \(D\) is \(p(x) :- A\) then \(D\) contains also the declaration \(p(y) :- \exists x_{\text{true}} A\).

\(^4\) 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.
Using the transition system described by (the rules in) Table 1 we can define the following notion of observables which considers the input–output of terminating computations, including the deadlocked ones. Other notions of observables (e.g., timed traces) could also be interesting in this context. We consider here the following notion because this is the one usually considered in the papers dealing with semantics of ccp languages (e.g., see [12]). Furthermore, it can be used as the basis to define compositional proof-systems along the lines developed in [9]. Here and in the following $\rightarrow^*$ denotes the reflexive and transitive closure of the relation $\rightarrow$.

**Definition 2.3** (Observables). Let $A$ be an agent. We define $C_{\text{o}}(A) = \{ \langle c, d \rangle \mid \langle A, c \rangle \rightarrow^* \langle B, d \rangle \Rightarrow \}$.

### 3. PROGRAMMING IDIOMS AND EXAMPLES

We show now how some typical reactive programming idioms can be derived from the basic combiners of tccp.

**Delay.** The delay constructs

$$\text{tell}(c) \rightarrow A \quad \text{and} \quad \text{ask}(c) \rightarrow A$$

delay the execution of $A$ (after the execution of $\text{tell}(c)$ and $\text{ask}(c)$) $\delta$ time-units. These constructs can be defined inductively in our programming language as follows: $\text{tell}(c) \rightarrow A$ denotes the agent $(\text{tell}(c) \rightarrow (\downarrow_\delta A), \text{where} \downarrow_{\delta + 1} A$ is inductively defined by $\text{tell}(\text{true}) \rightarrow (\downarrow_\delta A)$, with $\downarrow_0 A$ being defined by $A$ (and similarly for $\text{ask}(c) \rightarrow A$).

**Time-out.** The timed guarded choice agent

$$\sum_{i=1}^{n} \text{ask}(c_i) \rightarrow A_i \text{ time-out}(m) B$$

waits at most $m$ time-units ($m \geq 0$) for the satisfaction of one of the guards. Before this time-out the process behaves just like the guarded choice: As soon as there exist enabled guards, one of them (and the corresponding branch) is nondeterministically selected. After waiting for $m$ time-units, if no guard is enabled, the timed choice agent behaves as $B$. This agent can be defined inductively as follows. Let us denote by $A$ the agent $\sum_{i=1}^{n} \text{ask}(c_i) \rightarrow A_i$. In the base case, $m = 0$, we define $\sum_{i=1}^{n} \text{ask}(c_i) \rightarrow A_i \text{ time-out}(0) B$ as the agent

$$\text{now } c_1 \text{ then } A \text{ else } (\text{now } c_2 \text{ then } A \text{ else } ...)$$
For the inductive step we define $\sum_{i=1}^{n} \text{ask}(c_i) \rightarrow A_i \text{ time-out}(m) \ B$ as

$$\sum_{i=1}^{n} \text{ask}(c_i) \rightarrow A_i \text{ time-out}(0) \left( \sum_{i=1}^{n} \text{ask}(c_i) \rightarrow A_i \text{ time-out}(m-1) \ B \right).$$

It is immediate to check that the above inductively defined agent has the expected operational behavior. Consider, for example, the base case. If the current store entails one of the guards, say $c_j$, then (by rule R3 the agent $\sum_{i=1}^{n} \text{ask}(c_i) \rightarrow A_i$ is immediately evaluated; that is, the agent $A_j$ is evaluated at the next time instant. Otherwise, the agent $B$ is evaluated at the next time instant. Note also that the cascade of now then else commands does not impose any priority on the guards of the time-out mechanism. In fact, if a $c_j$ (argument to a now) is satisfied, then the whole nondeterministic choice agent $A$ is evaluated.

Watchdogs. These are typical preemption primitives of such languages as ESTEREL. Watchdogs are used to interrupt the activity of a process on a signal from a specific event: In our framework, since events are expressed by constraints, a watchdog can be defined as the process

$$\text{do } A \text{ watching } c$$

which behaves as $A$, as long as $c$ is not entailed by the store; when $c$ is entailed, the process $A$ is immediately aborted. As discussed above, we have instantaneous reaction 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.

Previous watchdog agents can be defined in terms of the other constructs of the language as follows. Assume that there exists an (injective) renaming function $\rho$ which, given a procedure name $p$, returns a new name $\rho(p)$ which is not used elsewhere in the program. Moreover, let us use now $c$ else $B$ as a shorthand for now $c$ then stop else $B$. Then we have the following translation $\Rightarrow$:

$\text{do } \text{tell}(d) \text{ watching } c \quad \Rightarrow \text{now } c \text{ else } \text{tell}(d),$ 

$\text{do } \sum_{i=1}^{n} \text{ask}(c_i) \rightarrow A_i \text{ watching } c \quad \Rightarrow \text{now } c \text{ else } \sum_{i=1}^{n} \text{ask}(c_i) \rightarrow \text{do } A_i \text{ watching } c,$

$\text{do } \text{now } d \text{ then } A \text{ else } B \text{ watching } c \quad \Rightarrow \text{now } d \text{ then } \text{do } A \text{ watching } c$

$\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \]}
where we assume that, for any procedure \( p \) declared as \( p(x) :\land A \), a declaration
\[ p(p(x)) :\land do \rho(A) \text{ watching } c \] is added, where \( \rho(A) \) denotes the agent obtained
from \( A \) by replacing in it each occurrence of any procedure \( q \) by \( \rho(q) \).\(^5\) The
assumption in the case of the \( \exists x A \) agent is needed for correctness. In practical cases
it can be satisfied by suitably renaming the variables associated to signals. The pre-
vious translation can be easily extended to the case of the agent \( do A \text{ watching } c \) else \( B \), which behaves as the previous watchdog and also activates the process \( B \)
when \( A \) is aborted (i.e., when \( c \) is entailed). In the following we will then use also
this form of watchdog.

The assumption on the instantaneous evaluation of \( \text{now } c \) is essential in order to
obtain a preemption mechanism which can be expressed in terms of \( \text{now} \) then \( \text{else} \) primitive. In fact, if the evaluation of \( \text{now } c \) took one time-unit then this
unit delay would change the compositional behavior of the agent controlled by the
watchdog. Consider, for example, the agent \( A : \text{tell}(a) \rightarrow \text{tell}(b) \) which takes two
time-units to complete its computation. The agent \( A^*: \text{now } c \text{ else } \text{tell}(a) \rightarrow \text{now } c
\text{ else } \text{tell}(b) \) (resulting from the translation of \( \text{do } A \text{ watching } c \)) behaves composi-
tionally as \( A \), unless a \( c \) signal is detected, in which case the evaluation of \( A \) is inter-
rupted. On the other hand, if the evaluation of \( \text{now } c \) took one time-unit then \( A^* \)
would take four time-units and would not behave anymore as \( A \) when \( c \) is not pre-
sent. In fact, in this case, the agent \( A \parallel B \) would produce \( d \) while \( A^* \parallel B \) would not,
where \( B \) is the agent \( \text{tell}(\text{true}) \rightarrow \text{now } a \text{ then } \text{tell}(d) \text{ else stop} \).

3.1. A System Controller

As a simple example of a \textit{tccp} program let us consider a system \( s(Ex) \) consisting
of two processes \( p_1 \) and \( p_2 \) which perform some time-critical activities, reacting to
external inputs transmitted on the channel \( Ex \). The system is continuously checked
by a controller which receives a stream of \( \text{ok} \) messages by each process \( p_i \). Each
\( \text{ok} \) message is sent at unpredictable time instants; however, it is assumed that each \( p_i \)
is working correctly iff it sends the next \( \text{ok} \) within \( n \) time-units from the previous
one. When this limit is exceeded by a process \( p_i \) the controller aborts the whole
system, starts a recovery routine \( rr \) for the activity of \( p_i \), and then restarts the
system. The system \( s(Ex) \) is implemented by the following program where we do
not detail the specific tasks of the \( p_i \)’s and of the recovery routines:

\[
s(Ex) :\\exists \text{Alarm}, O_1, R_1, O_2, R_2
\begin{align*}
&\{(\text{do } p_1(Ex,O_1,R_1) \parallel p_2(Ex,O_2,R_2) \text{ watching Alarm = on}) \\
&\quad \text{controller}(O_1,O_2,R_1,R_2)\}\end{align*}
\]

controller(\(O_1,O_2,R_1,R_2):=\exists A1,A2
\begin{align*}
&\{(\text{do } c(O_1,A1) \parallel c(O_2,A2) \text{ watching Alarm = on else}) \\
&\quad (\text{now } (A1 = \text{on } \sqcup A2 = \text{on}) \text{ then } rr(R1) \parallel rr(R2) \text{ else})\}\end{align*}

\(^5\) These problems are, of course, a matter of the garbage collection in the implementation. However,
we would like to point out here that in many practical cases the theoretically unbound growing of the
store in \textit{(timed) ccp} is not a serious problem, since many variables are used only locally and the memory
allocated for them can be reused.
now A1 = on then rr(R1) else
now A2 = on then rr(R2))
| restart(Ex))
c(O,A) :- ask(∃Y.O = [ok | Y]) → (∃Y tell(O = [ok | Y]) → c(Y,A))
timeout(n) tell(Alarm = on ∧ A = on)

The reading of this program should be immediate: The call of s(Ex) activates in parallel the controller and the agents p1 and p2 which use the channel Ex to communicate with the external environment and two internal channels: Oi (i ∈ {1, 2}) is used to send the ok messages to the controller while Ri allows the process to pass information on the current status to the recovery routine. The agents p1 and p2 are in the scope of a watchdog which is controlled by the signal Alarm: As soon as the event Alarm = on is detected the activity of these two agents is interrupted.

The controller activates two parallel copies (one for each pi) of the process c(O,A) which, using a time-out construct, checks for the correct emission of the ok signals on the channel O. Whenever an interval of more than n time-units is detected between two next ok messages (on the same channel) the variable Alarm is set to on in order to signal an error. Also the variable Ai is set to on in order to identify the process pi which caused the error. When an error is detected the controller interrupts the activity of the agent c(O1,A1) | c(O2,A2) (since this is in the scope of a watchdog) and starts the appropriate recovery routine rr(Ri) (depending on the value of Ai) together with the restart(Ex) routine (this will restart the whole system later on). Notice that the nesting of now then else constructs in the previous program allows one to express priorities among guards which are evaluated within the same time instant. Moreover, since the variables Alarm,O1,R1,O2,R2 are local to the agent s(Ex), two next activations of the agent s(Ex) could use the same memory locations for these variables, thus avoiding the unbound growing of the occupied memory. Analogously for the variables A1 and A2 in controller and Y in c(O,A).6

3.2. Railroad Crossing

Next we consider a standard example which models the real-time control of a crossing of n train-tracks similar to that one shown in [30]. The behavior of trains on track i is modeled by the following declaration.

\[ \text{train}(T_i) := (\text{ask(true) → train}(T_i)) + \exists T, V(\text{tell}(T_i = [s | T]) \rightarrow s(\text{tell}(T = [o | V]) \rightarrow \text{train}(V))) \]

The passing of trains on the i-th track is thus described by a stream Ti of signals s and o. The signal s indicates the entering of a train (on track i) in the crossing area. The signal o, on the other hand, indicates that a train has exited the crossing area.

6 The reason for having a renamed version of each procedure p is that in general we cannot change its original declaration, since other occurrences of p (not in the scope of an watchdog) may need it.
area. It is assumed that it takes a train $\delta > 0$ time-units to pass the crossing area (recall that $\text{tell}(c) \rightarrow A$ and $\text{ask}(c) \rightarrow A$) and delay the execution of $A$ $\delta$ time-units after the execution of $\text{tell}(c)$ and $\text{ask}(c)$.

The streams $T_1, ..., T_n$ are processed by the controller which keeps track of the number of trains present in the crossing area (which is maximal $n$) and which controls the crossing by means of a stream $K$ of signals $l$ (lower) and $r$ (raise). Let $T$ denote $T_1, ..., T_n$ and denote the result of replacing $T_i$ in $T_1, ..., T_n$ by $T_i$. Furthermore we introduce the notation $\exists x(c \rightarrow A)$ as an abbreviation of $\text{ask}(\exists x c) \rightarrow \exists \text{tell}(c) \parallel A).$ We define

$$\text{control}_0(T, K) := \bigoplus_{i=1}^{n} \exists T (T_i = [s \mid T] \rightarrow \exists L (\text{tell}(K = [l \mid L]) \rightarrow \text{control}_1(T_r, L)))$$

$$\text{control}_1(T, K) := \bigoplus_{i=1}^{n} \exists T (T_i = [s \mid T] \rightarrow \text{control}_2(T_r, K))$$

$$+$$

$$\bigoplus_{i=1}^{n} \exists T (T_i = [o \mid T] \rightarrow \exists L (\text{tell}(K = [r \mid L]) \rightarrow \text{control}_0(T_r, L)))$$

$$...$$

$$\text{control}_j(T, K) := \bigoplus_{i=1}^{n} \exists T (T_i = [s \mid T] \rightarrow \text{control}_{j+1}(T_r, K))$$

$$+$$

$$\bigoplus_{i=1}^{n} \exists T (T_i = [o \mid T] \rightarrow \text{control}_{j-1}(T_r, K))$$

$$...$$

$$\text{control}_n(T, K) := \bigoplus_{i=1}^{n} \exists T (T_i = [o \mid T] \rightarrow \text{control}_n - 1(T_r, K))$$

Finally, the crossing itself reads the stream $K$ of signals emitted by the controller and indicates the completion of the actions of raising and lowering the gates by a stream $C$ of signals $d$ (down) and $u$ (up). Assuming that lowering the gates takes $\delta_1$ time-units, the action of lowering the gates is modeled by

$$\text{lower}(K, C) := \exists L (K = [l \mid L] \rightarrow d, \exists D (\text{tell}(C = [d \mid D]) \rightarrow \text{raise}(L, D)))$$

The procedure raises models the raising of the gates. Raising the gates is assumed to take $\delta_2$ time-units. When raising the gates, if a lower signal arrives then the action of raising the gates has to be aborted and the action of lowering the gates initiated. This can be described by means of the $\text{do A watching c else B}$ construct previously described. We define

$$\text{raise}(K, C) := \exists L (K = [r \mid L] \rightarrow$$

$$\text{do up}(L, C) \text{ watching } \exists M (L = [l \mid M])$$

$$\text{else } \exists M (\text{tell}(L = [l \mid M]) \parallel \text{down}(M, C))$$
where the procedures up and down are defined as follows.

\[
\begin{align*}
\text{up}(K, C) & := \downarrow_{\delta} \Diamond \text{tell}(C = [u \mid D]) \rightarrow \text{lower}(K, D) \\
\text{down}(K, C) & := \downarrow_{\delta} \Diamond \text{tell}(C = [u \mid D]) \rightarrow \text{raise}(K, D)
\end{align*}
\]

Given the above procedure declarations the behavior of the train-crossing system is described by the agent

\[
\text{train}(T_1) \parallel \cdots \parallel \text{train}(T_n) \parallel \text{control}_0(T, K) \parallel \text{lower}(K, C)
\]

in the empty store \text{true} (which represents the initial situation; i.e., no trains are in the crossing area and the gates are up).

We observe that this program provides a correct and realistic description of a railroad crossing assuming that the basic time-unit of the underlying computational model is small with respect to the external delays \(\delta, \delta_1, \delta_2\). For example, it takes the controller \(n\) time-units to process the simultaneous crossing of \(n\) trains. In real-life situations, however, we may safely assume that none of those trains will exit the crossing before the controller has processed them all. In our model this assumption can be formalized simply be requiring that \(n\) is (sufficiently) smaller than \(\delta\).

4. THE DENOTATIONAL MODEL

It is easy to see that the operational semantics which associates to an agent \(A\) its observables \(O_{io}(A)\) is not compositional. In this section we define a compositional characterization of the operational semantics obtained by using sequences of pairs of finite constraints, so called \emph{timed reactive sequences}, to represent tccp computations. These sequences are similar to those used in the semantics of dataflow languages [28] of (standard) ccp [12] and of imperative languages [10, 15].

We introduce a denotational model which associates to an agent a set of (timed) reactive sequences of the form

\[
\langle c_1, d_1 \rangle \cdots \langle c_n, d_n \rangle \langle d, d' \rangle
\]

where a pair of constraints \(\langle c_i, d_i \rangle\) represents a computation step performed by a generic agent at time \(i\): Intuitively, the agent transforms the global store from \(c_i\) to \(d_i\) or, in other words, \(c_i\) is the assumption on the external environment while \(d_i\) is the contribution of the agent itself. The last pair denotes a \emph{stuttering step} in which no further information can be produced by the agent, thus indicating that a \emph{resting point} has been reached.

Since in tccp computations the store evolves monotonically, it is natural to assume that reactive sequences are monotonically increasing. So in the following we will assume that each timed reactive sequence \(\langle c_1, d_1 \rangle \cdots \langle c_{n-1}, d_{n-1} \rangle \langle c_n, d_n \rangle\) satisfies the following condition, \(d_i \vdash c_i\) and \(c_j \vdash d_{j-1}\), for any \(i \in [1, n-1]\) and
Since the constraints arising from computation steps are finite, we also assume that a reactive sequence contains only finite constraints.7

The set of all reactive sequences is denoted by \( \mathcal{S} \) and its typical elements by \( s, s_1, \ldots \), while sets of reactive sequences are denoted by \( S, S_1, \ldots \) and \( \emptyset \) indicates the empty reactive sequence. Furthermore, given a sequence \( s = \langle c_1, d_1 \rangle \langle c_2, d_2 \rangle \cdots \langle c_{n-1}, d_{n-1} \rangle \langle c_n, e_n \rangle \), we define first \((s) = c_1 \) and result \((s) = e_n \), while \( \cdot \) denotes the operator which concatenates sequences. Operationally the reactive sequences of an agent are generated as follows.

**Definition 4.1.** We define the semantics \( R(A) : \mathcal{P}(\mathcal{S}) \rightarrow \mathcal{P}(\mathcal{S}) \) by

\[
R(A) = \{ \langle c, d \rangle \mid w \in \mathcal{S} \mid \langle A, c \rangle \rightarrow \langle B, d \rangle \text{ and } w \in R(B) \} \\
\cup \\
\{ \langle c, c \rangle \mid w \in \mathcal{S} \mid \langle A, c \rangle \rightarrow \langle B, c \rangle \text{ and } w \in R(A) \cup \{ \varepsilon \} \}.
\]

Formally \( R \) is defined as the least fixed-point of the corresponding operator \( \Phi : \mathcal{P}(\mathcal{S}) ightarrow \mathcal{P}(\mathcal{S}) \) defined by

\[
\Phi(I)(A) = \{ \langle c, d \rangle \mid w \in \mathcal{S} \mid \langle A, c \rangle \rightarrow \langle B, d \rangle \text{ and } w \in I(B) \} \\
\cup \\
\{ \langle c, c \rangle \mid w \in \mathcal{S} \mid \langle A, c \rangle \rightarrow \langle B, c \rangle \text{ and } w \in I(A) \cup \{ \varepsilon \} \}.
\]

The ordering on \( \mathcal{P}(\mathcal{S}) \) is that of (point-wise extended) set-inclusion (it is straightforward to check that \( \Phi \) is continuous).

**Definition 4.2.** Let \( s = \langle c_1, d_1 \rangle \langle c_2, d_2 \rangle \cdots \langle c_{n-1}, d_{n-1} \rangle \langle c_n, c_n \rangle \) be a reactive sequence. We say that \( s \) is connected if \( c_i = d_{i+1} \) for each \( i \in [2, n] \).

According to the previous definition a sequence is connected if all the information assumed on the external environment is produced by the agent itself, apart from the initial input constraint. Thus a connected sequence \( s = \langle c_1, c_2 \rangle \langle c_3, c_3 \rangle \cdots \langle c_{n-1}, c_n \rangle \langle c_n, c_n \rangle \) represents a tccp computation where \( c_1 \) is the input constraint, while \( c_n \) is the result. It follows immediately from the definition of \( R \) that we can obtain the observables of the agent \( A \) from the connected sequences in \( R(A) \). So we have the following result whose proof is immediate.

**Proposition 4.3.** For any agent \( A \) we have,

\[
\mathcal{O}(A) = \{ \langle c, d \rangle \mid \text{there exists a connected sequence } s \in R(A) \text{ such that} \\
\text{first}(s) = c \text{ and result}(s) = d \}.
\]

In order to show that \( R \) is compositional we introduce the following semantic operators.

7 Note that here we implicitly assume that if \( c \) is a finite element then also \( \exists \_c c \) is finite.
Definition 4.4. Let \( S, S_1, \) and \( S_2 \) be sets of reactive sequences and \( c, c_i \) be constraints. Then we define the operators \( \sum, \nbow, \) and \( \exists x \) as follows:

**Guarded choice.**

\[
\sum_{i=1}^{n} c_i \rightarrow S_i = \{ s \cdot s' \in \mathcal{F} \mid s = \langle d_1, d_1 \rangle \ldots \langle d_m, d_m \rangle, \ d_j \vdash c_i \ 	ext{for each} \ j \in [1, m-1], \ i \in [1, n], \\
\ d_m \vdash c_k \ \text{and} \ s' \in S_k \ \text{for an} \ h \in [1, n] \} \\
\cup \\
\{ s \in \mathcal{F} \mid s = \langle d_1, d_1 \rangle \ldots \langle d_m, d_m \rangle, \ d_j \vdash c_i \ 	ext{for each} \ j \in [1, m], \ i \in [1, n] \}. 
\]

**Parallel composition.** Let \( \overline{\oplus} : \mathcal{F} \times \mathcal{F} \rightarrow \mathcal{F} \) be the (commutative and associative) partial operator defined as follows:

\[
\langle c_1, d_1 \rangle \ldots \langle c_n, d_n \rangle \langle d, d \rangle \overline{\oplus} \langle c_1, e_1 \rangle \ldots \langle c_n, e_n \rangle \langle d, d \rangle = \langle c_1, d_1 \uplus e_1 \rangle \ldots \langle c_n, d_n \uplus e_n \rangle \langle d, d \rangle.
\]

We define \( S_1 \overline{\oplus} S_2 \) as the point-wise extension of the above operator to sets.

**The non-operator.**

\[
\nbow(c, S_1, S_2) = \{ s \in \mathcal{F} \mid s = \langle c', d \rangle \ldots s' \ 	ext{and either} \ c' \vdash c \ 	ext{and} \ s \in S_1 \ 	ext{or} \ c' \not\vdash c \ 	ext{and} \ s \in S_2 \}.
\]

The hiding operator. We first need the following notions similar to those used in [11]: Given a sequence \( s' = \langle c_1, d_1 \rangle \ldots \langle c_n, c_n \rangle \), we say that \( s' \) is \( x \)-connected if

- \( \exists_x c_1 = c_1 \) (that is, the input constraint of \( s' \) does not contain information on \( x \)) and
- \( \exists_x c_i \uplus d_{i-1} = c_i \) for each \( i \in [2, n] \) (that is, each assumption \( c_i \) does not contain any information on \( x \) which has not been produced previously in the sequence by some \( d_j \)).

A sequence \( s \) is \( x \)-invariant if

- for all computation steps \( \langle c, d \rangle \) of \( s, d = \exists_x d \uplus c \) holds.

The semantic hiding operator then can be defined as follows:

\[
\exists x S = \{ s \in \mathcal{F} \mid \text{there exists} \ s' \in S \ 	ext{such that} \ \exists_x s = \exists_x s', \ s' \ 	ext{is} \ x \text{-connected and} \\
s \ 	ext{is} \ x \text{-invariant} \}.
\]
A few explanations are in order here. Concerning the semantic choice operator, a sequence in \( \sum_{i=1}^{n} c_i \rightarrow S_i \) consists of an initial period of waiting for (a constraint stronger than) one of the constraints \( c_i \). During this waiting period only the environment is active by producing the constraints \( d_i \) while the process itself generates the stuttering steps \( \langle d_i, d_i \rangle \). Here we can add several pairs since the external environment can take several time-units to produce the required constraint. When the contribution of the environment is strong enough to entail a \( c_h \) 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 (the \( c_i \)'s) and that they have the same length (in all other cases the parallel composition is assumed being undefined).

In the definition of \( \overline{3} \) we say that a sequence is \( x \)-connected if no information on \( x \) is present in the input constraints which has not been already accumulated by the computation of the agent itself. A sequence is \( x \)-invariant if its computation steps do not provide more information on \( x \). So, we have the following theorem whose proof is straightforward and therefore omitted.

**Theorem 4.5.** The semantics \( R \) satisfies the equations of Table 2.

### 4.1. Full abstraction for tccp

The model defined in the previous subsection is correct; however, it introduces unnecessary distinctions or, in other words, it is not fully abstract. In fact, as shown by the following example, the semantics \( R \) distinguishes tccp agents whose observables (as defined by \( O_{io} \)) are the same under any possible context. Here and in the following loop is defined by the declaration \( \text{loop} : \text{&} \rightarrow \text{ask(true)} \rightarrow \text{loop} \) and a context \( C[ ] \) is simply an agent with a hole. The agent \( C[A] \) then represents the result of replacing the hole in \( C[ ] \) by \( A \).

**TABLE 2**

<table>
<thead>
<tr>
<th>Denotational Semantics of tccp</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E1 ) ( R(\text{stop}) = { \langle c_1, c_1 \rangle, \langle c_2, c_2 \rangle, \ldots, \langle c_n, c_n \rangle \in S \mid n \geq 1 } )</td>
</tr>
<tr>
<td>( E2 ) ( R(\text{tell}(c)) = { \langle d, d \cup c \rangle : x \in S \mid x \in R(\text{stop}) } )</td>
</tr>
<tr>
<td>( E3 ) ( R \left( \sum_{i=1}^{n} \text{ask}(c_i) \rightarrow A_i \right) = \sum_{i=1}^{n} c_i \rightarrow R(A_i) )</td>
</tr>
<tr>
<td>( E4 ) ( R(\text{now } c \text{ then } A \text{ else } B) = \text{now}(c, R(A), R(B)) )</td>
</tr>
<tr>
<td>( E5 ) ( R(A \mid B) = R(A) \mid R(B) )</td>
</tr>
<tr>
<td>( E6 ) ( R(\exists x A) = \exists x R(A) )</td>
</tr>
<tr>
<td>( E7 ) ( R(p(x)) = R(\text{tell}(true) \rightarrow A) \mid p(x) : A \in D )</td>
</tr>
</tbody>
</table>
Example 4.6. Consider the \textit{tccp} agents

\begin{align*}
A & : \text{ask}(\textit{true}) \rightarrow C \quad \text{and} \quad B : \text{ask}(\textit{true}) \rightarrow C \\
& + \\
& \text{ask}(\textit{true}) \rightarrow D
\end{align*}

where \(C\) and \(D\) are the agents

\begin{align*}
C & : \text{tell}(c) \rightarrow (\text{ask}(d) \rightarrow \text{stop}) \\
& + \\
& \text{ask}(\textit{true}) \rightarrow \text{loop}
\end{align*}

\begin{align*}
D & : \text{tell}(\textit{true}) \rightarrow (\text{ask}(d) \rightarrow \text{stop}) \\
& + \\
& \text{ask}(\textit{true}) \rightarrow \text{loop}
\end{align*}

with \(d \geq c\) and \(c \neq d\). We have \(R(A) \neq R(B)\), since \(<\textit{true}, \textit{true}>, <\textit{true}, \textit{true}>, <d, d> \in R(A) \setminus R(B)\). However, the agents \(A\) and \(B\) cannot be distinguished by any \textit{tccp} context. In fact, since \(A\) contains \(B\), clearly \(C_{\textit{w}}[C'[B]] \subseteq C_{\textit{w}}[C'[A]]\) for any context \(C'[\_]\). Moreover, also the other inclusion holds: If the agent \(D\) produces a result in the context \(C'[\_]\), then the constraint \(d\) required by the \textit{ask}(\textit{d})\) has to be already produced by the external environment \(C'[\_]\) when the \textit{tell}(\textit{true})\) is performed (otherwise the rule of maximal parallelism forces the computation to enter the loop branch). Since \(c \leq d\), clearly if we replace \textit{tell}(\textit{true})\) by \textit{tell}(\textit{c})\) we cannot observe any difference. So also the agent \(C\) can be successfully evaluated in \(C'[\_]\) and it produces the same result as \(D\). Note that the presence of \(+\) and of the loop agent in the definition of \(C\) and \(D\) is essential (otherwise \(A\) and \(B\) could be distinguished).

In order to identify agents like the previous ones and to obtain a fully abstract semantics (\textit{w.r.t.} \(\ell_{\textit{w}}\)) we need a suitable abstraction on denotations. However, due to the presence of the \textit{now then else} construct and of maximal parallelism, we cannot use the abstraction which has been used in [12] for \textit{ccp} since this would be incorrect. In fact, it is easy to construct \textit{tccp} contexts which distinguish (according to our notion of observables) most of the programs which are observably equivalent under any \textit{ccp} context. Consider, for example, the agents \(A : \text{tell}(c \sqcup d)\) and \(B : \text{tell}(e) \rightarrow \text{tell}(d)\) where \(c \neq d\). These two agents are identified by the fully abstract semantics for \textit{ccp}; indeed they are observably equivalent (\textit{w.r.t.} input–output) under any \textit{ccp} context. However, they can be distinguished by the following \textit{tccp} (and also \textit{ccpm}) context

\begin{align*}
C & : [\_] \rightarrow \text{tell}(\textit{true}) \rightarrow (\text{ask}(c \sqcup d) \rightarrow \text{stop}) \\
& + \\
& \text{ask}(c) \rightarrow \text{tell}(e)
\end{align*}

where \((c \sqcup e) \sqcup d \neq c \sqcup d\), since \(<\textit{true}, c \sqcup d> \in C_{\textit{w}}[C'[A]] \setminus C_{\textit{w}}[C'[B]]\).

Analogous examples can be done by using the \textit{now then else} construct. So, the full abstraction problem for \textit{tccp} (and for \textit{ccpm}) cannot be reduced to the analogous one for \textit{ccp}. Indeed, as discussed more precisely later in this section, our
full abstraction result requires the ability to specify the difference between an assumption \( c_i \) and the previous contribution \( d_{i-1} \). Such a difference is formalized by using the weak relative pseudo-complement \( c_i \setminus d_{i-1} \) which has been defined in [21] for (semi) lattices by relaxing the standard notion of relative pseudo-complement [6]. In our setting, a constraint system \( \mathcal{C} \) is weakly relative pseudo-complemented if, for each pair \( c, d \) of constraints in \( \mathcal{C} \) such that \( c \leq d \), there exists a (unique) constraint \( d \setminus c \) (called the weak relative pseudo-complement of \( c \) w.r.t. \( d \)) such that the following hold:

1. \( c \uplus (d \setminus c) = d \) and
2. if \( c \uplus d' = d \) for some \( d' \) then \( (d \setminus c) \leq d' \).

So, the weak relative pseudo-complement of \( c \) w.r.t. \( d \) represents the least amount of information which has to be added to \( c \) to obtain \( d \). If in the conditions above we replace \( = \) for \( \geq \) and we consider any pair of constraints (so, \( c \) does not need to be \( \leq d \)) then we obtain the more common notion of relative pseudo-complement.

A lattice in which for any given pair of elements \( c, d \) there exists the relative pseudo-complement (and therefore also the weak relative pseudo-complement) of \( c \) w.r.t. \( d \) is called Browerian. Well-known results [6] ensure that the elements of any finite distributive lattice form a Browerian lattice, that any chain is a Browerian lattice, and that a lattice is Browerian iff it is completely meet distributive. Furthermore, clearly any Boolean lattice is also Browerian. As for constraint systems, in practice most of them consist of sets of first order formulas built from some primitive constraints (i.e., atomic formulas) by using the usual first order connectives. Clearly, if one does not impose any restriction on connectives, then the resulting constraint systems are (weakly) relatively pseudo-complemented, since the first order formulas (modulo logical equivalence) form a Boolean algebra (also called the Lindenbaum algebra). However, often only some connectives are allowed in constraints (typically existential quantification and conjunction) and therefore one cannot express anymore the (weak) relative pseudo-complement.

It is worth noting that some practical constraint systems are weakly relative pseudo-complemented. This is the case, for example, of the Gentzen constraint system which has been used for real-time computation in default tcc [41].

For our full abstraction result we also require that the constraint system is a finitary domain, i.e., that for each finite (algebraic) element \( c_0 \in \mathcal{C} \) the set \( \{ d_0 \mid d_0 \leq c_0 \text{ and } d_0 \text{ is finite} \} \) is finite. This assumption is satisfied by several constraint systems (e.g., all the constraint systems considered in [41], namely Herbrand, finite domains, and Gentzen).

Following the standard practice [12], in order to obtain a fully abstract semantics we “saturate” a denotation by adding all those reactive sequences which do not introduce new observables under any context. Here and in the rest of this section we assume that the given constraint system is a finitary domain and is weakly relative pseudo-complemented.

---

8 It is easy to see that if a lattice is relatively pseudo-complemented then it is also weakly relatively pseudo-complemented, while in general the converse does not need to hold [21].
Definition 4.7 (Saturation). We define the less-connected relation on sequences, denoted by $\lesssim$, as follows. Let $s, s'$ be reactive sequences. Then

- $s \lesssim s'$ (s is less connected than $s'$) iff for some sequences $s_1$ and $s_2$ we have that $s = s_1 \cdot \langle a, b \rangle \cdot (c, d') \cdot s_2$, $s' = s_1 \cdot \langle a, b' \rangle \cdot (c, d') \cdot s_2$, and $(c \cdot b') \leq (c \cdot b)$.

Moreover, we define the (equivalence) relation $\simeq$ as follows:

- $s \simeq s'$ iff the sequences $s$ and $s'$ differ only in the number of repetitions of the last element.

Given a set of reactive sequences $S$, we denote by $\llbracket S \rrbracket$ the least set $S'$ such that the following holds:

1. $S \subseteq S'$,
2. if $s' \in S'$ and either $s \lesssim s'$ or $s \simeq s'$, then $s \in S'$.

So, given a set of sequences $S$, the saturation $\llbracket \cdot \rrbracket$ is defined point-wise on $S$ and adds all those sequences which differ from those already in $S$ either in the fact that they are less connected or in the number of stuttering steps at the end. Intuitively, the fact that $s$ is less connected than $s'$ means that the gaps existing between what is produced ($d_i$) and what is assumed at the next time instant ($c_{i+1}$) are bigger in $s$ than in $s'$. In other words, when composing sequences via the $\parallel$ operator, $s$ needs more tell contributions than $s'$ in order to obtain a connected sequence, so less connected sequences can be added safely to the denotation of an agent. Notice that from the above definition it follows immediately that $\llbracket \cdot \rrbracket$ is extensive, monotonic, and idempotent. That is, $\llbracket \cdot \rrbracket$ is a closure operator on $(\mathcal{S}, \subseteq)$.

The fully abstract semantics $R_{\llbracket \cdot \rrbracket}$ is obtained by simply applying the function $\llbracket \cdot \rrbracket$ to $R(A)$; that is, $R_{\llbracket \cdot \rrbracket}(A) = \llbracket R(A) \rrbracket$. In the following we will prove the compositionality, correctness, and full abstraction results for $R_{\llbracket \cdot \rrbracket}$. The proof of the following theorem is deferred to the Appendix.

Theorem 4.8 (Compositionality of $R_{\llbracket \cdot \rrbracket}$). Let $A$, $B$, and $A_i$ be generic tccp agents. Then $R_{\llbracket \cdot \rrbracket}$ satisfies the following equalities:

1. $R_{\llbracket \sum_{i=1}^{n} c_i \rightarrow A_i \rrbracket} = \llbracket \sum_{i=1}^{n} c_i \rightarrow R(A_i) \rrbracket$,
2. $R_{\llbracket \text{now } c \text{ then } A \text{ else } B \rrbracket} = \llbracket \text{now } (c, R(A), R(B)) \rrbracket$,
3. $R_{\llbracket A \parallel B \rrbracket} = \llbracket R(A) \parallel R(B) \rrbracket$,
4. $R_{\llbracket 3x A \rrbracket} = \llbracket 3x R(A) \rrbracket$.

The semantics $R_{\llbracket \cdot \rrbracket}$ is correct, since the abstraction $\llbracket \cdot \rrbracket$ does not introduce any connected sequence giving a new input–output pair. Thus we have the following.

Theorem 4.9 (Correctness of $R_{\llbracket \cdot \rrbracket}$). For any agent $A$ we have

$$\llbracket A \rrbracket = \{ \langle c, d \rangle \mid \text{there exists a connected sequence } s \in R_{\llbracket \cdot \rrbracket}(A) \text{ such that } \text{first}(s) = c \text{ and result}(s) = d \}.$$  

Proof. First observe that, by definition of $\llbracket \cdot \rrbracket$, clearly all the sequences in $\llbracket S \rrbracket$ are obtained from sequences in $S$ by using a finite number of $\simeq$ and $\lesssim$ operations.
We then show that, given a set of reactive sequences \( S \), if \( s \in \pi(S) \setminus S \) is obtained by applying one abstraction operation and \( s \) is a connected sequence, then there exists \( s' \in S \) such that \( s' \) is connected, \( \text{first}(s) = \text{first}(s') \), and \( \text{result}(s) = \text{result}(s') \). The thesis then follows from an obvious inductive argument and from Proposition \( 4.3 \).

Since \( s \in \pi(S) \setminus S \) clearly \( s \) is obtained from another sequence \( s' \) by using \( \pi \). We consider two cases corresponding to the two components of the abstraction \( \pi \).

If \( s \approx s' \) clearly \( s \) is connected iff \( s' \) is connected. Moreover we have \( \text{first}(s) = \text{first}(s') \) and \( \text{result}(s) = \text{result}(s') \).

Assume now that \( s \approx s' \) where \( s \) is the connected sequence

\[
\langle c, c_1 \rangle \langle c_1, c_2 \rangle \cdots \langle c_{i-1}, c_i \rangle \langle c_i, c_{i+1} \rangle \cdots \langle c_n, d \rangle \langle d, d \rangle
\]

and (without loss of generality) \( s' \) is the sequence

\[
\langle c, c_1 \rangle \langle c_1, c_2 \rangle \cdots \langle c_{i-1}, c'_{i} \rangle \langle c_i, c_{i+1} \rangle \cdots \langle c_n, d \rangle \langle d, d \rangle.
\]

From the definition of \( \approx \) it follows that \( (c_i, c'_i) \not\approx (c_i, c_i) \). The definition of weak relative pseudo-complement implies that \( c_i = c'_i \) and therefore \( s' \) is also a connected sequence with \( \text{first}(s) = \text{first}(s') \) and \( \text{result}(s) = \text{result}(s') \), which concludes the proof.

Finally, we can prove the full abstraction result.

**Theorem 4.10 (Full abstraction).** Assume that the constraint system is weakly relative pseudo-complemented. Then, for any pair of tcp agents \( A \) and \( B \), \( \mathcal{R}_d(A) = \mathcal{R}_d(B) \iff \mathcal{C}_d(A) = \mathcal{C}_d(B) \) for each context \( C \cdot \).

*Proof.* Since the “only if” part follows from Theorem \( 4.9 \), it suffices to prove the “if” part. We prove the contrapositive by showing that if \( \mathcal{R}_d(A) \neq \mathcal{R}_d(B) \) then we can define an agent \( C_A \) such that \( \mathcal{C}_d(A) \neq \mathcal{C}_d(B) \). The proof is by contradiction.

Assume, without loss of generality, that there exists a sequence \( s = \langle c_1, d_1 \rangle \langle c_2, d_2 \rangle \cdots \langle c_n, d_n \rangle \in \mathcal{R}_d(A) \setminus \mathcal{R}_d(B) \). By definition of \( \mathcal{R}_d \) we can assume that either \( n = 1 \) or \( c_n \neq c_{n-1} \), then we define inductively the agent \( C_A \) which recognizes the sequence \( s \) as follows,

1. For \( s = \langle c, c \rangle \) we define \( C_A = \text{stop} \).
2. For \( s = \langle c_1, d_1 \rangle \langle c_2, d_2 \rangle \cdots \langle c_n, d_n \rangle \), \( n > 1 \), we define \( C_A \) as the agent

   \[
   \text{tell}(c_2, d_1) \rightarrow \text{(now } c'_1 \text{ then loop else)}
   \]

   \[
   \text{now } c'_2 \text{ then loop else}
   \]

   \[
   \vdots
   \]

   \[
   \text{now } c'_m \text{ then loop else}
   \]

   \[
   \text{now } c_2 \text{ then } C_{A_2} \text{ else loop)}
   \]

where \( s_2 \) is the sequence \( \langle c_2, d_2 \rangle \cdots \langle c_n, d_n \rangle \) and \( c'_1, c'_2, ..., c'_m \) are all the (finitely many) finite constraints such that \( c_2 < c'_1 \leq c_n \) for each \( i \in [1, m] \). As usual, \text{loop} is
defined by the declaration \( \text{loop} := \text{ask(true)} \rightarrow \text{loop} \) and \( c_2 \downarrow d_1 \) denotes the weak relative pseudo-complement of \( d_1 \) w.r.t. \( c_2 \). From the definition of the operational semantics it follows immediately that \( \langle c_1, c_n \rangle \in C_\text{op}(A \upharpoonright C_s) \). Assume now that \( \langle c_1, c_n \rangle \in C_\text{op}(B \upharpoonright C_s) \) holds. From Theorem 4.9, the definition of \( R_s \), and the extensivity of \( x \) it follows that there exist two sequences \( s' \in R_s(C_s) \) and \( s'' \in R_s(B) \) such that \( \text{first}(s') = \text{first}(s'') = c_1 \), \( \text{result}(s') = \text{result}(s'') = c_n \), and \( s' \parallel s'' \) is a connected sequence. From the definition of \( C_s \) given before it follows that if \( s'' \in R_s(C_s) \), \( \text{first}(s') = c_1 \), and \( \text{result}(s') = c_n \) then \( s' \) has the form

\[
\langle c_1, c_1 \downarrow (c_2 \downarrow d_1) \rangle \langle c_2, c_2 \downarrow (c_3 \downarrow d_2) \rangle \cdots \langle c_n, c_n \rangle.
\]

Moreover, since \( s' \parallel s'' \) is a connected sequence, from the definition of \( \parallel \) it follows that \( s'' \) has the form

\[
\langle c_1, e_1 \rangle \langle c_2, e_2 \rangle \cdots \langle c_n, e_n \rangle,
\]

where \( c_i \subseteq e_j \) and \( e_j \downarrow (c_{j+1} \downarrow d_j) = c_{j+1} \), for each \( i \in [1, n - 1] \). Now, let \( j \) be the least index \( i \) such that \( e_i \neq d_i \) (such a \( j \) exists, because by hypothesis \( s \notin R_s(B) \)). Since \( e_j \downarrow (c_{j+1} \downarrow d_j) = c_{j+1} \) the definition of the weak relative pseudo-complement implies that \( (c_{j+1} \downarrow e_j) \subseteq (c_{j+1} \downarrow d_j) \). Therefore we have that the sequence

\[
s_j: \langle c_1, d_1 \rangle \cdots \langle c_j, d_j \rangle \langle c_{j+1}, e_{j+1} \rangle \cdots \langle c_n, e_n \rangle
\]

is in \( R_s(B) \), since \( R_s(B) \) is closed under the relation \( \subseteq \). By repeating this argument for the sequences \( s_j \), with \( j \leq n \), we obtain that \( s \in R_s(B) \), which contradicts the hypothesis. This shows that \( \langle c_1, c_n \rangle \notin C_\text{op}(B \upharpoonright C_s) \) and concludes the proof.

It is worth noting that in the above proof, differently from the case of \( \text{ccp} \), one cannot recognize a reactive sequence \( s = \langle c_1, d_1 \rangle \cdots \langle c_n, e_n \rangle \) by simply mirroring it, i.e., by defining a context which “asks what \( s \) tells” (the \( d_i \)'s) and “tells what \( s \) asks” (the \( c_i \)'s). Such a construction can be used in the case of \( \text{ccp} \) [12] because \( \text{ccp} \) has an interleaving model for \( \parallel \). Therefore, when composing (in parallel) \( s \) and its mirror image, one can simply alternate their actions. Here, because of maximal parallelism, we cannot allow such an interleaving and in order to recognize \( s \) we use a context which fills the gaps between what \( s \) tells (\( d_i \)) and what \( s \) asks at the next step (\( c_{i+1} \)). This is formally expressed by using the weak relative pseudo-complement of \( d_i \) w.r.t. \( c_{i+1} \): In fact we construct a context whose denotation contains a sequence \( s' \) which, at each step, asks the same as \( s \) and tells \( c_{i+1} \downarrow d_i \).\(^9\) In order to guarantee the existence of these constraints which fill the gaps we have to assume that the constraint system is weakly relative pseudo-complemented. We also need to assume that \( \mathcal{E} \) is a finitary domain, since otherwise we should use an infinitary agent to obtain the distinguishing sequence.\(^10\)

\(^9\) To be more precise, in order to satisfy the monotonicity requirement, also, the \( c_i \) has to be added, so \( s' \) tells the constraint \( c_i \downarrow (c_{i+1} \downarrow d_i) \).

\(^10\) In the previous proof, the cascade of now \( c_i \) then loop else constructs is needed for correctness. Such a cascade would be infinitary if there were infinitely many finite constraints between \( c_2 \) and \( c_n \).
Some additional assumptions on the constraint system cannot be avoided if one wants to obtain a fully abstract semantics based on reactive sequences and by using a point-wise defined saturation condition. This is the content of the following proposition, which justifies our use of some additional structure in the constraint system, even though we did not prove that it is the minimal one needed to obtain the full abstraction result.

**Proposition 4.11.** Let $R$ be defined as in Definition 4.1 and let $\pi: \wp(\mathcal{S}) \to \wp(\mathcal{S})$ be a compositional saturation operator defined point-wise. If $R_a$ is compositional and correct in the sense of Theorem 4.9 then $R_a$ is not fully abstract for some constraint system.

**Proof.** The proof is by contradiction. Assume that $R_a$ is fully abstract and consider the constraint system $\langle \{\text{true}, d_1, d_2, d_3, \text{false} \}, \subseteq, \sqcup, \text{true}, \text{false} \rangle$, where $d_1$, $d_2$, and $d_3$ are not comparable (notice that this is a complete lattice which is not weakly relative pseudo-complemented). Then consider the $tccp$ ($ccpm$) agents $A: D_1 + D_2$ and $B: D_2 + D_3$ where, for $i \in [1, 3]$, $D_i$ is the agent

$$\text{ask(true)} \rightarrow \text{tell}(d_i) \rightarrow (\text{ask(false)} \rightarrow \text{stop} + \text{ask(true)} \rightarrow \text{loop})$$

and loop is defined as before. The sequence $s_i = \langle \text{true}, \text{true} \rangle \langle \text{true}, d_i \rangle \langle \text{false}, \text{false} \rangle$ is in $R(D_i) \setminus R(D_j)$, for each $i, j \in [1, 3]$ with $i \neq j$. Therefore, for each $j \in [1, 3]$, there exists a context $C_j = \text{tell}(d_j) \rightarrow \text{stop}$ such that, for any $i \in [1, 3]$ with $i \neq j$, $\langle \text{true}, \text{false} \rangle \in C_i(D_i) \parallel D_j$.

Since $\pi$ is a saturation operator we have $s_i \in R_{\pi}(D_1)$. Moreover, $s_i \not\in R_{\pi}(D_2) \cup R_{\pi}(D_3)$ must hold, since otherwise from the composibility of $R_a$ and Theorem 4.9 we would obtain that either $\langle \text{true}, \text{false} \rangle \in C_i(D_2) \parallel D_3$ or $\langle \text{true}, \text{false} \rangle \in C_i(D_3) \parallel D_2$, which is incorrect.

On the other hand, by definition of $+$ and the fact that $\pi$ is defined point-wise, it follows that $R_{\pi}(A) = R_{\pi}(D_1 + D_2) = R_{\pi}(D_1) \cup R_{\pi}(D_2)$ and analogously $R_{\pi}(B) = R_{\pi}(D_2) \cup R_{\pi}(D_3)$. Moreover, note that every finite computation that can be performed by $D_1$ (by $D_3$) can be performed by using either $D_2$ or $D_3$ (or $D_1$). Therefore, for any context $C$, we have $C_i(A) = C_i(B)$. From the full abstraction of $R_a$ it follows that $R_{\pi}(A) = R_{\pi}(B)$ and therefore that $R_{\pi}(D_1) \cup R_{\pi}(D_2) = R_{\pi}(D_2) \cup R_{\pi}(D_3)$ holds. This contradicts the fact that $s_i \in R_{\pi}(D_1) \setminus (R_{\pi}(D_2) \cup R_{\pi}(D_3))$ and concludes the proof.

### 4.2. Full Abstraction for ccp with Maximal Parallelism

In this section we consider the full abstraction problem for the language $ccpm$ obtained from $tccp$ by dropping the now then else statement or, equivalently, obtained from $ccp$ by interpreting the parallel operator in terms of maximal parallelism rather than interleaving.

Clearly the semantics $R_a$ introduced in the previous section is correct also for $ccpm$, being this a sublanguage of $tccp$. However it is not anymore fully abstract as shown by the following example.

---

11 We say that $\pi: \wp(\mathcal{S}) \to \wp(\mathcal{S})$ is defined point-wise if $\pi(S) = \bigcup_{x \vdash S} x'(s)$ for some $x' \in \mathcal{S} \to \mathcal{S}$. 
Example 4.12. Consider the ccpm agents

\[ \begin{align*}
A: & \text{ask(true) } \rightarrow \text{tell(c)} \\
B: & \text{ask(true) } \rightarrow \text{tell(c)} \\
& + \text{ask(true) } \rightarrow \text{tell(true)} \rightarrow \text{tell(c)}
\end{align*} \]

We have that \( R(A) \neq R(B) \), since \( s = \langle \text{true} \rangle \langle \text{true} \rangle \langle \text{true} \rangle \langle \text{true} \rangle \langle c \rangle \langle c \rangle \in R(A) \backslash R(B) \). Indeed, the agents A and B can be distinguished by the tccp context

\[ C: [ \ ] \rightarrow \text{tell(true)} \rightarrow \text{tell(true)} \rightarrow \text{now} \ c \ \text{then loop \ else \ stop.} \]

However, the agents A and B cannot be distinguished by any ccpm context. In fact, since A contains B, clearly \( C_{\text{cp}}(C[B]) \subseteq C_{\text{cp}}(C[A]) \) for any context \( C[\ ] \). Moreover, because of the monotonicity of ccpm computations, also the other inclusion holds: In fact, if the second branch of the agent A produces a result in the ccpm context \( C[\ ] \) then we can replace \( \text{tell(true)} \rightarrow \text{tell(c)} \) by \( \text{tell(c)} \) and obtain the same result, since \( \text{true} \subseteq c \). So also the agent B can be successfully evaluated in \( C[\ ] \) and it produces the same result as A.

As it results from the previous example, the now then else construct allows more distinguishing contexts, since it permits us to check for the absence of information. Therefore, in order to obtain a fully abstract semantics for ccpm, we need a saturation operator which is an abstraction of \( x \). As previously mentioned, also in the case of ccpm we cannot apply the saturation conditions which are used to obtain a fully abstract semantics for ccp and we need some further condition on the constraint system. In the rest of this section we then assume that the constraint system is relatively pseudo-complemented, i.e., that for each pair of constraints\(^{12}\) in \( C \) there exists a (unique) constraint \( d \backslash c \) \(^{13}\) (called the relative pseudo-complement of \( c \) w.r.t. \( d \)) such that the following hold:

1. \( c \sqcup (d \backslash c) \geq d \) and
2. if \( c \sqcup d' \geq d \) for some \( d' \) then \( (d \backslash c) \leq d' \).

The saturation on sets of reactive sequences is then defined analogously to the previous case, except that we use relative pseudo-complement rather than its weak version.

**Definition 4.13** (Saturation). We now define the weakly less connected relation \( \ll' \) on sequences as follows. Let \( s \) and \( s' \) be reactive sequences. Then

- \( s \ll' s' \) (\( s \) is weakly less connected than \( s' \)) if for some sequences \( s_1 \) and \( s_2 \) we have that \( s = s_1 \cdot \langle a, b \rangle \langle c, d \rangle \cdot s_2 \), \( s' = s_1 \cdot \langle a, b' \rangle \langle c', d \rangle \cdot s_2 \), \( \langle c' \rangle \backslash b' \subseteq \langle c \rangle \backslash b \), and \( c' \models c \).

Given a set of reactive sequences \( S \), we denote by \( \beta(S) \) the least set \( S' \) such that the following hold:

\(^{12}\) Actually, we use this condition only for pairs \( c, d \) such that \( c \subseteq d \).

\(^{13}\) For the sake of simplicity we use here the same notation used for weak relative pseudo-complement.
S \subseteq S',

(ii) if s' \in S' and either s \ll s' or s \simeq s', then s \in S' (\simeq is defined as in Definition 4.7).

The abstraction \( \beta \) is then defined as \( \pi \), provided that we consider weakly less connected sequences rather than less connected ones. As resulting from the previous definition \( \beta \) is coarser than \( \pi \), since if \( s \ll s' \) then \( s \ll s' \) holds. As expected, the fully abstract semantics \( R_\beta \) for \textit{ccpm} is obtained by simply applying \( \beta \) to \( R \) (that is, \( R_\beta(A) = \beta(R(A)) \)). The following compositionality and correctness results are proved similarly to the analogous one given for \( R_\pi \), so their proofs are omitted.

**Theorem 4.14 (Compositionality).** Let \( A, B, \) and \( A_i \) be generic \textit{ccpm} agents. Then the following equalities hold:

1. \( R_\beta(\sum_{i=1}^n \text{ask}(c_i) \rightarrow A_i) = \beta(\sum_{i=1}^n c_i \rightarrow R_\beta(A_i)) \).
2. \( R_\beta(\text{now } c \text{ then } A \text{ else } B) = \beta(\text{now}(c, R_\beta(A), R_\beta(B))) \).
3. \( R_\beta(A \parallel B) = \beta(R_\beta(A) \parallel R_\beta(B)) \).
4. \( R_\beta(\exists xA) = \beta(\exists xR_\beta(A)) \).

**Theorem 4.15 (Correctness of \( R_\beta \)).** For any agent \( A \) we have

\[
C_\mu(A) = \{ (c,d) \mid \text{there exists a connected sequence } s \in R_\beta(A) \text{ such that } c = \text{first}(s) \text{ and } d = \text{result}(s) \}.
\]

Finally we can have the following.

**Theorem 4.16 (Full abstraction).** Assume that \( \mathcal{C} \) is relatively pseudo-complemented. Then, for any pair of \textit{ccpm} agents \( A \) and \( B \), \( R_\beta(A) = R_\beta(B) \) if \( C_\mu(C[A]) = C_\mu(C[B]) \) for each context \( C[ ] \).

**Proof.** The “only if” part follows from Theorem 4.15. The proof of the “if” part is similar to the proof of Theorem 4.10 by using the following context \( C_n \) to recognize the sequence \( s = \langle c_1, d_1 \rangle \langle c_2, d_2 \rangle \cdots \langle c_n, d_n \rangle \). Let us define \( c_i = c_{i+1} \setminus d_i \) for \( i \in [1, n-1] \). Then in case \( n \) is even \( C_n \) is defined as

\[
\begin{align*}
\text{ask}(c_1) \rightarrow & \quad \text{tell}(e_2) \rightarrow \quad \cdots \rightarrow \text{ask}(c_{n-1}) \rightarrow \text{stop} \\
\text{+} \rightarrow & \quad \text{+} \\
\text{ask(true)} \rightarrow \text{loop} \quad & \quad \text{ask(true)} \rightarrow \text{loop} \\
\| \\
\text{tell}(e_1) \rightarrow & \quad \text{ask}(c_2) \rightarrow \quad \cdots \rightarrow \text{tell}(e_{n-1}) \rightarrow \quad \text{ask}(c_n) \rightarrow \text{stop} \\
\text{+} \rightarrow & \quad \text{+} \\
\text{ask(true)} \rightarrow \text{loop} \quad & \quad \text{ask(true)} \rightarrow \text{loop}
\end{align*}
\]
while in case \( n \) is odd \( C_s \) is defined as
\[
\text{ask}(c_1) \rightarrow \text{tell}(e_2) \rightarrow \ldots \rightarrow \text{tell}(e_{n-1}) \rightarrow \text{ask}(c_n) \rightarrow \text{stop}
\]
\[
+ \rightarrow +
\]
\[
\text{ask}(\text{true}) \rightarrow \text{loop}
\]
\[
\}
\]
\[
\text{ask}(\text{true}) \rightarrow \text{loop} + \rightarrow +
\]
\[
\text{tell}(e_1) \rightarrow \text{ask}(c_2) \rightarrow \ldots \rightarrow \text{ask}(c_{n-1}) \rightarrow \text{stop}
\]
\[
+ \rightarrow +
\]
\[
\text{ask}(\text{true}) \rightarrow \text{loop} \quad \text{ask}(\text{true}) \rightarrow \text{loop}
\]

As usual, \text{loop} is defined by the declaration \text{loop} := \text{ask}(\text{true}) \rightarrow \text{loop}.

Also in the previous proof the context which recognizes a sequence \( s \) is obtained by filling the gaps existing between what \( s \) tells and what \( s \) asks at the next step. Differently from the case of \text{tccp}, these gaps are expressed here by relative pseudo-complement rather than by weak relative pseudo-complement. It can be shown that the semantics \( R_d \) is correct also for \text{ccp}. It is not fully abstract, as shown by the agents \( A' \) and \( B' \) at the beginning of this subsection (the fully abstract semantics for \text{ccp} in \cite{12} could be obtained by imposing a further abstraction).

5. COMPARING \text{tccp}, \text{ccpm}, AND \text{ccp} VIA EMBEDDING

In this section we show that the semantic differences among \text{tccp}, \text{ccpm}, and \text{ccp} that we have discussed in the previous section correspond to different expressive powers for these three languages. We compare them by using the notion of embedding that we discussed in the Introduction. To this aim we use the following abstract notion of observables \( \mathcal{O}_s \) which essentially distinguishes finite computations from infinite ones.

**Definition 5.1.** Let \( A \) be a generic (either \text{tccp} or \text{ccpm}) agent. We define
\[
\mathcal{O}_s(A) = \{ \theta \mid \text{there exists } c \in \mathcal{C} \text{ s.t. } \langle A, c \rangle \rightarrow^* \langle B, d \rangle \rightarrow \text{ and } \theta = \alpha(A, c \cdot B, d) \}
\]
where \( \alpha \) is any total (abstraction) function from the set of sequences of configurations to a suitable set.

Since our separation results are given w.r.t. \( \mathcal{O}_s \), they hold for any concrete observables which can be seen as an instance of \( \mathcal{O}_s \) (e.g., input–output pairs, resting points, finite traces). In the following we denote by \( \mathcal{A}_t, \mathcal{A}_m, \) and \( \mathcal{A}_c \), the \text{tccp}, \text{ccpm}, and \text{ccp} agents, respectively, and we assume that the observables \( \mathcal{O}_s: \mathcal{A}_t \rightarrow \text{Obs}_t, \mathcal{O}_m: \mathcal{A}_m \rightarrow \text{Obs}_m, \text{ and } \mathcal{O}_c: \mathcal{A}_c \rightarrow \text{Obs}_c \) are all instances of \( \mathcal{O}_s \).

As mentioned in the Introduction, some restrictions on the decoder and the compiler are needed in order to use embedding as a tool for language comparison. In general, it is quite natural to require that the decoder cannot extract any information from an empty set and, conversely, that it cannot cancel completely all the
information which is present in a nonempty set describing a computation. Therefore, denoting by $\text{Obs}$ the observables of the target language, we require that

\begin{itemize}
  \item[(i)] $\forall O \in \text{Obs}, \mathcal{D}(O) = \emptyset$ iff $O = \emptyset$.
\end{itemize}

Furthermore, as discussed in [13], it is reasonable to require that the compiler $\mathcal{C}$ is a morphism w.r.t. the parallel and the choice operator. So, as in [13], we use also the following conditions:

\begin{itemize}
  \item[(ii)] $\mathcal{C}(A \parallel B) = \mathcal{C}(A) \parallel \mathcal{C}(B)$.
  \item[(iii)] $\mathcal{C}(\sum_{i=1}^{n} \text{ask}(c_i) \rightarrow A_i) = \sum_{i=1}^{n} \mathcal{C}(\text{ask}(c_i) \rightarrow A_i)$.
\end{itemize}

(however, our first separation result uses only (ii)). Clearly $\text{ccpm}$ can be embedded into $\text{tccp}$, the former being a sublanguage of the latter. As for the reverse, it is intuitively clear that the presence of the now then else construct augments the expressivity of the language, since it allows us to check also for the absence of information. In general, in order to prove that a source language cannot be embedded into a target one, we exhibit a semantic property of the abstract observables which holds for the target language and not for the source one. In our case we can simply observe that, due to the fact that the store grows monotonically in $\text{ccpm}$ computations, if a $\text{ccpm}$ agent $A \parallel B$ has a finite computation then both $A$ and $B$ have a finite computation. Thus we have the following proposition whose proof is immediate.

**Proposition 5.2.** Let $A$ be a $\text{ccpm}$ agent. If $\mathcal{O}_m(A) = \emptyset$ then $\mathcal{O}_m(A \parallel B) = \emptyset$ for any other $\text{ccpm}$ agent $B$.

On the other hand, the previous proposition does not hold for $\text{tccp}$. In fact, the presence of the now then else construct enforces a kind of nonmonotonic behavior: Adding more information to the store can inhibit some computations, since the corresponding else branches are discarded. Thus we have the following theorem.

**Theorem 5.3.** When considering any notion of observables which is an instance of $\mathcal{O}_m$, the language $\text{tccp}$ cannot be embedded into $\text{ccpm}$ while satisfying the conditions (i) and (ii).

**Proof.** The proof is by contradiction. Consider the $\text{tccp}$ agents $A$ and $B$ defined below

\begin{align*}
A &: \text{tell}(c) \quad \text{and} \quad B &: \text{now } c \text{ then loop else tell(true) } \rightarrow (\text{ask}(c) \rightarrow \text{stop}) \\
& \quad + \quad \text{ask(true) } \rightarrow \text{loop}
\end{align*}

(where loop is defined by the declaration $\text{loop} := \text{tell(true) } \rightarrow \text{loop}$) and assume that there exist their translations $\mathcal{C}(A)$ and $\mathcal{C}(B)$ in $\text{ccpm}$ and there exists a decoder $\mathcal{D}: \text{Obs}_m \rightarrow \text{Obs}$, which satisfy the requirements (i), (ii) in this section. From the definition of $\mathcal{C}$ and an inspection of the agent $B$ it follows that $\mathcal{C}(B) = \emptyset$. Condition (i) on the decoder implies that $\mathcal{O}_m(\mathcal{C}(B)) = \emptyset$, since $\mathcal{C}(B) = \emptyset$ (by $\mathcal{C}(\mathcal{E}(\mathcal{C}(B)))$). Moreover, condition (ii) on the compiler and Proposition 5.2 imply that
TABLE 3

The Transition Rule for Interleaving

<table>
<thead>
<tr>
<th>Ri</th>
<th>( \langle A, c \rangle \rightarrow \langle A', d \rangle )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( \langle A \parallel B, c \rangle \rightarrow \langle A' \parallel B, d \rangle )</td>
</tr>
<tr>
<td></td>
<td>( \langle B \parallel A, c \rangle \rightarrow \langle B \parallel A', d \rangle )</td>
</tr>
</tbody>
</table>

\( C_m(\mathcal{C}(A \parallel B)) = C_m(\mathcal{C}(A) \parallel \mathcal{C}(B)) = \emptyset. \) Now clearly the agent \( A \parallel B \) has a terminating computation for the input constraint \( \text{true} \). Therefore \( C_i(A \parallel B) \neq \emptyset \) and this, together with previous equalities and condition (i), gives a contradiction, thus concluding the proof.

Since Proposition 5.2 holds also when considering standard \( ccp \) agents we have also the following.

**Corollary 5.4.** When considering as observables instances of \( C_m \), the language \( tccp \) cannot be embedded into \( ccp \) while satisfying the conditions (i) and (ii) above.

We compare now \( ccpm \) and \( ccp \) by showing that the former language is strictly more expressive than the latter. The syntax of \( ccp \) is defined as in Definition 2.2 and its operational semantics is given by the transition system \( T' \), obtained from the one in Table 1 by replacing rules \( R7 \) and \( R8 \) for rule \( Ri \) contained in Table 3. Since in the following it will be clear from the context which transition system is being used, to simplify the notation we will denote by \( \rightarrow \) also the relation defined by \( T' \).

To embed \( ccp \) into \( ccpm \) it is sufficient to modify the \( ccpm \) guarded choice in such a way that its evaluation can be arbitrarily delayed.\(^{14}\) So, given a \( ccp \) agent \( A \), define (inductively) its \( ccpm \) translation \( \mathcal{F}_d(A) \) as the \( ccpm \) agent obtained from \( A \) by replacing each occurrence of a guarded choice agent \( A \# n_i=1 \text{ask}(c_i) \) for the agent \( p_A \) declared as

\[
p_A := \left( \sum_{i=1}^{n} \text{ask}(c_i) \rightarrow \mathcal{F}_d(A_i) \right) + \text{ask}(\text{true}) \rightarrow p_A.
\]

The translation of a set of declarations \( D \) into \( ccpm \), denoted by \( \mathcal{F}_d(D) \), is obtained in the obvious way by applying \( \mathcal{F}_d \) to all the agents appearing in \( D \) and by augmenting \( D \) with the declarations for all the agents \( p_A \) introduced by \( \mathcal{F}_d \). This translation allows one to simulate the interleaving execution model of standard \( ccp \) by using maximal parallelism, since the branch \( \text{ask}(\text{true}) \rightarrow p_A \) in the definition of \( p_A \) allows one to postpone the evaluation of the agent \( A \). The correspondence result is expressed by the following.

\(^{14}\) Alternatively, we could delay the evaluation of the tells.
Proposition 5.5. Let $A$ be a ccp agent. There exists a derivation $(A, c) \rightarrow^* (B, d)$ for a given set of declarations $D$ iff there exists a derivation $(\mathcal{F}_D(A), c) \rightarrow^* (\mathcal{B}', d)$ for the set of declarations $\mathcal{F}_D(D)$.

Proof. Immediate.

So ccp can be embedded into ccpm. To show that ccpm cannot be embedded into ccp we observe that, given an input constraint $c$, if the ccp agent $A \parallel B$ has a finite derivation then so does the agent $A \parallel (A+B)$. This property can be easily proved by noting that, due to interleaving, if the agent $B$ is selected in the derivation of $A \parallel B$ then it can be selected also in the derivation of $A \parallel (A+B)$, while if $B$ is not selected then one can always select $A$ in $A+B$ and obtain a finite derivation for it. Thus we have the following.

Proposition 5.6. Let $A$ and $B$ be ccp agents. If $\mathcal{E}(A \parallel B) \neq \emptyset$ then $\mathcal{E}(A \parallel (A+B)) \neq \emptyset$.

On the other hand, previous property does not hold for ccpm. In fact, even though $A \parallel B$ has a successful derivation, it can happen that (the guard in) $B$ is enabled by the constraints produced by $A$. In this case, due to maximal parallelism, the computation for $A \parallel (A+B)$ can be forced to choose $A$ in the choice $(A+B)$ and therefore to enter a wrong (i.e., nonterminating) branch. Thus we have the following theorem where, for technical reasons, the abstract observables $\mathcal{E}_{a}$ are assumed to be obtained from $\mathcal{E}_{a}$ by considering only the input constraint $true$ (rather than a generic $c$) in its definition.

Theorem 5.7. When considering as observables instances of $\mathcal{E}_{a}$, the language ccpm cannot be embedded into ccp while preserving conditions (i), (ii), and (iii).

Proof. Similar to the proof of Theorem 5.3 by considering the ccpm agents

$$A: \text{ask}(true) \rightarrow \text{tell}(c) \rightarrow \text{tell}(true) \rightarrow \text{tell}(true) \rightarrow (\text{ask}(d) \rightarrow \text{stop})$$

$$+ \quad \text{ask}(true) \rightarrow \text{loop}$$

and

$$B: \text{ask}(c) \rightarrow \text{tell}(d)$$

by noting that $\mathcal{E}_{a}(A \parallel B) \neq \emptyset$ while $\mathcal{E}_{a}(A \parallel (A+B)) = \emptyset$ and by using Proposition 5.6.

It is worth noting that deterministic ccpm is exactly deterministic ccp, as shown by the following proposition whose proof is immediate and therefore omitted. Recall that deterministic ccp($m$) is obtained by imposing $n=1$ in the choice construct $\sum_{i=1}^{n} \text{ask}(c_i) \rightarrow \text{A}_i$.

Proposition 5.8. Let $D.A$ be a deterministic ccp($m$) process. There exists a derivation $(A, c) \rightarrow^* (B, d)$ by using the transition system in Table 1 iff there exists a derivation $(A, c) \rightarrow^* (B, d)$ by using the transition system $T'$ (obtained from the one in Table 1 by replacing rules $R7$ and $R8$ for rule $R1$).
6. RELATED WORK

As mentioned in the Introduction there are three main differences between our approach and that one pursued in [40] and [41].

First, the computational model of both the languages tcc [40] and default tcc [41] is inspired by that one of synchronous languages: Computation proceeds in “bursts of activity” and in each phase a deterministic ccp (or default ccp) process is executed to produce a response to an input produced by the environment. This process accumulates monotonically information in the store, according to the standard ccp computational model, until it reaches a resting point, i.e., a terminal state in which no more information can be generated. When the resting point is reached, the absence of events can be checked and it can trigger actions in the next time interval. Therefore, 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 introduce directly 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. As previously discussed, some syntactic restrictions are needed also in our case to obtain bounded response time, that is, to be able to statically determine the maximal length of each time-unit. However, 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.

A second difference relies in the transfer of information across time boundaries. In tcc and default tcc the programmer has to transfer explicitly the (positive) information from a time instant to the next one by using special primitives which 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. Only a limited form of recursion is allowed across time boundaries, since this ensures that tcc and default tcc programs can eventually be compiled to finite state automata [40, 41].

On the contrary, no explicit transfer is needed in tccp, since the computational model is based on the monotonic evolution of the store which is usual in ccp.

It is worth noting that this difference affects the expressive power of the language. In fact, default tcc has been directly designed as a non-Turing-powerful language (default tcc programs can be compiled to finite state automata [41]). Furthermore, assuming that recursive procedures (across time boundaries) are without parameters also tcc programs can be compiled to finite state automata [40]. It is worth noting that this is the case also when procedures have formal parameters, provided that these parameters are distinct variables (as it is in standard ccp): In fact, if the procedure p is defined by p(x) :- A, then procedure call p(y) can be
replaced by $\exists x(p \parallel tell(d_{xy}))$, provided that the definition of $p$ is replaced by $p := A$. Since deterministic $tccp$ is a Turing-powerful language, this shows that also when restricting to the deterministic fragment $tccp$ cannot be embedded into $default tcc$ nor in $tcc$ (assuming in the last case that only distinct variables appear as procedure parameters).

A third relevant difference is in the fact that $tcc$ and $default tcc$ are deterministic languages while our language allows for nondeterminism. Indeed, the simplicity of both the $tcc$ and the $default tcc$ semantic domains is due to the restriction to deterministic programs: An extension to nondeterminism would require complicated semantic structures based on sets of sequences. On the other hand, our nondeterministic timed extension of $ccp$ allows us to define a reasonably simple denotational (fully abstract) semantics based on sequences.

To summarize, even though our proposal shares with $tcc$ and $default tcc$ many similarities, its original motivation and possible applications are different. $Default tcc$ (which can be considered as the successor of $tcc$) has been inspired by the ESTEREL-like languages and therefore it is mainly a language for programming real-time kernels. As such, it does not need to be Turing powerful, does not need to include nondeterminism, and has to allow for strong preemption [5]. Strong preemption is important for some applications (see [4]); however its increased expressive power comes with a price, since in general paradoxes can arise. Semantically, these problems are treated in $default tcc$ by using assumptions about the future evolution of the system.

On the other hand, our (Turing-powerful) language provides a formalism for specifying large concurrent timed systems, in the spirit of the timed process algebras. In this general context of specification formalisms weak preemption often suffices while nondeterminism is essential, as witnessed by the fact that all the existing timed process algebras include a nondeterministic choice operator. Furthermore, since the style of programming for $tccp$ is essentially the same as the one of $ccp$, $tccp$ provides a higher level language w.r.t. the formalisms based on timed process algebras, thus simplifying the specification and prototyping of large systems.

7. CONCLUSIONS

We have defined the language $tccp$, a timed extension of $ccp$, and we have defined a fully abstract model for it and for its sublanguage $ccpm$. We have also studied the expressive power of these languages.

Due to the presence of maximal parallelism, the semantics we have defined and the proofs of full abstraction are completely different from the ones existing for standard $ccp$ [12]. Fully abstract semantics for timed $ccp$ languages are given also in [40, 41]. However, the languages considered in these papers are different from $tccp$ since they do not assume maximal parallelism and they restrict to deterministic programs. For this reason the results in [40, 41] are substantially different from ours.

The fully abstract semantics of $tccp$ and $ccpm$ are more concrete than the one for $ccp$; i.e., they need less identifications. This reflects the fact that $tccp$ and $ccpm$ are more expressive than $ccp$. 


More precisely, we have shown that these three languages have a strictly decreasing expressive power, since \( tccp \) cannot be embedded in \( ccpm \) which, in turn, cannot be embedded into \( ccp \). The first result is due to the presence of the now then else construct (in \( tccp \)) which enforces a kind of nonmonotonic behavior since it allows us to check for absence of information. For example, assuming a finite set of function symbols allows us to check whether a variable is not instantiated, similarly to the \( \text{Var}(x) \) built-in of Prolog. The fact that \( ccpm \) is more expressive than \( ccp \) is due to the presence of maximal parallelism which augments control over the (global) choice. In fact, in the presence of maximal parallelism one can force the computation to discard some (nonenabled) 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. In other words, the languages \( ccpm \) and \( tccp \) are sensitive to delays in adding constraints to the store, whereas this is not the case for \( ccp \).

We are currently following two lines of research. We are investigating the extension of previous results to consider also confluent \( ccp \) languages [17, 31] and infinite computations. Preliminary results show that also in this case \( tccp \) is more expressive than \( ccp \) which, in turn, is more expressive than confluent \( ccp \). In this case the separation results show that fair merge [34] can be expressed in \( tccp \) and not in \( ccp \), while angelic merge [34] can be expressed in \( ccp \) and not in confluent (in the sense of [17]) \( ccp \).

A second line of research concerns the definition of tools for the verification and the analysis of \( tccp \) programs, following the guidelines of [9] and [16]. In particular, we are now studying an extension based on temporal logic [35] of the proof system defined in [9] to reason about the correctness of \( tccp \) programs.

**APPENDIX**

**Theorem 4.8 (Compositionality of \( R_x \)).** Let \( A, B, \) and \( A_i \) be generic \( tccp \) agents. Then the following equalities hold.

1. \( R_x(\sum_{i=1}^{n} \text{ask}(c_i) \rightarrow A_i) = \alpha(\sum_{i=1}^{n} c_i, R_x(A_i)) \),
2. \( R_x(\text{now } c \text{ then } A \text{ else } B) = \alpha(\text{now } c, R_x(A), R_x(B)) \),
3. \( R_x(A \land B) = \alpha(R_x(A) \parallel R_x(B)) \),
4. \( R_x(\exists \text{x } A) = \alpha(\exists \text{x } R_x(A)) \).

**Proof.** We prove the cases of the now-construct, the parallel composition, and the hiding operator (the remaining one is treated similarly).

1. Since \( R(\text{now } c \text{ then } A \text{ else } B) = \text{now } c, R(A), R(B) \) we need to show that \( \alpha(\text{now } c, R(A), R(B)) = \alpha(\text{now } c, R_x(A), R_x(B)) \).

   The proof of the inclusion \( \alpha(\text{now } c, R(A), R(B)) \subseteq \alpha(\text{now } c, R_x(A), R_x(B)) \) follows from the fact that \( \alpha \) is a closure operator and from the monotonicity of \( \text{now} \).

   To prove the other inclusion it suffices to show that \( \text{now } c, R_x(A), R_x(B) \subseteq \alpha(\text{now } c, R(A), R(B)) \) and then to apply the fact that \( \alpha \) is a closure operator.
Consider a sequence \( s = \langle c', d' \rangle \cdot s_1 \in \text{not}(c, R_1(A), R_1(B)) \). Then, by definition of \( \text{not} \), either \( c' \not\sqsubset c \) and \( s \in R_1(A) \) or \( c' \not\sqsupset c \) and \( s \in R_1(B) \) holds. We consider only the case \( c' \not\sqsubset c \) (the case \( c' \not\sqsupset c \) is analogous). Since \( \pi \) is defined point-wise, there exists \( s' \in R(A) \) such that \( s \in \pi(s') \). By definition of \( \pi \), \( \text{first}(s) = \text{first}(s') = c' \) and therefore, by definition of \( \text{not} \), \( s' \in \text{not}(c, R(A), R(B)) \). Finally, since \( \pi \) is defined point-wise, \( s \in \pi(s') \subseteq \pi(\text{not}(c, R(A), R(B))) \) which concludes the proof of this case.

2. Analogously to the previous case it is sufficient to prove \( \pi(R_1(A) \upharpoonright R_1(B)) = \pi(R(A) \upharpoonright R(B)) \). We consider the two inclusions separately.

\((\subseteq)\) The proof is straightforward by using the fact that \( \pi \) is a closure operator and the monotonicity of \( \upharpoonright \).

\(\subseteq\) We prove that \( R_1(A) \upharpoonright S_2 \subseteq \pi(R(A) \upharpoonright S_2) \), where either \( S_2 = R(B) \) or \( S_2 = R_1(B) \). Then the thesis follows from the fact that \( \pi \) is a closure operator and by symmetry of \( \upharpoonright \).

Consider a sequence \( s \in R_1(A) \upharpoonright S_2 \). If \( s \in R(A) \upharpoonright S_2 \) then the thesis follows by extensivity of \( \pi \). Otherwise there exist \( s_1 \in R_1(A) \setminus R(A) \) and \( s_2 \in S_2 \) such that \( s = s_1 \upharpoonright s_2 \). By definition of \( \pi \), there exists \( s' \in R(A) \) such that \( s_1 \in \pi(s') \); namely, \( s_1 \) is obtained from \( s' \) by applying some \( \preceq \)- and \( \succeq \)-reduction steps. Since \( \preceq \)-reduction does not modify any repetition of the last pair of a sequence, we can assume without loss of generality that the \( \preceq \)-reduction steps are performed before the \( \succeq \)-reduction steps.

The proof is then by induction on the number \( k \) of applications of \( \preceq \)-reduction steps.

\((k = 0)\) In this case we can perform only \( \succeq \)-reduction steps; hence, \( s_1 \succeq s'_1 \) holds. From the definition of \( R \) it follows that, for each \text{tcp} agent \( C \) and for each sequence \( \hat{s} \in R(C) \),

\[
\text{if } \hat{s} \succeq \hat{s}' \text{ and } \text{length}(\hat{s}) \geq \text{length}(\hat{s}') \text{ then } \hat{s} \in R(C)
\]

holds. Moreover the same property holds if we substitute \( R(C) \) for \( R_1(C) \).

Since \( s_1 \notin R(A) \), \( s'_1 \in R(A) \), and \( s_1 \succeq s'_1 \), from (1) it follows that \( \text{length}(s_1) < \text{length}(s'_1) \).

The definition of \( \upharpoonright \) implies that \( \text{length}(s_1) = \text{length}(s_2) \). Then, since \( s_2 \in S_2 \), from (1) it follows that there exists \( s'_2 \in S_2 \) such that \( s_2 \succeq s'_2 \) and \( \text{length}(s'_2) = \text{length}(s'_1) \).

Therefore, by definition of \( \succeq \) and of \( \upharpoonright \), there exists \( s = s'_1 \upharpoonright s'_2 \in R(A) \upharpoonright S_2 \) such that \( s \succeq s' \). By applying a \( \succeq \)-reduction step to \( s' \), we have that \( s \in R_1(A) \upharpoonright S_2 \) which completes the base case.

\((k > 0)\) Consider \( s_1 = \langle c_1, d_1 \rangle \cdots \langle c_n, c_n \rangle \in R_1(A) \setminus R(A) \) and \( s_2 = \langle c_1, f_1 \rangle \cdots \langle c_n, c_n \rangle \in S_2 \) such that \( s = s_1 \upharpoonright s_2 = \langle c_1, d_1 \uplus f_1 \rangle \cdots \langle c_n, c_n \rangle \) and \( s_1 \) is obtained from \( s'_1 \in R(A) \) by applying \( k \preceq \)-reduction steps. From the definition of \( \preceq \)-reduction step it follows that there exists a sequence \( s'_1 \in R(A) \) obtained from \( s'_1 \) by applying \( k - 1 \preceq \)-reduction steps and there exists \( i \in \{1, n - 1\} \), such that

\[
s'_1 = \langle c_1, d_1 \rangle \cdots \langle c_i, d_i \rangle \langle c_{i+1}, d_{i+1} \rangle \cdots \langle c_n, c_n \rangle \quad \text{and} \quad c_{i+1} \uplus d'_i \preceq c_{i+1} \uplus d_i.
\]
By definition of $\overline{1}$, $s_1^2 \overline{1} s_2 = \langle c_1, d_1 \sqcup f_1 \rangle \cdots \langle c_n, d_n \sqcup f_n \rangle \overline{1} s_2$, and by inductive hypothesis $s_1^2 \overline{1} s_2 \in R_2(A \overline{1} S_2)$. Then in order to prove the thesis we have only to show that $s_1 \overline{1} s_2$ is less connected than $s_1^2 \overline{1} s_2$, i.e., that $c_{i+1} \setminus (d_i \sqcup f_i) \leq c_{i+1} \setminus (d_i \sqcup f_i)$. Let us define $w = c_{i+1} \setminus (d_i \sqcup f_i)$. By definition of weak relative pseudo-complement, $d_i \sqcup f_i \sqcup w = c_{i+1}$ holds; therefore $c_{i+1} \setminus d_i \leq f_i \sqcup w$. By (2) it follows that $c_{i+1} \setminus d_i \leq f_i \sqcup w$ and therefore, by definition of weak relative pseudo-complement, $d_i \sqcup f_i \sqcup w = c_{i+1}$. Thus $c_{i+1} \setminus (d_i \sqcup f_i) \leq w = c_{i+1} \setminus (d_i \sqcup f_i)$ which completes the proof for this case.

3. By definition of $\preceq$ and by compositionality of $R$, it is sufficient to prove that $\alpha(\exists x R(A)) = \alpha(\exists x R(A))$. The inclusion $\alpha(\exists x R(A)) \subseteq \alpha(\exists x R(A))$ is immediate, since $x$ is a closure operator and $\exists x$ is monotonic. In order to prove the other inclusion we show that $\exists x R(A) \subseteq \alpha(\exists x R(A))$. The thesis then follows since $x$ is a closure operator.

Consider a sequence $s = \langle e_1, f_1 \rangle \langle e_2, f_2 \rangle \cdots \langle e_n, f_n \rangle \in \exists x R(A)$. If $s \in \exists x R(A)$ then $s \in \exists x R(A)$, since $x$ is extensive. Assume now that $s \notin \exists x R(A)$. By definition of $\exists x$, $s \notin \forall x$ and there exists $s_1 \in R(A) \setminus R(A)$ such that $\exists x, s = \exists x s_1$. $s_1$ is x-connected and $s$ is x-invariant. Moreover, by definition of $\exists x$, there exists $s'_1 \in R(A)$, such that $s_1$ is obtained from $s'_1$ by applying some $\equiv$- and $\simeq$-reduction steps. Analogously to the previous case the proof is now by induction on the number $k$ of applications of $\equiv$-reductions.

(k = 0) In this case we perform only $\simeq$-reduction steps; therefore, $s_1 \simeq s'_1$. By definition of $\simeq$, it is easy to check that $s_1 \in \forall x$ and is x-connected if and only if $s'_1 \in \forall x$ and is x-connected. Moreover, since $s'_1 \in R(A)$ the definition of $\exists x$ implies that there exists $s' \in \exists x R(A)$ such that $s \simeq s'$. By applying a $\simeq$-reduction step to $s'$ we have that $s \in \exists x R(A)$ and thus the thesis follows.

(k > 0) Assume that

$$s = \langle e_1, f_1 \rangle \langle e_2, f_2 \rangle \cdots \langle e_n, f_n \rangle \in \exists x R(A) \setminus \exists x R(A)$$

and

$$s_1 = \langle c_1, d_1 \rangle \langle c_2, d_2 \rangle \cdots \langle c_n, c_n \rangle \in R(A) \setminus R(A),$$

where $s$ is x-connected, $s$ is x-invariant, and $\exists x s_1 = \exists x s$. Moreover, assume that $s_1$ is obtained from $s'_1 \in R(A)$ by applying $k$ $\equiv$-reduction steps. By definition of a $\equiv$-reduction step, there exists a sequence $s'_1 \in R(A)$ obtained from $s'_1$ by applying $k - 1$ $\equiv$-reduction steps and there exists $i \in [1, n - 1]$, such that

$$s'_1 = \langle c_1, d_1 \rangle \cdots \langle c_i, d'_i \rangle \langle c_{i+1}, d_{i+1} \rangle \cdots \langle c_n, c_n \rangle \quad \text{and} \quad c_{i+1} \setminus d'_i \leq c_{i+1} \setminus d_i.$$

(3)

In order to prove the thesis we now prove the following four points:

(a) We first show that $s'_1$ is x-connected. Since $s_1$ is x-connected it is sufficient to prove that $\exists x c_{i+1} \sqcup d'_i = c_{i+1}$. Since $s_1$ is x-connected $\exists x c_{i+1} \sqcup d_i = c_{i+1}$ holds and hence, by definition of weak relative pseudo-complement, $c_{i+1} \setminus d_i \not\leq \exists x c_{i+1}$. From (3) it follows that $c_{i+1} \setminus d'_i \not\leq \exists x c_{i+1}$ and therefore, by definition of weak relative pseudo-complement, $\exists x c_{i+1} \sqcup d'_i = c_{i+1}$ holds.
(b) Next we show that \( s^* \in \mathcal{A} \), where the sequence \( s^* \) is defined as follows:

\[
\begin{align*}
  s^* &= \langle e_1, f_1 \rangle \cdots \langle e_i, f'_i \rangle \langle e_{i+1}, f_{i+1} \rangle \cdots \langle e_n, e_n \rangle \quad \text{and} \quad f'_i &= e_i \sqcup \exists_s d'_i.
\end{align*}
\]

Since by hypothesis \( s \in \mathcal{A} \), we have only to prove that \( e_i \leq f'_i \leq e_{i+1} \). The first inequality follows by construction, since \( f'_i = e_i \sqcup \exists_s d'_i \). The second one holds since \( s^*_i \in \mathcal{A}, s \in \mathcal{A} \) and therefore \( d'_i \leq e_{i+1} \) and \( e_i \leq e_{i+1} \). Moreover, since \( \exists_s s_1 = \exists_s s \), we have that \( \exists_s e_{i+1} = \exists_s e_{i+1} \). Then by the axioms for \( \exists_s \), it follows that \( f'_i = e_i \sqcup \exists_s d'_i \leq e_i \sqcup \exists_s e_{i+1} \leq e_{i+1} \).

(c) Now we prove that \( \exists_s s^*_i = \exists_s s^* \). Since \( \exists_s s_1 = \exists_s s \), the definition of \( s^* \) implies that we have only to prove that \( \exists_s d'_i = \exists_s f'_i \). We have the following equalities

\[
\begin{align*}
  \exists_s f'_i &= (\text{by definition}) \\
  \exists_s (e_i \sqcup \exists_s d'_i) &= (\text{by the axioms for } \exists_s) \\
  \exists_s e_i \sqcup \exists_s d'_i &= (\text{since } \exists_s s_1 = \exists_s s \text{ implies } \exists_s e_i = \exists_s c_i) \\
  \exists_s e_i \sqcup \exists_s d'_i &= (\text{by monotonicity of } \exists_s, \text{ since } c_i \leq d'_i) \\
  \exists_s d'_i.
\end{align*}
\]

(d) Finally we show that \( s^* \) is \( x \)-invariant. Since \( s \) is \( x \)-invariant, from the construction of \( s^* \) it is sufficient to show that \( f'_i = \exists_s f'_i \sqcup e_i \) and this follows immediately from the previous equalities.

Since \( s^*_i \in R_x(A) \) from the previous four points and the definition of \( \overline{3} \) it follows that \( s^* \in \overline{3} R_x(A) \). Therefore, by inductive hypothesis, \( s^* \in \mathcal{A} (\overline{3} R_x(A)) \) and to complete the proof we have to show that \( s \) is less connected than \( s^* \), i.e., that

\[
e_{i+1} \setminus f'_i \leq e_{i+1} \setminus f_i.
\]

Let \( w = e_{i+1} \setminus f_i \). Since \( s \) is \( x \)-invariant and \( \exists_s s_1 = \exists_s s \) it follows that \( f_i = e_i \sqcup \exists_s f_i = e_i \sqcup \exists_s d_i \). By definition of weak relative pseudo-complement,

\[
e_i \sqcup \exists_s d_i \sqcup w = e_{i+1}
\]

holds. Therefore, from the axioms for \( \exists_s \) and the equality \( \exists_s s_1 = \exists_s s \) it follows that

\[
\exists_s d_i \sqcup \exists_s (e_i \sqcup w) = \exists_s e_{i+1} = \exists_s c_{i+1}.
\]

Since \( s_1 \) is \( x \)-connected, \( \exists_s e_{i+1} \sqcup d_i = e_{i+1} \) holds; therefore the axioms for \( \exists_s \) together with (5) imply that

\[
d_i \sqcup \exists_s (e_i \sqcup w) = d_i \sqcup \exists_s c_{i+1} = c_{i+1}.
\]
Moreover, from the definition of weak relative pseudo-complement and (3) it follows that \( c_{i+1} \cup d_i \leq c_{i+1} \cup d_i \leq \exists_x(e_i \cup w) \) and therefore \( d_i' \cup \exists_x(e_i \cup w) = c_{i+1} \geq d_i \). This, together with the axioms for \( \exists_x \) implies that

\[
\exists_x d_i' \cup \exists_x(e_i \cup w) \geq \exists_x d_i .
\] (7)

Now, we have the following inequalities

\[
\begin{align*}
e_{i+1} & \geq (\text{since } f_i' \leq e_{i+1} \text{ and } w \leq e_{i+1}) \\
f_i' \cup w & = (\text{by definition of } f_i') \\
\exists_x d_i' \cup e_i \cup w & = (\text{by the axioms for } \exists_x) \\
\exists_x d_i' \cup e_i \cup w \cup \exists_x(e_i \cup w) & \geq (\text{by (7)}) \\
\exists_x d_i \cup e_i \cup w & = (\text{by (4)}) \\
e_{i+1} & 
\end{align*}
\]

which imply that \( e_{i+1} = f_i' \cup w \). Therefore \( e_{i+1} \setminus f_i' \leq w = e_{i+1} \setminus f_i \) which completes the proof.

ACKNOWLEDGMENTS

The authors thank Luca Aceto for interesting discussions on the subject and the anonymous referees for their precise and helpful comments.

Received March 3, 1999

REFERENCES


42. Scott, D. (1982), Domains for denotational semantics, in “Proc. of ICALP.”