URL: http://www.elsevier.nl/locate/entcs/volume27.html 15 pages

On the expressiveness of Timed Concurrent Constraint Programming ⋆

Simone Tini

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

Abstract

We prove that both the synchronous data flow language Lustre restricted to finite value types and the synchronous state oriented language Argos are embedded in the synchronous paradigm Timed Concurrent Constraint (`tcc`). In fact, for each of the two languages we provide a `tcc` language encoding it compositionally w.r.to the structure of programs. Moreover, we prove that the "strong abortion" mechanism of the synchronous imperative language Esterel can be encoded in `tcc`.

1 Introduction

Synchronous languages [1,9] have been proposed to program *reactive systems* [11], namely systems which maintain an ongoing interaction with their environment at a rate controlled by this. The life of a reactive system is divided into instants, namely moments in which it is stimulated by the environment and it must react. Every reaction of the system produces a response which is expected by the environment within a bounded time, at least within the next instant, so that reactions do not overlap. Synchronous formalisms are based on the *synchronous hypothesis* [3] which states that reactions of a system are instantaneous. This hypothesis simplifies reasoning about reactive systems, since it presents at least two advantages. The first is that the construct of parallel composition does not give rise to nondeterminism, namely the various components of a system act synchronously, and their actions cannot arbitrarily interleave. The second is that time is treated as any other external event, namely no special construct to deal with physical time is needed. In fact, the notion of physical time is replaced by a notion of ordering among events.

⋆ Research partially supported by ESPRIT Working Group "Concurrent Constraint Programming for Time Critical Applications" (COTIC), Project Number 23677.

Since the eighties, two classes of synchronous formalisms have been developed: the class of state oriented synchronous languages (*Esterel* [3], *Statecharts* [10], *Argos* [13]), which are tailored for programming reactive systems where the control handling aspects are predominant, and the class of declarative data flow synchronous languages (*Lustre* [5], *Signal* [7]), which are tailored for programming reactive systems where the data processing aspects are predominant.

More recently, the paradigms *Timed Concurrent Constraint* (`tcc`) and *Timed Default Concurrent Constraint* (`tdcc`) have been introduced in [16,17] to integrate synchronous programming with the paradigms Concurrent Constraint (`cc`) [15] and Default Concurrent Constraint (`dcc`) [8], respectively. `cc` is based on the idea that systems of agents execute asynchronously and interact by posting and checking constraints in a shared pool of information (*store*). `dcc` extends `cc` since agents may check the absence of constraints in the store. `tcc` and `tdcc` extend `cc` and `dcc`, respectively, with constructs to sequentialize, w.r.to sequences of instants, agent interaction with the store. Moreover, according to the synchronous hypothesis, agents are able to post and check constraints instantaneously.

In this paper we investigate the expressiveness of `tcc` and `tdcc` by relating their expressive power with the expressive power of other synchronous languages. According to [18], a language L is more expressive than a language L' (L *embeds* L') if there exists an encoding of L' in L which preserves the meaning of programs. This means that there exists a map from the semantic domain of L' to the semantic domain of L such that the meaning of each program P' of L' is mapped to the meaning of the program of L encoding P'. As argued in [4], an interesting property of this encoding is the compositionality w.r.to the syntax of L'.

We prove that the state oriented language Argos is embedded in `tcc`. In fact, we provide an encoding of Argos in a `tcc` language. This encoding is compositional w.r.to the syntax of Argos and is linear w.r.to the size of Argos programs. This result suggests that `tcc` is well suited to efficiently encode the *weak abortion* mechanism [2] which is offered by Argos and by other state oriented synchronous languages. This mechanism permits to abort a process and to allow it to perform its last reaction in the instant in which it is aborted. The *strong abortion* mechanism [2] differs from the previous one because the aborted process is not allowed to react in the instant in which it is aborted. We prove that also this mechanism can be encoded in `tcc`. This result is interesting because the strong abortion mechanism is offered by some synchronous languages, such as Esterel. Note that in [17] it is guessed that this mechanism can be encoded in `tdcc` but not in `tcc`.

Then we consider the subset of Lustre restricted to finite value types. We denote this subset of Lustre by Lustre$_{\mathcal{F}}$ and we prove that it is embedded in `tcc`. In fact, we provide an encoding of Lustre$_{\mathcal{F}}$ in a `tcc` language. This encoding is compositional w.r.to the syntax of Lustre$_{\mathcal{F}}$ and is linear w.r.to

the size of Lustre$_{\mathcal{F}}$ programs and w.r.to the cardinality of values of types. However, we conjecture that one cannot encode the full language Lustre in any `tdcc` language. These results depend on the fact that Lustre offers a mechanism to memorize information which is not available in `tdcc`. In order to simulate this mechanism in `tdcc`, one must transfer explicitly information from one instant to the subsequent one. This can be done only if types have finite values and, in this case, it is efficient only if types have few values. Note that the subset of Lustre which can be efficiently encoded in `tcc` is interesting. In fact, it is well known that the subset of Lustre restricted to boolean types is sufficient to encode any Finite State Machine.

2 An introduction to `tcc` and `tdcc`

In this section we introduce informally the paradigms `tcc` and `tdcc`. For a formal and complete treatment, we refer to [16,17].

`tdcc` is a family of languages parametric w.r.to a *constraint system* which determines the information that can be treated by agents. A constraint system $\mathcal{C}$ is a tuple $\langle D_{\mathcal{C}}, \vdash_{\mathcal{C}}, Var_{\mathcal{C}} \rangle$, where $D_{\mathcal{C}}$ is a set of primitive constraints (*tokens*), $\vdash_{\mathcal{C}} \subseteq Fin(D_{\mathcal{C}}) \times D_{\mathcal{C}}$ is an *entailment relation*, and $Var_{\mathcal{C}}$ is a set of variables. Tokens express possibly partial information on the value of the variables in $Var_{\mathcal{C}}$. A set of tokens $\{c_1, \ldots, c_n\}$ entails a token c if the information given by c follows from the information given by $c_1, \ldots, c_n$. We consider the following syntax for `tdcc`:

$P ::= \texttt{In} : seqvar \ ; \ D \ ; \ A \ ; \ \texttt{Out} : seqvar$
$seqvar ::= \epsilon \mid X, seqvar$
$A ::= \texttt{tell}\ a \mid \texttt{if}\ a\ \texttt{then}\ A \mid \texttt{if}\ a\ \texttt{else}\ A \mid (A, A) \mid \texttt{next}\ A \mid p$
$D ::= \epsilon \mid p = A \ , \ D$

where a, X, $seqvar$, p, A, D, P are metavariables for tokens, variables, sequences of variables, procedures, agents, sequences of declarations and programs, respectively. We obtain a `tdcc` language by choosing a suitable constraint system. The syntax of `tcc` is obtained by removing the construct `if _ else _` . Here we do not consider the construct of `tcc` to define local variables. This can be used to encode both the analogous construct of Lustre and the construct of Argos to define local signals. Also these constructs are not considered here.

Each `tcc` program P interacts with the environment by exchanging information at each instant. In fact, at each instant the environment stimulates P by posting in the store a set of tokens expressing information on input variables, and P reacts by posting in the store a set of tokens expressing information on output variables. All information is removed from the store between any instant and the subsequent one. The operational semantics of P is given in terms of sequences of pairs of sets of tokens $(\Delta_1, \Delta'_1), \ldots, (\Delta_n, \Delta'_n), \ldots$, where Δ_n is the set of tokens prompted by the environment at the n^{th} instant and

Δ'_n is the set of tokens posted in the store by P at the same instant, for every $n \geq 1$.

According to the synchronous hypothesis, constructs `tell`, `if _ then _`, `if _ else _` and $(\_, \_)$ do not take time, while construct `next` takes exactly one unit of time.

The agent `tell` a posts the token a in the store and then terminates. Both agents `if` a `then` A and `if` a `else` A query the store about the validity of token a. If the store entails a then the first agent behaves as A else it terminates. Symmetrically, if the store entails a then the second agent terminates else it behaves as A. The agent (A_1, A_2) is the synchronous parallel composition of A_1 and A_2. Tokens posted in the store by A_1 are available to A_2 at the same instant, and conversely. The agent `next` A will behave as A at the next instant.

In order to specify cyclic behaviors, an agent can be a (recursive) procedure. Two syntactic restrictions are required. The first is that recursion is guarded, namely that if a call of procedure p is in the body of p then it must be in the body of a `next`. The second is that procedures have no parameters, so that at runtime there is a finite number of procedure calls.

As in [16,17], we write a for `tell` a, we denote by `always` A the agent p, with $p = (A, \texttt{next}\ p)$, which starts A at each instant. Moreover, we denote by $\{A_i \mid 1 \leq i \leq n\}$ the agent $(\ldots(A_1, A_2), A_3), \ldots, A_n)$. Finally, we denote by `if` $a \vee b$ `then` A the agent (`if` a `then` A, `if` b `then` A) and by `if` $a \wedge b$ `then` A the agent `if` a `then if` b `then` A.

3 Embedding Argos and strong abortion in tcc

In this section we show that Argos is embedded in `tcc`. We begin by introducing informally Argos. Then we define the `tcc` language $\texttt{tcc}_{\text{Argos}}$ and we provide a compositional encoding of Argos in $\texttt{tcc}_{\text{Argos}}$. Finally, we prove that the strong abortion mechanism can be encoded in $\texttt{tcc}_{\text{Argos}}$.

3.1 An introduction to Argos.

An Argos program P has an interface w.r.to the environment, consisting of a set of input signals I_P and a set of output signals O_P, and a body, consisting of an Argos agent. Agents may be Mealy machines, Mealy machines refined by agents, and parallel composition of agents. We assume a set of signals $\mathcal{S}$ with $I_P, O_P \subseteq \mathcal{S}$ for every program P. Each Argos agent has a graphical representation. We choose to present a process-like syntax:

$P ::= \texttt{In} : seqsig\ ;\ A\ ;\ \texttt{Out} : seqsig$
$A ::= M \mid M \rhd \{A_1, \ldots, A_n\} \mid A \parallel A$
$seqsig ::= \epsilon \mid s, seqsig$

where s, *seqsig*, A, M, P are metavariables for signals, sequences of signals, agents, Mealy machines and programs, respectively.

A *Mealy machine* M is a tuple $(S_M, s_{0_M}, I_M, O_M, T_M)$, with $S_M = \{s_{0_M}, \ldots, s_{n_M}\}$ a set of states, $s_{0_M} \in S_M$ the initial state, I_M a set of input signals, O_M a set of output signals and $T_M \subseteq S_M \times \mathcal{B}(I_M) \times 2^{O_M} \times S_M$ a set of transitions. Here, $\mathcal{B}(I_M)$ denotes the set of boolean expressions with variables in I_M. The set $\mathcal{B}(I_M)$ is ranged over by b.

A Mealy machine M starts running in its initial state s_{0_M}. Whenever M is in state s_{h_M} and the environment prompts a set of signals $I \subseteq I_M$, a transition (s, b, O, s') is *triggered* if and only if $s = s_{h_M}$ and expression b evaluates to true, provided that each variable in b evaluates to true if it appears in I, while it evaluates to false otherwise. If (s_{h_M}, b, O, s_{k_M}) is triggered then it *fires*, so that the output signals in O are broadcast to the environment, M leaves state s_{h_M} and it reaches the state s_{k_M}.

Whenever M is in state s_{h_M} and no transition is triggered then M does not leave s_{h_M}. Since only deterministic programs are admitted, it is required that $b_1 \wedge b_2 \equiv \mathit{false}$ for boolean expressions b_1 and b_2 such that $(s_{h_M}, b_1, O_1, s_{k_{1_M}}), (s_{h_M}, b_2, O_2, s_{k_{2_M}}) \in T_M$.

The agent $A = M \rhd \{A_0, \ldots, A_n\}$ denotes the Mealy machine M *refined* by the agents $A_0, \ldots, A_n$. A global state of A is formed by a state $s_{h_M} \in S_M$ and by a state of A_h, $0 \leq h \leq n$. At the first instant, both M and A_0 start running. If a state s_{h_M} is reached by M at the i^{th} instant and is left by M at the j^{th} instant, then A_h starts running from its initial state at the $(i+1)^{th}$ instant and performs its last reaction at the j^{th} instant. In practice, the machine M can *activate* and *weakly abort* the agents $A_1, \ldots, A_n$. Now, A has $I_A = I_M \cup \bigcup_{0 \leq h \leq n} I_{A_h}$ as input set of signals and $O_A = O_M \cup \bigcup_{0 \leq h \leq n} O_{A_h}$ as output set of signals.

The agent $A = A_1 \parallel A_2$ is the parallel composition of A_1 and A_2. A global state of A is a pair consisting of a state of A_1 and a state of A_2. A reaction of A consists of both a reaction of A_1 and a reaction of A_2. The agent A has $I_A = (I_{A_1} \cup I_{A_2}) \setminus (O_{A_1} \cup O_{A_2})$ as input set of signals and $O_A = O_{A_1} \cup O_{A_2}$ as output set of signals. Signals broadcast by A_1 are sensed both by the environment and by A_2, and conversely. These instantaneous communications may give rise to *causality paradoxes* which originate nondeterminism or nonreactivity (namely the inability of agents to react). Argos rejects statically programs where causality paradoxes may appear. We will assume to deal with correct programs.

3.2 The `tcc` *language* $\texttt{tcc}_{\text{Argos}}$

Let us assume the constraint system $\mathcal{A} = \langle D_{\mathcal{A}}, \vdash_{\mathcal{A}}, Var_{\mathcal{A}} \rangle$ such that:

- $Var_{\mathcal{A}}$ contains both $\mathcal{S}$ and the variables $in\_s_{h_M}$ and $out\_s_{h_M}$, for s_{h_M} a state of a Mealy machine M;
- $D_{\mathcal{A}}$ contains tokens of the form $b = t$ and $b = f$, where b is a boolean expres-

sion with variables in $Var_{\mathcal{A}}$ and t and f denote true and false, respectively;

- $\vdash_{\mathcal{A}}$ is the least relation such that:

$$\{b = t\} \vdash_{\mathcal{A}} \neg b = f \qquad \{b_1 = f\} \vdash_{\mathcal{A}} b_1 \wedge b_2 = f$$
$$\{b = f\} \vdash_{\mathcal{A}} \neg b = t \qquad \{b_1 = f, b_2 = f\} \vdash_{\mathcal{A}} b_1 \vee b_2 = f$$
$$\{b_1 = t\} \vdash_{\mathcal{A}} b_1 \vee b_2 = t \qquad \{b_1 = t, b_2 = t\} \vdash_{\mathcal{A}} b_1 \wedge b_2 = t.$$

We denote by $\texttt{tcc}_{\text{Argos}}$ the tcc language obtained by instantiating tcc over the constraint system $\mathcal{A}$. Our aim is to give an encoding of Argos into $\texttt{tcc}_{\text{Argos}}$. Intuitively, given a signal s, the token $s = t$ encodes the fact that s is broadcast. Given a state s_{h_M} of a Mealy machine M, the token $in\_s_{h_M} = t$ encodes the fact that M enters s_{h_M}, while the token $out\_s_{h_M} = t$ encodes the fact that M leaves s_{h_M}.

We define now the $\texttt{tcc}_{\text{Argos}}$ construct "`do _ watch _`" which will be used to simulate the weak abortion mechanism of Argos. Given a $\texttt{tcc}_{\text{Argos}}$ agent A and a boolean expression b, the agent `do` A `watch` b starts behaving as A and is weakly aborted when $b = f$ is not entailed by the store. The construct `do _ watch _` is defined as follows:

$$\texttt{do}\ a\ \texttt{watch}\ b\ =\ a$$
$$\texttt{do}\ (\texttt{if}\ a\ \texttt{then}\ A)\ \texttt{watch}\ b\ =\ \texttt{if}\ a\ \texttt{then do}\ A\ \texttt{watch}\ b$$
$$\texttt{do}\ (A_1, A_2)\ \texttt{watch}\ b\ =\ (\texttt{do}\ A_1\ \texttt{watch}\ b, \texttt{do}\ A_2\ \texttt{watch}\ b)$$
$$\texttt{do}\ (\texttt{next}\ A)\ \texttt{watch}\ b\ =\ \texttt{if}\ b = f\ \texttt{then next}\ (\texttt{do}\ A\ \texttt{watch}\ b)$$
$$\texttt{do}\ p\ \texttt{watch}\ b = \texttt{do}\ A\ \texttt{watch}\ b,\ \text{where } p = A.$$

We define now the $\texttt{tcc}_{\text{Argos}}$ construct "`_ init _`" which will be used to simulate the activation mechanism of Argos. Given a $\texttt{tcc}_{\text{Argos}}$ agent A and a boolean expression b, the agent A `init` b checks the token $b = t$ at each instant. If this is entailed by the store, then A `init` b starts A at the subsequent instant. The agent A `init` b is defined as follows:

$$A\ \texttt{init}\ b = \texttt{always}\ (\texttt{if}\ b = t\ \texttt{then next}\ A).$$

3.3 An encoding of Argos in $\texttt{tcc}_{\text{Argos}}$.

We define both a function $\langle\ \_\ \rangle$ from Argos agents to $\texttt{tcc}_{\text{Argos}}$ agents and a function $[\ \_\ ]$ from Argos agents to sets of declarations of $\texttt{tcc}_{\text{Argos}}$ procedures. The encoding of an Argos program P with body A is a $\texttt{tcc}_{\text{Argos}}$ program denoted by $\langle P \rangle$ and having $[A]$ as declaration set and $\langle A \rangle$ as body.

Intuitively, the prompting to an Argos agent A of a set of signals I by the environment corresponds to the prompting to the $\texttt{tcc}_{\text{Argos}}$ agent $\langle A \rangle$ of the set of tokens $\{s = t \mid s \in I\} \cup \{s = f \mid s \in I_A \setminus I\}$. The broadcasting of a set of signals $O \subseteq O_A$ by A corresponds to the posting in the store of the set of tokens $\{s = t \mid s \in O\} \cup \{s = f \mid s \in O_A \setminus O\}$ by $\langle A \rangle$.

$$\langle (S_M, s_{0_M}, I_M, O_M, T_M) \rangle = p_{0_M}$$

$$[(S_M, s_{0_M}, I_M, O_M, T_M)] = \{p_{h_M} = A_{h_M} \mid 0 \leq h \leq n\}, \text{ where}$$
$$A_{h_M} = (\{\texttt{if } b = t \texttt{ then}(\mathcal{O}(O), \texttt{next } p_{k_M}, out\_s_{h_M} = t, in\_s_{k_M} = t) \mid (s_{h_M}, b, O, s_{k_M}) \in T_M\},$$
$$\texttt{if } R_{h_M} = t \texttt{ then } (\texttt{next } p_{h_M}, out\_s_{h_M} = f))$$

$$\langle M \rhd \{A_0, \ldots, A_n\} \rangle = (\langle M \rangle, \{(\texttt{do } \langle A_h \rangle \texttt{ watch } out\_s_{h_M}) \texttt{ init } in\_s_{h_M} \mid 0 \leq h \leq n\},$$
$$\texttt{do } \langle A_0 \rangle \texttt{ watch } out\_s_{0_M})$$

$$[M \rhd \{A_0, \ldots, A_n\}] = [M] \cup \bigcup_{0 \leq h \leq n} [A_h]$$

$$\langle A_1 \parallel A_2 \rangle = (\langle A_1 \rangle [O'_{A_1}/O_{A_1}], \langle A_2 \rangle [O''_{A_2}/O_{A_2}], \{\texttt{always(if } s' = t \vee s'' = t \texttt{ then } s = t,$$
$$\texttt{if } s' = f \wedge s'' = f \texttt{ then } s = f) \mid s \in O_A\})$$

$$[A_1 \parallel A_2] = [A_1][O'_{A_1}/O_{A_1}] \cup [A_2][O''_{A_2}/O_{A_2}]$$

$$\langle \texttt{In} : s_1, \ldots, s_n; A; \texttt{Out} : s_{n+1}, \ldots, s_{n+m} \rangle = \texttt{In} : s_1, \ldots, s_n; [A]; \langle A \rangle; \texttt{Out} : s_{n+1}, \ldots, s_{n+m}$$

Fig. 1. The functions $\langle \_ \rangle$ and $[\_]$.

The definitions of $\langle \_ \rangle$ and $[\_]$ are in Figure 1. Before explaining these definitions, we need some notations.

Given an Argos agent A and a set of signals $O \subseteq O_A$, we denote by $\mathcal{O}(O)$ the $\texttt{tcc}_{\text{Argos}}$ agent $(\{s = t \mid s \in O\}, \{s = f \mid s \in O_A \setminus O\})$ which simulates the broadcasting of signals in O.

Given a Mealy machine M and a state $s_{h_M} \in S_M$, we denote by R_{h_M} the boolean expression $\bigwedge_{(s_{h_M}, b, O, s_{k_M}) \in T_M} \neg b$. If R_{h_M} evaluates to t and M is in state s_{h_M} then M does not leave s_{h_M}, since no transition is triggered.

For a Mealy machine M, $\langle M \rangle$ is the procedure call p_{0_M} and $[M]$ contains the declaration of a procedure p_{h_M} for each state $s_{h_M} \in S_M$. Each transition (s_{h_M}, b, O, s_{k_M}) of M corresponds to a parallel component of the body A_{h_M} of p_{h_M} which checks the token $b = t$ and, if this is entailed by the store, performs the following actions:

- it posts in the store $s = t$ (resp. $s = f$) for each $s \in O$ (resp. $s \in O_M \setminus O$)
- it activates the procedure p_{k_M} at the next instant
- it posts in the store the tokens $out\_s_{h_M} = t$ and $in\_s_{k_M} = t$. These tokens permit to weakly abort the $\texttt{tcc}_{\text{Argos}}$ agent $\langle A_h \rangle$ at the current instant and to activate the agent $\langle A_k \rangle$ at the next instant if M is refined by Argos agents $A_0, \ldots, A_n$.

Moreover, a parallel component of A_{h_M} checks the token $R_{h_M} = t$ and, if this is entailed by the store, it posts the token $out\_s_{h_M} = f$ in the store, representing that M does not leave s_{h_M}. So, p_{h_M} posts either $out\_s_{h_M} = t$ or $out\_s_{h_M} = f$ in the store.

The encoding of $M \rhd \{A_0, \ldots, A_n\}$ is obtained from the encodings of M and $A_0, \ldots, A_n$. The $\texttt{tcc}_{\text{Argos}}$ agent $\langle A_h \rangle$ is activated at the $(n+1)^{th}$ instant if $in\_s_{h_M} = t$ is entailed by the store at the n^{th} instant, namely if M enters s_{h_M} at the n^{th} instant. The agent $\langle A_h \rangle$ is weakly aborted when the token

$out\_s_{h_M} = t$ is entailed by the store, namely when M leaves s_{h_M}. The agent $\langle A_0 \rangle$ is activated also at the first instant, since A_0 starts running in the first instant.

The encoding of $A_1 \parallel A_2$ cannot be obtained as the parallel composition of the encodings of A_1 and A_2 since it may happen that $\langle A_1 \rangle$ posts the token $s = t$ in the store and $\langle A_2 \rangle$ posts the token $s = f$ in the store, when s is broadcast by A_1 and not by A_2. We denote by $\langle A_1 \rangle [O'_{A_1}/O_{A_1}]$ the $\texttt{tcc}_{\text{Argos}}$ agent obtained from $\langle A_1 \rangle$ by renaming each signal $s \in O_{A_1}$ in the body of a `tell` by s', and we denote by $\langle A_2 \rangle [O''_{A_2}/O_{A_2}]$ the $\texttt{tcc}_{\text{Argos}}$ agent obtained from $\langle A_2 \rangle$ by renaming each signal $s \in O_{A_2}$ in the body of a `tell` by s''. Now, when either $s' = t$ or $s'' = t$ is entailed by the store then $\langle A_1 \parallel A_2 \rangle$ posts the token $s = t$ in the store. Otherwise, it posts the token $s = f$ in the store.

It is immediate to observe that the encoding of Argos in $\texttt{tcc}_{\text{Argos}}$ given by the functions $\langle \_ \rangle$ and $[\_]$ is linear w.r.to the size of Argos programs.

We observe that in [14] a translation of Argos into boolean equations has been given. Our technique to encode the mechanisms of activation and weak abortion has some analogies with that of [14].

The following theorem states the soundness of our encoding, namely that there exists a correspondence between the operational meaning of an Argos program P and the operational meaning of the $\texttt{tcc}_{\text{Argos}}$ program $\langle P \rangle$.

Theorem 3.1 *The following facts are equivalent.*

- *The Argos program P reacts to a sequence of sets of input signals $I_1, \ldots, I_n, \ldots$ by broadcasting sets of output signals $O_1, \ldots, O_n, \ldots$.*
- *The $\texttt{tcc}_{\text{Argos}}$ program $\langle P \rangle$ reacts to a sequence of sets of tokens $\Delta_1, \ldots, \Delta_n, \ldots$, with $\Delta_n = \{s = t \mid s \in I_n\} \cup \{s = f \mid s \in I_P \setminus I_n\}$, by producing sets of tokens $\Delta'_1, \ldots, \Delta'_n, \ldots$, with $\Delta'_n \supseteq \{s = t \mid s \in O_n\} \cup \{s = f \mid s \in O_P \setminus O_n\}$.*

3.4 Encoding the strong abortion mechanism.

We define the $\texttt{tcc}_{\text{Argos}}$ construct "`do _ strong watch _`" such that, given a $\texttt{tcc}_{\text{Argos}}$ agent A and a boolean expression b, the agent `do` A `strong watch` b starts behaving as A and is strongly aborted in the instant in which $b = f$ is not entailed by the store. The construct `do _ strong watch _` is defined as follows:

`do` a `strong watch` b $=$ `if` $b = f$ `then` a
`do` (`if` a `then` A) `strong watch` b $=$ `if` a `then do` A `strong watch` b
`do` (A_1, A_2) `strong watch` b $=$ (`do` A_1 `strong watch` b, `do` A_2 `strong watch` b)
`do` (`next` A) `strong watch` b $=$ `if` $b = f$ `then next` (`do` A `strong watch` b)
`do` p `strong watch` b $=$ `do` A `strong watch` b, where $p = A$.

Now, given an agent A and a signal s, the agent `do` A `strong watch` s is strongly aborted when $s = f$ is not entailed by the store. This means that

`do` A `strong watch` s is strongly aborted when $s = t$ is entailed by the store. In fact, we have assumed that for each signal $s \in I_A$, the environment prompts either $s = t$ or $s = f$ at each instant. Moreover, at each instant each $\texttt{tcc}_{\text{Argos}}$ agent posts either $s = t$ or $s = f$ in the store for each output signal s.

We note that in [17] it is shown how one can encode a construct equivalent to `do _ strong watch` s in `tdcc`, but it is conjectured that strong abortion cannot be encoded in `tcc`. So, we have proved the `tcc` is more powerful than expected.

4 Embedding Lustre in `tcc`

In this section we show that $\text{Lustre}_{\mathcal{F}}$ is embedded in `tcc`. We begin by introducing informally Lustre. Then we define the `tcc` language $\texttt{tcc}_{\text{Lustre}}$ and we provide a compositional encoding of $\text{Lustre}_{\mathcal{F}}$ in $\texttt{tcc}_{\text{Lustre}}$.

4.1 An introduction to Lustre.

A Lustre program has as body a set of equations of the form $X = E$, with X a variable and E an expression. Both X and E denote *flows*, namely pairs consisting of a sequence of values of a given type and of a sequence of instants (*clock*). A flow takes the n^{th} value of its sequence at the n^{th} instant of its clock. The equation $X = E$ assigns to X the flow of E.

A program has a *basic clock* such that at each instant of this clock the environment prompts the value of the input variables. Slower clocks can be defined by means of flows with boolean values.

Expressions are constructed from constants and variables by means of *data operators* and *temporal operators*. Data operators are usual operators over basic types which operate pointwise on the sequences of values of their operands. Temporal operators operate explicitly over flows:

- `pre` ("previous") acts as a memory: if $(e_1, e_2, \ldots, e_n, \ldots)$ is the sequence of values of an expression E then `pre`(E) has the clock of E as clock and $(\perp, e_1, \ldots, e_{n-1}, \ldots)$ as sequence, where $\perp$ represents an undefined value.
- $\rightarrow$ ("followed by"): if E and F are expressions with the same clock and with sequences $(e_1, e_2, \ldots, e_n, \ldots)$ and $(f_1, f_2, \ldots, f_n, \ldots)$ then $E \rightarrow F$ has the clock of E and F as clock and $(e_1, f_2, \ldots, f_n, \ldots)$ as sequence.
- `when` samples an expression according to a slower clock: if E is an expression and E' is a boolean expression with the same clock then E `when` E' has the clock defined by E' and the sequence extracted from the one of E by keeping only those values of indexes corresponding to t values in the sequence of E'.
- `current` interpolates an expression on the clock immediately faster than its own: if E is an expression whose clock is not the basic one, and E' defines this clock, then `current`(E) has the clock of E' as clock, and at any instant of this clock it takes the value of E at the last time when E' was t.

The effect of operators `when` and `current` is showed below.

X	x_1	x_2	x_3	x_4	x_5	x_6	x_7
E'	t	f	t	t	f	t	f
$Y = X\,\texttt{when}\,E'$	x_1		x_3	x_4		x_6	
$\texttt{current}(Y)$	x_1	x_1	x_3	x_4	x_4	x_6	x_6

As argued in [5], we can assume without loss of generality that all operators are applied to variables. In fact, every equation can be transformed into an equivalent set of equations satisfying this requirement by introducing auxiliary variables. So, we consider the following syntax for Lustre:

$P ::= \texttt{In} : varseq\ ;\ Eqs\ ;\ \texttt{Out} : varseq$
$varseq ::= \epsilon \mid X, varseq$
$Eqs ::= Eq \mid Eq\ ,\ Eqs$
$Eq ::= X = E$
$E ::= k \mid X \mid \texttt{dop}(X, \ldots, X) \mid X \rightarrow X \mid \texttt{pre}(X) \mid X\,\texttt{when}\,X \mid \texttt{current}(X)$

where k, X, $varseq$, `dop`, E, Eq, Eqs, P are metavariables for constants, variables, sequences of variables, data operators, expressions, equations, sets of equations and programs, respectively.

The Lustre compiler checks that the following requirements are satisfied:

- operators are applied to operands having the same clock
- any output variable is defined by exactly one equation
- any output variable does not depend on itself.

The operational semantics of a program P with input variables $\vec{Y} = \{Y_1, \ldots, Y_n\}$, output variables $\vec{X} = \{X_1, \ldots, X_n\}$ and sets of equations $\vec{X} = \vec{E}$ is given in terms of sequences of pairs $(\vec{Y} = \vec{y^1}, \vec{X} = \vec{x^1}), \ldots, (\vec{Y} = \vec{y^n}, \ldots, \vec{X} = \vec{x^n}), \ldots$, where y_i^n is the value assigned to the input variable Y_i by the environment at the n^{th} instant of the basic clock, and x_j^n is either $\perp$, if the clock of the output variable X_j does not evaluate to t at the n^{th} instant of the basic clock, or k, if the expression E_j evaluates to k at the n^{th} instant of the basic clock.

4.2 The `tcc` *language* $\texttt{tcc}_{\text{Lustre}}$.

We consider the constraint system $\mathcal{L} = \langle D_{\mathcal{L}}, \vdash_{\mathcal{L}}, Var_{\mathcal{L}} \rangle$ such that:

- $Var_{\mathcal{L}}$ contains the variables X, ck_X, P_X, C_X for each variable X in the syntax of Lustre;
- $D_{\mathcal{L}}$ is the set of tokens of the form $X = E$, with X a variable in $Var_{\mathcal{L}}$ and E an expression over variables in $Var_{\mathcal{L}}$;
- $\vdash_{\mathcal{L}}$ is the least relation such that $\{X = E, Y = X\} \vdash_{\mathcal{L}} Y = E$.

We denote by $\texttt{tcc}_{\text{Lustre}}$ the tcc language obtained by instantiating tcc over the constraint system $\mathcal{L}$.

4.3 An encoding of Lustre$_{\mathcal{F}}$ in $\texttt{tcc}_{\text{Lustre}}$.

We define both a function $\ll \_ \gg$ from Lustre$_{\mathcal{F}}$ equations to $\texttt{tcc}_{\text{Lustre}}$ agents and a function $[[\_]]$ from Lustre$_{\mathcal{F}}$ equations to sets of declarations of $\texttt{tcc}_{\text{Lustre}}$ procedures. Both functions are extended to sets of equations. The encoding of a Lustre$_{\mathcal{F}}$ program P with body Eqs is a $\texttt{tcc}_{\text{Lustre}}$ program, denoted by $\ll P \gg$, with $[[Eqs]]$ as set of declarations of procedures and with body derived from $\ll Eqs \gg$.

Intuitively, the prompting of a set of values $\vec{y}$ for the input variables $\vec{Y}$ to a Lustre$_{\mathcal{F}}$ program P by the environment corresponds to the prompting of the set of tokens $\vec{Y} = \vec{y}$ to the $\texttt{tcc}_{\text{Lustre}}$ program $\ll P \gg$. At each instant, and for each output variable X, the program $\ll P \gg$ posts tokens constraining the variable ck_X either to t, if X has the basic clock as clock, or to the value of the boolean variable B, if B defines the clock of X. So, ck_X represents the clock of X. Moreover, if X evaluates to k in P then $\ll P \gg$ posts in the store the token $X = k$. If X evaluates to $\perp$ in P then $\ll P \gg$ posts in the store the token $X = \perp$.

The definitions of $\ll \_ \gg$ and $[[\_]]$ are in Figure 2, where in the definition of $\ll X = \texttt{pre}(Y) \gg$ we have assumed that Y ranges over $\{v_1, \ldots, v_n\}$ and in the definition of $\ll X = \texttt{current}(Y) \gg$ we have assumed that $B_1, \ldots, B_m$ are the boolean variables which appear in P in the body of a `when`, namely which define clocks of expressions.

At each instant, the agent $\ll X = k \gg$ constrains ck_X to take the value t and X to take the value k. This reflects that X has the basic clock as clock and that it always evaluates to k.

At each instant, the agent $\ll X = Y \gg$ constrains ck_X to take the value of ck_Y and X to take the value of Y. This reflects that X has the clock of Y as clock and that it always evaluates as Y.

At each instant, the agent $\ll X = \texttt{dop}(X_1, \ldots, X_n) \gg$ constrains ck_X to take the value of ck_{X_1} and X to take the value of $\texttt{dop}(X_1, \ldots, X_n)$. This reflects that X has the clock of ck_{X_1} as clock and that it always evaluates as $\texttt{dop}(X_1, \ldots, X_n)$. Note that $ck_{X_1} = ck_{X_i}$ for every $2 \leq i \leq n$.

Let us consider now the agent $\ll X = Y_1 \rightarrow Y_2 \gg$. It constrains ck_X to take the value of ck_{Y_1} at each instant, to reflect that X has the clock of Y_1 as clock. The procedure p_X constrains X to take the value of Y_1, while the procedure q_X constrains X to take the value of Y_2. The procedure p_X is active from the first instant up to the first instant in which the clock of X is t; q_X is activated afterwards.

At each instant, the agent $\ll X = Y$ `when` $B \gg$ posts in the store the token $ck_X = B$ reflecting that B is the clock of X. Moreover, $\ll X = Y$ `when` $B \gg$ constrains X to take either the value of Y, if B has value t, or $\perp$, otherwise.

$$[[X = E]] \;=\; \begin{cases} \{p_X = A_{p_X}, q_X = A_{q_X}\} \text{ if } E = Y_1 \to Y_2 \text{ for some } Y_1, Y_2 \\ \emptyset \qquad\qquad\qquad\qquad\qquad \text{otherwise} \end{cases}, \text{ where}$$

$A_{p_X} = (\texttt{if } ck_X = t \texttt{ then } (X = Y_1, \texttt{next } q_X), \texttt{if } ck_X = f \texttt{ then } (X = Y_1, \texttt{next } p_X),$
$\quad\texttt{if } ck_X = \bot \texttt{ then } (X = Y_1, \texttt{next } p_X))$
$A_{q_X} = (X = Y_2, \texttt{next } q_X)$

$\ll X = k \gg = \texttt{always } (ck_X = t, X = k)$

$\ll X = Y \gg = \texttt{always } (ck_X = ck_Y, X = Y)$

$\ll X = \texttt{dop } (X_1 \ldots, X_n) \gg = \texttt{always } (ck_X = ck_{X_1}, X = \texttt{dop } (X_1 \ldots, X_n))$

$\ll X = Y \texttt{ when } B \gg = \texttt{always } (ck_X = B, \texttt{if } B = t \texttt{ then } X = Y,$
$\qquad\qquad\qquad\qquad\qquad\qquad\texttt{if } B = f \texttt{ then } X = \bot, \texttt{if } B = \bot \texttt{ then } X = \bot)$

$\ll X = Y_1 \to Y_2 \gg = (p_X, \texttt{always } ck_X = ck_{Y_1})$

$\ll X = \texttt{pre}(Y) \gg =$
$(P_Y = \bot, \texttt{always}(ck_X = ck_Y,$
$\qquad\texttt{if } ck_Y = t \texttt{ then } X = P_Y, \texttt{if } ck_Y = f \vee ck_Y = \bot \texttt{ then } X = \bot,$
$\qquad\texttt{if } Y = v_1 \texttt{ then next } P_Y = v_1, \ldots, \texttt{if } Y = v_n \texttt{ then next } P_Y = v_n,$
$\qquad\texttt{if } P_Y = v_1 \wedge (ck_Y = f \vee ck_Y = \bot) \texttt{ then next } P_Y = v_1,$
$\qquad\vdots$
$\qquad\texttt{if } P_Y = v_n \wedge (ck_Y = f \vee ck_Y = \bot) \texttt{ then next } P_Y = v_n,$
$\qquad\texttt{if } P_Y = \bot \wedge (ck_Y = f \vee ck_Y = \bot) \texttt{ then next } P_Y = \bot))$

$\ll X = \texttt{current}(Y) \gg =$
$(C_Y = \bot, \texttt{always}(\texttt{if } ck_Y = t \texttt{ then } X = Y, \texttt{if } ck_Y = f \texttt{ then } X = C_Y,$
$\qquad\texttt{if } ck_Y = \bot \texttt{ then } X = \bot,$
$\qquad\texttt{if } ck_Y = B_1 \texttt{ then } ck_X = ck_{B_1}, \ldots, \texttt{if } ck_Y = B_m \texttt{ then } ck_X = ck_{B_m},$
$\qquad\texttt{if } Y = v_1 \texttt{ then next } C_Y = v_1, \ldots, \texttt{if } Y = v_n \texttt{ then next } C_Y = v_n,$
$\qquad\texttt{if } C_Y = v_1 \wedge (ck_Y = f \vee ck_Y = \bot) \texttt{ then next } C_Y = v_1,$
$\qquad\vdots$
$\qquad\texttt{if } C_Y = v_n \wedge (ck_Y = f \vee ck_Y = \bot) \texttt{ then next } C_Y = v_n,$
$\qquad\texttt{if } C_Y = \bot \wedge (ck_Y = f \vee ck_Y = \bot) \texttt{ then next } C_Y = \bot))$

$\ll Eq, Eqs \gg = (\ll Eq \gg, \ll Eqs \gg)$
$[[Eq, Eqs]] = [[Eq]] \cup [[Eqs]]$

$\ll \texttt{In} : Y_1, \ldots, Y_h \;;\; Eqs \;;\; \texttt{Out} : X_1, \ldots, X_k \gg =$
$\texttt{In} : Y_1, \ldots, Y_h \;;\; [[\, Eqs \,]] \;;\; (\ll Eqs \gg, \texttt{always } (ck_{Y_1} = t, \ldots,\; ck_{Y_h} = t)) \;;\; \texttt{Out} : X_1, \ldots, X_k$

Fig. 2. The functions $\ll \_ \gg$ and $[[\,\_\,]]$.

The agent $\ll X = \texttt{pre}(Y) \gg$ defines an auxiliary variable P_Y carrying the value taken by Y at the last instant in which the clock of Y was t. When the clock of Y is t then the agent $\ll X = \texttt{pre}(Y) \gg$ constrains X to take the value of P_Y else it constrains X to take $\perp$.

The agent $\ll X = \texttt{current}(Y) \gg$ constrains ck_X to take the value of ck_{B_i}, where B_i is the clock of Y. This reflects that X has as clock the clock of the clock of Y. The variable C_Y plays the same rôle of P_Y in the body of $\ll X = \texttt{pre}(Y) \gg$. Finally, $\ll X = \texttt{current}(Y) \gg$ constrains X to take either the value of Y, if the clock of Y is t, or the value of C_Y, if the clock of Y is f, or $\perp$, otherwise. Note that the clock of Y is either t or f iff the clock of X is t.

For each Lustre$_{\mathcal{F}}$ program P, the body of $\ll P \gg$ has `always` $(ck_{Y_1} = t, \ldots, ck_{Y_h} = t)$ as parallel component, to reflect that all input variables have as clock the basic clock.

It is immediate to observe that the encoding of Lustre$_{\mathcal{F}}$ in $\texttt{tcc}_{\text{Lustre}}$ given by the functions $\ll \_ \gg$ and $[\![\_]\!]$ is linear w.r.to the size of Lustre$_{\mathcal{F}}$ programs and w.r.to the cardinality of the values of types.

Now, in order to encode the equation $X = \texttt{pre}(Y)$, we need to transfer explicitly the value of Y from an instant to the subsequent one, since `tcc` (like `tdcc`) does not offer any mechanism to memorize information. This would not be possible if Y ranges over an infinite set of values. So, we conjecture that the full language Lustre can be encoded neither in `tcc` nor in `tdcc`.

The following theorem states the soundness of the encoding of Lustre$_{\mathcal{F}}$ in $\texttt{tcc}_{Lustre}$, namely that there exists a correspondence between the operational meaning of a Lustre$_{\mathcal{F}}$ program P and the operational meaning of its encoding $\ll P \gg$.

Theorem 4.1 *The following facts are equivalent:*

- *A Lustre program P with input set of variables $\vec{Y} = \{Y_1, \ldots, Y_h\}$ and output set of variables $\vec{X} = \{X_1, \ldots, X_k\}$ reacts to an input sequence $\vec{Y}=\vec{y^1}, \ldots, \vec{Y}=\vec{y^n}, \ldots$ by giving the output sequence $\vec{X}=\vec{x^1}, \ldots, \vec{X}=\vec{x^n}, \ldots$*
- *The $\texttt{tcc}_{\text{Lustre}}$ program $\ll P \gg$ reacts to a sequence of sets of tokens $\Delta_1, \ldots, \Delta_n, \ldots$, with $\Delta_n = \vec{Y}=\vec{y^n}$, by producing the sequence of sets of tokens $\Delta'_1, \ldots, \Delta'_n, \ldots$ with $\Delta'_n \vdash X_j = x_j^n$, and $\Delta'_n \vdash ck_{X_j} = t$ iff the clock of X_j at instant n is t.*

5 Conclusion

We have investigated the expressiveness of the synchronous paradigm `tcc`. We have defined the `tcc` languages $\texttt{tcc}_{\text{Argos}}$ and $\texttt{tcc}_{\text{Lustre}}$ and we have proved that they encode compositionally the synchronous state oriented language Argos and the synchronous data flow language Lustre$_{\mathcal{F}}$, respectively. We have also

proved that `tcc` encodes the strong abortion mechanism. Finally, we have conjectured that `tdcc` is not sufficiently powerful to encode the full language Lustre.

Note that a language embedding both $\texttt{tcc}_{\text{Argos}}$ and $\texttt{tcc}_{\text{Lustre}}$ could be used for merging Argos and Lustre. Other proposals for merging synchronous languages can be found in [12], where Argos is mapped to Lustre, and in [14], where the idea is to map both Argos and Lustre to the `DC` code [6].

References

[1] Benveniste, A. and Berry, G. (Editors): *Another Look at Real-Time Systems.* Special Issue of Proceedings of the IEEE **79**, 1991.

[2] Berry, G.: *Preemption in Concurrent Systems.* Proc. of FSTTCS '93, Springer LNCS 761, 1993.

[3] Berry, G. and Gonthier, G.: *The Esterel Synchronous Programming Language: Design, Semantics, Implementation.* Science of Computer Programming **19**, 1992.

[4] de Boer, F. and Palamidessi, C.: *Embedding as a Tool for Language Comparison.* Information and Computation **108**, 1994.

[5] Caspi, P., Halbwachs, N., Pilaud, P., Plaice, J.: *LUSTRE, a Declarative Language for Programming Synchronous Systems.* Proc. of POPL '87, ACM Press, 1987.

[6] C2-A-SYNCHRON. The common format of synchronous languages. The declarative code `DC` version 1.0. Technical report, Synchron project, 1995.

[7] Le Guernic, P., Benveniste, A., Bournai, P. and Gonthier, T.: *SIGNAL: a Data Flow Oriented Language for Signal Processing.* Technical Report IRISA Report 246, Rennes, France, 1985.

[8] Gupta, V., Jagadeesan, R. and Saraswat, V.A.: *Models for Concurrent Constraint Programming.* Proc. of CONCUR '96, Springer LNCS 1119, 1996.

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

[10] Harel, D.: *Statecharts: a Visual Formalism for Complex Systems.* Science of Computer Programming **8**, 1987.

[11] Harel, D. and Pnueli, A.: *On the Development of Reactive Systems.* In K.R. Apt (Editor), Logic and Models of Concurrent Systems, NATO, ASI-13, Springer, 1985.

[12] Jourdan, M., Lagnier, F., Maraninchi, F. and Raymond, P.: *A multiparadigm language for reactive systems.* In 5^{th} IEEE International Conference on Computer Languages, Toulose, IEEE Computer Society Press, 1994.

[13] Maraninchi, F.: *Operational and Compositional Semantics of Synchronous Automaton Composition.* Proc. of CONCUR '92, Springer LNCS 630, 1992.

[14] Maraninchi, F. and Halbwachs, N.: *Compiling Argos into Boolean Equations.* Proc. of FTRTFTS '96, Springer LNCS 1135, 1996.

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

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

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

[18] Shapiro, E.Y.: *Embedding Among Concurrent Programming Languages.* Proc. of CONCUR '92, Springer LNCS 630, 1992.