Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 188 (2007) 143–155
www.elsevier.com/locate/entcs

A Framework for Timed Concurrent Constraint Programming with External Functions[1]

M. Alpuente[a,2] B. Gramlich[b,3] A. Villanueva[a,4]

[a] *DSIC, Technical University of Valencia*
Valencia, Spain

[b] *Faculty of Informatics*
Vienna University of Technology
Vienna, Austria

Abstract

The *timed concurrent constraint programming* language (tccp in short) was introduced for modeling reactive systems. This language allows one to model in a very intuitive way typical ingredients of these systems such as timeouts, preemptions, etc. However, there is no natural way for modeling other desirable features such as functional computations, for example for calculating arithmetic results. In fact, although it is certainly possible to implement such kind of operations, each single step of the computation takes time in tccp, and avoiding interferences with the intended overall behavior of the (reactive) system is quite involved.
In this paper, we propose an extension of tccp for modeling *instantaneous* computations which improves the expressiveness of the language, in the sense that operations that are cumbersome to implement in pure tccp, are executed by calling an efficient, external functional engine, while the tccp programmer can focus on the pure, and usually more complex, reactive part of the system. We also describe a case study which motivates the work, and discuss how the new capability presented here can also be used as a new tool for developers from the verification point of view.

Keywords: Timed Concurrent Constraint language, Functional features, Case study

1 Introduction

The programming language *Timed Concurrent Constraint Programming* (tccp in short) was introduced by F. de Boer *et al.* in [3] for modeling reactive systems, i.e., concurrent systems which continuously interact with the user (and generally do not terminate). tccp was defined as an extension of the ccp model introduced by

[1] This work has been partially supported by the EU (FEDER) and the Spanish MEC under grants TIN2004-7943-C04 and HA2006-0007, the "ICT for EU-India Cross-Cultural Dissemination Project under grant ALA/95/23/2003/077- 054", and the Valencian Government under grant GV06/285.

[2] Email: alpuente@dsic.upv.es

[3] Email: gramlich@logic.at

[4] Email: villanue@dsic.upv.es

1571-0661/$ – see front matter
doi:10.1016/j.entcs.2007.05.043

Saraswat in [12], which was ideally thought of as a simple but powerful model for concurrency. The tccp language introduced two main modifications to the original ccp model. On the one hand, an implicit notion of (discrete) time was included in the semantics of tccp. On the other hand, a new *conditional* agent was introduced, which is able to handle negative information that can be used to model timeouts and preemptions.

In this paper, we propose an extension of tccp for modeling *instantaneous* functions which allows us to simplify and speed up arithmetic calculations. Although it is possible to implement arithmetic functions in tccp, the resulting implementation of such functions is quite far from being intuitive. Moreover, these computations consume unspecified amount of time, thus making the synchronization of processes more difficult. For example, a given process might need some data that another process is committed to compute, and this computation might take some time depending on the data size. Thus, the calculation might slow down and eventually disorder the overall execution of the system. We illustrate this problem by means of an example in Section 3.

The new capability presented here can be used as a new tool for developers from the verification point of view. It is well-known that verifying concurrent systems is highly complex. In the context of tccp, where the synchronization among processes is manually programmed, badly implemented calculations may cause synchronization errors and even mask other communication anomalies which then become more elusive to capture. In such cases, the possibility to perform an independent verification for the reactive and the functional components of the system can be a very helpful facility. External functions written in a functional language can be seen as a specification of tccp function implementations and the programmer can check the implementation of the whole tccp system by using the version with the external functions. Assertions which use the external functions can also be introduced in the tccp program, thus automatically verifying that they are satisfied during the program execution.

In Section 2 we first introduce the tccp language, then in Section 3 we motivate the proposed extension of the language by means of an example. We also provide the semantics for the new construct. In Section 4 we illustrate the proposed extensions by means of a representative example (the model of a coffee machine). We discuss how we can use the new features to check tccp programs in Section 5. Finally, some lines of further work and our conclusions are in Section 6.

2 The tccp language

The concurrent constraint programming framework (ccp) was defined as a simple but powerful model for concurrent systems. Over the last decades, the model has been extended in different ways, tccp being one of these extensions. tccp is a concurrent constraint language with a notion of time and a mechanism to capture negative information. Similarly to other languages of the ccp family, tccp is parametric w.r.t. an underlying constraint system. This implies that, at each time instant, there ex-

ists a global store which contains the information accumulated up to that specific time instant.

The underlying constraint system determines the atomic propositions and constraints of the language. In the following we recall the essential aspects of tccp. A tccp program $P ::= D.A$ consists of a set of declarations D and an agent A. A declaration D is defined as a set of declarations $D.D$ or a clause of the form $D ::= p(\overline{x}) :- A$ where $\overline{x}$ is a (possibly empty) list of variables, and A is an agent. Finally, agents are defined as:

$$A ::= \mathsf{tell}(c) \mid \sum_{0<i<j} \mathsf{ask}(c_i) \rightarrow A_i \mid \mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, A \mid A||A \mid \exists x\, A \mid p(\overline{v})$$

The *tell* agent $\mathsf{tell}(c)$ adds the (atomic) constraint c to the global store. c must be a constraint from the underlying constraint system.. The semantics of the agent establishes that the constraint c is only available in the following time instant. In other words, the execution of the *tell* agent takes one instant of time. The *choice* agent $\sum_{0<i<j} \mathsf{ask}(c_i) \rightarrow A_i$ corresponds to non-deterministic choice. It executes one of the branches A_i among theses whose guard c_i is satisfied by the store at that time. When a branch is taken, the execution of the corresponding A_i agent starts in the following time instant. This means that also the execution of the *choice* agent takes one instant of time. If no guard is entailed by the store, then the *choice* agent suspends. In such cases, it is executed again in the following time instant.

Both the *tell* and the *choice* agents exist in the ccp paradigm (i.e., the model without any notion of time). The only difference here is that their execution consumes one time instant in tccp. The *conditional* agent $\mathsf{now}\, c\, \mathsf{then}\, A_1\, \mathsf{else}\, A_2$ is new in tccp. This agent introduces the capability to capture negative information and, thus, to model features such as *timeouts* or *preemptions*. The *conditional* agent checks if the constraint c is satisfied by the store. In that case, the agent A_1, corresponding to the *then* branch, is executed. Otherwise, the agent corresponding to the *else* branch (A_2) is run. It is important to remark that, in contrast to the choice agent, the execution of the corresponding agent (A_1 or A_2) starts at the same time instant as the *conditional* agent, i.e., at the same time instant as the guard is checked. Note that another difference to the *choice* agent is the fact that the conditional agent never suspends: The execution always continues either with the *then* branch, or with the *else* one.

The ||-agent is the *parallel* agent, which is also defined in the ccp model. The semantics of the parallel agent in ccp follows the interleaving approach whereas, in the tccp language, parallelism is maximal parallelism. This means that each time we have two or more parallel agents which can be executed, all of them are executed concurrently, i.e., their execution starts at the same time instant. The semantics for the *hiding* agent $\exists x\, A$ coincides with the ccp version. This agent can be seen as an existential quantification of variable x in agent A. In that way, we make variable x local to agent A. Finally, $p(\overline{v})$ is the *procedure call* agent. If there exists a clause of the form $p(\overline{x}) :- A$ in the set of declarations, then the body A is executed in the

following time instant.. Similarly to the *choice* and *tell* agents, also the *procedure call* consumes time.

As said before, tccp is parametric w.r.t. a *cylindric constraint system*. Let us briefly introduce the constraint system underlying the language. Intuitively, a simple constraint system is a set of atomic constraints and an entailment relation $\vdash$ which satisfies some specific properties. Formally:

Definition 2.1 (Simple Constraint System [12]) *A* simple constraint system *is a structure $\langle \mathcal{C}, \vdash \rangle$ where $\mathcal{C}$ is the set of atomic constraints and the entailment relation $\vdash \subseteq \wp(\mathcal{C}) \times \mathcal{C}$ satisfies:*

(i) $u \vdash C$ *for all* $C \in u$

(ii) $u \vdash C$ *if* $u \vdash C'$, $\forall C' \in v$, *and* $v \vdash C$

The entailment relation can be extended to $\wp(\mathcal{C}) \times \wp(\mathcal{C})$ in the normal way. We can obtain a cylindric constraint system, by adding an existential quantification operator to a simple constraint system. Formally:

Definition 2.2 (Cylindric Constraint System [12]) *A tuple $\langle \mathcal{C}, \vdash, Var, \exists \rangle$ is a* cylindric constraint system *iff $\langle \mathcal{C} \vdash \rangle$ is a simple constraint system, Var is a denumerable set of variables and, for each $x \in Var$, there exists a function $\exists_x : \wp(\mathcal{C}) \to \wp(\mathcal{C})$ such that, for each $u, v \in \wp(\mathcal{C})$:*

(i) $u \vdash \exists_x u$,

(ii) $u \vdash v$ *then* $\exists_x u \vdash \exists_x v$,

(iii) $\exists_x(u \cup \exists_x v) = \exists_x u \cup \exists_x v$,

(iv) $\exists_x(\exists_y u) = \exists_y(\exists_x u)$.

3 Instantaneous functions in tccp

Let us motivate our proposal by means of a simple example: A tccp program which defines a deterministic arithmetic function. tccp was not specifically defined for specifying this kind of programs but rather reactive systems, and for this reason the code appears quite unnatural and clumsy.

We assume that the underlying constraint system supports Presburger Arithmetic (that is, the first-order theory of the natural numbers with addition). Assume also that no mechanism to ensure that a variable is non-free is provided in the constraint system. Also recall that in tccp there is no sequential composition agent. Therefore, sequentialization must be achieved by explicit synchronization.

In the following program, the clause `mult(N,M,Z,S)` returns in `Z` the product `N*M`. We manually synchronize the procedure calls by using an auxiliary variable `S` which ensures that the arithmetic calculations (in the last line) are not attempted before the recursive call has been successfully executed.

```
mult(N,M,Z,S) :-
    now (M=1) then
        (tell(Z=N) || tell(S=1))
```

```
else
    ∃M',Z',S'(tell(M' is M - 1) ||
              mult(N,M',Z',S') ||
              ask(S'=1) → (tell(Z is N+Z') || tell(S=1)).
```

For example, the execution of the agent (goal) `mult(3,3,Result,Sync)` instantiates, after a certain amount of time, variable `Result` to 9, and `Sync` to 1, which signals the termination of the process. This particular execution takes 8 instants of time, and what is more important, the time needed to finish a computation directly depends on the input values of the call. A detailed trace of this execution can be found in [2].

This is a very simple example, and we could have used an underlying constraint system where the product operator was defined, but it is not realistic to assume that we have a constraint system able to compute any function. In [14], the relation between constraint systems and first-order logic is established. In that context, a constraint system is defined as a pair (Σ, Δ), where Σ is a signature specifying constants, functions and predicate symbols, and Δ is a consistent first-order theory over Σ. The authors then define that $c \vdash d$ iff the formula $c \Rightarrow d$ is true in all models of Δ and, for operational reasons, they require $\vdash$ to be decidable. This restriction also indicates which kind of constraint systems underlying tccp we can reasonably assume and use.

Many complex arithmetical functions such as the factorial, square roots, etc. impose strong dependencies among the data which can only be achieved in tccp by a contrived sequentialization of processes as shown in the example above.

In the following, we present a declarative mechanism for supporting numerical calculations in tccp (i.e., function calls have no side effects: Identical calls at different points in time yield identical results). We propose a simple though practical classical approach where the logic language is interfaced to an external functional language. This has the advantage that the semantics of tccp can be easily adapted. More sophisticated languages integrating functions and constraint exist such as Mercury [13], Oz [15], Ciao [8], Curry [7], Toy [11], Gödel [10], Slam-sl [9], but none of them extends the ask-tell paradigm with time.

3.1 The semantics of tccp with a function call agent

In Figure 1 we recall the original operational semantics of tccp. Let us add a new *function call* agent in the syntax of the language. We write $\mathsf{Y} \leftarrow e$ to denote a call e to an external function. External functions are not defined by explicit tccp rules, but their semantics is determined by an external implementation which simply requires e to be sufficiently instantiated. The interface to the external functional engine consists of a binary predicate eval(e,V). The second argument must be a free variable which is instantiated to the result v of the function call e after successful execution. In that case, eval(e,V) becomes true. Then, in the following time instant, the constraint $\mathsf{Y} = \mathsf{V}$ is added to the store. We assume that this process takes a constant amount of time: One time instant.

r1 $\langle \mathsf{tell}(c), s\rangle \longrightarrow \langle \mathsf{stop}, s \sqcup c\rangle$

r2 $\langle \sum_{i=0}^{n} \mathsf{ask}(c_i) \rightarrow A_i, s\rangle \longrightarrow \langle A_j, s\rangle$ if $0 \leq j \leq n$ and $s \vdash c_j$

r3 $$\frac{\langle A, s\rangle \longrightarrow \langle A', s'\rangle}{\langle \mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, B, s\rangle \longrightarrow \langle A', s'\rangle}$$ if $s \vdash c$

r4 $$\frac{\langle B, s\rangle \longrightarrow \langle B', s'\rangle}{\langle \mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, B, s\rangle \longrightarrow \langle B', s'\rangle}$$ if $s \nvdash c$

r5 $$\frac{\langle A, s\rangle \not\longrightarrow}{\langle \mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, B, s\rangle \longrightarrow \langle A, s\rangle}$$ if $s \vdash c$

r6 $$\frac{\langle B, s\rangle \not\longrightarrow}{\langle \mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, B, s\rangle \longrightarrow \langle B, s\rangle}$$ if $s \nvdash c$

r7 $$\frac{\langle A, s\rangle \longrightarrow \langle A', s'\rangle \text{ and } \langle B, s\rangle \longrightarrow \langle B', s''\rangle}{\langle A||B, s\rangle \longrightarrow \langle A'||B', s' \sqcup s''\rangle}$$

r8 $$\frac{\langle A, s\rangle \longrightarrow \langle A', s'\rangle \text{ and } \langle B, s\rangle \not\longrightarrow}{\langle A||B, s\rangle \longrightarrow \langle A'||B, s'\rangle}$$

r9 $$\frac{\langle A, s_1 \sqcup \exists x\, s_2\rangle \longrightarrow \langle A', s'\rangle}{\langle \exists^{s_1} x\, A, s_2\rangle \longrightarrow \langle \exists^{s'} x\, A', s_2 \sqcup \exists x\, s'\rangle}$$

r10 $\langle p(x), s\rangle \longrightarrow \langle A, s\rangle$ if $p(x) :- A \in D$

Fig. 1. Original operational semantics of tccp

Below we provide the operational semantics for the function call agent [5]:

$$\frac{}{\langle \mathsf{Y} \leftarrow e, st\rangle \longrightarrow \langle \mathsf{stop}, st \sqcup \{\mathsf{Y} = \mathsf{V}\}\rangle} \quad \text{if } \exists\, \mathsf{V}.\mathsf{eval}(e, \mathsf{V})$$

Note that the computed value v is only available in the store in the following time instant. This is similar to what happens with the *tell* agent, that is, the execution of the function call agent takes exactly one time unit.

We have reimplemented in Curry ([7]) our previous tccp interpreter which we had written in Prolog. The new prototype includes the above construct for function calls. The system can be found in `http://www.dsic.upv.es/~villanue/tccp-func/`. We use Curry itself as the functional engine, external to the tccp interpreter. This allows us to program functions in a natural way while making use of advanced features such as type inference, higher-order features, partial application, equational constraint solving, etc. However, we do not exploit the power of logical variables in function calls which are available in Curry because we are only interested in deterministic computations. We represent all non-determinism within tccp. This is compatible with the basic ask-tell principle and very similar to the classical connection of external functions to logic programs [5].

[5] See [3] to recall the original semantics of tccp.

4 A Case Study

In this section we present a case study where the use of instantaneous functions arises very naturally. The program models a coffee machine with the following facilities:

- coins of 2€, 1€, 0.5€, 0.2€, 0.1€, 0.05€, 0.02€ and 0.01€ are accepted;
- the user can have some coffee (`coffee`), some milk coffee (`mcoffee`), only milk (`milk`), some tea (`tea`), only a cup (`cup`) and he can also choose to cancel the process (`cancel`);
- if the user makes a choice, but has not previously introduced enough money, the machine requests more money; moreover, no drink is produced;
- if the user introduces more money than necessary, the machine returns the change to the user;
- if the machine has not enough coins for returning the extra money to the user (the difference between the price of the product and the total amount introduced), then it does not supply the product, but it returns the whole amount of money introduced by the user;
- the machine has different boxes for each kind of coin, thus it is able to concurrently return a coin of each kind.

Next we explain the most important points in the specification of the coffee machine. The complete code of this program can be found in [2]. First of all, let us recall that streams are used in tccp to record the change of state. Each single variable is associated to a stream (implemented as a logical list); that is, each element of the list represents the value of the variable at a given time instant. This allows us to handle imperative variables in the same way as logical lists are used in concurrent logic languages. We write $X = [Y|Z]$ for denoting a stream X recording the current value Y of the considered variable and the stream Z of future values of the same variable. Streams are also used in tccp as explicit communication channels between tccp agents as illustrated in this section.

Now, we are ready to explain the clause which models the user behavior. In the figure we only show a fragment and we write `[...]` to indicate that some code has been omitted:

```
user(Free,Free',Order,Cash,Cash') :-
  ∃Free'',Order',Cash''(
   (ask(Free=[idle|Free']) → ∃C (tell(Cash = [C|Cash']) ||
                                 insertCoin1(C,Cash')) +
    [...]
```

```
    ask(Free=[idle|Free']) → tell(Order=[coffee|Order']) +
    ask(Free=[idle|Free']) → tell(Order=[mcoffee|Order']) +
    [...]
    ask(Free=[idle|Free']) → tell(Order=[cancel|Order']) +
    ask(Free=[idle|Free']) → tell(true)) ||
  user(Free',Free'',Order',Cash',Cash'')).
```

The `user` clause has five parameters:

- `Free` is a stream whose values can be either `idle` (when the machine is idle) or `busy` (when the machine is processing an order).
- `Free'` is an output stream parameter and corresponds to the tail of `Free`.
- `Order` is an output stream, whose head contains the selected order.
- `Cash` is a stream containing tuples with the money introduced by the user, classified by kinds of coins.
- `Cash'` is the tail of `Cash` and corresponds to the updated money after the user has introduced some more money.

The body of this clause is simply a non-deterministic choice between the different actions that the user can perform, including the choice *do nothing* (the last one). In case the user had introduced some money, the corresponding branch (depending on the kind of coin) calls a devoted procedure `insertCoin` which records this amount.

```
insertCoin1(C,Cash) :-
  ∃ X1,X2,X3,X4,X5,X6,X7,X8 (
    tell(C=c(X1,X2,X3,X4,X5,X6,X7,X8)) ||
    ask(true)→ ∃X'(tell(X' is X1+1) ||
                   tell(Cash=[c(X',X2,X3,X4,X5,X6,X7,X8)|_])))
```

In the `insertCoin1` clause, the first `tell` agent stores in each $\mathtt{X_i}$ the number of coins of each kind introduced up to that time instant. After that, first the variable corresponding to the kind of coin introduced is updated, and then the tuple storing the total number of coins is modified. In order to ensure that variable $\mathtt{X_i}$ is instantiated before the update is done, a delay is forced by calling agent `ask(true)`.

The `coffeeMachine` procedure models the behavior of the coffee machine. The `Case` argument is similar to the `Cash` one discussed above, and stores the total number of coins in the machine case, that can be returned to the user.

The following program excerpt corresponds to the actions that the coffee machine performs when the user presses the `coffee` button:

```
coffeeMachine(Free,Free',Ordr,Ordr',Cash,Case,Case',Outpt,Chnge) :-
  ∃C,Inpt(tell(Case=[C|Case']) ||
          tell(Ordr=[_|Ordr']) ||
          tell(Cash=[Inpt|_]) ||
          ask(Ordr=[coffee|_]) →
              ∃ N(N ← paid(Inpt) ||
                  ask(true) →
                      now N≥0.3 then
                          ∃ Chng(Chng ← change(0.3,C,Inpt) ||
                                 tell(Free=[working|_]) ||
                                 tell(Outpt=[coffee|_]) ||
```

```
                            tell(Change'=[Chng|_]) ||
                            giveChange(Ordr',Chng,Case,Case'))
                    else tell(Outpt=[moreMoney|_])) +
        ask(Ordr=[milk|_]) →
            ∃ N(N ← paid(Inpt) ||
                ask(true) →
                    now N≥0.35 then
                        ∃ Chng(Chng ← change(0.35,C,Inpt) ||
                            tell(Free=[working|_]) ||
                            tell(Outpt=[milk|_]) ||
                            tell(Change'=[Chng|_]) ||
                            giveChange(Ordr',Chng,Case,Case'))
                    else tell(Outpt=[moreMoney|_])) +
        [...]  +
        ask(Ordr=[cancel|_]) →
            giveChange(Ordr',Free,C,Case,Case')) ||
coffeeMachine(Free',Free'',Ordr',Ordr'',Cash',Case',Case'',Chnge',Outpt')
```

The structure of the declaration is as follows: Depending on the product chosen by the user (which is recorded in the `Order` stream), the machine checks the stream `Cash` to determine if enough money has been introduced. Here we use a function `paid` to calculate the total amount of money introduced. This function is externally implemented (in Curry) by simply adding the values of the different kinds of coins recorded in stream `Cash`.

```
data Case = C Int Int Int Int Int Int Int Int
paid ::  Case -> Int
paidC x1 x2 x3 x4 x5 x6 x7 x8 =
 x1*2 + x2*1 + x3*0.5 + x4*0.2 + x5*0.1 + x6*0.05 + x7*0.02 + x8*0.01
```

Whenever the total amount is greater than the price of the product, we start the process of supplying the product and (if necessary) returning the change, which depends on the number of coins of each kind the machine has. Note that `change` is again an external function, which represents the number of coins that the machine must return to the user. The following code excerpt shows the implementation of the `change` function, where `Price` represents the cost of the chosen product, `C` the number of coins in the case of the machine, and `Input` the coins that have been introduced by the user. The auxiliary function `coinsFor` calculates the total amount of each kind of coin that the machine will give back to the user.

```
change ::  Int -> Coins -> Coins -> Coins
change Price Input Case = coinsFor (Price - (paid Input)) Case

coinsFor ::  Int -> Coins -> Coins
coinsFor x (C c1 c2 c3 c4 c5 c6)
  | x >= 2 & c1 > 0 =
      plus (C 1 0 0 0 0 0 0 0)
           coinsFor (x-2) (C (c1-1) c2 c3 c4 c5 c6 c7 c8)
  | x >= 1 & c2 > 0 =
      plus (C 0 1 0 0 0 0 0 0)
           coinsFor (x-1) (C c1 (c2-1) c3 c4 c5 c6 c7 c8)
  [...]
  | x >= 0.01 & c8 > 0 =
      plus (C 0 0 0 0 0 0 0 1)
```

```
            coinsFor (x-0.01) (C c1 c2 c3 c4 c5 c6 c7 (c8-1))
  | otherwise = (C -1 -1 -1 -1 -1 -1 -1)

plus ::  Coins -> Coins -> Coins
plus (C x1 x2 x3 x4 x5 x6 x7 x8) (C y1 y2 y3 y4 y5 y6 y7 y8) =
  C (x1+y1) (x2+y2) (x3+y3) (x4+y4) (x5+y5) (x6+y6) (x7+y7) (x8+y8)
```

Procedure `giveChange` returns the coins to the user and also updates the machine case. The coins are eventually returned in parallel whenever possible. Note that this procedure uses the result calculated by the function `change` described above. In case there are not enough coins for returning the required change, the `change` function returns the value `(C -1 -1 -1 -1 -1 -1 -1)`. In this way, this situation could be detected by simply introducing a check about the returned value `Chng` in the `coffeeMachine` process. However, for simplicity, we have not implemented this checking in the current version of the code.

```
giveChange(Order,Change,Case,Case') :-
    ∃D1,D2,D3,D4,D5,D6,D7,D8,X1,X2,X3,X4,X5,X6,X7,X8,C1,C2,C3,C4,C5,C6,C7,C8 (
          tell(Change = c(D1,D2,D3,D4,D5,D6,D7,D8)) ||
          tell(Case=[c(X1,X2,X3,X4,X5,X6,X7,X8)|Case']) ||
          ask(true) → (tell(C1 is X1-D1) || tell(C2 is X2-D2) ||
                       [...]
                       tell(C8 is X8-D8) ||
                       tell(Order=[no|_]) || tell(Free=[idle|_]))).
```

Finally, the `system` procedure synchronizes the machine and the user declarations. Initially, the coffee machine has two coins of each class.

```
system(Case,Output) :- ∃ Free,Order,Cash,Cash',Case' (
      tell(Free=[idle|_]) ||
      tell(Order = [no|_]) ||
      tell(Cash = [c(0,0,0,0,0,0,0,0)|Cash']) ||
      user(Free,Order,Cash,Cash') ||
      Case = [c(2,2,2,2,2,2,2,2)|Case']) ||
      coffeeMachine(Free,Order,Cash,Case,Case',Output)).
```

5 Analysis and Run-time Verification

In this section we illustrate how instantaneous functions can be used in tccp to verify some (static as well as dynamic) properties of the system.

Let us enumerate some properties that the user could be interested to check in the model of the coffee machine:

(i) If an order is initiated and sufficient money has been introduced, then eventually the order is correctly completed (i.e., the product is supplied, the change is correctly returned, the Case is consistently updated and the status of the machine is reset); otherwise the machine returns the money.

(ii) If the user does not introduce enough money for the selected product, then no action is performed unless the cancel button is pressed. In that case, inserted coins are returned to the user.

Let us (partially) specify the former property above. Namely, that the Case is correctly updated after the completion of an order. We have to check that the new value of the machine case coincides with the amount Case + Cash - Change. In order to verify this property, we introduce a new external function `check` that performs the required computation.

```
check ::  Coins -> Coins -> Coins -> Coins -> Bool
check (C x1 x2 x3 x4 x5 x6 x7 x8) (C y1 y2 y3 y4 y5 y6 y7 y8)
      (C z1 z2 z3 z4 z5 z6 z7 z8) (C w1 w2 w3 w4 w5 w6 w7 w8) =
               y1+w1-x1 == z1 && y2+w2-x2 == z2 && y3+w3-x3 == z3 &&
               y4+w4-x4 == z4 && y5+w5-x5 == z5 && y6+w6-x6 == z6 &&
               y7+w7-x7 == z7 && y8+w8-x8 == z8
```

Assume that the definition of the function call `check(Change,C,C',Cash)` is written in Curry and that the calls to this function are correctly sequentialized and synchronized with the rest of the tccp code:

```
coffeeMachine(...)  :-
[...]
    ask(Order=[cancel|_]) → giveChange(Order,Input,Case,Case')) ||
        check2(Change',Case',Case'',Cash') ||
        coffeeMachine(Free',Free'',Order',Order'',Cash',Case',
                      Case'',Change', Output'))
[...]

check2(Change,Case,Case',Cash) :-
       ∃B(B ← check(Change,Case,Case',Cash) ||
          ask(true) → now (B=0) then stop else skip.
```

These function calls in the program can be seen as a means to introduce invariants or assertions along the code. Such invariants are checked during system execution, and the execution is interrupted in case one of the assertions is not satisfied. Obviously, the invariants should not corrupt the behavior of the original system, and it is the programmer who must take care of the synchronization.

It is also possible to program such checks at the program goal level, by using shared variables to synchronize the main process with a dedicated monitor.

```
coffeeMachine(Free,Free',Order,Order',Cash,Case,Case',Change,Output)
|| check2(Change,Case,Case',Cash)
```

Regarding the dynamic properties, we can combine our methodology with the constraint temporal logic defined in [4,1] which we illustrate as follows.

Consider again property (i) above. On one hand, we have to check that, whenever the `coffee` order is given to the machine, the latter eventually produces either the `coffee` output or the `moreMoney` output. In the constraint temporal logic this

can be written as:

$$\Box(\texttt{Order = coffee} \rightarrow \Diamond(\texttt{Output = coffee} \vee \texttt{Output = moreMoney}))$$

More specifically, when the order `coffee` is given to the machine, and providing that enough money has been introduced, then the machine must produce `coffee` and returns the change. This can be specified as follows:

$$\Box(\texttt{Order = coffee} \rightarrow \texttt{money\_received} \rightarrow (\Diamond(\texttt{Output = coffee} \wedge \texttt{change\_returned})))$$

where `money_received` and `change_returned` are atomic predicates corresponding to boolean properties that would have to be updated by the program at the appropriate time instant. Intuitively, `money_received` should have to be updated each time the system detects that the amount of money introduced is enough to serve the chosen product. We could also check whether the machine's state is `working` whenever `money_received` is true.

The second considered property checks the case when the user does not introduce enough money. In such a case, if the `cancel` order is given, then the machine should give back the whole amount of money introduced by the user up to that time instant. This additional new dynamic property could be defined as follows:

$$\begin{aligned}\Box((&\texttt{Order = coffee} \wedge \neg\texttt{money\_received}) \rightarrow \\ &((\texttt{Output = idle} \wedge \neg\texttt{money\_received})\, \mathcal{U}\, (\\ &\quad (\texttt{Order = cancel} \wedge \Diamond(\texttt{change\_returned})) \\ &\quad \vee \texttt{money\_received})))\end{aligned}$$

where `change_returned` is another atomic predicate corresponding to the appropriate boolean property. The $\mathcal{U}$ operator denotes the classic until operator of the temporal logic, defined also for the logic of [4].

We are currently improving our implementation by interfacing it with more powerful external constraint solvers in Curry. As further work, we also plan to develop a methodology to verify tccp code with external functions by adapting the model checkers developed in [6].

6 Conclusions

In a timed concurrent system, where data are shared among the processes and one process might wait for another to end, efficient synchronization policies are desired that can detect the bottlenecks and speed up the execution of the system. As a system with a support for time-dependent parallelism of the processes, tccp falls into this category, and functional computations (especially arithmetic ones) turn out to be actual bottlenecks in synchronizing the processes. Since tccp provides no support for sequential computation, performing a simple arithmetic computation may waste a considerable amount of time and also suspend the progress of all other processes that require the result of this computation. In order to overcome these drawbacks, we have developed a functional engine that can be plugged into the tccp system. This significantly optimizes the performance of the system and brings

more usability to tccp. The case study presented in the paper shows the usefulness of incorporating such aspects in tccp, especially in practical domains where it is convenient to distinguish arithmetic computations from other parts of a problem solving strategy.

References

[1] M. Alpuente, M.M. Gallardo, E. Pimentel, and A. Villanueva. A Real-Time Logic for tccp verification. *J. of Universal Computer Science*, to appear, 2006.

[2] M. Alpuente, B. Gramlich, and A. Villanueva. Timed Concurrent Constraint programming with External Functions. Technical Report DSIC-II/13/06, DSIC, Tech. Univ. Valencia, December 2006. http://www.dsic.upv.es/~villanue/agv06-tr.pdf.

[3] F. S. de Boer, M. Gabbrielli, and M. C. Meo. A Timed Concurrent Constraint Language. *Information and Computation*, 161:45–83, 2000.

[4] F. S. de Boer, M. Gabbrielli, and M. C. Meo. A Temporal Logic for reasoning about Timed Concurrent Constraint Programs. In G. Smolka, editor, *Proceedings of 8th International Symposium on Temporal Representation and Reasoning*, pages 227–233. IEEE Computer Society Press, 2001.

[5] S. Bonnier and J. Maluszynski. Towards a Clean Amalgamation of Logic Programs with External Procedures. In *Proceedings of the 5th Conference on Logic Programming & 5th Symposium on Logic Programming*, pages 311–326. MIT Press, 1988.

[6] M. Falaschi and A. Villanueva. Automatic Verification of Timed Concurrent Constraint programs. *Theory and Pract. of Logic Programm.*, to appear, 2006.

[7] M. Hanus (ed.). Curry: An Integrated Functional Logic Language (Vers. 0.8.2), March 2006. Available at http://www.informatik.uni-kiel.de/~curry.

[8] M. Hermenegildo. Some Methodological Issues in the Design of CIAO - A Generic, Parallel, Concurrent Constraint System. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, volume 874 of *LNCS*, pages 123–133. Springer-Verlag, 1994.

[9] A. Herranz and J.J. Moreno-Navarro. Slam-sl tutorial. Technical report, Babel Group, School Of Computer Science, Technical University of Madrid, 2001. Based on A. Herranz's PhD. Thesis.

[10] P.M. Hill and J.W. Lloyd. *The Gödel Programming Language*. MIT Press, 1994. ISBN 0-262-08229-2.

[11] F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of Int'l Conference on Rewriting Techniques and Applications*, volume 1631 of *LNCS*, pages 244–247. Springer-Verlag, 1999.

[12] V. A. Saraswat. *Concurrent Constraint Programming Languages*. The MIT Press, Cambridge, MA, 1993.

[13] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29:17–64, 1996.

[14] F. Valencia. Decidability of Infinite-State Timed CCP Processes and First-Oder LTL. *Theoretical Computer Science*, 330(3):577–607, 2005.

[15] P. Van Roy, P. Brand, D. Duchier, S. Haridi, M. Henz, and C. Schulte. Logic programming in the context of multiparadigm programming: the Oz experience. *Theory and Practice of Logic Programming*, 3(6):715–763, 2003.