Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 258 (2009) 63–77
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

The tccp Interpreter

Alexei Lescaylle[1,2] Alicia Villanueva[3]

DSIC, Universidad Politécnica de Valencia
Valencia, Spain

Abstract

The *Timed Concurrent Constraint* language (tccp in short) is a declarative language inspired by some formalisms specially developed to deal with concurrent and reactive systems. It is defined as a timed extension of the Saraswat's *Concurrent Constraint Paradigm* (ccp in short) which makes the new language suitable for modeling reactive systems. The computational model of ccp is based on agents generating (*telling*) and checking (*asking*) partial information (*constraints*) on a global *store* (a conjunction of constraints). tccp inherits the main characteristics of ccp: it is parametric w.r.t. a constraint system and non-deterministic. This work presents the tccpInterpreter system, which is the result of implementing the tccp language in Maude. Maude has been shown to be well suited for the task of prototyping the semantics of programming languages. Our tccpInterpreter parses a given program and simulates its behavior, also allowing us to reuse the Maude features to execute and analyze tccp programs.

Keywords: Tool demonstration, Timed Concurrent Constraint Language, Maude

1 Introduction

The Saraswat's *Concurrent Constraint Programming* paradigm, ccp in short [14], is a simple but powerful model for expressing concurrent systems. The computational model is based on agents interacting among them by adding and consulting partial information (constraints) in a global *store*. ccp replaces the classical notion of store as a collection of valuations of variables by the

---

[1] This work has been supported by the Spanish MEC under grant TIN2007-68093-C02-02, by the Generalitat Valenciana under grant GV/2009/024 and by the Universidad Politécnica de Valencia, under grant PAID-06-07 (TACPAS).

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

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

doi:10.1016/j.entcs.2009.12.005

notion of store as a conjunction of constraints. The *Timed Concurrent Constraint* language, tccp in short [5], extends the ccp paradigm with a notion of time that makes the language suitable for modeling reactive systems [8], namely systems which maintain an ongoing information exchange with their environment at run-time.

The tccp language inherits some significant characteristics from ccp: it is parametric w.r.t. a given constraint system, non-deterministic, and the stored information grows monotonically. The underlying constraint system specifies which kind of constraints the system will be able to handle, and the relation among them. The non-deterministic behavior allows us to have more compact and precise specifications for large systems. Finally, the monotonic behavior of the store implies that information cannot be canceled, thus it is necessary to use *streams* to model the classical evolution of variable values.

The temporal model introduced in tccp is based on a global clock that synchronizes the execution. The *agent-based* model provides an intuitive way to specify reactive and embedded systems. Thanks to the new features (w.r.t. ccp), the language is able to capture typical behaviors of reactive systems such as *time-outs*, *time-delays* or *watchdogs*. Furthermore, since time is embedded in the semantics of the language, it is possible to naturally use the (constrained version of) Linear Temporal Logic (LTL) proposed in [5] to specify and model check properties of tccp programs [3,2,6].

The rewriting logic-based and high-performance reflective specification language Maude [4,1] has been proposed for the task of building and analyzing a wide range of applications. In particular, rewriting logic [12] can deal with state and concurrent computations and has been used as a semantic framework for giving executable semantics to (concurrent) languages and models. Maude supports structured theory specifications, algebraic data types and function specification in rich equational logics. We assume that the reader has a basic knowledge of Maude[4].

As an example of the use of Maude as a semantic framework to provide executable semantics, we find in [7] JavaFAN (Java Formal ANalyzer), which is a tool that formally specifies the Java semantics in Maude. In [16] it is presented an interpreter of LOTOS based on the symbolic semantics for Full LOTOS [11]. In [17], the operational semantics of CCS [13] is implemented in Maude. CCS is in some sense similar to tccp. For example, operators like Nil or Choice are present in both languages. However, there are also important differences: in tccp the model for concurrency is based on maximal parallelism whereas in CCS it is used the interleaving model. In tccp, agents interact by using a global store (adding or consulting information) whereas in CCS, a

---

[4] For more detailed documentation about Maude, the interested reader can consult [4,1].

process may use communication ports to interact with other processes.

In this work we present the tccpInterpreter system: an interpreter for the tccp language. The tool is the implementation of the tccp formalism in Maude. The interpreter parses the tccp program, executes it (simulating its behavior), and also some analysis of the tccp program can be run. The analysis is carried out by using the Maude's *search* command which explores all the possible computations of a program, looking for safety violations when desiderated. Finally, it is important to note that tccpInterpreter incorporates some notions from [2] that were not present in the original definition of the language but that make the tccp framework more flexible.

To our knowledge, there is just an implementation of tccp. In [15] it was presented a prototype developed in the Mozart-Oz language. Mozart-Oz [9] is a multi-paradigm language allowing multi-threaded higher order programs to be directly executed in a distributed open system. However, the proposal is not publicly available and does not support the new features of tccp presented during the last years.

This work is organized as follows. In Section 2 we present the basic notions of the tccp language. Then, in Section 3 we partially describe how the syntax and operational semantics of tccp have been implemented in Maude. A model for a specific constraint system is shown in Section 4. Section 5 is devoted to show the functionality of the tool by using an illustrative example and, finally, in Section 6 we conclude and give some directions for future work.

2 The tccp language

Let us first recall the syntax of the agents of the tccp language. A tccp program P is of the form $D.A$ where D is a set of declarations of the form $\mathsf{p}(x)\,{:}{-}\,B$ and A is an agent that initiates the execution. We assume that c and c_i are finite constraints (i.e., elements) of the underlying constraint system. Then, a tccp agent is defined as follows:

$$A, B ::= \mathsf{skip} \mid \mathsf{tell}(c) \mid \textstyle\sum_{i=1}^{n} \mathsf{ask}(c_i) \rightarrow A_i \mid \mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, B \mid A || B \mid \exists x\, A \mid \mathsf{p}(x)$$

In brief, the skip agent does nothing. tell(c) adds the constraint c to the store. The choice agent $\sum_{i=1}^{n} \mathsf{ask}(c_i) \rightarrow A_i$ models the non-determinism: it checks whether the store satisfies the constraints c_i and non-deterministically executes one of the agents A_i whose condition c_i is satisfied. In case no condition c_i is entailed, the choice agent *suspends* (it will be executed again in the following time instant). The conditional agent now c then A else B executes agent A if the store satisfies c, otherwise executes B. Note that due to the partial nature of the stored information, the fact that c is not satisfied does not imply that $\neg c$ is satisfied. $A \parallel B$ executes the two agents A and B in parallel following the *maximal parallelism* model. The $\exists x\, A$ agent is used to define the variable x local to the process A. Finally, $\mathsf{p}(x)$ is the procedure call agent where x denotes the set of parameters of the process p.

The notion of time is modeled in the language following the idea that updates and consults to the store takes one time unit. Therefore, only the tell, choice and procedure call agents consume time. The rest of agents are considered instantaneous.

Similarly to ccp, the store in the original model of tccp behaves monotonically. Thus, it is not possible to change the value of a given variable along the time. This problem can be solved by using *streams*. For instance, we write $X = [1|Z]$ to denote a variable X recording the current value 1. The variable Z represents the future values of X. To ease the tasks related to streams manipulation, we use the modified computation model for tccp presented in [2] where the notion of global store was replaced by the notion of *structured store*. A structured store consists of a sequence of stores where each store of the sequence contains only the constraints added in the corresponding time instant.

tccpInterpreter considers three new agents introduced in [10] to deal in a more compact way with some common behaviors regarding streams. The $\mathsf{ask\text{-}tell}(S, V)$ agent consults the current value of the stream S and updates the store by instantiating the fresh variable V to such value. The $\mathsf{update}(S, V, T)$ agent inserts the value of the variable V in the stream S. T represents the future values of S. Finally, the $\mathsf{assign}(S, V_f, V_a)$ agent consults whether the value of V_f can be recovered from (any of the values stored in) the stream S; in that case, it instantiates V_a to the recovered value.

We can see in Figure 1 the operational semantics of the language borrowed from [2] that it is slightly different from the original model in [5] due to the new computational model based on structured stores. It is given by means of a transition relation between configurations, which are composed by an agent and the store *st* (at the current time instant t). The symbols $\doteq$, $\div$ and $\dot{\supset}$ are related to the operations of consulting the last value and tail of streams and the containment of a value in a stream. Finally, functions *free* and *len* check whether a variable is instantiated, and compute the length of a given stream, respectively.

Let us describe some of these semantic rules. The first rule **R1** specifies the evolution of the tell agent: it reaches a skip agent in the following time instant with the given store *st* augmented with the constraint c. Rule **R2** states that A_j is executed in the following time unit whenever *st* entails the condition c_j. Regarding the conditional agent, **R3** models the case when the

R1	$\langle \mathsf{tell}(c), st\rangle_t \longrightarrow \langle \mathsf{skip}, st \sqcup_{t+1} c\rangle_{t+1}$	
R2	$\langle \sum_{i=0}^{n} \mathsf{ask}(c_i) \rightarrow A_i, st\rangle_t \longrightarrow \langle A_j, st\rangle_{t+1}$	if $0 \leq j \leq n$,$st \vdash_t c_j$
R3	$\dfrac{\langle A, st\rangle_t \longrightarrow \langle A', st'\rangle_{t+1}}{\langle \mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, B, st\rangle_t \longrightarrow \langle A', st'\rangle_{t+1}}$	if $st \vdash_t c$
R4	$\dfrac{\langle B, st\rangle_t \longrightarrow \langle B', st'\rangle_{t+1}}{\langle \mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, B, st\rangle_t \longrightarrow \langle B', st'\rangle_{t+1}}$	if $st \not\vdash_t c$
R5	$\dfrac{\langle A, st\rangle_t \not\longrightarrow}{\langle \mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, B, st\rangle_t \longrightarrow \langle A, st\rangle_{t+1}}$	if $st \vdash_t c$
R6	$\dfrac{\langle B, st\rangle_t \not\longrightarrow}{\langle \mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, B, st\rangle_t \longrightarrow \langle B, st\rangle_{t+1}}$	if $st \not\vdash_t c$
R7	$\dfrac{\langle A, st\rangle_t \longrightarrow \langle A', st'\rangle_{t+1},\ \ \langle B, st\rangle_t \longrightarrow \langle B', st''\rangle_{t+1}}{\langle A \parallel B, st\rangle_t \longrightarrow \langle A' \parallel B', st' \sqcup st''\rangle_{t+1}}$	
R8	$\dfrac{\langle A, st\rangle_t \longrightarrow \langle A', st'\rangle_{t+1},\ \ \langle B, st\rangle_t \not\longrightarrow}{\langle A \parallel B, st\rangle_t \longrightarrow \langle A' \parallel B, st'\rangle_{t+1}}$	
R9	$\dfrac{\langle A, st_1 \sqcup \exists x\, st_2\rangle_t \longrightarrow \langle A', st'\rangle_{t+1}}{\langle \exists^{st_1} x\, A, st_2\rangle_t \longrightarrow \langle \exists^{st'} x\, A', st_2 \sqcup \exists x\, st'\rangle_{t+1}}$	
R10	$\langle \mathsf{p}(x), st\rangle_t \longrightarrow \langle A, st\rangle_{t+1}$	if $\mathsf{p}(x) : -A \in D$
R11	$\langle \mathsf{ask\text{-}tell}(S, C_v), st\rangle_t \longrightarrow \langle \mathsf{skip}, st \sqcup_{t+1} C_v = V_C\rangle_{t+1}$	if $st \models_t S \doteq V_C$,$\mathit{free}(C_v)$, $len(S, st, t) > 0$
R12	$\langle \mathsf{update}(S, V_i, C_t), st\rangle_t \longrightarrow \langle \mathsf{skip}, st \sqcup_{t+1} S = [V_i \mid C_t]\rangle_{t+1}$	if $st \models_t \mathit{free}(S)$,$\mathit{free}(C_t)$
R13	$\langle \mathsf{update}(S, V_i, C_t), st\rangle_t \longrightarrow \langle \mathsf{skip}, st \sqcup_{t+1} T_C = [V_i \mid C_t]\rangle_{t+1}$	if $st \models_t S \dot{\div} T_C$,$\mathit{free}(C_t)$, $len(S, st, t) > 0$
R14	$\langle \mathsf{assign}(S, V_f, V_a), st\rangle_t \longrightarrow \langle \mathsf{skip}, st \sqcup_{t+1} V_a = V_f\rangle_{t+1}$	if $st \models_t S \dot{\supset} V_f$,$\mathit{free}(V_a)$

Fig. 1. Operational semantics of the tccp language.

condition holds. In case that the agent A with the current store st can evolve in the agent A' and a new store st', then A' is executed in the following time instant. **R7** models the evolution of the $\parallel$ agent: in case that A with store st is able to evolve to the agent A' with st', and also B with store st is able to evolve into the agent B' with st'', then $A' || B'$ is executed in the following time instant with st' augmented with st''.

3 The interpreter implementation

The tccpInterpreter system is the result of the implementation in Maude of the tccp formalism, i.e., the language operational semantics plus a specific constraint solver. The tool takes as input the specification of a tccp program and simulates its behavior following the semantics of the language. tccpInterpreter consists of approximately 1080 lines of code divided in six Maude modules. Each module models one or more of the entities of tccp: agents, constraints, programs, the store, the underlying constraint system, the operational semantics, etc. Maude allows us to implement a constraint solver for the language or to use an existing one to handle constraints.[5]

3.1 Syntactic objects

The representation of the syntax of tccp in Maude is quite intuitive for all tccp constructs. Agents are defined to be terms of *sort* `TccpAgent`. For instance, the tell agent is encoded by using a Maude constructor symbol with identifier `tell` followed by the given constraint (a term of sort `TccpConstraint`):

```
op tell_ :  TccpConstraint -> TccpAgent .
```

The conditional agent is encoded by defining the identifier `now` followed by a boolean constraint (term of sort `TccpBoolean`), the `then` block which contains an agent, and the `else` block with another agent.

```
op now_then_else_ :  TccpBoolean TccpAgent TccpAgent -> TccpAgent .
```

The choice agent is encoded by using two Maude constructor symbols. The first one models a single branch of a choice: the identifier `ask` is followed by a Boolean constraint, the arrow `->` and an agent. The second one models the composition of two or more branches:

```
op ask_->_ :  TccpBoolean TccpAgent -> TccpChoice .
op _+_     :  TccpChoice TccpChoice -> TccpChoice [assoc comm] .
```

Note that the operator `_+_` is labeled with the attributes `assoc` and `comm` since it is associative and commutative.

The parallel agent is encoded by using the constructor symbol `||` composed by two agents.

```
op _||_ :  TccpAgent TccpAgent -> TccpAgent [assoc] .
```

The rest of tccp agents are encoded similarly. The system models all the agents appearing in Figure 1, including those introduced in [10].

3.2 The Operational Semantics

The operational semantics of tccp are encoded in Maude as transitions over configurations by means of Maude rules. For readability, each rule is labeled with an identifier. One configuration (of sort `TccpConfig`) contains

---

[5] It is possible to interact with Maude from other platforms, for example from Java.

a triple with the set of declarations of the given `tccp` program (a term of sort `TccpDeclarationSet`), the agent to be executed (a term of sort `TccpAgent`) and the current store (a term of sort `TccpStructuredStore`):

```
op < _, _, _ > :  TccpDeclarationSet TccpAgent TccpStructuredStore ->
                  TccpConfig .
```

The (structured) store at a given time instant is encoded as a store (a term of sort `TccpStore`) and a natural number between brackets denoting the current time instant:

```
        op _{_} :  TccpStore Nat -> TccpStructuredStore .
```

In the following, we show how some of the semantics' rules are specified in Maude. Let us start by describing the case for the conditional agent, modeled by the Maude conditional rules `now-true` and `now-false`. Maude conditional rules are of the form `crl` T `=>` T' `if` C and state that one term T rewrites (`=>`) to a second term T' whenever the condition C is satisfied. Then, the `now-true` rule shown below is executed just when its condition holds. In other words:

(i) the store `TpSt` represents all the information stored in `SS{t}` up to the current time instant,

(ii) `TpSt` satisfies the constraint `CtBl` of the conditional `tccp` agent. This is checked by means of the function `consultTccpStore (TpSt , CtBl)`, and

(iii) `Ag1'` is the result of executing `Ag1` with the current store.

```
crl [now-true]:  < DcSt , now(CtBl) then Ag1 else Ag2 , SS{t} > =>
                 < DcSt , Ag1' , SS1{k} >
    if TpSt := returnGlobalStoreFromStructuredStoreList (SS{t}) ∧
       consultTccpStore (TpSt , CtBl) == ctrue ∧
       < DcSt , Ag1 , SS{t} > => < DcSt , Ag1' , SS1{k} > .
```

`consultTccpStore (TpSt , CtBl)` gets as input the store `TpSt` and the boolean constraint `CtBl` and returns `ctrue` when the store entails the given constraint or `cfalse` otherwise. When the condition holds, we reach the configuration `< DcSt , Ag1' , SS1{k}` resulting of executing the agent in the `then` branch of the conditional agent. The `now-false` rule is defined similarly:

```
crl [now-false]:  < DcSt , now (CtBl) then Ag1 else Ag2 , SS{t} > =>
                  < DcSt , Ag2' , SS2{k} >
    if TpSt := returnGlobalStoreFromStructuredStoreList (SS{t}) ∧
       consultTccpStore (TpSt , CtBl) == cfalse ∧
       < DcSt , Ag2 , SS{t} > => < DcSt , Ag2' , SS2{k} > .
```

The following code excerpt describes the rules modeling the semantics of the choice agent. The rule `ask-true` specifies the case when a choice agent with a single branch can be executed. In this case, the agent `ask(CtBl) -> Ag` evolves to a configuration containing the original declarations set `DcSt`, the agent to be executed `Ag` and the structured store resulting from updating the given store `SS{t}` with the empty store (`strue`). The symbol `=>` is also used to incrementally identify the components of a structured store. The conditional rule states that the agent is executed only when the store `TpSt` satisfies the constraint `CtBl` of the agent, checked by means of `consultTccpStore (TpSt , CtBl) == ctrue`.

```
crl [ask-true]:  < DcSt , ask(CtBl) -> Ag , SS{t} > =>
                 < DcSt , Ag ,(SS{t} => strue {t + 1})>
    if TpSt := returnGlobalStoreFromStructuredStoreList (SS{t}) ∧
       consultTccpStore (TpSt , CtBl) == ctrue .
```

The conditional rule `choice-true` models the case when the choice agent has more than one branch and one of them can be executed. Since the operator `_+_` is associative and commutative, we have to describe just the first branch (`ask(CtBl) -> Ag`) of the given agent:

```
crl [choice-true]:  < DcSt , ((ask(CtBl) -> Ag) + AgCh) , SS{t} >
                    => < DcSt , Ag , (SS{t} => strue {t + 1}) >
    if TpSt := returnGlobalStoreFromStructuredStoreList (SS{t}) ∧
       consultTccpStore (TpSt , CtBl) == ctrue .
```

Finally, the `choice-false` rule models the case when the choice agent suspends, meaning that none of the constraints appearing in the choice agent `AgChS` is satisfied. In this case, the agent `AgChS` is executed in the following time instant:

```
crl [choice-false] :  < DcSt , AgChS , SS{t} > =>
                      < DcSt , AgChS , (SS{t} => strue {t + 1}) >
    if TpSt := returnGlobalStoreFromStructuredStoreList (SS{t}) ∧
       consultTccpStore (TpSt , AgChS) == cfalse .
```

The parallel agent is specified as the following conditional rule where it is reached the configuration resulting of executing, at the same time, the agents `Ag1` and `Ag2`. The execution of `Ag1` produces `Ag1'` and the structured store $\texttt{SS}_1\{\texttt{k}\}$, whereas the execution of `Ag2` produces `Ag2'` and the structured store $\texttt{SS}_2\{\texttt{k}\}$. Then, the resulting configuration contains the parallel composition of both `Ag1'` and `Ag2'`, and the structured store resulting of joining the structured stores: $(\texttt{SS}_1 \wedge \texttt{SS}_2)\{\texttt{k}\}$).

```
crl [parallel]:  < DcSt , Ag1 || Ag2 , SS{t} > =>
                 < DcSt , Ag1' || Ag2', (SS1 ∧ SS2){k} >
    if < DcSt , Ag1 , SS{t} > => < DcSt , Ag1' , SS1{k} > ∧
       < DcSt , Ag2 , SS{t} > => < DcSt , Ag2' , SS2{k} > .
```

The remaining rules are defined similarly.

4 The underlying constraint solver

Other important point in the `tccp` framework is the interaction with the underlying constraint solver. Typically, the constraint solver must be able of solving arithmetic and boolean constraints, and to perform some operations with streams. These goals can be achieved in an elegant way implementing the constraint system in Maude. Once defined the types of the expressions and the syntax of the operators needed to handle constraints, we specify the rules describing the evolution of each possible combination, thus the satisfaction relation.

The sort `TccpArithmetic` is used to represent the data types for arithmetic operations:

```
subsorts Float TccpVariable < TccpArithmetic .
```

Currently, `TccpArithmetic` includes floating-point numbers and variables. The following operators represent the *sum*, *rest*, *multiplication* and *division* of two arithmetic terms (`TccpArithmetic`) returning another arithmetic term, respectively:

```
ops _+'_ _-'_ :  TccpArithmetic TccpArithmetic ->
                 TccpArithmetic [prec 33 gather (E e)] .
ops _*'_ _/'_ :  TccpArithmetic TccpArithmetic ->
                 TccpArithmetic [prec 31 gather (E e)] .
```

The attribute `prec` sets the precedence of the operators given as a natural number: a lower value indicates a tighter binding and the attribute `gather (E e)` restricts the precedence of `TccpArithmetic` terms that are allowed as arguments. Both mechanisms avoid possible ambiguities arising when parsing

`TccpArithmetic` terms.

The operation semantics for the constraint system is modeled by using Maude equations in the module `TCCP-STORE`. For example, we define a Maude equation to add two numbers, another equation to add a variable (of sort `TccpVariable`) whose value must be recovered from the current store and a number, etc. We have an operator `evalTccpArithmetic` that, given a `TccpArithmetic` expression and the current store, returns the expected result. In case that the expression cannot be evaluated, it returns the original expression. The following equation specifies the simple case when, given the store `TpSt`, we add two floating numbers: `Ft1` and `Ft2`.

```
eq evalTccpArithmetic (Ft1 +' Ft2 , TpSt) = Ft1 + Ft2 .
```

Below we show the case for the sum of two variables `TpVar1` and `TpVar2` given the store `TpSt`. By means of the operator `evalArithmeticVariableInStore` we can recover from the store the value of `TpVar1` and `TpVar2`. In case that both values are floating numbers, `evalArithmetic` returns the sum of both:

```
ceq evalTccpArithmetic (TpVar1 +' TpVar2 , TpSt) = Ft1 + Ft1
     if Ft1 := evalArithmeticVariableInStore (TpVar1 , TpSt) ∧
        Ft1 =/= noIsFloat ∧
        Ft2 := evalArithmeticVariableInStore (TpVar2 , TpSt) ∧
        Ft2 =/= noIsFloat .
```

When a computation cannot be taken, then the input expression is returned:

```
ceq evalTccpArithmetic (TpAr , TpSt) = TpAr [owise] .
```

The expression `TccpBoolean` is used to represent the data types needed to handle boolean constraints:

```
  subsorts TccpExpression TccpArithmetic < AuxConstraint .
  subsorts Float TccpConstant TccpVariable < TccpExpression .
  subsorts TccpTerm TccpStream < TccpExpression .
ops _<'_ _>'_ _=='_ _!='_ _<='_ _>='_ : AuxConstraint AuxConstraint ->
                                        TccpBoolean [prec 37] .
op _and'_ : TccpBoolean TccpBoolean -> TccpBoolean [prec 55 assoc comm] .
op _or'_ : TccpBoolean TccpBoolean -> TccpBoolean [prec 59 assoc comm] .
op not' (_) :  TccpBoolean -> TccpBoolean [prec 53] .
```

We can consult whether two numbers (`Float`) are equal, whether one is smaller than the other, one variable (`TccpVariable`) is greater or equal to another (their values are recovered from the store), etc. Regarding streams, we can recover the values stored in a stream, the current tail, and also the current value of a stream (the last added value).

5 Running the interpreter

The interface of our tool is guided in a Maude console. To run the tool, we have to use the Maude command *load* `file-name`:

```
Maude> load .../tccpInterpreter.maude
```

Once the interpreter is loaded, we can use the Maude commands to invoke actions. For example, we can use the command *red* `expression` to parse or to identify an expression (an entity of the language). The command checks the given expression and returns the type or the sort associated to it. In other words, it tries to reduce the given expression following the specified grammar. The following example shows the output of Maude when reducing a tccp agent.

```
Maude> red tell ('X :=' 1.)  .
reduce in TCCP-INTERPRETER : tell('X :=' 1.0) .
rewrites:  5 in 0ms cpu (0ms real) (~ rewrites/second)
result TccpAgent:  tell('X :=' 1.0)
```

`TCCP-INTERPRETER` is the main module of the tccpInterpreter. The example shows that the given expression is an agent of the language.

We can also use the command *rew* `expression` to explore the possible behavior of a tccp program. For example:

```
Maude> rew < DcSt , tell('C :=' 2.)  , (strue {0}) > .
rewrite in TCCP-INTERPRETER : < DcSt,tell('C :=' 2.0),strue{0} > .
rewrites:  16 in 0ms cpu (0ms real) (~ rewrites/second)
result TccpConfig:  < DcSt,skip,(strue{0}) => ('C :=' 2.0){1} >
```

The execution of the given tell agent creates a new structured store with the information `('C := 2.0){1}` that is added to the initial store `strue{0}`.

Finally, the *search* command allows us to explore the reachable state space in different ways. We write:

```
search Term1 =>* Term2 .
```

to carry out the proof from the term `Term1` consisting of none, one, or more steps (=>*) to the pattern `Term2` to be reached.

5.1 Illustrative example

Here we describe a more elaborated example of an interaction with the tccpInterpreter system. In Figure 2 we show the specification in tccp of a part of a microwave oven controller that we have borrowed from [6]. To make the description clearer we show a labeled version of the declaration. Labels appear within braces { }:

```
{D} {ld} microwave_error(Door,Button,Error) :-
     {le0}∃ D,B,E ({lp1}({lt2}tell(Error=[_|E]) ‖
                   {lp3}({lt4}tell(Door=[_|D]) ‖
                   {lp5}({lt6}tell(Button=[_|B]) ‖
                   {lp7}({ln8}now(Door=[open|D] ∧ Button=[on|B]) then
                              {lp9}({le10}∃E1({lt11}tell(E=[yes|E1])) ‖
                                    {le12}∃B1({lt13}tell(B=[off|B1])))
                             else{le14}∃E1({lt15}tell(E=[no|E1])) ‖
                                 {lc16}microwave_error(D,B,E))))).
```

Fig. 2. The `microwave_error` declaration in tccp.

The declaration D models the process of detecting whether the door of the microwave is open at the same time that it is turned-on. This situation is controlled by the conditional agent in `ln8`. In case the condition holds, the process forces (with the tell agent in `lt13`) the microwave to be turned-off in the following time instant. Moreover, an error signal must be emitted (agent `lt11`). If the condition does not hold, then the system emits (via another tell agent `lt15`) a signal of *no error* that will be available in the store at the following time instant. These signals may be captured by other processes, thus it can be seen that the store allows the synchronization of processes. Finally, the procedure call agent `microwave_error(D,B,E)` models the recursion of the system.

By using the following command in the Maude console, once loaded the tccpInterpreter, the system simulates the behavior of the given declaration D[6].

```
Maude> search < D,'microwave_error (['open|'_],['on|'_],['no|'_]),
                        (strue{0}) > =>* < D,Ag,St > .
```

The first term specifies the configuration, composed by the declaration D, the procedure call agent `'microwave_error (['open|'_],['on|'_],['no|'_])` and the empty store at time instant 0 (`strue{0}`). The proof consists in reaching the second term that specifies the configuration containing D, an agent `Ag` and the structure store `St`. By using the non-instantiated variables `Ag` and `St` we can simulate the behavior of the given procedure call agent at each time unit. Note that we can perform a different proof by using a specific agent or a specific structured store in the second term.

The recursive procedure call agent (`lc16`) causes the system not to end, but this is the expected behavior in the tccp execution model. Therefore, we have to deal with infinite sets of states. To make the execution finite, we can use the Maude debugging feature [4] to capture each step of the computation, or to use a ceiling of time-units in the evolution of a tccp specification.

In the following we show a part of the Maude output for the execution

---

[6] For readability, we use D instead of the whole code of the declaration.

of the command described previously. It shows the resulting store at time instant `2`. In the execution graph, at time instant `0` the store is empty. At time instant `1`, the store contains the information resulting by the procedure call in the first term, where the parameters of the call are instantiated. Finally, at time instant `2` the store contains the information added by the tell agents `lt11` and `lt13` (the constraint of the conditional agent `ln8` is satisfied), and the information added by the second procedure call `lc16`:

```
(strue {0}) =>
((('Button :=' ['on | 'TailStr']) ('Door :=' ['open | 'TailStr])
  ('Error :=' ['no | 'TailStr'']))  {1}) =>
(('B :=' ['off | 'B1]) ('Button' :=' 'B) ('E :=' ['yes | 'E1])
  ('Error' :=' 'E) ('TailStr :=' 'D) ('TailStr' :=' 'B)
  ('Door' :=' 'D) ('TailStr'' :=' 'E)) {2} => ...
```

The system returns the final configuration reached by the given specification when it ends. The most relevant information in the configuration is the resulting structured store which can be used later to reason with the given specifications.

6 Conclusions and Future Work

We have presented the tccpInterpreter system, an interpreter for the tccp language that, given the specification of a tccp program, is able to simulate the corresponding behavior of such program following the semantics of the language. It has been implemented in Maude, an executable rewriting logic language that allows a precise specification of tccp describing, in a intuitive way, all the entities of the language such as the underlying constraint system, agents, and its operational semantics.

We have presented how the Maude system can be used as a semantic framework and metalanguage to build an entire environment and mechanisms for the execution of the formal specification language tccp. Maude leads to an perspicuous formulation in the task of specifying transition systems. It presents a rich notation supporting formal specification and implementation of concurrent systems. In this paper, we demonstrate the feasibility and the interest of formalizing the behavior of tccp with the Maude language.

We have described the functionality of tccpInterpreter by using a practical example. The tool is publicly available at the url http://www.dsic.upv.es/~villanue/tccpInterpreter/ and http://www.dsic.upv.es/~alescaylle/tccp.html. To our knowledge, there was no adequate and public implementation of tccp so far.

One of the important advantages of this implementation is that once we have the tccp language encoded in Maude, we can use the Maude related-tools to reason about tccp programs, for example, for model checking. This interpreter allows us to explore the particular features of tccp and its behavior (maximal parallelism and the underlying constraint system).

We plan to extend our tool in several ways. To improve the interface of the system we plan to construct a graphical web interface. We plan to study both, how to carry out the implementation of the model-checking algorithm proposed in [6] for tccp programs, and how to adjust the Maude's model-checker to verify tccp programs. In this way we aim at comparing both approaches to determine which one is the most appropriate in terms of efficiency.

Acknowledgement

We gratefully thank Pedro Ojeda Forés for his unconditional help in the implementation of the tccpInterpreter system.

References

[1] Maude Web Site. http://maude.csl.sri.com/, 2009.

[2] M. Alpuente, M. M. Gallardo, Ernesto Pimentel, and A. Villanueva. Verifying Real-Time Properties of tccp Programs. *Journal of Universal Computer Science*, 12(11):1551–1573, 2006.

[3] M. Alpuente, M.M. Gallardo, E. Pimentel, and A. Villanueva. A semantic framework for the abstract model checking of tccp programs. *Theoretical Computer Science*, 346(1):58–95, 2005.

[4] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[5] F. S. de Boer, M. Gabbrielli, and M. C. Meo. A Temporal Logic for Reasoning about Timed Concurrent Constraint Programs. In *TIME '01: Proceedings of the Eighth International Symposium on Temporal Representation and Reasoning (TIME'01)*, page 227, Washington, DC, USA, 2001. IEEE Computer Society.

[6] M. Falaschi and A. Villanueva. Automatic Verification of Timed Concurrent Constraint Programs. *Theory and Practice of Logic Programming*, 6(3):265–300, May 2006.

[7] A. Farzan, F. Cheng, J. Meseguer, and G. Rosu. Formal analysis of Java programs in JavaFAN. In *Proceedings of Computer-aided Verification (CAV'04)*, volume 3114 of *LNCS*, pages 501 – 505, 2004.

[8] D. Harel and A. Pnueli. On the development of reactive systems. pages 477–498, 1985.

[9] S. Haridi, P. Van Roy, P. Brand, and C. Schulte. Programming Languages for Distributed Applications. *New Generation Computing*, 16(3):223–261, 1998.

[10] A. Lescaylle and A. Villanueva. Verification and Simulation of protocols in the declarative paradigm. Technical report, DSIC, Univeridad Politécnica de Valencia, 2008. Available at http://www.dsic.upv.es/~alescaylle/files/dea-08.pdf.

[11] L. Logrippo, M. Faci, and M. Haj-Hussein. An introduction to LOTOS: learning by examples. *Comput. Netw. ISDN Syst.*, 23(5):325–342, 1992.

[12] N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. 4(1):1–36, January 2004.

[13] R. Milner. *Communication and concurrency.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[14] V. A. Saraswat, R. Jagadeesan, and V. Gupta. Foundations of Timed Concurrent Constraint Programming. In *Proc. of LICS'94*. IEEE CS, 1994.

[15] T. Sjöland, E. Klintskog, and S. Haridi. An interpreter for Timed Concurrent Constraints in Mozart (extended abstract). 2001.

[16] A. Verdejo. A tool for Full LOTOS in Maude. Technical Report 123-02, Dpto. Sistemas Informaticos y Programacion, Universidad Complutense de Madrid, 2002.

[17] A. Verdejo and N. Marti-Oliet. Implementing CCS in Maude 2. Technical report, 2002.