The tccp Interpreter

Alexei Lescaylle\textsuperscript{1,2}

\textit{DSIC, Universidad Politécnica de Valencia}\nValencia, Spain

Alicia Villanueva\textsuperscript{3}

\textit{DSIC, Universidad Politécnica de Valencia}\nValencia, Spain

\begin{abstract}

The \textit{Timed Concurrent Constraint Language} (tccp in short) is a constraint-based concurrent language inspired in process algebra. The language is well-suited for the specification of concurrent and reactive systems. tccp is parametric w.r.t. a constraint system, what is a main characteristic of the \textit{Concurrent Constraint Paradigm} of Saraswat. \textit{Maude} is an executable rewriting logic language specially well suited for the specification of distributed systems. The \textit{tccp Interpreter} system is the result of implementing the tccp language (the constraint system, agents and semantics) in \textit{Maude}. It parses the program and mimics in an automatic way its behavior allowing us to use the \textit{Maude} features to execute and analyze tccp programs.

\textit{Keywords:} Tool demonstration, Timed Concurrent Constraint Language, Maude

\end{abstract}

\section{Introduction}

The \textit{Concurrent Constraint Programming} paradigm, ccp in short \cite{13}, is a simple but powerful model for expressing concurrent systems. Systems are specified as agents executing asynchronously and interacting by adding and checking constraints (partial information) in a \textit{store}. The \textit{Timed Concurrent Constraint Language}, tccp in short \cite{5}, extends the ccp paradigm with a notion of time. This extension makes the language suitable for modeling reactive systems \cite{7}, namely systems which maintain an ongoing information exchange with their environment at run-time.

\textit{tccp} combines the operational view of process algebra \cite{11} with a declarative perspective based upon first-order logic \cite{12}. The language is parametric w.r.t. a
constraint system which specifies the constraints that can be handled by telling or asking actions performed by the agents of the language during an execution.

The language has some particular features: a declarative and concurrent nature, a model based on agents, a notion of time that synchronizes all agents, and the non-determinism. The non-deterministic behavior of \texttt{tccp} allows us to have compact and precise specifications of systems whereas the \textit{agent-based} model provides an intuitive way to specify reactive and embedded systems. It allows to capture typical behaviors of these systems such as \textit{time-outs, time-delays} or \textit{watchdogs}. Furthermore, the notion of time makes possible to use the (constrained version of) Linear Temporal Logic (LTL) proposed in [5] to specify properties of \texttt{tccp} programs and that can be checked by a \textit{model checker\cite{3,2,6}}.

The rewriting logic-based and high-performance reflective specification language \texttt{Maude\cite{4,1}} has been proposed for the task of building and analyzing a wide range of applications. In particular, rewriting logic\cite{10} can deal with state and concurrent computations and has been used as a semantic framework for the task of giving executable semantics to a wide range of languages and models of concurrency. \texttt{Maude} supports structured theory specifications, algebraic data types and function specification in rich equational logics, a high-level formalization of models and their prototyping and analysis. In this work we assume that the reader has a basic knowledge of \texttt{Maude}.

To our knowledge, there is a unique prototype of interpreter for \texttt{tccp} that was implemented by using the Mozart-Oz language\cite{14}. Mozart-Oz\cite{8} is a multi-paradigm language allowing multi-threaded higher order programs to be directly executed in a distributed open system. The proposal in \cite{14} has some restrictions and is not publicly available. In this work we present \texttt{tccpInterpreter}, an interpreter for the \texttt{tccp} language implemented in \texttt{Maude}. The interpreter automatically parses a \texttt{tccp} program and simulates the evolution of a given \texttt{tccp} program. The system incorporates some notions from\cite{2} that make the \texttt{tccp} framework more flexible. We show how it is possible to implement in an intuitive and precise way the \texttt{tccp} formalism in \texttt{Maude}.

This work is organized as follows. In Section 2 we describe the implementation process of the interpreter, an excerpt of the semantics implementation and a specific constraint system. Section 3 is devoted to show the functionality of the tool by using an illustrative example. In Section 4 we draw the conclusions and future work.

2 The interpreter implementation

The \texttt{tccpInterpreter} system, a \texttt{Maude} application, is the result of the implementation of the \texttt{tccp} formalism, i.e., the language operational semantics plus a specific constraint solver. The tool takes as input the specification of a \texttt{tccp} program and simulates its behavior: it shows the evolution of agents in the program depending on the current store (called \textit{Structured Store\cite{4}}), that also evolves along the time.

\texttt{tccpInterpreter} consists of approximately 1080 lines of code divided in six \texttt{Maude}
modules. Each module models one or more of the entities of \texttt{tccp}: agents, constraints, streams, declarations, programs, the store, the underlying constraint system, the operational semantics, etc. \texttt{Maude} allows us to implement a constraint solver for the language or to use an existing one to handle constraints.\footnote{We can interact with \texttt{Maude} from other platforms, for example with Java.}

2.1 Syntactic objects

The representation of the syntax of \texttt{tccp} in \texttt{Maude} is quite intuitive for all \texttt{tccp} constructs. Let us first recall the syntax of the \texttt{tccp} language:

\[ A, A_1, A_2 ::= \text{skip} \mid \text{tell}(c) \mid \sum_{i=1}^{n} \text{ask}(c_i) \rightarrow A_i \mid \text{now } c \text{ then } A_1 \text{ else } A_2 \mid A_1 || A_2 \mid \exists x A \mid p(x) \]

A \texttt{tccp} program \( P \) consists of a set of declarations \( D \) of the form \( p(x) ::= A \) and an initial agent to be executed. In brief, the choice agent \( \sum_{i=1}^{n} \text{ask}(c_i) \rightarrow A_i \) models the non-determinism of the system. It checks whether the store satisfies the constraints \( c_i \) and non-deterministically executes (in the following time instant) one of the agents \( A_i \), provided its constraint \( c_i \) was satisfied. In case no condition \( c_i \) is entailed, the choice agent \texttt{suspends}. The conditional agent \texttt{now } \( c \text{ then } A_1 \text{ else } A_2 \) executes the agent \( A_1 \) if the store satisfies \( c \), otherwise executes \( A_2 \). It provides the ability to describe notions such as \texttt{timeout} or \texttt{preemption}. These notions are necessary to model reactive systems.

\texttt{tccp} has an implicit notion of time in its semantics. There are agents that consume one time unit for their execution. The choice agent is one of these consuming-time agents whereas the conditional agent is instantaneous. Let us show the operational semantics extracted from [2] associated to these two agents. It is given as a transition relation between configurations, where a configuration is composed of an agent and the current store \( st \). The first rule \textbf{R2} states that \( A_j \) is executed in the following time unit whenever \( st \) entails the condition \( c_j \). Regarding the conditional agent, \textbf{R3} models the case when the condition holds. In case that the agent \( A \) with the current store \( st \) can evolve in the agent \( A' \) and the new store \( st' \), then \( A' \) is executed in the following time instant. If \( A \) cannot evolve, it is executed in the next time instant (rule \textbf{R5}).

\[
\begin{align*}
\textbf{R2} & \quad \langle \sum_{i=0}^{n} \text{ask}(c_i) \rightarrow A_i, st \rangle_t \rightarrow \langle A_j, st \rangle_{t+1} & \text{if } 0 \leq j \leq n, st \models c_j \\
\textbf{R3} & \quad \langle A, st \rangle_t \rightarrow \langle A', st' \rangle_{t+1} & \text{if } st \models c \\
\textbf{R4} & \quad \langle \text{now } c \text{ then } A \text{ else } B, st \rangle_t \rightarrow \langle A', st' \rangle_{t+1} & \text{if } st \not\models c \\
\textbf{R5} & \quad \langle A, st \rangle_t \rightarrow \langle A, st \rangle_{t+1} & \text{if } st \models c \\
\textbf{R6} & \quad \langle \text{now } c \text{ then } A \text{ else } B, st \rangle_t \rightarrow \langle B, st \rangle_{t+1} & \text{if } st \not\models c 
\end{align*}
\]

The choice agent is encoded by using two \texttt{Maude} constructor symbols. The first one models the behavior of a single branch in a choice agent. The symbol \texttt{ask} is followed by a boolean constraint (sort \texttt{TccpBoolean}), the arrow \( \rightarrow \) and an agent...
Lescaylle and Villanueva

The second one models the composition of two or more branches:

\[
\text{op ask} \rightarrow : \text{TccpBoolean TccpAgent} \rightarrow \text{TccpChoice} \\
\text{op } + : \text{TccpChoice TccpChoice} \rightarrow \text{TccpChoice} \quad \text{[assoc comm]}
\]

Note that the definition of the operator \(+\) is labeled with the attributes \text{assoc} and \text{comm} since it is associative and commutative.

The conditional agent is encoded by using one \text{Maude} constructor symbol. It has the symbol \text{now} followed by a boolean constraint, the \text{then} block which contains an agent and the \text{else} block with another agent.

\[
\text{op now then else : TccpBoolean TccpAgent TccpAgent} \rightarrow \text{TccpAgent}
\]

The rest of the agents of \text{tccp} are encoded in a similar way. We also model the new agents introduced in [9] to mechanize some operations over \text{streams}, used in \text{tccp} to model the evolution of variable values along the time.

\subsection{The Operational Semantics}

The operational semantics of \text{tccp} are encoded into \text{Maude} as transitions over configurations where one configuration contains a triple with the given \text{tccp} program (\text{TccpDeclarationSet}), the agent to be executed and the current store. By using the \text{Maude} constructor symbol \langle \_ , \_ , \_ \rangle we represent a configuration of the system:

\[
\text{op } \langle \_ , \_ , \_ \rangle : \text{TccpDeclarationSet TccpAgent TccpStructuredStore} \rightarrow \text{TccpConfig}
\]

The following code excerpt describes the rules modeling the semantics of the choice agent. Each rule is labeled with an identifier for readability. The rule \text{ask-true} specifies the case when a choice agent with a single branch can be executed. In this case, the agent \text{ask(CtBl)}\rightarrow \text{Ag} evolves to a configuration (on the right-hand side of the =\rightarrow symbol) containing the original declaration set \text{DcSt}, the agent to be executed \text{Ag} and the structured store resulting from updating \text{SS\{t\}} with the empty store (\text{strue}) since the choice agent adds no new information. The symbol \Rightarrow is used to incrementally identify the components of a structured store.

\[
\text{crl [ask-true]: } \langle \text{DcSt} , \text{ask(CtBl)}\rightarrow \text{Ag} , \text{SS\{t\}} \rangle =\rightarrow \\
\langle \text{DcSt} , \text{Ag} , (\text{SS\{t\}} \Rightarrow \text{strue \{t + 1\}}) \rangle
\]

if \text{TpSt} := \text{returnGlobalStoreFromStructuredStoreList (SS\{t\})} ∧ \text{consultTccpStore (TpSt , CtBl)} == \text{ctrue}.

The transition is modeled using a conditional rule, thus it is executed only when the store \text{TpSt}, representing all the information stored in \text{SS\{t\}} so far, satisfies the constraint \text{CtBl} of the agent, checked by means of \text{consultTccpStore (TpSt , CtBl)} == \text{ctrue}. The operator \text{consultTccpStore} gets as input the store \text{TpSt} and the boolean constraint \text{CtBl}, and it returns \text{ctrue} when the store entails the given constraint or \text{cfalse} otherwise.

The conditional rule \text{choice-true} specifies the case when the choice agent has more than one branch and one of them can be executed. Note that the operator \text{+} is associative and commutative, thus we can describe the first branch of the agent

\footnote{\{t\} denotes the current time instant t.}
+, considering the rest of the branches in the second component \( \text{AgCh} \):

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

Finally, the \text{choice-false} rule models the case when the choice agent suspends, meaning that none of the constraints appearing in the choice agent \( \text{AgChS} \) is satisfied by the store \( (\text{consultTccpStore} (\text{TpSt} , \text{AgChS}) == \text{cfalse}) \). In this case, the agent \( \text{AgChS} \) is executed in the following time instant:

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

The rules for the implementation of the semantics of the conditional agent are specified in a similar way. They model the case when the store satisfies the constraint of the agent (rule \text{now-true}) and the case when it does not (rule \text{now-false}). In the first case, the rule evolves to the configuration resulting of executing the agent in the \text{then} part (Ag1). The execution of \( \text{Ag1} \) produces \( \text{Ag1}' \) and the structured store \( \text{SS1}\{k\} \) where \( k > t \). In the second rule, the agent \( \text{Ag2} \) is executed since the constraint was not entailed. In this second case, \( \text{Ag2} \) produces \( \text{Ag2}' \) and the structured store \( \text{SS2}\{k\} \).

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

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

The rest of the rules describing the operational semantics of the language are defined similarly.

2.3 The underlying constraint solver

Other important point in the \text{tccp} framework is the implementation of the constraint solver. In our case, 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 \text{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.

The expression \( \text{TccpArithmetic} \) is used to represent the data types for arithmetic operations:
Lescaylle and Villanueva

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:

\[
\begin{align*}
\text{ops} & \quad \mathtt{+_\ast/} : \ TccpArithmetic \ TccpArithmetic \to \\
& \quad \text{TccpArithmetic} \ [\text{prec} \ 33 \ \text{gather} \ (E \ e)] . \\
\text{ops} & \quad \mathtt{-_\ast/} : \ TccpArithmetic \ TccpArithmetic \to \\
& \quad \text{TccpArithmetic} \ [\text{prec} \ 31 \ \text{gather} \ (E \ e)] . 
\end{align*}
\]

The attribute \text{prec} sets the precedence of the operators given as a natural number, where a lower value indicates a tighter binding and the attribute \text{gather} \ (E \ e) restricts the precedences of \text{TccpArithmetic} terms that are allowed as arguments. Both mechanisms avoid possible ambiguities arising in the parsing of \text{TccpArithmetic} terms.

The result of each operator is modeled by using Maude equations depending on all the possible combinations that may be generated. For example, we need an equation to add two numbers, we need an equation to add a variable and a number and viceversa, etc. We have an operator \text{evalTccpArithmetic} that, given a \text{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 rule specifies the simple case when, given the store TpSt, we add two floating numbers, Ft1 and Ft2:

\[
eq \text{evalTccpArithmetic} \ (Ft1 +\ Ft2 , \ TpSt) = Ft1 + Ft2 .
\]

The following rule specifies when, given the store TpSt, we add two variables: TpVar1 and TpVar2. By means of the operator \text{evalArithmeticVariableInStore} we can recover from the store the value of TpVar1 and TpVar2. In case that both values are floating numbers, \text{evalArithmetic} returns the sum of both:

\[
\begin{align*}
\text{ceq} & \quad \text{evalTccpArithmetic} \ (TpVar1 +\ TpVar2 , \ TpSt) = Ft1 + Ft1 \\
& \quad \text{if} \ Ft1 := \text{evalArithmeticVariableInStore} \ (TpVar1 , \ TpSt) \ \land \\
& \quad \text{Ft1} /= noIsFloat \ \land \\
& \quad \text{Ft2 := evalArithmeticVariableInStore} \ (TpVar2 , \ TpSt) \ \land \\
& \quad \text{Ft2} /= noIsFloat .
\end{align*}
\]

The following rule states that when no previous rule can be taken, then the input expression is returned:

\[
\text{ceq} \quad \text{evalTccpArithmetic} \ (TpAr , \ TpSt) = TpAr [\text{owise}] .
\]

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

\[
\begin{align*}
\text{subsorts} & \quad \text{TccpExpression} \ TccpArithmetic \ < \ \text{AuxConstraint} . \\
\text{subsorts} & \quad \text{Float} \ TccpConstant \ TccpVariable \ < \ \text{TccpExpression} . \\
\text{subsorts} & \quad \text{TccpTerm} \ TccpStream \ < \ \text{TccpExpression} .
\end{align*}
\]
In this way, we can consult whether two numbers (Float) are equals, 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 perform certain operation with them. For instance, we can recover the values stored in a stream, the current tail, and also the current value of a stream (the last added value).

3 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-SEMANTICS : tell('X :=' 1.0) .
rewrites: 5 in 0ms cpu (0ms real) (~ rewrites/second)
result TccpAgent: tell('X :=' 1.0)

TCCP-SEMANTICS is a module of the tccpInterpreter. The example shows that the given expression is an agent (in particular of sort TccpAgent) 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-SEMANTICS : < DcSt , tell('C :=' 2.0) , strue{0} > .
rewrites: 16 in 0ms cpu (0ms real) (~ rewrites/second)
result TccpConfig: < DcSt , skip , (strue{0}) \Rightarrow ('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 that has to be reached Term2.
3.1 Illustrative example

Here we describe a more elaborated example of an interaction with the tccp Interpreter system. In Figure 1 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 {}:

\[
\begin{align*}
\{D\} & \{ld\} \text{ microwave_error(Door,Button,Error) :- } \\
& \{le0\} \exists D,B,E \ (\{lp1\}\{lt2\}tell(\text{Error=}[_{E1}] ) \ || \\
& \{lp3\}\{lt4\}tell(\text{Door=}[_{D}] ) \ || \\
& \{lp5\}\{lt6\}tell(\text{Button=}[_{B}] ) \ || \\
& \{lp7\}\{ln8\}\text{now}(\text{Door=}[_{open}|D] \land \text{Button=}[_{on}|B]) \text{ then} \\
& \{lp9\}\{le10\}\exists E1(\{lt11\}tell(\text{E=}[_{yes}|E1]) ) \ || \\
& \{le12\}\exists B1(\{lt13\}tell(\text{B=}[_{off}|B1]) ) \\
& \text{else}\{le14\}\exists E1(\{lt15\}tell(\text{E=}[_{no}|E1]) ) \ || \\
& \{lc16\}\text{microwave_error}(D,B,E)))).
\end{align*}
\]

Fig. 1. 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 \(ln8\). In case the condition holds, the process forces (with the \(tell\) agent \(lt13\)) 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 \(D\)\(^7\).  

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

\(^7\) For readability, we use \(D\) instead of the entire code of the declaration.
time-units in the evolution of a tccp specification.

In the following we show a part of the Maude output for the execution of the command described previously. It shows the resulting store at the time instant 2. In the execution graph, at the time instant 0 the store is empty. At the 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 the 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, and so on:

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

The system returns the final configuration reached by the given specification when it ends. The important element of the configuration is the resulting structured store which can be used later to reason with the given specifications.

4 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. The generated Maude descriptions have been validated using the platform supporting this 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 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 can establish a comparison which determines which approach is the most appropriate.

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 4590 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] D. Harel and A. Pnueli. On the development of reactive systems. pages 477–498, 1985.

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

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

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

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

[12] C. Olarte and F. D. Valencia. The expressivity of universal timed CCP: undecidability of Monadic FLTL and closure operators for security. In PPDP ’08: Proceedings of the 10th international ACM SIGPLAN conference on Principles and practice of declarative programming, pages 8–18, New York, NY, USA, 2008. ACM.

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

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