Verification and Simulation of protocols in the declarative paradigm

CANDIDATE: Alexei Lescaylle

SUPERVISOR: Alicia Villanueva

– October 2008 –

This work has been supported by the Spanish MEC/MICINN under grant TIN2007-68093-C02-02, HA2006-0007, and by the Universidad Politécnica de Valencia Valencia, program PAID-06-07 (TACPAS).
Author’s address:

Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de Vera, s/n
46022 Valencia
España
# Contents

1 Introduction .............................................. 3

2 The \texttt{tccp} Language ................................. 7
   2.1 The new computational model ......................... 9
   2.2 An extended computational model for \texttt{tccp} .......... 10
       2.2.1 The new agents .............................. 12

3 Modeling Communication Protocols in the Timed Concurrent
   Constraint Language .................................. 13
   3.1 Protocol specification using \texttt{tccp} ............. 13
       3.1.1 Basic Terms and Common Actions .............. 14
       3.1.2 Environment .................................. 15
       3.1.3 Communication Protocols ..................... 18
       3.1.4 Principals .................................. 19
   3.2 Verification ...................................... 23
       3.2.1 Checking the Needham-Schroeder Protocol ..... 24
       3.2.2 Checking the Otway-Rees protocol .......... 24

4 A tool for Generating a Symbolic Representation of \texttt{tccp} executions .................. 27
   4.1 A symbolic representation of \texttt{tccp} executions .... 27
       4.1.1 The implementation .......................... 28
   4.2 Practical examples ................................ 31
       4.2.1 The \texttt{microwave_error} System ............. 31
       4.2.2 The \texttt{build} System ....................... 35

5 Conclusions and Future Work ......................... 39

Bibliography .............................................. 41
Abstract

The Concurrent Constraint Paradigm (ccp, Saraswat 1993) is a powerful model based on the notion of store-as-constraint instead of the classical notion of store-as-valuation. This means that its computational model is based on states consisting of a combination of constraints, opposite to the traditional model where states consist of a valuation of variables. This notion of state achieves a natural compression in the space state.

The Timed Concurrent Constraint language (tccp in short) is a declarative, concurrent and non-deterministic programming language defined by F. de Boer et al. as an extension of the ccp where the authors introduced some features such as a notion of time and new agents to specify reactive and embedded systems.

In this work, we propose tccp as a specification language for communication protocols. To this end, we endow the language with some new agents that make the modeling of protocols clearer. We adopt the popular Dolev-Yao intruder model. Moreover, we present the StructGenerator system which, given the specification of a tccp program, constructs a symbolic representation (a tccp Structure) modeling the behaviour of such tccp program. The resulting structure is the input of a model-checking algorithm defined for tccp programs. Such structure can be seen as a variant of a Kripke Structure where the features of the ccp framework impose some difficulties when dealing with equivalence of states. Finally, we cover the design and implementation of the StructGenerator system.

With the aim to help the understanding of our contribution, we show along this work some illustrative examples.
0. Abstract
1 Introduction

The Concurrent Constraint Paradigm (ccp in short) [27] is a simple but powerful model based on the notion of store-as-constraint instead of the classical notion of store-as-valuation. Therefore, the computational model is based on states which are composed of a conjunction of constraints instead of being defined as a valuation of variables. The Timed Concurrent Constraint language (tccp in short) was introduced in [6] as an extension of ccp for specifying reactive systems in an intuitive way. The authors introduced a notion of time within the language semantics and new agents to handle negative information, i.e., information that is not present in the system. Negative information is needed for modeling behaviours such as time-outs or preemption in reactive systems.

During the last years, verification techniques for concurrent and reactive systems have been widely developed. The model-checking technique [13] is a formal technique that allows one to verify whether a property is satisfied by a model. This technique suffers the so-called state explosion problem which motivates the development of many optimization approaches to mitigate it. We can find in the literature many optimizations based on abstract interpretation, partial order, symbolic representations, etc. for different modeling and (fragments of) programming languages. The ccp framework in general, and the tccp language in particular, thanks to the store-as-constraint approach, symbolically represents sets of classical states as conjunctions of constraints, thus achieving a natural compression of the search space. Note that this fact does not prevent one from applying other optimization techniques such as [1, 2].

This work is divided in two parts, in the first one, we propose tccp as a suitable specification language for communication protocols since it poses certain features such as its declarative and concurrent nature, the notion of time within its semantics, non-determinism, etcetera. In particular, the non-deterministic behaviour of tccp allows us to have compact and precise specifications of systems whereas the agent-based model, which is enriched in this work with some new agents, provides an intuitive way to specify both, actions that are basic to any protocol, and protocol participants. Note that the notion of time makes possible the interaction among principals and to use the (constrained version of) Linear
Temporal Logic (LTL) [14] to model-check temporal properties over tccp protocol specifications. To this end, we can use the formal verification techniques defined for tccp in [2, 3, 17].

We show our approach to the protocol specification and verification problem as follows. We first provide some auxiliary operations that are usually performed during a protocol run. Following the Dolev-Yao model [16], we present a general definition of the environment (the model for the network) where protocols are executed. We also give a generic guideline for the specification of the particular behaviour of protocol participants. We use an asynchronous model for handling messages between participants. This means that each time a message is sent, two asynchronous actions are performed: a principal sends the message, and, maybe later, a second principal receives it. Finally, we show how to reason about security properties by using the LTL logic proposed in [14]. In particular, we are able to check whether there exists a state where an intruder knows or discloses a secret. We illustrate our proposal modelling the behaviour of the Needham-Schroeder public key authentication protocol [23] and the Otway-Rees symmetric key authentication protocol [26].

In the second part of this work, we present the StructGenerator system, which automatically constructs a symbolic representation (called tccp Structure) of a tccp program. The tccp Structure was defined in the framework of a model-checking algorithm that was proposed in [17] for the verification of tccp programs. This structure, which is the input of the verification phase, is a variant of the Kripke Structure. Due to the nature of the ccp model, where nodes consist of a composition of constraints, the relation among states in the construction of the graph is non trivial due to the monotonic behaviour of the store in tccp. To make the tccp framework more flexible, we use a new version of the computational model of tccp so the model-checking algorithm differs in some aspects from the one proposed in [17], by incorporating some notions from [3]. We also describe the design and implementation of the StructGenerator system and we demonstrate its functionality carrying out the execution of two practical examples.

Related work. There are many approaches related to the specification and verification of communication protocols [20, 12]. The closest work to ours is the recent paper [25], where the expressiveness of the utcc language is illustrated by specifying a security protocol. Our work differs from [25] in that, even if both languages are descendants from ccp, their features and nature are quite different: tccp is non-deterministic whereas utcc is not; utcc uses replication whereas tccp does not, etc. These differences make the specification of systems different in the two languages.

In the literature, we find also logic programming languages used to analyze communication protocols. For instance, [21] implements a reachabilty analysis

Other declarative languages have also been shown suitable for the protocol specification and analysis problems. In [15], the specification language Maude was shown as an alternative approach. Haskell has also been considered for analyzing communication protocols. In [4] we can find how Maude and Haskell can be used for the modeling of protocols.
2

The tccp Language

The Timed Concurrent Constraint Language (tccp) is a concurrent declarative language defined in [5] as an extension of the ccp framework [27]. In the ccp paradigm, the notion of store as valuation from Von Neumann is replaced by the notion of store as constraint. The computational model is based on a global store where constraints are accumulated and a set of agents that may interact with others via the store. The ccp model is non-deterministic and there is no notion of time defined. Intuitively, the execution of a tccp program evolves by asking and telling information to the store. It is a simple but powerful model in which partial information can easily be handled. The model is parametric w.r.t. a cylindric constraint system $C$. The interested reader can find in [5, 27] more details about the constraint system.

The temporal extension of tccp introduces a notion of time within the semantics, i.e., no special agent for time passing is defined. We can find other approaches to the temporal extension of ccp in the literature that make different design choices [28, 18, 24]. In tccp the actions of asking and telling information to the store consume time. In practice this means that both, consults and updates to the store, take time.

In tccp, it is possible to model behaviours typical from reactive and/or embedded systems. In these systems, the absence of information can cause the execution of an action. tccp introduces the conditional agent in order to model such behaviours. Let us briefly recall the syntax of the language:

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

where $c, c_i$ are finite constraints (i.e., atomic propositions) of $C$. A tccp process $P$ is an object of the form $D.A$, where $D$ is a set of procedure declarations of the form $p(\bar{x}) :- A$, and $A$ is an agent. Intuitively, the skip agent does nothing. tell($c$) adds the constraint $c$ to the store, being $c$ available only in the subsequent time instant. The choice agent $\sum_{i=1}^{n} \text{ask}(c_i) \rightarrow A_i$ checks whether the store satisfies the guards, and non-deterministically executes (in the following time instant) one of the agents $A_i$ provided its guard $c_i$ were satisfied. In case no condition $c_i$ is entailed, the choice agent suspends. The conditional agent (now $c$ then $A_1$ else
A2) executes agent A1 if the store satisfies c, otherwise executes A2. A1||A2 executes the two agents A1 and A2 in parallel (the concurrent model used is *maximal parallelism*). The ∃xA agent is used to define variables local to the process A. Finally, p(π) is the procedure call agent.

In the semantics of tccp, the only agents that consume time are the tell(c), ∑ni=1ask(ci) → Ai, and p(π). The store in the original tccp model can be seen as a blackboard where information is continuously written and never canceled, thus differently from other extensions. Stores grow monotonically, thus it is not possible to change the value of a given variable. In order to model the evolution of variable values along the time we use *streams* in a similar way as logical lists are used in logic languages. We write X = [Y | Z] for denoting a stream X recording the current value Y of an imperative-style variable, and the stream Z of future values of such variable.

This model presents the problem that we cannot retrieve in a simple way the order in which the information has been added to the store. Problem that arises since the information is added without keeping track of the insertion order so that we cannot recover the time instant when a constraint has been added.

The following example illustrates the execution of a tccp program.

\[
\text{fib}(X,Y) :- \\
\quad \text{now}(X = 1) \text{ then tell}(Y = 1) \text{ else} \\
\quad \text{now}(X = 2) \text{ then tell}(Y = 1) \text{ else} \\
\quad \exists X1,X2,Y1,Y2( \\
\quad \quad \text{tell}(X1 \text{ is } X - 1) || \\
\quad \quad \text{tell}(X2 \text{ is } X - 2) || \\
\quad \quad \text{ask(true)} \rightarrow \text{fib}(X1,Y1) || \\
\quad \quad \text{ask(true)} \rightarrow \text{fib}(X2,Y2) || \\
\quad \quad \text{ask(isNumber}(Y1) \land \text{isNumber}(Y2)) \rightarrow \text{tell}(Y \text{ is } Y1 + Y2)).
\]

By means of the predicate fib/2 we can calculate the sequence of numbers named *Fibonacci numbers*. The first and the second conditional agents (now) add to the store the constraint Y = 1, the result of the calculation, in case that the input number X is 1 or 2. When the second now does not holds, which means that X is greater than 2, we declare four local variables X1,X2,Y1 and Y2 to continue the calculation. To this end, we instantiate X1 to the result of the first subtraction (X1 is X - 1) and X2 to the result of the second subtraction (X2 is X - 2). In parallel, we let pass a time unit by using the ask(true) agent since the information added by the tell agents will be available in the next time instant and then we call the function recursively using both procedure call agents fib(X1,Y1) and fib(X2,Y2). The last choice agent will remain suspended while Y1 and Y2 are not numbers (they are not instantiated to their respective values); otherwise the resulting value Y is calculated and added to the store.
2.1 The new computational model

In [3], a new computational model was proposed in which a new notion of store, called *structured store*, was defined. A structured store consists of a timed sequence of stores where each store only contains the information added at a given time instant. It is defined as follows:

**Definición 2.1.1 (Structured Store [3])** A *structured store* is an infinite indexed sequence of stores where the \(i^{th}\) component of the sequence \(s_t\) is denoted as \(s^i_t\), and it represents the store at time \(i\).

Thus, we can observe and analyze the evolution of the structured store through time. The new computational model allows us to recover the order in which the information is added during a computation. In this work we consider that framework instead of the original one since, as we will show later, it makes simpler some tasks that are commonly performed by protocol participants and makes easy the implementation of the **StructGenerator** system.

Following [3], \(s^i_t\) represents the information added at the \(i^{th}\) time instant; \(s^i_t \models c\) checks whether the information stored up to the \(t^{th}\) time instant entails \(c\); finally, \(s^i_t \sqcup t\{c\}\) is used to add the constraint \(c\) at the \(t^{th}\) time instant.

Next, we show a example that combines the notion of structured store and the use of stream:

**Example 2.1.2** The structured store \(s^n_t\) is produced by the execution of a given program where, among several operations, we have that at each time instant \((n \in \mathbb{N})\) the stream \(X\) is incremented. Streams \(X^{n-1}\) range over \(\mathbb{N}\):

- \(s^0_t : X\)
- \(s^1_t : X = [0|X'], \ldots\)
- \(s^2_t : X' = [1|X''], \ldots\)
- \(s^3_t : X'' = [2|X'''], \ldots\)
- \(\ldots\)
- \(s^n_t : X^{n-1} = [n - 1|X^n], \ldots\)

Then, we can use the entailment relation \((\models)\) over \(s^n_t\) in order to reason with the given program, i.e., \(\text{ask}(X = 3) \rightarrow \text{skip}\), in this case the consult holds since in the time instant 3 \(X\) is equal to 2 so the program skip (it does nothing).
The interested readers can find more detailed examples in [3]. The operational semantics of the new model is shown in Fig. 2.1. We show the transition relation \( \rightarrow \) of the operational semantics where \( \rightarrow \in (A \times \text{STORE} \times N)^2 \), \( A \) is a set of tccp agents, and \( N \) is the domain of natural numbers. In the figure, the symbol \( \not\rightarrow \) is used to indicate that it is not possible to evolve by using relation \( \rightarrow \), i.e., the execution suspends.

| R1 | \( \langle \text{tell}(c), st \rangle_t \rightarrow \langle \text{skip}, st \sqcup \cdot t+1 \cdot c \rangle_{t+1} \) |
| R2 | \( \sum_{i=0}^{n} \langle \text{ask}(c_i), A_i, st \rangle_t \rightarrow \langle A_j, st \rangle_{t+1} \) if \( 0 \leq j \leq n \)
| R3 | if \( st \vdash_t c_j \) |
| R4 | \( \langle A, st \rangle_t \rightarrow \langle A', st' \rangle_{t+1} \) if \( st \not\vdash_t c \) |
| R5 | \( \langle A, st \rangle_t \not\rightarrow \) if \( st \not\vdash_t c \) |
| R6 | \( \langle A, st \rangle_t \rightarrow \langle A', st' \rangle_{t+1} \) if \( st \not\vdash_t c \) |
| R7 | \( \langle A, st \rangle_t \rightarrow \langle A', st' \rangle_{t+1} \) if \( st \not\vdash_t c \) |
| R8 | \( \langle A, st \rangle_t \rightarrow \langle A', st' \rangle_{t+1} \) if \( st \not\vdash_t c \) |
| R9 | \( \langle A, st \rangle_t \rightarrow \langle A', st' \rangle_{t+1} \) if \( st \not\vdash_t c \) |
| R10 | \( \langle p(x), st \rangle_t \rightarrow \langle A, st \rangle_{t+1} \) if \( p(x) : -A \in D \) |

Figure 2.1: Structured operational semantics.

Given a tccp program \( P \), an agent \( A_0 \), and an initial structured store \( st^0 = \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \cdot \·

2.2 An extended computational model for tccp

By using the new computational model described in [3], structured store, we can easily, for instance, recover the last value added to a stream, or add a new value to a stream. In the following, we present the definitions that allow us to ease these operations. Let us first recall the notion of current value from [3].
Definición 2.2.1 (Current Value [3]) Let \( X \) be a stream, \( st \) a structured store and \( t \in \mathbb{N} \). Then, \( V_C \) is the current value of \( X \) in \( st \) at instant \( t \), denoted by \( st \models_t X = V_C \), iff \( \exists m > 0 \) such that:

\[
\begin{align*}
st \models_t \exists A_1 \ldots \exists A_m \exists V_C \exists A s. & \ X = [A_1, \ldots, A_{m-1}, V_C|As] \text{ and } \\
st \not\models_t \exists A' \exists A s'. & \ A s' = [A'|As']
\end{align*}
\]

In addition, under these conditions, we also say that the length of stream \( X \) in \( st \) at time \( t \) is \( m \), in symbols \( \text{len}(X, st, t) = m \).

Now we define the notion of current tail, needed in this work to easily add values at the last position of a stream.

Definición 2.2.2 (Current Tail) Let \( X \) be a stream, \( st \) a structured store, and \( t \in \mathbb{N} \). Then, \( T_C \) is the current tail of \( X \) in \( st \) at instant \( t \), denoted by \( st \models_t X = T_C \), iff \( \text{len}(X, st, t) = m \) and \( m > 0 \) such that:

\[
\begin{align*}
st \models_t \exists A_1 \ldots \exists A_m \exists T_C. & \ X = [A_1, \ldots, A_m|T_C]
\end{align*}
\]

Next, we define the notion of value retrieved from a stream. We say that a given value can be retrieved from a stream if it unifies with one of the components of this stream.

Definición 2.2.3 (Retrieved Value) Let \( X \) be a stream, \( st \) a structured store and \( t \in \mathbb{N} \). Then, a given value \( V \) is retrieved from \( X \) in \( st \) at instant \( t \), denoted by \( st \models_t X \triangleright V \), iff \( \text{len}(X, st, t) = m \), \( m > 0 \) and \( V \) unifies (=) with some value of the stream \( X \), such that:

I) \( st \models_t \exists A_1 \ldots \exists A_m \exists T_C. \ X = [A_1, \ldots, A_m|T_C], \neg \text{free}(V), V \neq \ldots \), \( \exists i. V = A_i, 1 \leq i \leq m \)

II) \( st \models_t \exists A_1 \ldots \exists A_m \exists T_C. \ X = [A_1, \ldots, A_m|T_C], \neg \text{free}(V), V = [V_1, \ldots, V_n], \forall j \exists i. V_j = A_{i+(n-1)}, 1 \leq j \leq n, 1 \leq i \leq m \)

III) \( st \models_t \exists A_1 \ldots \exists A_m \exists T_C. \ X = [A_1, \ldots, A_m|T_C], \text{free}(V), V = A_i, 1 \leq i \leq m \)

The predicate free/\( T \) \( ^{1} \) is used to check whether a variable is non-instantiated in the store \( st \).

To be able to ask the constraint system whether a term can be retrieved from a stream in the store, we define a specific function as the interface of the Retrieved Value notion defined above.

Definición 2.2.4 (Find) Let \( S \) be a stream and \( V \) a given value, then \( \text{find}(S, V) \) returns true if \( V \) can be retrieved from the stream \( S \); otherwise returns false. Formally,

\[
\text{find}(S, V) = \begin{cases} 
\text{true} & \text{if } st \models_t S \triangleright V \\
\text{false} & \text{otherwise}
\end{cases}
\]

\(^{1}\)We assume that the constraint system has such predicate implemented.
2.2.1 The new agents

Many communication protocols need to generate nonces to guarantee freshness of messages. One way for generating these nonces is by recovering the last generated one and calculating the new nonce by applying to the recovered one a (more or less complicated) function. In this work, we follow this approach and, to make the specification simpler, we define two new tccp agents called ask-tell and update. These agents take advantage of the new computational model.

In tccp, consults do never add information to the store: They must be combined with tell agents to modify the store. The new ask-tell agent consults the store and updates it by instantiating a fresh variable to the value consulted:

Definición 2.2.5 Let $S$ be a stream and $V$ a fresh variable. The $\text{ask-tell}(S, V)$ agent recovers in $V$ the current value of the stream $S$. Moreover, it instantiates $V$ to such value. Formally,$^2$

\[ \begin{align*}
R_{12} & \quad \langle \text{ask-tell}(S, V), st \rangle_t \rightarrow \langle \text{skip}, st \rangle_{t+1} \quad \text{if } st \models_t \text{free}(S); \neg \text{free}(V) \\
R_{13} & \quad \langle \text{ask-tell}(S, V), st \rangle_t \rightarrow \langle \text{skip}, st \sqcup_{t+1} V = V_C \rangle_{t+1} \quad \text{if } st \models_t S \equiv V_C, \text{free}(V), \\
& \quad \quad \quad \text{len}(S, st, t) > 0
\end{align*} \]

When we use streams some operations that allow us to handle them are necessary, among them we have the values inserting in a stream and the values recovering from a stream. To this end, we define two agents, update and assign, which mechanize both process, respectively.

The new update agent adds a given value at the last position of a stream:

Definición 2.2.6 Let $S$ be a stream, $V$ a given value and $T$ a free variable. The $\text{update}(S, V, T)$ agent updates $S$ with $V$. Formally,

\[ \begin{align*}
R_{14} & \quad \langle \text{update}(S, V, T), st \rangle_t \rightarrow \langle \text{skip}, st \sqcup_{t+1} S = [V | T] \rangle_{t+1} \quad \text{if } \text{free}(S), \text{free}(T) \\
R_{15} & \quad \langle \text{update}(S, V, T), st \rangle_t \rightarrow \langle \text{skip}, st \sqcup_{t+1} T_C = [V | T] \rangle_{t+1} \quad \text{if } st \models_t S \equiv T_C, \\
& \quad \quad \quad \text{free}(T), \\
& \quad \quad \quad \text{len}(S, st, t) > 0
\end{align*} \]

Finally, we show the new agent assign, which is used to recover a value of the stream:

Definición 2.2.7 Let $S$ be a stream, $V_f$ a given value and $V_a$ a fresh variable. The $\text{assign}(S, V_f, V_a)$ agent instantiates $V_a$ to $V_f$ if the value $V_f$ can be retrieved from $S$. Formally,

\[ \begin{align*}
R_{16} & \quad \langle \text{assign}(S, V_f, V_a), st \rangle_t \rightarrow \langle \text{skip}, st \sqcup_{t+1} V_a = V_f \rangle_{t+1} \quad \text{if } st \models_t S \supseteq V_f, \text{free}(V_a)
\end{align*} \]

\[^2\text{We provide the operational semantics following the notation and model of [3].} \]
3

Modeling Communication Protocols in the Timed Concurrent Constraint Language

Designing communication protocols has demonstrated to be a difficult task due to some facts, among them, we have that (1) a protocol is executed in a complex environment (the network) where intruders may try to interfere in the protocol run, and (2) the level of abstraction chosen [22]. Therefore, there are multiple aspects that must be considered to analyze the correctness or security of a protocol since these aspects can arise many vulnerabilities. This situation makes it necessary to use formal approaches that allow one to automatize the design and verification of communication protocols.

In this chapter we describe how tccp can be used to model communication protocols. To achieve this goal, in Sections 3.1 we present the method to model some common actions that take place in different protocols, the environment (the network) and protocols principals. Finally, in Section 3.2 we describe how to specify properties related to the verification process.

3.1 Protocol specification using tccp

This section presents how to specify protocols in tccp. The specification can be divided in three parts: (1) some basic terms, and some common actions that usually take place during a protocol run; (2) the (hostile) environment which controls the protocol execution, and (3) the principals’ specific behaviour. (1) and (2) can be reused for any protocol since the (hostile) environment is endowed with the possible actions that an intruder can do during the execution of any protocol following the considered attacker model. The principals behaviour must be redefined for each different protocol since it depends on the protocol definition. We provide a guideline to transform its informal definition to the tccp specification.

We assume perfect cryptographic primitives. In particular, we assume that each principal can encrypt a message with the public key of any principal, and
one principal can decrypt a message only if it has the corresponding secret key.

### 3.1.1 Basic Terms and Common Actions

In this section, we first describe the basic terms used for the specification and then we introduce the common actions usually performed by principals.

The term \( \text{pair}(X, \text{pk}(X)) \) identifies the principal \( X \) as a participant of the protocol and his public key \( \text{pk}(X) \). \( \text{sk}(X) \) represents the secret key of principal \( X \). \( k(X, Y) \) specifies a key which is shared by two principals, \( X \) and \( Y \). \( \text{nonce}(X, S_n) \) stores in the stream \( S_n \) the nonces generated by principal \( X \). \( \text{key}(X, S_k) \) stores in the stream \( S_k \) the keys generated by principal \( X \). \( \text{cons}(C_1, C_2) \) models the composition of two components \( C_1 \) and \( C_2 \) (similar to the dot notation of Prolog). \( \text{item}(\text{Key}, C_n) \) represents an element of a message where \( \text{Key} \) can be a public or a shared key used to encrypt it (or the special value \( \text{plain} \) if it is not encrypted) and \( C_n \), which store terms of the form \( \text{cons}/2 \), represents its components. \( \text{msg}(\text{Items}) \) represents a message where \( \text{Items} \) is a stream which contains elements of the form \( \text{item}/2 \) corresponding to the content of the message. \( \text{know}(X, S_v) \) stores in the stream \( S_v \) the private knowledge of principal \( X \). \( \text{pknow}(S_p) \) stores the public knowledge of all participants of a protocol. \( \text{dv}(X, Y, Z, M, P) \) represents the fact that a principal \( X \), impersonating \( Y \), has sent a message \( M \) to \( Z \). \( P \) is instantiated to \( \text{ok} \) when the message has already been processed (traveled the network). \( \text{deliver}(S_d) \) stores in the stream \( S_d \) the messages sent by the participants of the protocol; i.e., plays the role of channel.

In the following, we show the declarations which represent common actions. As we have said before, many protocols use nonces to ensure freshness of messages. Let us first show how we can generate nonces in \text{TCCP}:

\[
\text{generator}(S, V) : \exists V', T (\text{ask-tell}(S, V') \parallel \text{ask}(true) \rightarrow \begin{cases} \text{now}(-\text{free}(V')) & \text{then} (V \leftarrow V' + 1) \parallel \text{ask}(true) \rightarrow \\ \text{update}(S, V, T) \end{cases} \text{else skip}).
\]

The action \( \text{generator} \) generates and stores in the stream \( S \) the new value \( V \). The \( \text{ask-tell}(S, V') \) agent instantiates \( V' \) to the current value of the stream \( S \); then, \( V \) is generated and finally the stream \( S \) is consistently updated with \( V \).

Next, the predicate \( \text{send}/3 \), models the action when principal \( A \) sends a message \( M \) to principal \( B \).

\[
\text{send}(A, B, M) : \exists S (\text{tell}(\text{pknow}(S))) \parallel \text{ask}(true) \rightarrow \begin{cases} \text{now} (\text{find}(S, \text{pair}(A, .)) \land \text{find}(S, \text{pair}(B, .))) & \text{then} \\ \exists S', P, T (\text{tell}(\text{deliver}(S'))) \parallel \text{ask}(true) \rightarrow \\ \text{update}(S', \text{dv}(A, A, B, M, P), T) \end{cases} \text{else skip}).
\]

A condition for the action \( \text{send} \) to be taken is that both, \( A \) and \( B \) participate in the protocol, for this reason, we first instantiate \( S \) to the stream that stores...
the public knowledge in the protocol; once instantiated, we check whether A and B are principals. Then, the process instantiates $S'$ to the stream of sent messages ($\text{deliver}/1$), and we update $S'$ with the corresponding term $dv/5$ which adds to the store that the message $M$ must be delivered to principal B. The asynchronous model for the communication becomes clear at this point.

Finally, the predicate $\text{dec}/2$ models the process of a principal decrypting a message. The principal X knows the content of each element of the message only if he/she knows the appropriate decryption key. $\text{dec}/2$ is divided into two main blocks: The first one models when the decryption process has finished. The guard of the first conditional agent, $\text{free}(M)$, holds this means that the message have been processed, thus the process ends ($\text{skip}$). The second block specifies the heart of the decryption process. We instantiate the variables required to check each element $E$ of the message. Then, the process checks whether the principal is able to decrypt $E$ (one of the guards of the conditional agents, for example ($K = \text{plain}$, holds). In such case, we can say that he knows the content $C$ of $E$ updating his stream of private knowledge ($\text{update}(Kv,C,F)$); Finally, if the principal is unable to decrypt it, then we say that he knows the encrypted element ($\text{update}(Kv,E,F)$). The process recursively processes the rest of the message.

$$\text{dec}(X,M) :- \text{now}(\text{free}(M)) \text{ then skip}$$
$$\text{else } \exists E,T,K,C,Kv,F(\text{tell}(M = [E|T]) \parallel \text{ask(true)} \rightarrow$$
$$\text{tell}(E = \text{item}(K,C)) \parallel \text{tell(know}(X,Kv)) \parallel \text{ask(true)} \rightarrow$$
$$\text{now}(K = \text{plain}) \text{ then update}(Kv,C,F)$$
$$\text{else now}(K = \text{pk}(X) \land \text{find}(Kv,sk(X))) \text{ then update}(Kv,C,F)$$
$$\text{else now}(K \neq \text{pk}(\_)) \land \text{find}(Kv,K) \text{ then update}(Kv,C,F)$$
$$\text{else } \text{update}(Kv,E,F) \parallel$$
$$\text{dec}(X,T)).$$

### 3.1.2 Environment

The design of protocols turns out problematic even assuming perfect cryptography. The problem is mainly due to the fact that principals communicate over a network controlled by an intruder who can intercept, analyze, and modify messages, being thus able to carry out malevolent actions. These capabilities correspond to the Dolev-Yao attacker [16]. In this section, we show the specification of the network model which represents also the intruder capabilities.

Let us first show two declarations used by the environment. The first one, $\text{sel}/2$, is used to fish messages from the network. In each iteration, $\text{sel}$ checks whether the stream of sent messages $S$ is empty (non-instantiated). If the condition holds, $\text{sel}$ finishes the execution; otherwise it looks into the parameters of a term $dv$ and checks whether the variable $P$ is instantiated to $\text{ok}$ (meaning that the message has already been processed). In that case, continues looking for a
message to fish. In case \( P \) is not instantiated, \( \text{sel} \) instantiates \( V \) to the found \( \text{dv} \) term.

\[
\text{sel}(S,V) :- \text{now}(\text{free}(S)) \text{ then skip}
\]
\[
\text{else} \; \exists \; A,A',B,M,P,S'(\text{tell}(S = [\text{dv}(A,A',B,M,P)|S']) \| \text{ask}(\text{true}) \rightarrow
\text{now}(P = \text{ok}) \text{ then } \text{sel}(S',V) \text{ else } \text{tell}(V = \text{dv}(A,A',B,M,P))).
\]

\( \text{compose}(X,M,M') \) looks into the message passed as argument \( M \) and composes a new message \( M' \) with the same number of components by using the knowledge of the intruder \( X \). The predicate ends when the guard \( \text{free}(M) \) holds which means that the message \( M \) has been processed and the content of the new message \( M' \) has been created. In case the guard is not satisfied the predicate non-deterministically can compose a new element (encrypted or not), or recover certain information of the form \( \text{item}(\_,\_) \) from the stream of private knowledge of the intruder, or use the same element \( E \) of the message \( M \) to created the new element of the message \( M' \) (\( \text{update}(M',\text{item}(\text{plain},C'),F) \), \( \text{update}(M',\text{item}(K,C'),F) \), etc.). The predicate processes the whole message recursively. \( \text{compose}/3 \) uses the auxiliary predicate \( \text{gen}(C,S_A,C') \) where given \( C \), a term of the form \( \text{cons}/2 \), and free variables \( S_A \) and \( C' \) the predicate will return in \( C' \) a new term of the same form \( \text{cons}/2 \) that will be created with some of values of the term \( \text{cons}/2 \) stored in \( C \). First, we show the declaration of the predicate \( \text{gen}/3 \) and then, the declaration of \( \text{compose}/3 \):

\[
\text{gen}(C,S_A,C') :- \exists \; C1,C2(\text{now}(\neg \text{free}(C)) \text{ then}
\text{now}(C = \text{cons}(\_,\_)) \text{ then}
\text{tell}(C = \text{cons}(C1,C2)) \| \text{ask}(\text{true}) \rightarrow
\exists \; F1,F2(\text{update}(S_A,C1,F1) \| \text{ask}(\text{true}) \rightarrow \text{update}(S_A,C2,F2) \|
\text{gen}(C2,S_A,C'))
\text{else assign}(S_A,\_,C1) \| \text{assign}(S_A,\_,C2) \|
\text{ask}(\text{true}) \rightarrow \text{tell}(C' = \text{cons}(C1,C2)))
\text{else skip}.
\]

\[
\text{compose}(X,M,M') :- \text{now}(\text{free}(M)) \text{ then skip else}
\exists \; E,T,Kv,Kp,C,Ax,C',F(\text{tell}(M = [E|T]) \| \text{ask}(\text{true}) \rightarrow \text{tell}(\text{know}(X,Kv)) \| \text{tell}(\text{pknow}(Kp)) \| \text{ask}(\text{true}) \rightarrow (\text{assign}(Kv,\text{cons}(\_,\_),C) \| \text{ask}(\text{true}) \rightarrow \text{gen}(C,Ax,C') \| \text{ask}(\neg \text{free}(C')) \rightarrow \text{update}(M',\text{item}(\text{plain},C'),F) +
\text{ask}(\neg \text{free}(C')) \rightarrow (\exists \; R,K(\text{assign}(Kp,\text{pair}(\_,\_),R) \| \text{ask}(\text{true}) \rightarrow \text{tell}(R = \text{pair}(\_,K)) \| \text{ask}(\text{true}) \rightarrow \text{update}(M',\text{item}(K,C'),F))) + \text{ask}(\text{true}) \rightarrow \text{assign}(Kv,\text{item}(\_,\_),C) \| \text{ask}(\text{true}) \rightarrow \text{update}(M',C,F) + \text{ask}(\text{true}) \rightarrow \text{update}(M',E,F) \| \text{compose}(X,T,M')).
\]
3.1. Protocol specification using tccp

In Fig. 3.1, we show the environment model coded in tccp. We have labeled the code to make clearer the description. First of all, it must be said that the environment is defined as a cycle where, during each iteration, one of the labelled actions 1a to 1e is non-deterministically executed when there is a message for processing.

```
environment(I):- ∃ D,Dv(tell(deliver(D))) || (ask(true)→ sel(D,Dv) +
    ask(true)→ assign(D,dv(_,_,_,_,ok),Dv)) ||
  ask(¬free(Dv))→ ∃ A,A′,B,M,P(tell(Dv = dv(A,A′,B, msg(M),P))) ||
  1a  ask(true)→ tell(P = ok) +
  1b  ask(true)→ dec(B,M) || tell(P = ok) +
  1c  ask(true)→ dec(I,M) || tell(P = ok) +
  1d  ask(true)→ ∃ P′,T(ask(true)→ update(D,dv(I,A,B, msg(M),P′),T) +
    ask(true)→ update(D,dv(I,B,A, msg(M),P′),T)) ||
    tell(P = ok)) +
  1e  ask(true)→ ∃ M′,P′,T(compose(I,M,M′)) ||
    ask(¬free(M′))→ update(D,dv(I,B,A, msg(M′),P′),T) +
    ask(¬free(M′))→ update(D,dv(I,A,B, msg(M′),P′),T) ||
    tell(P = ok)) ||
  environment(I)).
```

Figure 3.1: Procedure modeling the attacks that an intruder carries out in the hostile environment

First, we recover in D the stream of sent messages. In parallel, one of both actions sel(D,Dv) or assign(D,dv(_,_,_,_,ok),Dv)) can be executed. The first selects a message that have not been processed whereas the second selects a message that have been processed. In case there is a message to be processed (¬free(Dv)), the environment non-deterministically chooses one action among 1a to 1e. 1a represents a message loss. 1b models the case when the message is correctly delivered, thus the destination participant decrypts it. In 1c the intruder intercepts the message and tries to decrypt it. 1d model the cases when the intruder intercepts the message and impersonates A or B. Finally, in 1e the intruder composes a message and impersonates A or B.

Note that the blocking of communication capability of the Dolev-Yao intruder model is represented by action 1a; message intercepting is modelled by 1c, 1d and 1e, in particular 1c implements message decomposing, 1d message replaying whereas 1e implements message composing and replaying.
3.1.3 Communication Protocols

A protocol can be seen as a recipe that enables the connection and data exchange between two or more entities (principals) through messages. It describes how messages must be sent from one principal to another, and how these principals react when receiving a message. Principals usually play the role of initiator, responder or server. Messages consist of atoms such as principals’ names, or nonces. A nonce is a value (ideally) randomly generated by a principal and it is usually used to ensure the freshness of messages. There are different classifications of protocols depending on their goal, e.g., key establishment, real-time authentication, fresh key, key confirmation, etc. [8]. They can also be classified depending on the kind of keys that is used to encrypt and decrypt messages: shared key, public and private keys, key generation by a trusted server, etc. Along the paper, we will use two communication protocols as running examples. One of them uses pairs of public and private keys to ensure confidentiality of messages whereas the second one uses shared keys and a trusted server for the generation of the session key. Let us briefly describe them.

Needham-Schroeder public key authentication protocol

Below, we show the standard specification of the Needham-Schroeder public key authentication protocol [23] based on public-key cryptography without using trusted key servers. In the protocol, A and B (Alice and Bob in the literature) play the role of initiator and responder, respectively. Once the protocol has been completed, both principals are convinced about the identity of their partners. From that time instant, they can share critical information. In other words, the protocol ensures A to be communicating with B, and vice-versa.

1. $A \rightarrow B : \{A, N_A\}K_{BP}$
2. $B \rightarrow A : \{N_A, N_B\}K_{AP}$
3. $A \rightarrow B : \{N_B\}K_{BP}$

The protocol begins when $A$ sends to $B$ a message encrypted with $B$’s public key $K_{BP}$ which contains the identifier of $A$, and the nonce $N_A$ generated by $A$. $B$ decrypts that message by using his private key and sends to $A$ a message, encrypted with the public key of $A$, which contains the received nonce $N_A$ and a new generated nonce $N_B$. $A$ decrypts the message by using her secret key. Since the message contains the nonce $N_A$, $A$ can deduce that the message has recently been sent by $B$. Finally, $A$ sends the confirmation message to $B$, so that $B$ can infer that he is communicating to $A$ (he had previously sent $N_B$).
Otway-Rees symmetric key authentication protocol

The Otway-Rees symmetric key authentication protocol [26] is an authentication protocol based on symmetric key cryptography. It allows principals to communicate via a trusted third party (Server) which establishes a shared secret key ($SK_{AB}$) for secure communication between $A$ and $B$. We show the standard specification of the protocol below. $A$ wants to communicate with $B$ by using the server $S$. $S$ shares the keys $SK_{AS}$ and $SK_{BS}$ with $A$ and $B$, respectively. $N$ is a nonce generated by $A$ to identify the session.

1. $A \rightarrow B : N, A, B, \{N_A, N, A, B\}SK_{AS}$
2. $B \rightarrow S : N, A, B, \{N_A, N, A, B\}SK_{AS}, \{N_B, N, A, B\}SK_{BS}$
3. $S \rightarrow B : N, \{N_A, SK_{AB}\}SK_{AS}, \{N_B, SK_{AB}\}SK_{BS}$
4. $B \rightarrow A : N, \{N_A, SK_{AB}\}SK_{AS}$

$A$ generates the nonces $N$ and $N_A$; then, she sends to $B$ a message which contains the non-encrypted text $N, A, B$, and the elements $\{N_A, N, A, B\}$ encrypted with the secret key shared by her and the server ($SK_{AS}$). Then, $B$ forwards to $S$ the message received from $A$ together with a part containing $\{N_B, N, A, B\}$ encrypted with the key shared by him and $S$ ($SK_{BS}$). $S$ decrypts the message received from $B$ and checks whether elements $N, A, B$ from the two encrypted parts coincide. In that case, he generates the secret key $SK_{AB}$ and sends to $B$ a message which contains (1) the session identifier $N$, (2) a component encrypted with $SK_{AS}$ which contains $N_A$ and $SK_{AB}$ and (3) a component encrypted with the secret key $SK_{BS}$ which contains $N_B$ and $SK_{AB}$. Finally, $B$ forwards to $A$ a message containing the components (1) and (2) described above. When $A$ decrypts that message both participants, $A$ and $B$, know $SK_{AB}$.

3.1.4 Principals

Let us now show how the participants in protocols can be specified. Whereas the above described actions can be reused for modeling other protocols, the code for principals will be redefined for each different protocol. Each participant of the protocol is modeled as a tccp declaration. Let us show how to implement the two running examples of this paper. We consider these two examples to illustrate how the different approaches to encryption can be handled in tccp. Moreover, whereas the first protocol suffers a man-in-the-middle attack, the second one suffers a typing attack, so we show how it is possible to cover Dolev-Yao-based attacks as well as typing attacks.

Let us first show the specification of the Needham-Schroeder protocol.

The initiator $A$ recovers her public and private knowledge ($K$ and $Ka$, respectively) and her generated nonces $Kn$. Then, she non-deterministically decides (1)
provided she knows the public key of the responder B, then she sends to B a message containing her name A and a generated nonce Na, all encrypted with pk(B); or (2) provided she has received a message containing her generated nonce Na, then, she sends the confirmation message to the responder.

\[
\text{init}(A,B) : = \exists \ K, Ka, Kn, Na, M, T
\]
\[
(tell(pknow(K)) \land tell(know(A,Ka)) \land tell(nonce(A,Kn)) \land ask(true) \rightarrow
\]
\[
ask(find(K,\{\text{pair}(B,pk(B))\})) \rightarrow \text{generator}(Kn,Na) \land ask(\neg \text{free}(Na)) \rightarrow
\]
\[
tell(M=[\text{item}(pk(B),\{\text{cons}(A,Na)\})|T]) \land ask(true) \rightarrow \text{send}(A,B,msg(M)) +
\]
\[
ask(find(Ka,\{\text{cons}(Na,\{\})\})) \rightarrow \exists \ Inf,Nb(\text{assign}(Ka,\{\text{cons}(Na,\{\}), Inf\}) \land
\]
\[
ask(true) \rightarrow \text{tell}(Inf=\{\text{cons}(A,Na)\}) \land ask(true) \rightarrow
\]
\[
tell(M=[\text{item}(pk(B),\{\text{cons}(Nb,\{\})\})|T]) \land ask(true) \rightarrow \text{send}(A,B,msg(M))).
\]

The responder B, shown below, recovers from the store his public and private knowledge and his generated nonces Kn. Then, he non-deterministically decides (1) provided he knows the information resulting from the first message of the protocol, then he sends to A a message containing the received nonce Na and a new generated nonce Nb, all encrypted with pk(A). Note that the stream of private knowledge of the principal B can contain several terms of the form \text{cons}(A,\_), therefore the agent \text{assign}(Kb,\{\text{cons}(A,\{\}), Inf\}) can instantiate the variable Inf to any of the terms \text{cons}(A,\_) present in the stream of private knowledge of B; or (2) provided he knows the contents of the last message in the protocol, he finishes his execution.

\[
\text{resp}(B,A) : = \exists \ K, Kb, Kn, Nb
\]
\[
(tell(pknow(K)) \land tell(know(B,Kb)) \land tell(nonce(B,Kn)) \land ask(true) \rightarrow
\]
\[
ask(find(Kb,\{\text{cons}(A,\{\})\})) \rightarrow \exists \ Inf,Na,M,T (\text{assign}(Kb,\{\text{cons}(A,\{\}), Inf\}) \land
\]
\[
ask(true) \rightarrow \text{tell}(Inf=\{\text{cons}(A,Na)\}) \land \text{generator}(Kn,Nb) \land
\]
\[
ask(\neg \text{free}(Na) \land \neg \text{free}(Nb) \land \text{find}(K,\text{pair}(A,pk(A)))) \rightarrow
\]
\[
tell(M=[\text{item}(pk(A),\{\text{cons}(Na,Nb)\})|T]) \land
\]
\[
ask(true) \rightarrow \text{send}(B,A,msg(M)) +
\]
\[
ask(find(Kb,\{\text{cons}(Nb,\{\})\}) \rightarrow \text{skip}).
\]

We have still to connect the principals definitions with the environment. To this end, we define the main declaration of the protocol, run/3, that initializes the protocol run.
In section 3.2 we show how to formally verify that a chosen attack is, or is not, possible in the specification of the protocol by means of LTL formulas that the model must satisfy.

Following, we show how to specify the principals in the Otway-Rees protocol.

The initiator, non-deterministically, can do two things: (1), in case A knows a secret key to communicate with the server S, then she generates the nonces N and Na and sends to B the first message of the protocol; or (2) provided she knows the information resulting from decrypting the last message of the protocol, she ends the execution.

The responder B, shown below, recovers his private knowledge Kb and his generated nonces Kn. Then, he non-deterministically can (1) provided he knows the first message sent by A, sends to the server S the received message together with a new encrypted part containing a new nonce Nb generated by him; or (2) provided he knows the message sent by the server S, he gets the key generated by the server to be shared by B and A and sends to A the last message of the protocol which contains the shared key.
resp(B,A,S):- ∃ Kb,Kn,Inf,N,Nb,Ec,M,T
tell(know(B,Kb)) || tell(nonce(B,Kn)) || ask(true)→
ask(find(Kb,[cons(_,cons(A,B)),_]),Inf) → ask(true)→
assign(Kb,[cons(_,cons(A,B)),_],Inf) → ask(true)→
tell(Inf=[cons(N,cons(A,B)),Ec]) || generator(Kn,Nb) ||
ask(¬free(N) ∧ ¬free(Ec) ∧ ¬free(Nb) ∧ find(Kb,k(B,S)))→
tell(M=[item(plain,cons(N,cons(A,B))),Ec,
    item(k(B,S),cons(Nb,cons(N,cons(A,B))))]|T)) ||
ask(true)→ send(B,S,msg(M)) +
ask(find(Kb,[cons(N,[]),_,cons(Nb,_)]),Inf) → ask(true)→
tell(Inf=[cons(N,[]),Ec,cons(Nb,[])]) || ask(true)→
tell(M=[item(plain,cons(N,[])),Ec|T]) ||
ask(true)→ send(B,A,msg(M)).

The third participant of the protocol is the server S, which waits until receiving the message sent by B. Once received, he generates the shared key Kab for the communication between A and B, and sends a message to B that corresponds to the third message of the protocol.

server(S,A,B):- ∃ Ks,SK(tell(know(S,Ks))) || tell(key(S,SK)) || ask(true)→
ask(find(Ks,[cons(_,cons(A,B)),cons(_,cons(_,cons(A,B)))),
    cons(_,cons(_,cons(A,B)))),Inf,N,Na,Nb,Kab,M,T
(assign(Ks,[cons(_,cons(A,B)),cons(_,_),cons(_,_)],Inf) → ask(true)→
tell(Inf=[cons(N,cons(A,B)),cons(Na,cons(N,cons(A,B)))),
    cons(Nb,cons(N,cons(A,B)))),generator(SK,Kab) ||
ask(¬free(N) ∧ ¬free(Na) ∧ ¬free(Nb) ∧ ¬free(Kab) ∧ find(Ks,k(A,S)) ∧
    find(Ks,k(B,S)))→ tell(M=[item(plain,cons(N,[])),
    item(k(A,S),cons(Na,Kab)),item(k(B,S),cons(Nb,Kab))]|T)) ||
ask(true)→ send(S,B,msg(M))).

Finally, we have to connect the participants declarations and the environment. To this end, we define the run/4 process that initializes the protocol execution.
3.2. Verification

run(A,B,S,I):- \exists D_p, K_p, K_A, N_A, K_B, N_B, K_S, N_I, T^1, T^2, T^3, T^4, T^5, T^6, T^7, T^8, T^9, F^1, F^2, F^3, F^4, F^5 (tell(deliver(D_p)) || (pknow(K_p)) || (tell(know(A,K_A))) ||
next(K_A)) then update(K_p, pair(A,pk(A)), T^1) || update(K_A, k(A,S), T^2) ||
tell(N_A=[1|T^3]) else now(find(K_A,k(A,S))) then skip
else update(K_A,k(A,S), T^2) ||
tell(know(B,K_B)) || tell(nonce(B,N_B)) || ask(true) ->
now(free(K_B)) then update(K_p, pair(B,pk(B)), T^4) || update(K_B, k(B,S), T^5) ||
tell(N_B=[2|T^6]) else now(find(K_B,k(B,S))) then skip
else update(K_B,k(B,S), T^5) ||
tell(know(I,K_I)) || tell(nonce(I,N_I)) || ask(true) ->
now(free(K_I)) then update(K_p, pair(I,pk(I)), T^7) || update(K_I, k(I,S), T^8) ||
tell(N_I=[3|T^9]) else now(find(K_I,k(I,S))) then skip
else update(K_I,k(I,S), T^8) ||
tell(know(S,K_S)) || tell(key(S,S_K)) || ask(true) ->
now(free(K_S)) then update(K_p, pair(S,pk(S)), F^1) ||
update(K_S, k(A,S), F^2) || ask(true) -> update(K_S, k(B,S), F^3) ||
ask(true) -> update(K_S, k(I,S), F^4)
else now(find(K_S,k(A,S))) then skip else update(K_S,k(A,S), F^2) ||
now(find(K_S,k(B,S))) then skip else update(K_S,k(B,S), F^3) ||
now(find(K_S,k(I,S))) then skip else update(K_S,k(I,S), F^4) ||
now(free(K_S)) then tell(S_K=[5|F^5]) else skip ||
init(A,B,S) || resp(B,A,S) || server(S,A,B) || environment(I)).

Up to this point, we have given the guideline for the specification of communication protocols in ttcp by using a generic environment and some common declarations that ease our task. In the following, we illustrate how it is possible to verify properties over our specification.

3.2 Verification

We show how it is possible to verify properties of the protocols previously specified. We use the LTL logic presented in [14] that deals with constraints. We can use the model checker proposed in [17] which is based on this logic. Independently from the verification technique used, one very important task is to correctly identify which properties the protocol must satisfy. Note that, depending on the goal of the considered protocol, some properties may have no sense, whereas others are the ones characterizing the well-behavior of the system.

Intuitively, the LTL logic used in this work for the specification of properties is defined as the classical LTL logic (with temporal operators always □, eventually ◯, until U, next ◇), as well as an existential quantifier over variables and atomic propositions from C. In [3], a real-time extension of this logic was presented, allowing to specify (timed) quantitative properties.
### 3.2.1 Checking the Needham-Schroeder Protocol

Following the Dolev-Yao model, a man-in-the-middle attack was detected for this protocol. In the attack, $A$ initiates a protocol run with $I$ (the intruder), who (impersonating $A$) starts a second run of the protocol with $B$. In other words, the intruder asks $B$ to initiate a communication session saying that he is $A$. At the end of the attack, $B$ thinks he is communicating to $A$, which is false. An interesting property that this protocol should satisfy is that nonces are always known by at most two principals. The situation when more than two principals know a given nonce characterizes a protocol run where a man-in-the-middle attack has happened. The following property specifies this situation where $N$ is a nonce that can be generated by principal $A$, $B$ or the intruder $I$ so only the principal intended for the communication, as well as the principal that has generated it should know it:

$$\neg \diamond (\exists N, S_{kA}, S_{kB}, S_{kI}, S_{nA}, S_{nB}, S_{nI})(((\text{nonce}(A, S_{nA}) \land \text{find}(S_{nA}, N)) \lor \text{nonce}(B, S_{nB}) \land \text{find}(S_{nB}, N)) \lor (\text{nonce}(I, S_{nI}) \land \text{find}(S_{nI}, N))) \land$$
$$\text{know}(A, S_{kA}) \land \text{know}(B, S_{kB}) \land \text{know}(I, S_{kI}) \land$$
$$((\text{find}(S_{kA}, \text{cons}(N, _)) \lor \text{find}(S_{kA}, \text{cons}(N, _))) \land$$
$$\text{find}(S_{kB}, \text{cons}(N, _)) \lor \text{find}(S_{kB}, \text{cons}(N, _))) \land$$
$$\text{find}(S_{kI}, \text{cons}(N, _)) \lor \text{find}(S_{kI}, \text{cons}(N, _))))$$

Note that we are detecting the well-known attack, but thanks to the non-determinism and generality of the environment a number of traces can be analyzed.

We conclude this subsection saying that the protocol is safe with respect to this property which means that nonces $N_A$ and $N_B$ can only be known by principals $A$ and $B$:

$$\neg \diamond (\exists N_A, N_B, S_{kA}, S_{kB}, S_{kI}, S_{nA}, S_{nB}, S_{nI})(((\text{nonce}(A, S_{nA}) \land \text{find}(S_{nA}, N_A)) \land$$
$$\text{nonce}(B, S_{nB}) \land \text{find}(S_{nB}, N_B)) \land (\text{know}(A, S_{kA}) \land \text{know}(B, S_{kB}) \land$$
$$\text{know}(I, S_{kI}) \land \text{find}(S_{kA}, \text{cons}(N_A, _)) \land \text{find}(S_{kB}, \text{cons}(N_B, _)) \land$$
$$\text{find}(S_{kI}, \text{cons}(N_A, _)) \lor \text{find}(S_{kI}, \text{cons}(N_A, _))) \land$$
$$\text{find}(S_{kI}, \text{cons}(N_B, _)) \lor \text{find}(S_{kI}, \text{cons}(N_B, _))))$$

### 3.2.2 Checking the Otway-Rees protocol

Boyd [7] shows a typing attack for this protocol. The intruder $I$ (impersonating $B$) intercepts the first message of the protocol. Then builds the fourth message by removing the identifiers $A$ and $B$ from the plaintext part $N, A, B$ of the message. Thus, when $A$ decrypts the message he interprets the sequence $N, A, B$ as the session key that the Server had to build for the communication between $B$ and
him. Hence, the intruder can disclose any message encrypted with this sequence. Due to the way in which we have defined messages, our approach allows the misinterpretation of message segments.

We can specify as a goal of the protocol that the key received by $A$ and $B$ from the server must be equal. The following properties model this goal. Note that these properties do not guarantee the proper run of the protocol, they can only be used to verify that the protocol achieves the goal mentioned:

$$\diamond (\exists SK_s, K, Sk_A, Sk_B ((\text{key}(S, SK_s) \land \text{find}(SK_s, K) \land \text{know}(A, Sk_A) \land \text{find}(Sk_A, cons(\_ , K)) \land \text{know}(B, Sk_B) \land \text{find}(Sk_B, cons(\_ , K)))))$$

$$\diamond (\exists SK_s, K, Sk_A, Sk_B ((\text{key}(S, SK_s) \land \text{find}(SK_s, K)) \cup (\text{know}(B, Sk_B) \land \text{find}(Sk_B, cons(\_ , K)))) \cup (\text{know}(A, Sk_A) \land \text{find}(Sk_A, cons(\_ , K)))))$$

A desirable secrecy property of this protocol is that the intruder does not get any key that $A$ and $B$ has for secure communication:

$$\neg \diamond (\exists SK_s, K, Sk_A, Sk_B, Sk_I ((\text{key}(S, SK_s) \land \text{find}(SK_s, K) \land \text{know}(A, Sk_A) \land \text{find}(Sk_A, cons(\_ , K)) \land \text{know}(B, Sk_B) \land \text{find}(Sk_B, cons(\_ , K)) \land \text{know}(I, Sk_I) \land \text{find}(Sk_I, K))))$$

Finally, we show a property needed to guarantee the security of the protocol where we specify that the sequence $N, A, B$ can not be interpreted as a key by $A$ or $B$. It specifies that: never can happen that the term $\text{cons}(N, \text{cons}(A, B))$, equivalent to the sequence $N, A, B$ which contains the nonce $N$ created by principal $A$, and the identifiers of principals $A$ and $B$ could be interpreted by them as the key sent by the Server for the communication between them.

$$\neg \diamond (\exists Sn_A, N, Sk_A, Sk_B, ((\text{nonce}(A, Sn_A) \land \text{find}(Sn_A, N)) \land ((\text{know}(A, Sk_A) \land \text{find}(Sk_A, cons(\_ , cons(N, cons(A, B))))) \lor (\text{know}(B, Sk_B) \land \text{find}(Sk_B, cons(\_ , cons(N, cons(A, B))))))))$$
Modeling Communication Protocols in the Timed Concurrent Constraint Language
A tool for Generating a Symbolic Representation of tccp executions

A model-checking algorithm for tccp was proposed in [17] which given a tccp program transforms it into a symbolic representation (the tccp Structure) which is the input of the verification phase. The proposal in [17] is similar to the classical one, with the tccp Structure playing the role of the Kripke Structure. However, as we will show later, the tccp Structure differs from the Kripke Structure in some important points, so the verification algorithm had to be reformulated and adapted to the ccp model.

In this chapter, we describe the notion of tccp Structure where we present the main difficulties when dealing with the ccp model, we also show how we implemented the construction of such structure and finally, we show how to use the system by developing two running examples.

4.1 A symbolic representation of tccp executions

A graph structure (tccp Structure) for modeling tccp traces was proposed in [17] in the context of the definition of a model-checking algorithm. This structure can be seen as a variant of a Kripke Structure where, following the ccp model, the notion of state as valuation of variables is replaced by the notion of state as a conjunction of constraints. This means that a node in the tccp Structure can represent a set of classical nodes of a Kripke Structure. The StructGenerator system implements the construction of such structure. As we will show, the main difficulties for this constructions arise from the monotonic nature of the tccp store. Moreover, both in the construction of the tccp Structure phase and later in the verification phase, we have to take into consideration that, differently from the classical approach, the absence of some information in the store does not (necessary) mean that the negation of that information holds.

Let us briefly describe the symbolic representation. A state of the tccp Structure contains a set of atomic propositions; more specifically, it consists of a set of atomic constraints from the underlying constraint system in tccp. Each state of
the \texttt{tccp} Structure also contains a set of labels representing the current execution step. These labels are uniquely associated to each occurrence of an agent in the \texttt{tccp} program. Note that a pre-process for labeling the program is needed. Labels allow us to set those agents that must be executed in the following time instant.

Since during the execution of a program the store grows monotonically, by definition, and differently from the classical approach, there cannot be two states syntactically equal. This means that in order to define a finite representation of executions, a notion of equivalence between states (which eventually would determine the cycles in the graph structure) was needed. This notion is also necessary for having a finite construction algorithm.

Informally, we say that two states are equivalent if (1) the set of labels in both states coincide, and (2) the set of constraints in one state is equivalent to the set of constraints in the second state modulo renaming [17]. In order to deal with streams equivalence, we use the notion of current value (i.e., the more recent added value [3]) of the stream to set whether the stream is equal in two states. Note that the number of constraints defining a stream is always increasing (thus we never have two states syntactically equal) but, following the imperative-style notion of variable, at some specific execution point, we are interested just on the current value of such stream. The definition of state equivalence allows us to overcome the problem of termination of the algorithm caused by the monotonic behavior of the \texttt{tccp} store. Remember that \texttt{tccp} is used to model reactive systems. In case of being modeling arithmetic functionality, termination cannot be generally ensured.

4.1.1 The implementation

\texttt{StructGenerator} has been implemented in C++. It consists of approximately 3100 lines of code divided in 10 classes. Each class handles one of the entities of \texttt{tccp} (agents, constraints, declarations, stores, states, \ldots). We decided to implement the system in C++ since we needed to connect it to some constraint solvers, and we found some interfaces to ease these connections had been defined for that language. Moreover, we had to deal with complex data structures, thus C++ provided us the capability of defining classes and structuring them.

As we have said above, the \texttt{StructGenerator} system constructs in an automatic way the already described \texttt{tccp} Structure, so it aims to representing in a symbolic way the behavior of a \texttt{tccp} program. An algorithm for this construction was proposed in [17]. We have followed the main ideas in that algorithm, but, as mentioned above, we decided to refine some of the definitions by using the more flexible computational model in [3] which allowed us to deal more easily with streams.

Intuitively, the implementation moves through the following phases. First of all, given the specification of a \texttt{tccp} program in a text file, the system parses
the program where each occurrence of an agent is labeled with a different tag. Thereafter, the initial states are built and, from each initial state, we construct the states reached from such state following the semantics of the language. States are generated but are not introduced into the structure until confirmation. Confirmation is achieved when it is checked whether there exists no equivalent node in the structure. Let us describe the generation process from each node. This generation process is iteratively performed for each new state (initially for the initial states):

1. to generate the possible successors of a node following the semantics of the language,
2. to check whether there exist an equivalent node for each possible successors,
3. to confirm the introduction in the structure of the new node:
   (a) in case that an equivalent node does not exist, then the successor is added to the \texttt{tccp} structure
   (b) in case that an equivalent node exists, the calculated \textit{renaming} that makes the two states equivalent is associated to the edge connecting the original node to the equivalent one. The successor node is discarded.

The iterative process ends when all the nodes have been processed for successors generation. The key point for the construction is the second and third phases, where we look for equivalent nodes. As we have said before, this step is crucial to (partially) ensure the termination of the algorithm.

As shown in Fig. 4.1, the \texttt{StructGenerator} system has been structured in 2 modules, \texttt{Program Parser} and \texttt{Construction Process}. The \texttt{Program Parser} module takes as input the file that contains the program specification. Thereafter, by using the auxiliary procedures \texttt{Declaration Parser} and \texttt{Agent Parser}, generates the data structure \texttt{Declarations} which stores all the program information: agents, constraints, labeling, etc. The generated information \texttt{Declarations} is the input of the \texttt{Construction Process} module, which generates as output the graph structure. To this end, it uses a \texttt{Node Creation} process corresponding to the first phase in the iterative process above. In brief, it constructs the states of the structure by using three auxiliary functions: \texttt{instant}, \texttt{follows} and \texttt{find}\footnote{The interested reader can find in \cite{17} the definitions of the original \texttt{instant} and \texttt{follows} functions.}. Intuitively, \texttt{instant} and \texttt{follows} calculate, following the \texttt{tccp} semantics, (1) the constraints that an agent can add in a time instant and (2) the agents that must be executed in the following time instant, respectively. In other words, they calculate the contents of new nodes (labels and constraints). \texttt{find} is the function that looks for equivalent nodes in the structure as described above.
4. A tool for Generating a Symbolic Representation of tccp executions

The interface of our tool is console guided. To run the tool, we have to enter the command **StructGenerator**. Then, the system asks the user for the filename where the specification of the tccp program is written. Fig. 4.2 shows a simple execution.

```
user@pcname:~$ StructGenerator
Enter the filename of the tccp program(*.tccp or *.txt):
...microwave_error.txt
-----------------------------------------------
.....Creating graph of the declaration: microwave_error
  - creating children of the state: 1
  - creating children of the state: 2
  - creating children of the state: 3
  - creating children of the state: 4
...
```

Figure 4.2: StructGenerator run
### 4.2 Practical examples

In this section, we present two practical examples. The first one is the (partial) specification of a microwave oven controller that we have borrowed from [17]. The second one is the specification of the scheduler example used in [1] to motivate the symbolic version of the model checker. We first provide the tccp specification of the concurrent system and then we provide the tccp Structure generated by StructGenerator. We also show graphically the connection between nodes.

#### 4.2.1 The microwave_error System

In Fig. 4.3 we show the tccp specification of the system. To make clearer the relation between the specification and the generated graph, we show the labeled version of the program. Labels appear within braces {}.

The procedure declaration `microwave_error` models the process of detecting when the door of a microwave is open at the same time that the system 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. Note that the `tell` agent is executed at the same time instant when the error is detected, but the told constraint is only available to others in the following time instant. Moreover, an error signal must be emitted (agent `lt11`). If the condition does not hold, then the program emits (via another `tell` agent `lt15`) a signal of no error that will be available in the global 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.

Bellow we show the symbolic representation obtained when executing StructGenerator with input the (non-labeled version of) `microwave_error`. Currently,
the output is shown in a textual way, listing the set of states in the graph structure together with the relation between states. For the considered system, 7 states are generated. Each state contains an identifier, for example state 1. In this case, state 1 contains:

- a set of constraints: \( \text{yes}(\text{Door}=d(\text{open}) \mid D) \) and \( \text{Button}=b(\text{on}) \mid B) \),
- a set of labels: \( \text{lt2-microwave\_error} \), \( \text{lt4-microwave\_error} \), \( \text{lt6-microwave\_error} \), ...
- a set of defined variables: \( \text{Door}, \text{Button}, \text{Error}, E_1, B_1, D, B, E \)
- finally, looking at the children part, we can say that state 1 is linked with state 3 and state 4, with no renaming defined on the edge.

Graph(s) successfully created....
-- tree: 0 -> called: microwave\_error --
-- state 0 --
constraints: (true),
labels: ld-microwave\_error,
variables: Door,Button,Error,
children:
in tree: 0 -> state: 1 - renaming:
in tree: 0 -> state: 2 - renaming:
-- state 1 --
constraints: \( \text{yes}(\text{Door}=d(\text{open}) \mid D) \) and \( \text{Button}=b(\text{on}) \mid B) \),
labels: lt2-microwave\_error,lt4-microwave\_error,lt6-microwave\_error, lt11-microwave\_error,lt13-microwave\_error,lc16-microwave\_error,
variables: Door,Button,Error,E_1,B_1,D,B,E,
children:
in tree: 0 -> state: 3 - renaming:
in tree: 0 -> state: 4 - renaming:
-- state 2 --
constraints: \( \text{not}(\text{Door}=d(\text{open}) \mid D) \) and \( \text{Button}=b(\text{on}) \mid B) \),
labels: lt2-microwave\_error,lt4-microwave\_error,lt6-microwave\_error, lt15-microwave\_error,lc16-microwave\_error,
variables: Door,Button,Error,E_1,D,B,E,
children:
4.2. Practical examples

in tree: 0 -> state: 5 - renaming:
in tree: 0 -> state: 6 - renaming:

-- state 3 --
constraints: (Error=[e(\_)|E]),(Door=[d(\_)|D]),(Button=[b(\_)|B]),
(E=[e(yes)|E1]),(B=[b(off)|B1]), labels: lt2-microwave_error,lt4-microwave_error,lt6-microwave_error,
variables: Door,Button,Error,E1,B1,D,B,E,B1',E1',D',B',E', children:
in tree: 0 -> state: 3 - renaming: E/Error,E'/E,D/Door,D'/D,
B/Button,B'/B,E1'/E1,B1'/B1,D''/D',B''/B',
in tree: 0 -> state: 4 - renaming: E/Error,E'/E,D/Door,D'/D,
B/Button,B'/B,E1'/E1,B1'/B1,D''/D',B''/B',

-- state 4 --
constraints: (Error=[e(\_)|E]),(Door=[d(\_)|D]),(Button=[b(\_)|B]),
(E=[e(yes)|E1]),(B=[d(open)|B'], labels: lt2-microwave_error,lt4-microwave_error,lt6-microwave_error,
variables: Door,Button,Error,E1,B1,D,B,E,B1',E1',D',B',E', children:
in tree: 0 -> state: 5 - renaming: E/Error,E'/E,D/Door,D'/D,
B/Button,B'/B,E1'/E1,B1'/B1,D''/D',B''/B',
in tree: 0 -> state: 6 - renaming: E/Error,E'/E,D/Door,D'/D,
B/Button,B'/B,E1'/E1,D''/D',B''/B',

-- state 5 --
constraints: (Error=[e(\_)|E]),(Door=[d(\_)|D]),(Button=[b(\_)|B]),
(E=[e(no)|E1]),(B=[d(open)|B'], labels: lt2-microwave_error,lt4-microwave_error,lt6-microwave_error,
variables: Door,Button,Error,E1,B1,D,B,E,E1',B1',D',B',E', children:
in tree: 0 -> state: 3 - renaming: E/Error,E'/E,D/Door,D'/D,
B/Button,B'/B,E1'/E1,D''/D',B''/B',
in tree: 0 -> state: 4 - renaming: E/Error,E'/E,D/Door,D'/D,
B/Button,B'/B,E1'/E1,D''/D',B''/B',

-- state 6 --
constraints: (Error=[e(\_)|E]),(Door=[d(\_)|D]),(Button=[b(\_)|B]),
(E=[e(no)|E1]),(B=[b(on)|B'], labels: lt2-microwave_error,lt4-microwave_error,lt6-microwave_error,
In this example, just one tree is generated since there is just one procedure declaration. We can see this fact since all nodes are in tree: 0. The system generates as many trees as declarations in the program. Note that some relations, for example those of state 0 to state 1 and state 2, have no renaming. Renaming is associated to the relation only when cycles are established. We can graphically observe the relation between nodes in the generated structure in Fig. 4.4.

![Diagram](image)

Figure 4.4: The relation between nodes in the tccp Structure generated

Intuitively, state 1 models the case when the condition in the program holds (thus an error occurs), whereas state 2 represents when the condition does not hold. Consider now state 2. This state is related to states 5 and 6. These states are related to already existing nodes: state 5 travels to states 3 and 4.

---

2The figure does not show the renaming associated to each cyclic edge.
This means that the successors of state 5 are equivalent (following the definition of state equivalence informally given above) to these states. In these cases, the renaming that make the states equivalent is provided.

4.2.2 The build System

In order to illustrate how one tree is generated for each declaration, we show in Fig. 4.5 the labeled version of a tccp program example borrowed from [1]. It consists of two predicates. The first one, build, models the duration of three different tasks of the process of building a house. It gets the value of variables \( D_1, T_1 \) and \( E_1 \) – representing the duration of the tasks that must be scheduled – by calling the predicate get\_constraints. In parallel, an ask agent checks whether the value of the three variables are instantiated to integer numbers and, in that case, several constraints representing the scheduling restrictions are added to the store. Finally, a recursive call to the building process is made, allowing us to recalculate the planning schedule. The predicate get\_constraints is simpler since it just adds to the store the values of the different duration of tasks.

\[
\{ld\} \text{build}([PD|PD'],[PT|PT'],[PE|PE'],[PA|PA']) :-
\{le0\}\exists D_1, T_1, E_1 \{lp1\}\{lx2\} \text{get\_constraints}(D_1, T_1, E_1) \mid
\{la3\} \text{ask}(\text{atom}(D_1) \text{ and atom}(T_1) \text{ and atom}(E_1)) \rightarrow
\{lp4\}\{lt5\} (\text{tell}(PD+D_1=<PT) \mid
\{lp6\}\{lt7\} (\text{tell}(PT+T_1=<PE) \mid
\{lp8\}\{lt9\} (\text{tell}(PE+E_1=<PA) \mid
\{lc10\} \text{build}(PD',PT',PE',PA'))).
\]

\[
\{ld\} \text{get\_constraints}(W_1, I_1, P_1) :-
\{lp0\}\{lt1\} (\text{tell}(W_1) \mid
\{lp2\}\{lt3\} (\text{tell}(I_1) \mid
\{lt4\} \text{tell}(P_1))).
\]

Figure 4.5: The build system program in tccp

The resulting structure from the execution of StructGenerator is shown below. Note that in state 1, the set of labels contains the label to the procedure call agent lx2. state 1 is linked to nodes state 2 and state 3, that, as one can observe, have labels from agents in the get\_constraint declaration and in the build declaration.
Graph(s) successfully created....

-- tree: 0 -> called: build --

-- state 0 --
constraints: (true),
labels: ld-build,
variables: PD, PD', PT, PT', PE, PE', PA, PA',
children:
in tree: 0 -> state: 1 - renaming:

-- state 1 --
constraints: (true),
labels: lx2-build,la3-build,
variables: PD, PD', PT, PT', PE, PE', PA, PA', D1, T1, E1,
children:
in tree: 0 -> state: 2 - renaming:
in tree: 0 -> state: 3 - renaming:

-- state 2 --
constraints: (true),yes(atom(D1) and atom(T1) and atom(E1)),
labels: lt1-get_constraints,lt3-get_constraints,lt4-get_constraints,
lr5-build,lt7-build,lt9-build,lc10-build
variables: PD, PD', PT, PT', PE, PE', PA, PA', D1, T1, E1,
children:
in tree: 0 -> state: 4 - renaming:

-- state 3 --
constraints: (true),
not(atom(D1) and atom(T1) and atom(E1)),
labels: lt1-get_constraints,lt3-get_constraints,lt4-get_constraints,
la3-build,
variables: PD, PD', PT, PT', PE, PE', PA, PA', D1, T1, E1,
children:
in tree: 0 -> state: 5 - renaming:
in tree: 0 -> state: 6 - renaming:

-- state 4 --
constraints: (D1),(T1),(E1),(PD+D1=<PT),(PT+T1=<PE),(PE+E1=<PA),
labels: lx2-build,la3-build,
variables: PD, PD', PT, PT', PE, PE', PA, PA', D1, T1, E1, D1', T1', E1'
4.2. Practical examples

children:
in tree: 0 -> state: 2 - renaming: D1'/D1,T1'/T1,E1'/E1,
in tree: 0 -> state: 3 - renaming: D1'/D1,T1'/T1,E1'/E1,

-- state 5 --
constraints: (D1),(T1),(E1),yes(atom(D1) and atom(T1) and atom(E1)),
labels: lt5-build,lt7-build,lt9-build,lc10-build,
variables: PD, PD', PT, PT', PE, PE', PA, PA', D1, T1, E1,
children:
in tree: 0 -> state: 4 - renaming: PT/PD,T1/D1,PE/PT,E1/T1,PA/PE,

-- state 6 --
constraints: (D1),(T1),(E1),not(atom(D1) and atom(T1) and atom(E1)),
labels: la3-build,
variables: PD, PD', PT, PT', PE, PE', PA, PA', D1, T1, E1,
children:
in tree: 0 -> state: 5 - renaming:
in tree: 0 -> state: 6 - renaming:

-------------------------------------------------------------

-- tree: 1 -> called: get_constraints --

-- state 0 --
constraints: (true),
labels: ld-get_constraints,
variables: W1, I1, P1,
children:
in tree: 1 -> state: 1 - renaming:

-- state 1 --
constraints: (true),
labels: lt1-get_constraints,lt3-get_constraints,lt4-get_constraints,
variables: W1, I1, P1,
children:
in tree: 1 -> state: 2 - renaming:

-- state 2 --
constraints: (W1),(I1),(P1),
labels:
variables: W1, I1, P1,
children:
In Fig. 4.6 we graphically show the relation between states of the system. state 2 represents the case when the condition of the ask agent does hold whereas state 3 represents when the condition does not hold. state 2 is related to state 4 which is related again to state 2 and also to state 3. This is due to the execution of the procedure call agent lc10-build which models the recursive call to the building process. The second tree represents the predicate get_constraints composed by three states which describe the process of adding the corresponding variables to the store.
Conclusions and Future Work

We have shown how tccp is a suitable specification language for communication protocols and the StructGenerator system that, given the specification of a tccp program, constructs a symbolic representation of the set of all possible executions of the program.

On the one hand, features such as concurrency and non-deterministic nature are useful when specifying communication protocols. We have defined a translation from the informal specification of protocols to the formal one. To improve the compactness and clarity of models, we have defined new agents for tccp. We have also shown that many of the components in the defined model can be reused for the specification of other protocols. We have illustrated our approach using the Needham-Schroeder and Otway-Rees protocols and we show how to verify temporal properties.

On the other hand, the symbolic representation, the tccp Structure, generated by the StructGenerator system can be seen as a variant of a Kripke Structure where the notion of node is adapted to the ccp framework and, differently from the classical approach, a renaming may be associated to some edges. Due to the tccp model, the construction of such symbolic representation becomes non trivial, since due to the monotonic behavior of the store, we have to deal with infinite sets of states. To avoid this problem, a notion of equivalence among states is used, which, combined with the use of the current value of streams, allowed us to implement a finite process for the generation. The StructGenerator system is publicly available at the url http://www.dsic.upv.es/~villanue/tccp-StructGenerator/.

We plan to integrate in StructGenerator the new agents recently presented in [19]. We also plan to adapt the model-checking algorithm to this new framework (including the new agents and formulated within the structured computation model of [3]). Note that the tccp Structure generated by StructGenerator system has enough information to be the input of such model-checking tool. Finally, we would like to improve the interface of our system, by automatically showing the graphical version of the relation between nodes.
5. Conclusions and Future Work
Bibliography


