Universidad Politécnica de Valencia

Departamento de Sistemas
Informáticos y Computación

tesis de máster

The Timed Concurrent Constraint language in practice

Candidate:
Alexei Lescaylle Daudinot

Supervisor:
Alicia Villanueva García

– December 2009 –

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

Author's e-mail: alescaylle@dsic.upv.es

Author's address:

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

Dedicated to the memory of my parents,
Raquel Daudinot and
Gregorio Esmerido Lescaylle, and
my aunt Barbara Daudinot

Acknowledgments

I want to express my gratitude to my supervisor Alicia Villanueva García whose dedication and support have contributed largely, in my growth as a researcher.

I sincerely thank to my wife Isabel Serna Miquel and her family for their welcome, love and affection.

I thank from the bottom of my heart to my aunt Isabel Daudinot, her husband José Antonio, my cousins Mara, Bel, Papucho and Dayamí, and the rest of my family for their unconditional love, concern and confidence they placed on me.

Special thanks to my good friends Raunel, Jorge, Pedro, Lorena, Dani, Francho, Diego, Martina, Sonia, Ana, Aristides, Mauricio, we have spent special times together.

Thanks to the members of the ELP group, in particular to María Alpuente Frasnedo, for their welcome, confidence and support.

Alexei Lescaylle Daudinot
December the 10th, 2009.

Abstract

In his PhD thesis, Vijay A. Saraswat introduced the notion of computation as a constraint process, namely a process in which the computation is initiated through the statement of a relation between variables ranging over certain underlying domain of values, this process aims to obtain a characterization of such relation by accumulating consistent pieces of finite information (constraints) each stating some restriction to the feasible values of the variables. To study this kind of processes, he defined the *Concurrent Constraint Programming* framework (ccp in short) that replaces the notion of computation based on the von Neumann model where the state of the system is composed by a set of valuations of variables (*store-as-valuation*), by the notion of computation based on the idea that the state of the system is composed by a conjunction of partial information regarding the variables of the system (*store-as-constraint*). In other words, a store of the ccp model, instead of representing a specific point in the state space of the system, represents a set of states. The computational model of ccp consists of agents (processes) interacting asynchronously by using two basic operations: *tell* and *ask*. The *tell* operation adds a given constraint to the store whereas the *ask* operation checks whether or not a given constraint is entailed by the store. Constraints are never cancelled from the store. These features make ccp a simple and highly expressive paradigm for modeling concurrent systems.

There are several programming languages that extend the ccp model by introducing a notion of time in order to model systems where time aspects are essential, for example, reactive and real-time systems, distributed and concurrent systems. We consider the *Temporal Concurrent Constraint Programming* language (tccp in short). tccp is a declarative programming language that extends ccp by introducing a new agent to capture *negative* information (which is present at typical behaviors of reactive systems) and a global clock that synchronizes agents running concurrently.

In this thesis, we propose the tccp language for the specification and verification of communication protocols, in particular security protocols. We take advantage of the agent-based model and the underlying constraint system to represent the participants of a protocol and to carry out the verification process. The language *Universal Temporal ccp* (utcc in short) has recently been defined under the ccp paradigm to specify and verify mobile reactive systems. We study the relation between tccp and utcc. Specifically, we present a transformation from utcc into tccp that formally states the relation between the two languages. We prove the correctness of the transformation, i.e., that the transformed program preserves the behavior of the original one. The transformation makes possible to reuse the techniques and tools defined for tccp. Finally, we present an interpreter for the tccp language, implemented in Maude, that allows us to simulate the execution of a tccp (or utcc) program and to carry out certain kind of model-checking analysis of these programs.

Resumen

En su tesis doctoral, Vijay A. Saraswat introdujo la noción de computación como un proceso de restricciones, es decir, un proceso en el cual el cómputo se inicia a través de la declaración de una relación entre las variables de un determinado dominio, este proceso tiene como objetivo obtener una caracterización de la relación obtenida mediante la acumulación de información parcial sobre los posibles valores que pueden tomar las variables (restricciones). Para estudiar este tipo de procesos, definió el *Paradigma de Programación Concurrente basado en Restricciones* (ccp) el cual sustituye la noción del computo basado en el modelo de von Neumann, donde el estado del sistema está compuesto por un conjunto de valuaciones de variables (*store-as-valuation*), por la noción del computo basado en la idea en la cual el estado del sistema se compone de una conjunción de información parcial sobre las variables del sistema (*store-as-constraint*). En otras palabras, un store (memoria del sistema) del modelo ccp, en vez de representar un punto específico en el espacio de estados del sistema, representará un conjunto de éstos. El modelo computacional de ccp está basado en agentes (procesos) que interactúan de forma asíncrona mediante dos operaciones básicas: *tell* y *ask*. La operación *tell* añade una restricción al store mientras que la operación *ask* comprueba si una restricción se satisface o no en el store. Las restricciones que se añaden al store nunca se eliminan. Estas características convierten a ccp en un paradigma simple y muy expresivo para el modelado de sistemas concurrentes.

Existen varios lenguajes de programación que extienden el modelo ccp mediante la introducción de una noción de tiempo con el fin de modelar sistemas en los cuales aspectos relacionados con el tiempo son esenciales, por ejemplo, sistemas reactivos y de tiempo real, sistemas distribuidos y paralelos. En esta tesis, consideramos el lenguaje de programación *Temporal Concurrent Constraint Programming* (tccp). tccp es un lenguaje de programación declarativo, que extiende ccp mediante la introducción de un nuevo agente para capturar información *negativa* (comportamiento típico en los sistemas reactivos) y un reloj global que sincroniza los agentes que están siendo ejecutados simultáneamente.

En esta tesis, proponemos el lenguaje tccp para la especificación y verificación de protocolos de comunicación, en particular protocolos de seguridad. Para ello, aprovechamos el modelo basado en agentes y el sistema resolutor de restricciones subyacente para representar los participantes de un protocolo y para llevar a cabo el proceso de verificación en tccp. El lenguaje de programación *Universal Temporal ccp* (utcc) ha sido recientemente definido en el marco del paradigma ccp para especificar y verificar sistemas móviles reactivos. Por ello, estudiamos la relación que existe entre tccp y utcc. En concreto, se presenta una transformación que traduce programas utcc en programas tccp que expresa formalmente la relación entre los dos lenguajes. Por otro lado, demostramos que la transformación es correcta: el programa obtenido preserva el comportamiento original del programa dado. La transformación hace posible la reutilización de las técnicas y herramientas definidas para tccp. Por último, se presenta un intérprete para tccp, implementado en Maude, que nos permite simular la ejecución de un programa tccp (o utcc) y llevar a cabo cierto tipo de análisis, basado en la comprobación de modelos, sobre estos programas.

Contents

List of Figures

1 Introduction

The *Concurrent Constraint Programming* paradigm (ccp in short) was presented by Saraswat in [47, 48] to model concurrent systems. He introduced the notion of computation as a constraint process by replacing the notion of computation based on the von Neumann memory model where the state of the system is composed by a set of valuations of variables (*store-as-valuation*), by the notion of computation based on a model where the state of the system is composed of a conjunction of constraints (*store-as-constraint*). The main difference between the two models is that in the Saraswat's model the store can be seen as a set of valuations that provides partial information regarding the system variables. Therefore, a natural compression of the state space is achieved.

Processes (agents) in the ccp model run concurrently and interact among them asynchronously via the store. They use the basic operations *tell* and *ask* to update and consult the store, respectively. The operation *tell* allows us to add a given constraint to the store. More specifically, it conjoins such constraint to the constraints already in the store. The store grows monotonically, i.e., the added information is never canceled. The operation *ask* checks whether or not a given constraint is entailed by the store. Synchronization is achieved by using a blocking ask (*suspension*), which means that an agent is blocked if the store does not entails the checked constraint. A blocked agent proceeds when the store is strong enough to entail the constraint. In this way, ccp is a simple and highly expressive paradigm for modeling concurrent systems. Apart from the two mentioned operations, other operators are defined in ccp to model several situations such as hiding information, parallel composition or non-determinism. The hiding operator declares variables private to a process. The parallel one stands for the parallel execution of two given processes. The non-determinism allows an agent to consult multiple constraints simultaneously and, in case that any of them is entailed by the store, it executes the associated actions. The ccp model is parametric w.r.t. a constraint system that states the set and type of constraints that agents can use, as well as the relations among them.

The ccp model has been extended in different ways with a notion of time in order to specify and verify systems where *time* is involved, for example in, reactive systems [24] and, in general, applications with timing constraints. Therefore, some new languages have been defined. Some of these languages are the *Timed Default Concurrent Constraint* (dtcc) [22], the *Temporal Concurrent Constraint* language (tccp) [7], the *Non-deterministic Timed Concurrent Constraint* language (ntcc) [42], or the more recent *Universal Temporal Concurrent Constraint* language (utcc) [45]. Although these languages are defined under the same paradigm, the computational model of each one differs due to the different operators and assumptions to model the time.

We are interested in the tccp language that inherits from ccp all its agents as well as its main features (the non-deterministic nature, parametric w.r.t. an underlying constraint system, the monotonic growth of the store, etc.). tccp introduces a *conditional* agent to capture negative information, which is present at some typical behaviors of reactive systems. The notion of time is modeled as a global clock that synchronises the agents of the system under the assumption that basic operations *tell* and *ask* consume one time instant. The *parallel* agent (to model concurrency) is interpreted in terms of *maximal parallelism.* In a recent work [1], a new computational model for the tccp formalism has been proposed. It replaces the notion of the global store by the notion of the store as a timed sequence of stores (*structured store*); each store of the sequence contains the information added at a certain time instant. In this thesis, we consider this new tccp framework.

Reactive systems are systems that maintain an ongoing interaction with their environment [35]. Roughly speaking, they are processes that run in parallel and that are subject to timing constraints. Typical examples of reactive systems are operating systems, communication protocols, process controllers, among others. Communication protocols, and more precisely security protocols, are systems composed of a set of rules (*messages*) that enable the connection, communication and data exchange between two or more entities (called *principals*) in a secure way. Usually, they apply cryptographic methods to ensure the security in the session, but in many cases cryptography is not a guarantee for security. The increasing use of computer networks which make use of security protocols for web connections has made that the analysis of security protocols has gained considerable attention. Many protocols have been proven unsafe years after their definitions. Protocols are vulnerable mainly due to the fact that they are executed in a *hostile environment*, manipulated by an intruder, that controls the network. The intruder can intercept, compose, decompose, replace and replay messages in any session, being thus able to carry out malevolent actions. These capabilities correspond to the well-known Dolev-Yao attacker model [16].

Formalisms which take into account a notion time make the analysis of distributed systems in general, and security protocols in particular, more efficient and effective. The use of timestamps, timeouts and actions (i.e., retransmissions) which must to be executed when a timeout occurs have recently received some attentions [14]. By using tccp, thanks to its implicit notion of time, we can model in a elegant and precise way security protocols which are sensitive to the passage of time. Then, we can perform a precise verification process, improving not-timed approaches, for example these based on some process algebras [40, 26].

Contributions. We propose the tccp language for the specification and verification of communication protocols, in particular security protocols. We take advantage of the agent-based and constraint-based models to specify the interaction between the honest participants of a protocol and the hostile environment (an intruder) that controls the protocol execution. We define how to specify the role of each participant of a protocol and the environment as tccp declarations, and the properties representing the security requirements of the protocol. Then, by using the tccpInterpreter system [31] (a tccp interpreter implemented in the high-performance reflective language Maude [12, 13]) we simulate the execution of a given protocol and also verify some correctness properties.

In previous works [29, 30], we have proposed a similar methodology for specifying protocols, but it was not integrated with an interpreter. The main differences of this work is that we have improved clarity and compactness w.r.t. the previous proposals. We have used the power of the underlying constraint system to include some of the capabilities that were previously placed at the language abstraction level. These capabilities have been incorporated in the tccpInterpreter in such a way that we have provided the framework with a constraint system that includes the necessary operations. We have used the Needham-Schroeder and the Otway-Rees protocols as study cases to show our contribution on this field, but other protocols can also be modeled.

Recently, the utcc language [45] has been introduced to specify and verify mobile reactive systems. The authors showed one of the applications of the language by modeling the Needham-Schroeder protocol. We study the relation between tccp and utcc since they present different features in their definitions, thus the proposals for modeling protocols in both cases are quite different. We define a transformation from utcc programs into tccp programs that preserves the semantics of the original program. The two main challenges we had to overcome for the definition of the transformation were the definition of the abstraction operator of utcc that is not defined in tccp, and to mimic the notion of utcc time with the implicit time of tccp. The defined transformation provides us with an effective method to transform utcc programs into tccp programs, thus being able to apply other transformations such as the abstraction method of [2] or the interpreter presented in [32].

Finally, we describe the implementation of the mentioned interpreter for the tccp language. The tccpInterpreter system is a tool implemented in Maude [12, 13] that allows us to simulate the execution of a program and carry out certain kind of model-checking analysis of tccp programs. Maude has been proposed for the task of building and analyzing a wide range of applications. In particular, rewriting logic [37] can deal with state and concurrent computations and has been used as a semantic framework for giving executable semantics to (concurrent) languages and models. Maude supports structured theory specifications, algebraic data types and function specification in rich equational logics. These features allows us to implement in a elegant way the tccp formalism (the agent-based model, operational semantics and the underlying constraint system). Furthermore, by using the Maude's *search* command we can explore all the possible computations of a given tccp program, looking for safety violations when desiderated. We present the results obtained from carrying out the verification process of the considered protocols by using the interpreter.

Related Work. The verification of security protocols has been widely treated, in the literature through the use of different formalisms [11, 38]. Recently, in [45] was presented the utcc language which allows the specification of mobile behaviors and, due to its strong connection with Pnueli's Temporal logic, to prove reachability properties of utcc processes. In [45], each participant is modeled as an independent process and the underlying constraint system carries out some specific operations related to security protocols. One of the main differences between this approach and ours is that, following the Dolev-Yao model, in our case the attack is not explicitly modeled, but a generic specification for the intruder is used. This means that we are not detecting a known attack, but potentially any attack that the Dolev-Yao intruder is able to run. Moreover, although both languages belong to the same cc paradigm, they are quite different in nature: tccp is non-deterministic whereas utcc is deterministic, and utcc iteration is based on replication whereas tccp uses recursion. These differences make the specification of systems different in the two languages.

In [14], a method to model security protocols in a real-time scenario using reachability properties is proposed. We follow the same idea of [14] by specifying a general Dolev-Yao intruder model and by using model checking to verify the given protocol. In [4] a specific constraint system and some related techniques to analyze security protocols is developed. In tccp the specification and verification process does not depend directly on the constraint system. We just extend the underlying constraint system to represent the information resulting from the execution of a protocol such as the private knowledge of each principal, the representation of a message, etc.

In [51], Syverson and Meadows use a language based on temporal logic. The language allows one to specify security requirements. By using the NRL Protocol Analyzer [39], which is based on an extension of the Dolev-Yao model [16], they determine whether or not a specific protocol has met some of the requirements they specified. They discovered that an attack takes place in the analyzed protocol since a given specified requirement was not reached.

Maude and Haskell have been used to model security protocols [15, 3]. The formalization process in both approaches is similar. Each step of the analyzed protocol is encoded in two rules representing the sending and receiving of a message, respectively. The rewriting logic presented in Maude achieves an understandable formulation. This support is lacking in Haskell due to the use of set comprehension and auxiliary functions to specify the global model. In both formalisms, a model-checking technique is used to explore the state space for possible attacks.

Regarding the study of relations or transformations between languages, the expressive power of different temporal concurrent constraint languages have been previously studied [52, 7, 43, 42, 44]. In [52], it is proven that the temporal language defined in [19] embeds the synchronous data flow language Lustre [10] (restricted to finite value types) and the asynchronous state oriented language Argos [36]. Moreover, the strong abortion mechanism of Esterel can also be encoded in this language. In [7], it is shown that the notion of maximal parallelism of tccp is more expressive than the notion of interleaving parallelism of other concurrent constraint languages. Regarding the utcc language, the authors have shown in [44] that it is turing complete.

Finally, to our knowledge, there is just one implementation of tccp. In [50], it was presented a prototype developed in the Mozart-Oz language. Mozart-Oz [25] is a multi-paradigm language allowing multi-threaded higher order programs to be directly executed in a distributed open system. However, the proposal is not publicly available and does not support the new features of tccp presented during the last years.

As an example of the use of Maude as a semantic framework to provide executable semantics, JavaFAN (Java Formal ANalyzer) [18] is a tool that formally specifies the Java semantics in Maude. In [53], it is presented an interpreter of LOTOS based on the symbolic semantics for Full LOTOS [33]. The operational semantics of CCS [40] is implemented in Maude in [54]. CCS is in some sense similar to tccp. For example, operators like Nil or Choice are present in both languages. However, there are also important differences, for instance, in tccp the model for concurrency is based on maximal parallelism whereas in CCS it is used the interleaving model.

Organization. The thesis is organized as follows. In Chapter 2 we briefly describe the ccp framework and the tccp language, taking into account the new model defined in [1]. An example to show the functionality of the language is presented. Chapter 3 introduces some background material about security protocols. As an example, we describe the informal specification of a variant of the well-known Needham-Schroeder public key authentication protocol and the

Otway-Rees symmetric key authentication protocol.

In Chapter 4 we present our methodology for the formal specification in tccp of a protocol and the hostile environment, which models the actions an intruder may perform following the Dolev-Yao model [16]. We use the Needham-Schroeder and the Otway-Rees protocols as study cases to show the resulting encoding.

In Chapter 5 we introduce the transformation process from utcc into tccp. we describe the utcc language, the resulting encoding, an illustrative example and prove the correctness of our proposal.

In Chapter 6 we describe how the tccp language has been implemented in Maude. We show the codification of the syntax, operational semantics and model for a specific constraint system. Finally, to show one of the functionalities of the tccpInterpreter system, we describe the verification process carried out over the resulting specifications presented in Chapter 4. Finally, in Chapter 7 we conclude and give some directions for future work.

2
The (Timed) Concurrent Constraint Paradigm

In this chapter, we describe the Concurrent Constraint framework and the time extension we consider in the rest of the thesis.

2.1. The Concurrent Constraint Paradigm

Concurrent systems consist of multiple computing processes (agents) interacting among them. Examples of concurrent systems are communication systems based on *message-passing*, communication systems based on *shared-variables*, *synchronous* systems, *mobile* systems, *secure* systems, etc. There are many formalisms to deal with concurrent systems. They allow us to understand the behavior, evolution and the interaction of the components of such systems. For example, some of the main algebraic approaches to model concurrency are the process algebras of Milner's CCS [40], Hoare's CSP [26] and Bergstra's ACP [5].

The Concurrent Constraint Paradigm (ccp) [47, 48] has emerged as a simple and powerful paradigm to model concurrent systems. It extends and subsumes the concurrent logic programming [49] and constraint logic programming [27]. A key feature of ccp is that it replaces the notion of *store-as-valuation* from von Neumann by the notion of *store-as-constraint*. Therefore, in ccp the store can be seen as a set of valuations that provides partial information regarding the possible values that system variables can take. Agents of the model can modify the state of the system by telling and consulting constraints to the store, and they are synchronized by means of a *suspension* mechanism. This mechanism blocks an agent when the store does not entails the constraint the agent wishes to check, and it remains blocked until the store is strong enough to entail such constraint. Basically, the system evolves monotonically by accumulating information in the store. The ccp paradigm is parametric w.r.t. a *constraint system*. A constraint system specifies the basic constraints that agents can tell or ask during their executions.

Agents	$A, B ::= \mathsf{tell}(c)$	Tell
	$\mid A + B$	Choice actions
	$\mid A \parallel B$	Parallel composition
	$\mid X\hat{\ } A$	Hiding
	$\mid \mathsf{p}$	Procedure call
Procedure	$\mathsf{p} ::= \mathsf{g}(x_1, \ldots, x_n)$	
Declaration	$D ::= \mathsf{p} :: A$	Definition
	$\mid D.D$	Conjunction
Program	$P ::= \{D.A\}$	

Figure 2.1: ccp syntax.

Figure 2.1 shows the syntax for ccp programs. The *tell* adds a constraint to the store, the *choice* operator succeeds if either A or B succeeds. The *parallel* operator combines processes concurrently. The *hiding* operator introduces local variables to a process. Finally, the procedure call agent models recursion.

2.2. The Constraint System

The ccp framework is parametric w.r.t. a constraint system which states the constraints that can be used and the entailment relation among them. Let us recall the definition of constraint system described in [45]. A constraint system $\mathcal{C}$ can be represented as the pair (Σ, Δ) where Σ is a signature of function and predicate symbols and Δ is a first-order theory over Σ. Let $\mathcal{L}$ be the first-order language underlying $\mathcal{C}$ with a denumerable set of variables $Var = \{x, y, \ldots\}$, and logic symbols $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow, \exists, \forall$, *true* and *false*. The set of constraints $\mathcal{C} = \{c, d, \ldots\}$ are terms over $\mathcal{L}$. We say that c entails d in Δ, written $c \vdash_\Delta d$, iff $c \Rightarrow d \in \Delta$, in other words, iff $c \Rightarrow d$ is true in all models of Δ. In the following we use $\vdash$ instead of $\vdash_\Delta$ when confusion is not possible. For operational reasons, $\vdash$ is often required to be decidable.

We use $\vec{t}$ for a sequence of terms $t_1, \ldots, t_n$ with length $|\vec{t}| = n$. If $|\vec{t}| = 0$ then $\vec{t}$ is written as ϵ. We use $c[\vec{t} \backslash \vec{x}]$, where $|\vec{t}| = |\vec{x}|$ and x_i's are pairwise distinct, to denote c with the free occurrences of x_i replaced with t_i. The substitution $[\vec{t} \backslash \vec{x}]$ is similarly applied to other syntactic entities.

2.3. The tcc language

Saraswat, Jagadeesan and Gupta defined an extension over time of the ccp paradigm called *Temporal Concurrent Constraint* (tcc) [19]. The tcc language was inspired by synchronous languages such as Esterel [6], Lustre [23] or Signal

Agents	$A, B ::= \mathsf{tell}(c)$	Tell
	$\mid \mathsf{now}\, c\, \mathsf{then}\, A$	Positive Ask
	$\mid \mathsf{now}\, c\, \mathsf{else}\, A$	Negative Ask
	$\mid \mathsf{next}\, A$	Unit Delay
	$\mid \mathsf{abort}$	Abort
	$\mid \mathsf{skip}$	Skip
	$\mid A \parallel B$	Parallel Composition
	$\mid X\hat{}\ A$	Hiding
	$\mid \mathsf{p}$	Procedure Call
Procedure	$\mathsf{p} ::= \mathsf{g}(x_1, \ldots, x_n)$	
Declaration	$D ::= \mathsf{p} :: A$	Definition
	$\mid D.D$	Conjunction
Program	$P ::= \{D.A\}$	

Figure 2.2: tcc syntax.

[28]. The key idea in the definition of tcc was to introduce a notion of *discrete* time and some constructs allowing us to model typical behaviors of *reactive systems* such as *timeout* or *weak preemption.*

Reactive systems are programs that interact with their environment along the time, thus the concept of termination loses significance. They are essentially different from the functional systems whose behavior is simply described as a transformation of the input into the output. Moreover, in reactive systems it is important to capture when events are presented whereas in functional systems it does not matter when the input is given. Reactive systems usually are specified as concurrent systems where the environment is modeled as a concurrent process. Therefore, concurrency is a very important notion for such systems. Moreover, due to the increasingly use of computer networks such as Internet or mobile systems, concurrency has become more and more important.

The tcc language is well suited to specify reactive systems. In particular, due to its deterministic nature, tcc is suitable to model embedded systems (a subclass of reactive systems). In Figure 2.2, we show the syntax of the tcc language. Note that the choice agent is not defined in tcc. The Timing constructs: Timed Negative Ask, Unit Delay, and Abortion causes extension over time. Unit Delay states a process to be started in the next time instant. Timed Negative Ask is a conditional version of the Unit Delay, based on detection of negative information. It causes a process to be started in the next time instant if on quiescence of the current time instant, the store was not strong enough to entail a given information c. The Skip agent does nothing.

Other extensions over time were presented to improve the expressiveness of the ccp model. For example, the *Timed Default Concurrent Constraint Programming* was defined in [22]. This language allows one to model also *strong preemptions.* Moreover, in 1998 Gupta, Jagadeesan and Saraswat presented a language which incorporates a notion of *continuous* (or *dense*) time to the ccp model: the *hybrid* ccp language (hcc) [20]. The hcc language is able to model *hybrid systems* which can be defined as those systems that have a continuous behavior controlled by a discrete component. For example, a thermostat can be seen as an hybrid system. It has a continuous variable modelling the temperature, and a discrete control that turns-on or turns-off the system depending on the limits established for the temperature value. We can find in [21] some applicative examples for both the timed default concurrent constraint programming language and the hcc language.

Other recent models defined within the ccp family are the ntcc [42] and the utcc [45] languages. Both languages are essentially defined as an extension of the tcc. ntcc introduces the notion of non-determinism whereas utcc is deterministic.

2.4. The Universal Temporal Concurrent Constraint language

The Universal Timed Concurrent Constraint Language (utcc in short) [45] extends the deterministic timed language tcc to model mobile behavior. It introduces the parametric ask constructor $(\mathsf{abs}\,\vec{x};c)\,A$, replacing the native ask operation of tcc. The novelty is that the parametric ask not only consults whether or not a condition holds, but also binds the variables $\vec{x}$ in A to the terms that make c true. As in the tcc language, the notion of time is modeled as a sequence of *time intervals.* During each time interval, some *internal steps* are executed until the process reaches a resting point.

Figure 2.3 shows the syntax for ccp programs, borrowed from [45], where the variables in $\vec{x}$ are pairwise distinct.

Intuitively, skip does nothing; $\mathsf{tell}(c)$ adds c to the shared store. $A \parallel B$ denotes A and B running in parallel (interleaving) during the current time interval whereas $(\mathsf{local}\ \vec{x};c)\ A$ *binds* the set of fresh variables $\vec{x}$ in A. Thus, $\vec{x}$ are local to A under a constraint c. The *unit-delay* $\mathsf{next}\,A$ executes A in the next time interval. The *time-out* $\mathsf{unless}\,c\,\mathsf{next}\,A$ executes A in the next time interval iff c is not entailed at the resting point of the current time interval. These two processes explicitly control the time passing. The *replication* operator $!A$ is equivalent to execute $A \parallel \mathsf{next}\,A \parallel \mathsf{next}^2\,A \parallel \ldots$, i.e., unboundly many copies of A, one at a time. Finally, $(\mathsf{abs}\,\vec{x};c)\,A$ executes $A[\vec{t}\backslash\vec{x}]$ in the current time interval for *each term* $\vec{t}$ such that the condition $c[\vec{t}\backslash\vec{x}]$ holds in the store.

The structural operational semantics (SOS) of utcc is shown in Figure 2.4 [45]. It is given in terms of two transition relations between configurations. A configuration is of the form $\langle A, c\rangle$, where A is a process and c a store. The

Agents	$A, B ::= \mathsf{skip}$	Skip
	$\mid \mathsf{tell}(c)$	Tell
	$\mid A \parallel B$	Parallel composition
	$\mid (\mathsf{local}\, \vec{x}; c)\, A$	Local
	$\mid \mathsf{next}\, A$	Unit delay
	$\mid \mathsf{unless}\, c\, \mathsf{next}\, A$	Time out
	$\mid !A$	Replication
	$\mid (\mathsf{abs}\, \vec{x}; c)\, A$	Parametric ask
Declaration	$D ::= \mathsf{p} :: A$	Definition
	$\mid D.D$	Conjunction
Program	$P ::= \{D.A\}$	

Figure 2.3: utcc syntax.

$$\mathbf{R}_T \quad \frac{}{\langle \mathsf{tell}(c), d\rangle \longrightarrow \langle \mathsf{skip}, d \wedge c\rangle}$$

$$\mathbf{R}_P \quad \frac{\langle A, c\rangle \longrightarrow \langle A', d\rangle}{\langle A \parallel B, c\rangle \longrightarrow \langle A' \parallel B, d\rangle}$$

$$\mathbf{R}_L \quad \frac{\langle A, c \wedge (\exists \vec{x}\, d)\rangle \longrightarrow \langle A', c' \wedge (\exists \vec{x}\, d)\rangle}{\langle (\mathsf{local}\, \vec{x}; c)\, A, d\rangle \longrightarrow \langle (\mathsf{local}\, \vec{x}; c')\, A', d \wedge \exists \vec{x}\, c'\rangle}$$

$$\mathbf{R}_U \quad \frac{d \vdash c}{\langle \mathsf{unless}\, c\, \mathsf{next}\, A, d\rangle \longrightarrow \langle \mathsf{skip}, d\rangle}$$

$$\mathbf{R}_R \quad \frac{}{\langle !A, d\rangle \longrightarrow \langle A \parallel \mathsf{next}\, !A, d\rangle}$$

$$\mathbf{R}_A \quad \frac{d \vdash c[\vec{y}/\vec{x}] \quad |\vec{y}| = |\vec{x}|}{\langle (\mathsf{abs}\, \vec{x}; c)\, A, d\rangle \longrightarrow \langle A[\vec{y}/\vec{x}] \parallel (\mathsf{abs}\, \vec{x}; c \wedge \vec{x} \neq \vec{y})\, A, d\rangle}$$

$$\mathbf{R}_S \quad \frac{\gamma_1 \longrightarrow \gamma_2}{\gamma_1' \longrightarrow \gamma_2'} \quad \text{if } \gamma_1 \equiv \gamma_1' \text{ and } \gamma_2 \equiv \gamma_2'$$

$$\mathbf{R}_O \quad \frac{\langle A, c\rangle \longrightarrow^* \langle B, d\rangle \not\longrightarrow}{A \overset{(c,d)}{\Longrightarrow} F(B)}$$

Figure 2.4: Internal and observable reductions of utcc.

internal transition $\langle A, c\rangle \longrightarrow \langle A', c'\rangle$ states that the process A with current store c reduces, in one internal step, to the process A' with store c'. The *observable* transition $A \overset{(c,d)}{\Longrightarrow} B$ says that the process A on input c reduces, in one time interval, to the process B and output d. The function $F(B)$ is used to define the observable transition. It determines the utcc process to be executed in the following time instant looking to the next and unless processes occurring in A. In particular, $F(\mathsf{skip}) = \mathsf{skip}$, $F((\mathsf{abs}\,\vec{x}; c)\,Q) = \mathsf{skip}$, $F(P_1||P_2) = F(P_1)||F(P_2)$, $F((\mathsf{local}\ x; c)\,Q) = (\mathsf{local}\ x)\,F(Q)$, $F(\mathsf{next}\,Q) = Q$ and $F(\mathsf{unless}\,c\,\mathsf{next}\,Q) = Q$. The equivalence relation $\equiv$ states when two configurations are equivalent (for instance $A||B \equiv B||A)$[1].

2.5. The Temporal Concurrent Constraint language

The Temporal Concurrent Constraint Language (tccp) is a declarative language introduced in [7] to model more complex concurrent and reactive systems. The language inherits all the features of the ccp paradigm, including the monotonicity of the store, the non-determinism behavior, and that it is parametric w.r.t. an arbitrary constraint system. tccp introduces an operator for dealing with negative information and a newly implicit notion of time. The notion of time states that, instead of having time intervals as in tcc, each update or consult to the store takes one time instant. Let us briefly recall the syntax of tccp following [7] (Figure 2.5), where c, c_i are *finite constraints* (i.e., atomic propositions) of the underlying constraint system $\mathcal{C}$, and P is a tccp program of the form $D.A$, being D a set of procedure declarations of the form $\mathsf{p}(\vec{x})\ {:-}A$ and, A being an agent that initiates the execution of the program.

Intuitively, the skip agent does nothing; $\mathsf{tell}(c)$ adds the constraint c to the store; the choice agent $\sum_{i=1}^{n}\mathsf{ask}(c_i) \rightarrow A_i$ executes (in the following time instant) one of the agents A_i provided its guard c_i is satisfied. In case no condition c_i is entailed, the choice agent *suspends* (and it is again executed in the following time instant, until the store is able to entail the query). The choice agent models the non-determinism of the language. The new conditional agent now c then A else B executes agent A if the store satisfies c in the current time instant, otherwise executes B (in the same time instant). It models typical behaviors of reactive system such as *timeout* (the ability to wait for a specific signal for a limit of time, thus if such signal is not present yet then an exception is executed) or *preemption* (the ability to abort a process when a specific signal is detected); $A \parallel B$ executes the two agents A and B in parallel (following the *maximal parallelism* model which, instead of the *interleaving* model, executes the agents A and B at the same

[1]See [45] for a complete definitions.

Agents	$A, B ::= \mathsf{skip}$	- Inaction
	$\mid \mathsf{tell}(c)$	- Tell
	$\mid \sum_{i=1}^{n} \mathsf{ask}(c_i) \rightarrow A_i$	- Choice
	$\mid \mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, B$	- Conditional
	$\mid A \parallel B$	- Parallel
	$\mid \exists\, x\, A$	- Hiding
	$\mid \mathsf{p}(\vec{x})$	- Procedure Call
Declaration	$D ::= D.D$	
	$\mid \mathsf{p}(\vec{x})\, \text{:-}\, A$	
Program	$P ::= D.A$	

Figure 2.5: tccp syntax.

time); The $\exists\, x\ A$ agent, also called *hiding*, *restriction* or *locality*, is used to define the variable x local to the process A. We use $\exists\, \vec{x}\ A$ to represent $\exists\, x_1 \ldots \exists\, x_n\ A$; Finally, $\mathsf{p}(\vec{x})$ is the procedure call agent where $\vec{x}$ denotes the set of parameters of the process p. Regarding time passing, only the tell, choice and procedure call agents consume time.

As in ccp, the store behaves monotonically, thus it is not possible to change the value of a given variable along the time. Similarly to the logic approach, to model the evolution of variable values along the time we can use the notion of *stream*. For instance, given an *imperative-style* variable, we write $X = [Y \mid Z]$ to denote a stream X recording the current value Y, and the future values Z. In this thesis, we follow the modified computation model for tccp presented in [1] where the store was replaced by a *structured store* in order to ease the task of updating and retrieving information from the store. A *structured store* consists of a timed sequence of stores st_i where each store contains the information added at the i-th time instant.

Figure 2.6 shows the operational semantics of tccp, borrowed from [1], in terms of a transition system over configurations. Each transition step takes one time unit. The configuration $\langle A, st\rangle_t$ represents the agent A to be executed with the store st at the time instant t.

Let us describe the semantic rules. In the figure, symbol $\not\longrightarrow$ is used to indicate that it is not possible to execute the corresponding agent which means that the given agent suspends. The first rule **R1** describes how the tell agent at time instant t augments the store st by adding the constraint c and then skipping. Therefore, the constraint c will be available to other agents from the time instant $t + 1$. Rule **R2** states that A_j is executed in the following time instant whenever st entails the condition c_j. Regarding the conditional agent, **R3** models the case when the constraint c holds in the store st. Thus, if the agent A with the current store st is able to evolve into the agent A' and a new

R1	$\langle \mathsf{tell}(c), st\rangle_t \longrightarrow \langle \mathsf{skip}, st \sqcup_{t+1} c\rangle_{t+1}$	
R2	$\langle \sum_{i=0}^{n} \mathsf{ask}(c_i) \rightarrow A_i, st\rangle_t \longrightarrow \langle A_j, st\rangle_{t+1}$	if $0 \leq j \leq n, st \vdash_t c_j$
R3	$\dfrac{\langle A, st\rangle_t \longrightarrow \langle A', st'\rangle_{t+1}}{\langle \mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, B, st\rangle_t \longrightarrow \langle A', st'\rangle_{t+1}}$	if $st \vdash_t c$
R4	$\dfrac{\langle B, st\rangle_t \longrightarrow \langle B', st'\rangle_{t+1}}{\langle \mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, B, st\rangle_t \longrightarrow \langle B', st'\rangle_{t+1}}$	if $st \not\vdash_t c$
R5	$\dfrac{\langle A, st\rangle_t \not\longrightarrow}{\langle \mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, B, st\rangle_t \longrightarrow \langle A, st\rangle_{t+1}}$	if $st \vdash_t c$
R6	$\dfrac{\langle B, st\rangle_t \not\longrightarrow}{\langle \mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, B, st\rangle_t \longrightarrow \langle B, st\rangle_{t+1}}$	if $st \not\vdash_t c$
R7	$\dfrac{\langle A, st\rangle_t \longrightarrow \langle A', st'\rangle_{t+1},\ \ \langle B, st\rangle_t \longrightarrow \langle B', st''\rangle_{t+1}}{\langle A \parallel B, st\rangle_t \longrightarrow \langle A' \parallel B', st' \sqcup st''\rangle_{t+1}}$	
R8	$\dfrac{\langle A, st\rangle_t \longrightarrow \langle A', st'\rangle_{t+1},\ \ \langle B, st\rangle_t \not\longrightarrow}{\langle A \parallel B, st\rangle_t \longrightarrow \langle A' \parallel B, st'\rangle_{t+1}}$	
R9	$\dfrac{\langle A, st_1 \sqcup \exists x\, st_2\rangle_t \longrightarrow \langle A', st'\rangle_{t+1}}{\langle \exists^{st_1} x\, A, st_2\rangle_t \longrightarrow \langle \exists^{st'} x\, A', st_2 \sqcup \exists x\, st'\rangle_{t+1}}$	
R10	$\langle \mathsf{p}(\vec{x}), st\rangle_t \longrightarrow \langle A, st\rangle_{t+1}$ if $\mathsf{p}(\vec{x}) : -A \in D$	

Figure 2.6: tccp operational semantics

store st' then, A' is executed in the following time instant with the computed store st'. **R4** models the case when the condition c does not hold. In this case, B' is executed in the following time instant with the store st' computed by B. **R5** models the case when the condition c holds, but the agent A cannot evolve. When this occurs, A is executed again in the following time instant. Similarly, **R6** models the case when the condition c does not hold, and B cannot evolve. **R7** models the evolution of the parallel agent: if A with store st is able to evolve into A' with a new computed store st', and also B with store st is able to evolve into an agent B' with st'', then $A'||B'$ is run in the following time instant with the store resulting from the conjunction of st' and st''. **R8** models the case of the parallel agent when one of the branches evolves and the other does not. In this case, the configuration resulting of the execution of the branch that can evolve is executed in the following time instant in parallel with the branch that cannot evolve. **R9** specifies the evolution of the hiding agent. Intuitively, if there exists a transition $\langle A, st_1 \sqcup \exists x\, st_2\rangle_t \longrightarrow \langle A', st'\rangle_{t+1}$, then st' is the local information produced by A; moreover, st' must be hidden $(st_2 \sqcup \exists x\, st')$ from the main process. Finally, rule **R10** models the evolution of the procedure call agent. It executes in the following time instant the agent A, which is recovered from the procedure declaration.

Now we recall the definition of the observable behavior of the tccp language, which is described from the transition system in Figure 2.6. Let $d = d_0 \cdot d_1 \cdot \ldots \cdot d_i \cdot \ldots$ be a structured store where d_n is the information computed at time instant n.

Definición 2.5.1 (Observable Behavior of tccp programs [1]) *Given a tccp program P, an agent A_0, and an initial structured store $st = st^0_0 \cdot true^\omega$ where st^0_0 represents the first component of st^0, the* timed operational semantics *of P w.r.t. the initial configuration $\langle A_0, st^0\rangle$ is defined as:*

$$\mathcal{O}(P)[\langle A_0, st^0\rangle] = \{st = st^0_0 \cdot st^1_1 \cdot \ldots \mid \langle A_i, st^i\rangle_i \longrightarrow \langle A_{i+1}, st^{i+1}\rangle_{i+1} \textit{ for } i \geq 0\}$$

Thus, for each st^i incrementally built during the execution, the semantics only records its i^{th} component st^i_i, which corresponds to the constraints added at the time instant i.

An example. We can find in the literature a number of examples of systems that can be modeled using the tccp language. In the following, we describe a simple case to illustrate the use of tccp. In Figure 2.7 we present a tccp procedure declaration modeling a simplified module of a vending machine.

The declaration has two parameters P and S. P represents the product selected by the client whereas S represents the current credit available in the machine. The system is specified by means of a hiding agent introducing the local variables R, D and S' to the choice agent, that consists of five branches. In the first branch, the choice agent checks whether the product selected by the client is coded with the number 1. In this case, by means of the conditional agent, it checks that if the credit is greater or equal than the price of the selected product. Then, it emits in parallel two signals via tell agents. The first tell agent adds to the store the constraint $R := 1$, which means that the system must return the selected product; the second tell agent adds to the store the constraint $D := S - 1.5$, which means that the system must return the rest. If there was no enough credit, then by using the tell agent in the else block ($\mathsf{tell}(S' := 1.5)$), it emits a signal notifying the price of the selected product (the constraint $S' := 1.5$). The second, third and fourth branches model similar cases. Finally, the fifth branch specifies the case when the client cancels the operation. The system must return the current credit. This action is modeled when the tell agent adds the constraint $D := S$ to the store.

```
vendedor_products (P,S) :-
   ∃ R,D,S' ((ask(P == 1)→ now(S >= 1.5)
                              then (tell(R := 1) || tell(D := S − 1.5))
                              else tell(S' := 1.5) +
            (ask(P == 2)→ now(S >= 1.8)
                              then (tell(R := 2) || tell(D := S − 1.8))
                              else tell(S' := 1.8) +
            (ask(P == 3)→ now(S >= 0.8)
                              then (tell(R := 3) || tell(D := S − 0.8))
                              else tell(S' := 0.8) +
            (ask(P == 4)→ now(S >= 2.0)
                              then (tell(R := 4) || tell(D := S − 2))
                              else tell(S' := 2.0) +
             ask(P == "Canceled")→ tell(D := S)))))).
```

Figure 2.7: A tccp model for a module of a vending products machine

3 An overview about security protocols

In this chapter, we briefly discuss some basic notions about security protocols, also called cryptographic protocols.

From a general view, a protocol can be seen as a set of rules that make possible the connection, communication and data transfer between two or more entities (*principals*) across a network via *messages* exchange. These rules establish the syntax, semantics, and synchronization of the messages, which are usually composed of principals' names, *nonces* and/or *keys*. A nonce is a fresh value generated by a principal that is usually used to ensure the freshness of messages. A key is a piece of information used to encrypt and decrypt messages.

A security protocol is a concrete protocol that performs a security-related function (confidentiality, authenticity, etc.) and applies cryptographic methods to ensure the security of a message. There are different classifications of protocols depending on their goal [9] (i.e., key establishment, real-time authentication, etc.). They can also be classified depending on the kind of keys that is used to encrypt and decrypt messages (public key, shared key, etc.).

We use the Needham-Schroeder public key authentication protocol and the Otway-Rees symmetric key authentication protocol as study cases to show how tccp can be used to generically specify protocols. The first protocol uses the mechanism of public key encryption to ensure confidentiality of messages whereas the second uses shared keys and a trusted server for the generation of the session key.

We have chosen these two protocols since they use different encryption strategies and are vulnerable to different kinds of attacks.

3.1. The Needham-Schroeder protocol

We consider the simplified version of the Needham-Schroeder public key authentication protocol [41]. This authentication protocol allows principals A (*Alice*) and B (*Bob*) to communicate via the interchange of secret nonces which, besides to serve as nonces, serve as authenticators. It is based on the use of public keys cryptography and nonces, and aims to guarantee confidentiality and authentication, i.e., the nonce received by A from B and the nonce received by B from A must be knew only by them. The informal (standard) protocol definition is shown bellow:

$$\begin{array}{lll} 1. & A \rightarrow B: & \{A, N_A\}_{K_B} \\ 2. & B \rightarrow A: & \{N_A, N_B\}_{K_A} \\ 3. & A \rightarrow B: & \{N_B\}_{K_B} \end{array}$$

The protocol begins when A (initiator) sends to B (responder) a message, encrypted with B's public key K_B, containing her identifier A and a nonce N_A generated by her. When B receives the first message, he decrypts it by using his private key and sends to A the second message of the protocol that contains the received nonce N_A and a new nonce N_B generated by him. This message is encrypted with the public key of A. When A receives and decrypts the message by using her secret key, she deduces that, since the message contains the nonce N_A, it has recently been sent by B. Finally, A sends the confirmation message (the last message of the protocol) to B. In this way, B can infer that he is communicating to A since he had previously sent her the nonce N_B.

3.2. The Otway-Rees protocol

The Otway-Rees protocol symmetric key authentication protocol [46] 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 informal specification of the protocol as follows, where 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:

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

The protocol starts when A (initiator) sends to B (responder) a message which contains the non-encrypted text N, A, B and the text N_A, N, A, B encrypted with the secret key SK_{AB} shared by her and S (server), where N and N_A are the nonces generated by A. N is used to identify the session and N_A is used to ensure the freshness of the message. When B receives the first message, he forwards it to S with the text N_B, N, A, B encrypted with the key SK_{BS} shared by him and the server. When S receives and decrypts the message, it checks whether elements N, A, B from the three components of the message coincide. In that case, it generates the secret key SK_{AB} and sends to B a message containing 1) the session identifier N not encrypted, 2) the elements N_A and SK_{AB} encrypted with SK_{AS}, and 3) the elements N_B and SK_{AB} encrypted with SK_{BS}. When B receives the message, he extracts from the third component of the message the key SK_{AB}. Since the message contains the session identifier N and the third component contains the nonce generated by him, he deduces that such key has been generated by the server in the current session. Then, he forwards to A the message containing the first and the second components of the received message. When A receives and decrypts such message, she extracts from the second component of the message the key SK_{AB}. she also deduces that the key SK_{AB} has been generated by the server in the current session.

3.3. A classification of attacks

Security protocols are vulnerable to different forms of *attacks*. An attack take places when a intruder or attacker (a dishonest principal) exploits a specific feature or implementation bug of a protocol in order to carry out malevolent actions. Below, we mention some kind of attacks (see [11] for a detailed description).

- Parallel session: In this kind of attack an intruder forms messages for a given protocol run by using messages coming from another legitimate protocol session that is being concurrently executed. There are several forms of parallel session attacks such as oracle attacks, man-in-the-middle attacks or multiplicity attacks. A man-in-the-middle attack occurs when two principals believe they are mutually authenticated when, actually, the intruder is impersonating one of the principals in one session and the other principal in the second session.

- Type flaw: This attack takes place when a principal is induced by the intruder to erroneously interpret the structure of a message. For example, the principal might accept a nonce as a key when their length coincide.

- Denial-of-service: It is characterized by an explicit attempt to prevent legitimate users of a service from using it.

- Freshness: It is one of the most common attacks on authentication and key-establishment protocols. An intruder can get himself authenticated by replaying messages copied from a legitimate authentication session.

- Implementation-dependent: This kind of attack depends on how a given protocol is implemented.

3.4. The Dolev-Yao Intruder Model

Most proposals related to the specification and verification of protocols, including ours, are based on a set of assumptions. These assumptions are known as the Dolev-Yao intruder model which was developed on the basis of some assumptions described by Needham-Schroeder in [41] and Dolev-Yao in [16].

According to [16], messages are considered as indivisible abstract values, instead of sequences of bits, and the model of encryption is perfect (*black-box* model). These two assumptions simplifies the analysis of protocols, but they have the drawback of preventing the discovery of implementation dependent attacks. In general, the Dolev-Yao model consists of a set of assumptions representing a worst-case scenario where an intruder has complete control of the network (a hostile environment). The intruder has the capabilities of intercept, block, replay, compose (with the information in his/her possession) and decompose (provide he/she knows the appropriate decryption key) messages in the network. The intruder, as any participant in the protocol, knows the identity and the public key (in case of public key encryption) of the other principals, and contains a private memory to store the information acquired in a session. He/she is supposed not to know the private keys of the other participants, unless these have been disclosed in some way.

In order to better understand the problem of defining and analyzing protocols, in the following we present two attacks to the two considered protocols.

3.5. An attack to the Needham-Schroeder protocol

The Needham-Schroeder protocol was defined relying upon the assumption of perfect cryptography and that principals do not divulge secrets. However, it allowed the following man-in-the-middle attack discovered by Lowe in [34] where I_X denotes I impersonating X:

$$\begin{array}{lll}
1. & A \rightarrow I : & \{A, N_A\}_{K_I} \\
2. & I_A \rightarrow B : & \{A, N_A\}_{K_B} \\
3. & B \rightarrow A : & \{N_A, N_B\}_{K_A} \\
4. & A \rightarrow I : & \{N_B\}_{K_I} \\
5. & I_A \rightarrow B : & \{N_B\}_{K_B}
\end{array}$$

The man-in-the-middle attack describes how an intruder can discover a secret nonce. In the attack, A initiates a protocol run with I, 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. More precisely, the attack follows the following steps. First, A sends his name and a nonce to I. Later, I, impersonating A, sends to B the received message. Then, B, thinking that the received message comes from A, sends to her the received nonce and a new one generated by him (corresponding to the second step of the protocol). When A receives the message sent by B, she thinks that it comes from I (the second step of the first protocol run), and sends back to I the confirmation message that contains the nonce generated by B to A. Finally, I sends to B the same confirmation message. At the end of the attack, B thinks he is communicating with A, which is false.

3.6. An attack to the Otway-Rees protocol

Boyd, in [8], found one of the typing attacks that the Otway-Rees protocol suffers. In this attack, the intruder I intercepts the first message. Then, he/she creates the last message of the protocol, by eliminating from the non-encrypted text the identifiers A and B, and replies it to A impersonating B:

$$\begin{array}{l} 1.\ A \rightarrow I_B : N, A, B, \{N_A, N, A, B\}_{K_{AS}} \\ 4.\ I_B \rightarrow A : N, \{N_A, N, A, B\}_{K_{AS}} \end{array}$$

In particular, although the intruder cannot read the content of the encrypted part $\{N_A, N, A, B\}_{SK_{AS}}$ of the message, he/she can use it to send it (together with the session identifier N) to the honest participant A. This message replaces the legitimate last message of the protocol. The important point is that the principal A expects a message of the form $\{N_A, SK_{AB}\}_{SK_{AS}}$, being SK_{AB} the new key to be used to communicate with another participant, for example B. Therefore, she interprets that the sequence N, A, B is the expected key. From that moment, A encrypts its messages using as key the sequence N, A, B which the intruder knows since it corresponds with the non-encrypted text (the first component) of the message that the intruder can disclose. Therefore, the communication becomes insecure.

4 Modeling security protocols in tccp

In this chapter, we describe how we can specify in a general way in tccp the principals' roles of a given protocol and the intruder model of Dolev-Yao [16]. Specifically, we define some specific constraints (which are handled by the underlying constraint system) to model some of the concepts involved in a protocol definition. We specify a tccp declaration to model the role of each principal and a tccp declaration to model a hostile environment (modeling the intruder) that controls the protocols run. As practical examples, we show how to formally specify the Needham-Schroeder and the Otway-Rees protocols.

We can divide the specification process into three parts:

1. The representation of the concepts involved in a protocol by means of certain terms;
2. The encoding of the behavior of each participant of a protocol in a tccp declaration;
3. The codification as a tccp declaration of the *(hostile) environment*, a process that controls the protocol execution following the Dolev-Yao attacker model [16].

The specifications obtained from the first and third phases can be reused for the specification of other protocols. For the second phase, the example that we describe should serve as a guideline to transform the informal definition of a protocol to the corresponding tccp program.

4.1. Auxiliary concepts

Let us describe the concepts appearing in the informal protocol definition by means of terms which in our approach are handled as constraints.

We represent the private knowledge acquired by the members of a protocol with the term `know(`A`,`K`)` where A is the identifier of a principal, and K is a stream that stores the information. For instance, `know(`$alice$`,[a(`b`), n(`nN_A`),` `n(`nN_B`)|`T`])` states that the agent *alice knows* that b is the identifier of a protocol participant, and also knows the nonces nN_A and nN_B.

Messages involved in the protocol are represented by a term of the form `msg(`*ItemList*`)`. When a message is posted to the network, a term `chn(`*Msg*`,`*State*`)` is defined. The content of a message *ItemList* is a list of elements (or components). Each element is represented by a term of the form `enc(`*Key*`,`*Content*`)` stating that the information in *Content* has been encrypted with the key *Key*. A key is represented by terms `plain`, `k/1`, `pk/1`, or `sk/2`. `plain` means that the given content is non-encrypted, `k/1` is used to represent the public key of a principal whose identifier is the argument of the term, `pk/1` represents a private key which may be known by certain principals and finally, `sk/2` models a secret key shared by two principals whose identifiers are the arguments of the term. The information is represented as a list of atoms following the classical notation for the `cons` constructor of lists. The list of atoms may contain participant's identifiers, nonces, keys, etc...In `chn(`*Msg*`,`*State*`)`, the variables *Msg* and *State* represent the sent message and its status. There are two possible status: that the message has traveled the network but it has not been processed yet (*State* remains non-instantiated) or that the message has been already processed by the environment (*State* is instantiated to the value `ok`). We adopt an asynchronous model for message passing. This means that, when a principal sends a message, it is stored in the channel to be processed later. The term `rcv(`*Msg*`,`*State*`)` is used by the environment to deliver a message to the corresponding recipient. The variable *State* is instantiated to `ok` when the expected principal has processed the delivered message.

Finally, the term `blk(`*Msg*`)` is used to state that a message has been blocked, whereas `cnt(`*Content*`)` stores the information being the content of a message.

The constraint system. In addition to the representation of the information to be handled during the protocol run, some operations may be implemented in the underlying constraint system. Therefore, we assume that the constraint system provides the following functionality. When invoked, $new(N_A)$ generates a fresh (random) value for the given variable N_A. In our specifications, when we use this function, the variable passed as argument is never instantiated.

The function *free(Var)* returns a boolean value. It is used to check whether, given the current store, the given variable *Var* is instantiated or not. To recover certain information from a given stream, *recover(Stream,Info)* is defined. This function unifies the value matching *Info* in the stream. We can also use the auxiliary function *find(Stream,Info)* that checks whether *Info* can be recovered (returning the value *true*) or not (*false*). To determine this, the function checks if *Info* unifies with, at least, one of the values stored in the given stream. Note that, in contrast, *recover* returns the resulting substitution. For example, a call to *recover(*`[ a(`b`), n(`nN_A`), n(`nN_B`)|`T`]` *,* `a(`Ag`)`*)* would return $Ag = b$.

4.2. Encoding the Needham-Schroeder protocol into tccp

Let us now show how the participants of the protocol can be specified. The specification of principals must be redefined for each different protocol, and it is modeled as a tccp declaration.

The tccp declaration modeling the initiator A is shown below.

```
init (A,B) :-
 ∃ N_A,S_A,S_1,S_A1,N_B,S_2,S_3,S_A2
   (tell(N_A = new(N_A)) ||
   (tell(know(A,S_A)) ||
    ask(true)->
   1   (tell(chn(msg(enc(k(B),cons(A,cons(N_A,nil)))),S_1)) ||
       (tell(S_A = [(a(B),n(N_A)) | S_A1]) ||
   2    ask(rcv(msg(enc(k(A),cons(N_A,cons(N_B,nil)))),S_2) ∧ free(S_2))->
           (tell(rcv(msg(enc(k(A),cons(N_A,cons(N_B,nil)))),S_2) ∧ free(S_2)) ||
           (tell(S_2 = ok) ||
   3        ask(true)-> (tell(chn(msg(enc(k(B),cons(N_B,nil))),S_3)) ||
                          tell(S_A = [n(N_B) | S_A2]))))))))).
```

Following the steps in the informal protocol description, the initiator A generates a new nonce, stored in N_A, and states S_A as the stream that contains her private knowledge. Then, by means of the tell agent labeled with 1, she sends to the channel the message of the first protocol step. This means that she sends to B her identifier and the generated nonce, both things encrypted with the public key of B. This is modeled by `chn(msg(enc(k(`B`),cons(`A`,cons(`N_A`,`nil`)))),`S_1`)`. In parallel, she updates her private knowledge S_A with the identifier of the responder and the generated nonce. Again in parallel, the choice agent (labeled with 2) will be executed for detecting when A receives the second message of the protocol (characterized by the constraint `rcv(msg(enc(k(`A`),cons(`N_A`,cons(`N_B`,`nil`)))),`S_2`)` $\wedge$ `free(`S_2`)`, and `free(`S_2`)` ensures that the message has not been processed). When the condition holds, by means of the tell agents, she recovers in N_B the nonce generated by B and sets S_2 to `ok` (to ensure that the message is processed just one time). Then, she sends to B (the action labeled with 3) the confirmation message that contains the recovered nonce, and updates her private knowledge with such nonce.

The tccp declaration modeling the responder B is shown below.

```
resp (B) :-
 ∃ A,N_A,S_1,N_B,S_B,S_2,S_B1,S_3,S_B2
    ask(rcv(msg(enc(k(B),cons(A,cons(N_A,nil)))),S_1) ∧ free(S_1))->
       (tell(rcv(msg(enc(k(B),cons(A,cons(N_A,nil)))),S_1) ∧ free(S_1)) ||
       (tell(S_1 = ok) ||
       (tell(N_B = new(N_B)) ||
       (tell(know(B,S_B)) ||
        ask(true)->
          (tell(chn(msg(enc(k(A),cons(N_A,cons(N_B,nil)))),S_2)) ||
          (tell(S_B = [(a(A),n(N_A),n(N_B)) | S_B1]) ||
           ask(rcv(msg(enc(k(B),cons(N_B,nil))),S_3) ∧ free(S_3))->
             (tell(rcv(msg(enc(k(B),cons(N_B,nil))),S_3) ∧ free(S_3)) ||
             (tell(S_3 = ok) ||
              ask(true)-> tell(S_B = [secret(N_B) | S_B2]))))))))).
```

The execution of the responder B is suspended until the constraint in the first choice agent rcv(msg(enc(k(B),cons(A,cons(N_A,*nil*)))),S_1) $\wedge$ free(S_1) is satisfied. This situation models the moment in which he has received the first message of the protocol. Then, by means of the tell agents, he recovers in A and N_A the identifier of the initiator A and the nonce generated by her, respectively; sets S_1 to the value ok, generates the new nonce N_B and states that S_B is being to contain his private knowledge. After this, the specification sends to the channel the second message of the protocol. In particular, B sends to the principal whose identifier has been recovered from the received message the received nonce and a new one generated by him, all this encrypted with the public key of the target principal. Finally, he updates his private knowledge with the recovered information and the generated nonce. After having sent the second message, B waits until the constraint rcv(msg(enc(k(B),cons(N_B,*nil*))),S_3) $\wedge$ free(S_3) of the choice agent holds, which means that he has received the last message of the protocol. Then, he sets S_3 to the value ok and updates his private knowledge by storing the fact that the nonce previously generated by him N_B is secret.

4.3. Encoding the Otway-Rees protocol into tccp

Let us now show how the participants of the Otway-Rees protocol can be specified. They are modeled, each one, as a tccp declaration.

The tccp declaration modeling the initiator A is shown below.

```
init (A,B,S) :-
∃ N,N_A,S_A,S_1,S_{A_1},K_{AB},S_2,S_{A_2}
  (tell(N = new(N)) ||
  (tell(N_A = new(N_A)) ||
  (tell(know(A,S_A)) ||
   ask(true)->
     (tell(chn(msg((enc(plain,cons(N,cons(A,cons(B,nil)))),
          enc(sk(A,S),cons(N_A,cons(N,cons(A,cons(B,nil))))))),S_1)) ||
     (tell(S_A = [(a(B),a(S),n(N),n(N_A)) | S_{A_1}]) ||
      ask(rcv(msg((enc(plain,cons(N,nil)),
          enc(sk(A,S),cons(N_A,cons(K_{AB},nil))))),S_2) ∧ free(S_2))->
        (tell(rcv(msg((enc(plain,cons(N,nil)),
            enc(sk(A,S),cons(N_A,cons(K_{AB},nil))))),S_2) ∧ free(S_2)) ||
        (tell(S_2 = ok) ||
         ask(true)-> tell(S_A = [pk(K_{AB}) | S_{A_2}])))))))).
```

Following the steps in the informal protocol description, the initiator A generates, in parallel, the new nonces N and N_A, and states S_A as the stream that contains her private knowledge. Then, by means of a tell agent, she sends to the channel the message of the first protocol step. This means that she sends to B a message containing a non-encrypted part (`enc(`*plain*`,cons(`N`,cons(`A`,cons(`B`,`*nil*`))))`) and an encrypted part (`enc(sk(`A,S`),cons(`N_A`,cons(`N`,cons(`A`,cons(`B`,`*nil*`)))))`). In parallel, she updates her private knowledge S_A with the identifier of the responder, the server and the generated nonces. Again in parallel, a choice agent will be executed for detecting when A receives the last message of the protocol (characterized by the constraint `rcv(msg((enc(`*plain*`,cons(`N`,`*nil*`)),enc(sk(`A,S`),cons(`N_A`,cons(`K_{AB}`,`*nil*`))))),`S_2`)` $\wedge$ free(S_2), and free(S_2) ensures that the message has not been processed). When the condition holds, by means of the tell agents, she recovers in K_{AB} the key generated by the server S and sets S_2 to `ok` (to ensure that the message is processed just one time). Then, she updates her private knowledge with the recovered key.

The tccp declaration modeling the responder B is shown below.

```
resp (B,S) :-
∃ N,A,Enc_1,S_1,N_B,S_B,S_2,S_{B_1},Enc2,K_{AB},S_3,S_4,S_{B_2}
 ask(rcv(msg((enc(plain,cons(N,cons(A,cons(B,nil)))),Enc_1)),S_1) ∧
     free(S_1))->
   (tell(rcv(msg((enc(plain,cons(N,cons(A,cons(B,nil)))),Enc_1)),S_1) ∧
        free(S_1)) ||
   (tell(S_1 = ok) ||
   (tell(N_B = new(N_B)) ||
   (tell(know(B,S_B)) ||
    ask(true)->
      (tell(chn(msg((enc(plain,cons(N,cons(A,cons(B,nil)))),Enc_1,
      enc(sk($B,S$),cons($N_B$,cons($N$,cons($A$,cons($B$,$nil$)))))),$S_2$)) ||
  (tell($S_B$ = [(a($A$),a($S$),n($N$),n($N_B$)) | $S_{B_1}$]) ||
   ask(rcv(msg((enc($plain$,cons($N$,$nil$)),$Enc2$,
      enc(sk($B,S$),cons($N_B$,cons($K_{AB}$,$nil$)))),$S_3$) ∧ free($S_3$))->
    (tell(rcv(msg((enc($plain$,cons($N$,$nil$)),$Enc2$,
        enc(sk($B,S$),cons($N_B$,cons($K_{AB}$,$nil$)))),$S_3$) ∧ free($S_3$)) ||
    (tell($S_3$ = $ok$) ||
     ask($true$)-> (tell(chn(msg((enc($plain$,cons($N$,$nil$)),$Enc2$)),$S_4$)) ||
                 tell($S_B$ = [pk($K_{AB}$) | $S_{B_2}$])))))))))).
```

The execution of the responder B is suspended until the constraint in the first choice agent rcv(msg((enc($plain$,cons(N,cons(A,cons(B,nil)))),Enc_1)), S_1) ∧ free(S_1) is satisfied. This means that he has received the first message of the protocol. Then, by means of the tell agents, he recovers in N, A and Enc_1 the session identtifier, the identifier of the initiator A and the encrypted component of the message, respectively; sets S_1 to the value ok, generates the new nonce N_B and states that S_B is being to contain his private knowledge. After this, the specification sends to the channel the second message of the protocol. In particular, B sends to the server the received message with an encrypted component (enc(sk(B,S),cons(N_B,cons(N,cons(A,cons(B,nil)))))) generated by him. In parallel, he updates his private knowledge with the recovered information and the generated nonce (S_B = [(a(A),a(S),n(N),n(N_B)) | S_{B_1}]). After having sent the second message, B waits until the constraint rcv(msg((enc($plain$,cons(N,nil)),$Enc2$,enc(sk(B,S),cons(N_B,cons(K_{AB}, nil)))),S_3) ∧ free(S_3) of the choice agent holds, which means that he has received the third message of the protocol. When the condition holds, by means of the tell agents, she recovers in $Enc2$ and K_{AB} the component encrypted by the server S for A and the secret key generated by the server S, and sets S_3 to ok (to ensure that the message is processed just one time). After this, the specification sends to the channel the last message of the protocol. In particular, B forwards to the initiator A the message received from the server, but without the last component (chn(msg((enc($plain$,cons(N,nil)),$Enc2$)),S_4)). In parallel, he updates her private knowledge with the recovered key.

The tccp declaration modeling the server S is shown below.

```
server ($S$) :-
∃ $N,A,B,N_A,N_B,S_1,K_{AB},S_S,S_2,S_{S_1}$
 ask(rcv(msg((enc($plain$,cons($N$,cons($A$,cons($B$,$nil$)))),
          enc(sk($A,S$),cons($N_A$,cons($N$,cons($A$,cons($B$,$nil$))))),
          enc(sk($B,S$),cons($N_B$,cons($N$,cons($A$,cons($B$,$nil$)))))),$S_1$) ∧
    free($S_1$))->
   (tell(rcv(msg((enc($plain$,cons($N$,cons($A$,cons($B$,$nil$)))),
           enc(sk(A,S),cons(N_A,cons(N,cons(A,cons(B,nil))))),
           enc(sk(B,S),cons(N_B,cons(N,cons(A,cons(B,nil)))))),S_1) ∧
       free(S_1)) ||
   (tell(S_1 = ok) ||
   (tell(K_AB = new(K_AB)) ||
   (tell(know(S,S_S)) ||
    ask(true)->
      (tell(chn(msg((enc(plain,cons(N,nil)),
                   enc(sk(A,S),cons(N_A,cons(K_AB,nil))),
                   enc(sk(B,S),cons(N_B,cons(K_AB,nil))))),S_2)) ||
       tell(S_S = [(a(A),a(B),n(N),n(N_A),n(N_B),pk(K_AB)) | S_S1])))))).
```

The execution of the server S is suspended until the constraint in the first choice agent is entailed by the store. This situation models the case in which he/she has received the second message of the protocol. Note that the sequence cons(N,cons(A,cons(B,nil))) must be in the three components of the message. Then, by means of tell agents, he recovers in N, A, B, N_A and N_B the session identifier, the identifier of the initiator A, the identifier of the responder B, the nonce generated by A and the nonce generated by B, respectively; sets S_1 to the value ok, generates the secret shared key K_{AB} and states that S_S is being to contain his private knowledge. After this, the specification sends to the channel the third message of the protocol. In particular, S sends to the responder B (whose identifier has been recovered from the received message) the message chn(msg((enc($plain$,cons(N,nil)), enc(sk(A,S),cons(N_A,cons(K_{AB},nil))),enc(sk(B,S),cons(N_B,cons(K_{AB}, nil))))),S_2) composed of the session identifier in plain text, the component created for A encrypted with the key shared by her and the server containing the nonce generated by A and the secret key generated by the server, and the component created for B encrypted with the key shared by him and the server containing the nonce generated by B and the secret key generated by the server. Finally, the server updates his private knowledge with the recovered information and the generated key.

4.4. Encoding the Environment into tccp

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 a malicious agent (an intruder) who can intercept, analyze, and modify messages, being thus able to carry out malevolent actions.

In this section, we describe the specification of the network where the protocols are executed. We do not specify the intruder as a specific principal, but we encode it within the environment. The resulting specification allows the actions from the Dolev-Yao intruder model [16].

The environment is defined as a cycle where, during each iteration, a message is processed depending on whether the constraint (chn(M,S) $\wedge$ free(S)) of the first choice agent holds, which means that there is a message to process, or not. The tell agents after the condition recover in M the sent message, sets S to the value ok (to avoid processing a message twice) and states that S_I is going to store the intruder private knowledge. Then, one of the labeled actions (1a to 1d) is non-deterministically chosen to be executed. Finally, the environment is executed recursively in parallel.

```
environment(I) :- ∃M,S,SI,Si (
  ask(chn(M,S) ∧ free(S)) ->
     (tell(chn(M,S) ∧ free(S)) ||
     (tell(S = ok) ||
     (tell(know(I,S_I)) ||
  1a ((ask(true) -> tell(SI = [blk(M) | Si]) +
  1b  (ask(true)-> ∃S_1 tell(rcv(M,S_1)) +
  1c  (ask(true) ->
          now(M = msg(enc(k(I),_))) then
             ∃C,X,S1 (tell(M = msg(enc(k(I),C))) ||
                         ask(true) -> (tell(SI = [cnt(C) | Si]) ||
                                      (tell(recover(SI,a(X))) ||
                                       ask(true)->
                                          tell(rcv(msg(enc(k(X),C)),S1)))))
          else skip +
  1d   ask(true) ->
          now(find(SI,blk(_))) then
             ∃M',S1 (tell(recover(SI,blk(M'))) ||
                        ask(true) -> tell(rcv(M',S1)))
          else skip))) ||
     ask(true) -> environment(I)))))).
```

Let us describe the labeled actions 1a to 1d. We have labeled the code to improve readability of this description. The code in 1a models the situation in which the environment blocks the communication and as a consequence the intruder updates his private knowledge with the given term blk. 1b models the correct transmission of the message. The message is labeled with the term rcv being S_1 a fresh variable. Actions in 1c specify the case when the given message is encrypted with the public key of the intruder. Then, by using some tell agents, he recovers in C the content of such message and the identifier of the principal who is trying to communicate with him. Then, he composes a message with the recovered information and sends it to the honest principal whose identifier was previously recovered. Finally, 1d models the replication of message. In case that the intruder can recover from his private knowledge a message which has been blocked before, then he may deliver it.

To summarize, the *blocking of communication* capability of the Dolev-Yao intruder [16] is modeled by the code excerpt labeled with $1a$; The *message intercepting* capability is modeled by the fact that the environment is able to select the message to be processed. $1c$ implements the *message composing/decomposing* capability, and finally, the ability of *message replaying* is modeled by actions $1b$ and $1d$.

4.5. Concluding remarks

In this section, we have presented how the tccp language can be used for the specification of security protocols. We have established a direct correspondence between the principals' roles in the informal specification of a protocol and the resulting tccp formal specifications.

We have modeled a generic environment that includes the actions that an intruder can carry out in a protocol execution.

We do not specify explicitly the traces describing possible attacks, but thanks to the concurrency and the non-determinism, we include all the possible traces which can be analyzed later. In chapter 6, we describe the analysis process.

5 Embedding utcc into tccp

In this chapter, we describe the relation between utcc and tccp. We show how utcc can be embedded into tccp. It is not possible to have a direct transformation since the two languages have important differences in nature. In particular, both languages handle time differently, and the abstraction operator of utcc is not present in tccp. We define a transformation which, given the specification of a utcc program, constructs in an automatic way a tccp program that preserves the expected behavior of the original utcc program.

Let us first show the intuition of the transformation by means of an example.

5.1. Illustrative example

The following example is extracted from [45], where it was used to illustrate how mobility can be modeled in utcc:

Example 5.1.1 (Mobility [45]) *Let Σ be a signature with the unary predicates out_1, out_2, $\dots$ and a constant 0. Let Δ be the set of axioms over Σ valid in first-order logic. Consider the following process specifications.*

$P = (\mathsf{abs}\, y; out_1(y))\, \mathsf{tell}(out_2(y))$

and

$Q = (\mathsf{local}\, z)(\mathsf{tell}(out_1(z)) \| \mathsf{when}\, out_2(z)\, \mathsf{do\ next\ tell}(out_2(0)))$.

The process models a method to detect when a private information can become public in a given (non secure) channel (out_2). Thus, we have two channels called out_1 and out_2. Process P recovers from the first channel a value and forwards it to the second one. Q sends a private value (locally declared) z to out_1 and, in parallel, checks whether the second channel receives such value. In case the value appears in the out_2 channel, then, in the following time interval, the value 0 is sent to the second channel (modeling the vulnerability detection). By executing both processes in parallel $P \| Q$, the value 0 is produced in the second time interval.

Our transformation generates as many tccp procedure declarations as utcc processes are defined. In this case, we get the procedure declarations proc_P and proc_Q. Then, the initial call $P \| Q$ is transformed into the initial call (agent) of the tccp program.

We start by showing the transformation for the Q process. In this case, we have to pay special attention to model the utcc timed operator next, since it does not exist in tccp. The utcc process states that, when z is retrieved in the second channel, then in the following time interval (after the resting point) the value 0 is emitted to the channel. We define a global stream Syn that models when the utcc computation reaches a resting point. Having this in mind, we emit the 0 value when an ok is the value of the stream. We omit the details for the computation of the synchronization mechanism (the values for the synchronization stream), but we present it in the next subsection.

$$\begin{aligned} &\texttt{proc\_Q}() \text{ :- } \exists\, z\, (\mathsf{tell}(out_1(z)) \parallel \\ &\qquad\qquad \mathsf{ask}(out_2(z)) \rightarrow \mathsf{ask}(Syn \doteq \texttt{ok}) \rightarrow \mathsf{tell}(out_2(0))). \end{aligned}$$

The transformation for the P process is a bit more elaborated due to the use of the abs operator. Remember that (abs y; $out_1(y)$) tell($out_2(y)$) executes (in parallel) the instantiations (to the recovered values of y) of the tell agent. In order to handle the different possible instantiations, we use an auxiliary declaration $\texttt{abs}_i$ and a term $\texttt{subst}_\texttt{i} \in \Sigma$ to accumulatively store the processed instantiations.

$$\texttt{proc\_P}() \text{ :- } \exists\, y, S\, (\texttt{abs}_1(y, S) \parallel \mathsf{tell}(\texttt{subst}_1(S))).$$

$$\begin{aligned} &\texttt{abs}_1(y, S) \text{ :- } \exists\, t, S'\, ((\mathsf{now}(out_1(y)[t\backslash y] \wedge \neg(\mathsf{find}(S, \{[t\backslash y]\}))) \\ &\qquad\qquad \mathsf{then}\ (\mathsf{tell}(out_2(y)[t\backslash y]) \parallel \\ &\qquad\qquad\qquad (\mathsf{tell}(S = [\{[t\backslash y]\} \mid S']) \parallel \\ &\qquad\qquad\qquad\ \texttt{abs}_1(y, S))) \\ &\qquad\qquad \mathsf{else\ skip})). \end{aligned}$$

The arguments in the auxiliary call determine the pattern that is defined in the conditional agent. After executing the tell agent that updates the stream S (by adding the constraint $S = [\{[t\backslash y]\} \mid S']$), S includes the instantiation handled in the current recursive call. The computation stops when there is no instantiation retrieved from the store that is not already in the stream S (the constraint of the conditional agent does not hold). The last part of the example is the translation of the utcc initial process $P \| Q$:

$$\begin{aligned} &(\texttt{proc\_P} \parallel \texttt{proc\_Q} \parallel \mathsf{tell}(Syn = [\texttt{wait} \mid C_T]) \parallel \\ &\ \mathsf{utcc\_clock}(out_1(z) \wedge out_2(z), Syn) \parallel \mathsf{utcc\_clock}(out_2(0), Syn)) \end{aligned}$$

The two processes are called in parallel, as in the utcc example, and also the synchronization stream is initialized to the value wait, since the computation has not reached a resting point. Actually, also two clocks (one for each utcc resting point) are defined and executed in parallel to the above specification. Each clock updates the value of *Syn* whenever a condition (which characterizes a utcc resting point) holds. The conditions are statically computed, and we show the formalization in the following section. Here we simply show the declaration for these clocks. Each time a resting point is reached, the synchronization stream is updated to ok, thus all the agents that were waiting for this value, would start their execution. Then, the value is set again to wait.

```
utcc_clock($Store_1$,$Syn$) :-
  ask($Store_1$)$\rightarrow \exists\, Sn, Sn_1$((tell($Syn$ = [ok | $Sn$]) ||
                                   ask($true$)$\rightarrow$tell($Syn$ = [wait | $Sn_1$]))).
```

In Figure 5.1, we show the execution trace of the resulting tccp program. The table is interpreted from left to right, up to bottom. It is shown for each (tccp) time instant the current store and the agent to be executed. In the original utcc example, at the second resting point the value $out_2(0)$ is emmited. In the tccp trace, it can be seen that only after the first $Syn \doteq$ ok holds (thus the first resting point has been reached), the given value is emmited to the store. The value of the store at time instant 6 means the same as $Syn \doteq$ ok. We have simplified notation at instant 4 to improve clarity. Actually, the constraint $C_T = [\mathtt{ok} \mid Sn]$, which implies $Syn \doteq$ ok, is the information available in the store at time instant 4.

5.2. Formalization of the transformation

The transformation from utcc into tccp can be divided in two phases. The first one encodes each utcc process into tccp code, whereas the second one defines the necessary synchronization among the generated tccp agents. As we have shown, since the notion of time of both languages differs, we need to force the synchronization of processes in order to mimic the behavior of the original utcc program.

In the following, we define the τ_P function that transforms the utcc program into a tccp program. As mentioned above, this function includes a final phase where the synchronization mechanism complements the first transformation phase. Next we show the formalization. In Appendix B we show in pseudocode the mechanization of the process. We also prove that the built tccp program mimics the behavior of the given utcc program.

We say that an utcc program, similarly to a tccp program, is composed by a set of utcc declarations of processes and the utcc process that starts the execution

time	0	1
store	true	$Syn = [\texttt{wait} \mid C_T]$
agents	proc_P() $\|$ proc_Q() $\|$ tell($Syn = [\texttt{wait} \mid C_T]$) $\|$ utcc_clock($out_1(z) \wedge out_2(z)$,Syn) $\|$ utcc_clock($out_2(0)$,Syn)	abs$_1$(y, S) $\|$ tell(subst$_1$(S)) tell($out_1(z)$) $\|$ ask($out_2(z)$)$\rightarrow \ldots$ $\|$ ask($out_1(z) \wedge out_2(z)$)$\rightarrow \ldots$ $\|$ ask($out_2(0)$)$\rightarrow \ldots$
time	2	3
store	$out_1(z)$, subst$_1$(S)	$t = z$, $out_2(z)$, $S = [\{[t\backslash y]\} \mid S']$
agents	now($out_1(y)[t\backslash y] \wedge$ $\neg$(find($S, \{[t\backslash y]\}$))) then tell($out_2(y)$)$[t\backslash y]$ $\|$ tell($S = [\{[t\backslash y]\} \mid S']$) $\|$ abs(y, S) $\|$ ask($out_2(z)$)$\rightarrow \ldots$ $\|$ ask($out_1(z) \wedge out_2(z)$)$\rightarrow \ldots$ $\|$ ask($out_2(0)$)$\rightarrow \ldots$	now($out_1(y)[t'\backslash y] \wedge$ $\neg$(find($S, \{[t'\backslash y]\}$)))... skip $\|$ ask($Syn \doteq$ ok)$\rightarrow$ tell($out_2(0)$) $\|$ $\exists\, Sn, Sn_1$(tell($Syn = [\texttt{ok} \mid Sn]$)$\|$ ask($true$)$\rightarrow$ tell($Syn = [\texttt{wait} \mid Sn_1]$))$\|$ ask($out_2(0)$)$\rightarrow \ldots$
time	4	5
store	$Syn \doteq$ ok	$out_2(0)$, $Sn = [\texttt{wait} \mid Sn_1]$
agents	tell($out_2(0)$) $\|$ tell($Syn = [\texttt{wait} \mid Sn_1]$))$\|$ ask($out_2(0)$)$\rightarrow \ldots$	$\exists\, Sn', Sn'_1$(tell($Syn = [\texttt{ok} \mid Sn']$)$\|$ ask($true$)$\rightarrow$ tell($Syn' = [\texttt{wait} \mid Sn'_1]$))
time	6	7
store	$Sn_1 = [\texttt{ok} \mid Sn']$	$Sn' = [\texttt{wait} \mid Sn'_1]$
agents	tell($Syn = [\texttt{wait} \mid Sn'_1]$)	-

Figure 5.1: Trace of the resulting tccp program

of the program. Let us first introduce some notation. u$_d$ is a declaration from the set of utcc declarations, and u$_r$ is an utcc process. Moreover, we say that name(u$_d$) recovers the declaration name, and body(u$_d$) recovers the process on the rhs of the declaration. For example, name(P = tell($out_2(y)$)) returns P.

For each declaration u$_d$ in the set of declarations of the original utcc program, we define a tccp declaration in the tccp program. That tccp declaration has the form:

$$\mathsf{name}(\mathsf{u}_d) \;\texttt{:-}\; \tau_A(\mathsf{body}(\mathsf{u}_d))\ .$$

where τ_A is an auxiliary function that, given the utcc process u$_r$ (u$_r$ = body(u$_d$)), constructs a tccp agent that mimics its behavior.

Let us now describe the τ_A function. Depending on the form of the input process $\texttt{u}_r$, τ_A behaves differently. There are nine possible cases:

Case $\texttt{u}_r \equiv \mathsf{skip}$**.** The corresponding tccp agent is skip.[1]

Case $\texttt{u}_r \equiv \mathsf{tell(c)}$**.** The corresponding tccp agent is $\mathsf{tell}(c)$

Case $\texttt{u}_r \equiv (\mathsf{local}\ \vec{x}; c)\ A$**.** The corresponding tccp agent is $\exists^c\, \vec{x}\, (\tau_A(A))$
The superscript c denotes the initial store in the local computation in A, in the same sense as it is used in the utcc semantics for the local operator.

Case $\texttt{u}_r \equiv A \parallel B$**.** The corresponding tccp agent is $(\tau_A(A) \parallel \tau_A(B))$

Case $\texttt{u}_r \equiv \mathsf{next}\ A$**.** The corresponding tccp agent is $\mathsf{ask}(\mathit{Syn} \doteq \texttt{ok}) \rightarrow \tau_A(A)$

The Syn variable is a synchronization stream that is updated depending on the clock of the utcc program. The symbol $\doteq$ checks the last (current) value of the stream, which is updated to $\texttt{ok}$ each time a resting point is reached in the utcc program. We introduce later the synchronization mechanism that takes care of updating Syn.

Case $\texttt{u}_r \equiv \mathsf{unless}\ c\ \mathsf{next}\ A$**.** The corresponding tccp agent is
$\mathsf{ask}(\mathit{Syn} \doteq \texttt{ok}) \rightarrow \mathsf{now}\ c\ \mathsf{then\ skip\ else}\ \tau_A(A)$

Case $\texttt{u}_r \equiv\ !A$**.** The corresponding tccp agent is $(\tau_A(A) \parallel \mathsf{aux}_i)$
aux_i is an auxiliary (fresh) declaration defined to simulate the replication of utcc by means of the recursion capability of tccp. The definition of the new declaration is $\texttt{aux}_i$ `:-` $\mathsf{ask}(\mathit{Syn} \doteq \texttt{ok}) \rightarrow\ (\tau_A(A) \parallel \texttt{aux}_i)$., meaning that, at each resting point ($\mathit{Syn} \doteq \texttt{ok}$), the choice agent launches the execution of $\tau_A(A)$ in parallel with the procedure call $\texttt{aux}_i$ modeling recursion.

Case $\texttt{u}_r \equiv (\mathsf{abs}\,\vec{x}; c)\ A$**.** The corresponding tccp agent is

$\exists\, \vec{x}, S\ (\texttt{abs}_i(\vec{x}, S) \ ||\ \mathsf{tell}(\texttt{subst}_i(S)))$

and $\texttt{abs}_i(\vec{x})$ is an auxiliary (fresh) tccp declaration defined as follows:

```
abs_i($\vec{x}$, $S$):-$\exists\, \vec{t}, S'$ (now ($c[\vec{t}\backslash\vec{x}] \wedge \neg(\mathsf{find}(S, \{\vec{t}\backslash\vec{x}\}))$)
                 then ($\tau_A(A[\vec{t}\backslash\vec{x}])$ $\parallel$
                       tell($S = [\{\vec{t}\backslash\vec{x}\} \mid S']$) $\parallel$
                       abs_i($\vec{x}, S$))
                 else skip).
```

With $\texttt{subst}_i \in \Sigma$, i.e., it is a predicate handled by the constraint system $\mathcal{C}$.

[1]Since the semantics of both, the utcc version of the agent and the tccp version of the agent behave similarly and no confusion can arise, we don't distinguish them syntactically.

Case $u_r \equiv P$. The corresponding tccp agent is the call to the process name(P), i.e., proc_P.

Once all the utcc declarations are transformed, the second phase of the transformation τ_A can start. It consists in defining the clock that mimics the time passing in utcc. This is necessary since the explicit notion of time of utcc differs from the implicit one of tccp: A time unit in utcc may correspond with several time units in tccp. We use the stream *Syn* to simulate such clock. The stream may contain the following values:

- wait, that means that a resting point has not been reached.
- ok, that simulates that the resting point of the current time interval has been reached.

Remember that the initial call of the program initializes the stream *Syn* to the value wait.

As a result of this second transformation phase, a new declaration utcc_clock for each utcc resting point is introduced. These new declarations are executed in parallel with the initial call of the program. We need to run a pre-process that *computes* such clock, i.e., identifies the resting points of the original utcc program that define when a tick must occur. In the following, we show the function *instant* that computes the store generated during one utcc time instant. This information is used by the clock utcc_clock. The auxiliary function *follows* is used to compute the process that must be executed in the following time instant, and simulates the F function of the utcc semantics.

Given the utcc program u_p, let us assume that it possess n resting point. First, we define a tccp declarations of the following form.

```
utcc_clock(Store₁,Syn) :-
  ask(Store₁)→ ∃Sn,Sn₁((tell(Syn = [ok | Sn]) ||
                          ask(true)→tell(Syn = [wait | Sn₁]))).
```

Then, for each value computed by the functions *instant* and *follows* described below, we define in the initial term a call to such declaration with the computed values. This means we define n runs for the above declaration. When new computed values have already been computed (modulo renaming), then the process ends. More specifically, the computation starts by computing the *instant* for the initial call. Then, the function *follows* computes the process after the following resting point for which the *instant* is computed. The iteration proceeds until a *loop* is reached, i.e., the computed *instant* has already been computed (modulo renaming).

The function *instant* computes the information at the resting point of a given time interval, following the operational semantics of utcc:

$$
instant(\mathrm{u}_r, st) = \begin{cases}
st & \text{if } \mathrm{u}_r \equiv \mathsf{skip} \\
st \cup c & \text{if } \mathrm{u}_r \equiv \mathsf{tell}(c) \\
instant(A[\vec{x}'\backslash\vec{x}], c \wedge \exists\vec{x} st) & \text{if } \mathrm{u}_r \equiv (\mathsf{local}\ \vec{x}; c)\ A \text{ and} \\
 & \quad \vec{x}' \text{ are fresh variables} \\
instant(A_1, st) \wedge instant(A_2, st) & \text{if } \mathrm{u}_r \equiv A_1 \parallel A_2 \\
st & \text{if } \mathrm{u}_r \equiv \mathsf{next}\ A \\
st & \text{if } \mathrm{u}_r \equiv \mathsf{unless}\, c\, \mathsf{next}\, A \\
instant(A, st) & \text{if } \mathrm{u}_r \equiv\ !A \\
\bigwedge_{z\in\theta} instant(A[\vec{y}\backslash\vec{x}], st) & \text{if } \mathrm{u}_r \equiv (\mathsf{abs}\,\vec{x}; c)\, A \text{ and} \\
 & \quad st \vdash c[\vec{y}\backslash\vec{x}] \text{ and} \\
 & \quad \theta = \{\vec{y} \mid st \vdash c[\vec{y}\backslash\vec{x}]\} \\
instant(A, st) & \text{if } \mathrm{u}_r \equiv\ P = A
\end{cases}
$$

The function *follows* is similar to the future function F of utcc:

$$
follows(\mathrm{u}_r, st) = \begin{cases}
\mathsf{skip} & \text{if } \mathrm{u}_r \equiv \mathsf{skip} \\
\mathsf{skip} & \text{if } \mathrm{u}_r \equiv \mathsf{tell}(c) \\
(\mathsf{local}\ \vec{x})\ follows(A, st \wedge c) & \text{if } \mathrm{u}_p \equiv (\mathsf{local}\ \vec{x}; c)\ A \\
follows(A_1, st) \parallel follows(A_2, st) & \text{if } \mathrm{u}_p \equiv A_1 \parallel A_2 \\
A & \text{if } \mathrm{u}_p \equiv \mathsf{next}\ A \\
A & \text{if } \mathrm{u}_p \equiv \mathsf{unless}\, c\, \mathsf{next}\, A \text{ and} \\
 & \quad st \not\vdash c \\
!A & \text{if } \mathrm{u}_p \equiv\ !A \\
(\mathsf{abs}\ \vec{x}; c)\ follows(A, st \wedge c) & \text{if } \mathrm{u}_p \equiv\ (\mathsf{abs}\ \vec{x}; c)\ A \\
follows(A, st) & \text{if } \mathrm{u}_p \equiv\ P = A
\end{cases}
$$

5.2.1. Correctness of the transformation

We have defined a transformation from utcc programs into tccp programs in such a way that the resulting tccp program mimics the behavior of the original utcc program. In this section we show that our method is sound in the sense that each utcc trace of a program can be simulated by a tccp trace of the transformed version.

We fist introduce some notations. Similarly to [44], we say that elements $c_1, c_2, c'_1, c'_2, \ldots$ are the set of constraints defined by $\mathcal{C}$. Then, $\alpha, \alpha', \ldots$ denote infinite sequences of constraints where $\alpha = c_1.c_2\ldots$ and $\alpha' = c'_1.c'_2\ldots$. Finally, $\alpha(i)$ denotes the *i-th* element in α, for instance, $\alpha(2) = c_2$. Note that any constraint by itself is (or can represent) a store, so we can use both terminologies indistinctly.

Definición 5.2.1 (Input-Output Relations over utcc processes [44]) *Given the utcc process P, $\alpha = c_1.c_2\ldots$ and $\alpha' = c'_1.c'_2\ldots$, $P \overset{(\alpha,\alpha')}{\Longrightarrow}$ is used to represent $P = P_1 \overset{(c_1,c'_1)}{\Longrightarrow} P_2 \overset{(c_2,c'_2)}{\Longrightarrow} \ldots$. Then, the set*

$$io_{utcc}(P) = \{(\alpha,\alpha') \mid P \overset{(\alpha,\alpha')}{\Longrightarrow}\}$$

denotes the input-output behavior of P. This sequence can be interpreted as an interaction between the system P and an environment. At the time instant i, the environment provides an input c_i and P_i produces the output c'_i.

Let us next define the notion of inclusion of a utcc trace into a tccp trace. We write $\sqcup_{0\leq k\leq j} d$ with $k, j \in \mathbb{N}$ for the conjunction of all the stores 0 to j of the given structured store d.

Definición 5.2.2 (Entailment Relation) *Given the sequence of constraints $\alpha' = c'_1.c'_2\ldots$, and the structured store $d = d_0 \cdot d_1 \cdot d_3 \cdot \ldots$. Let $i, j, k \in \mathbb{N}$. We say that d entails the sequence of constraints α', denoted as $d \vdash_\tau \alpha'$, iff*

$$\forall i \exists j\,.\,(\sqcup_{0\leq k\leq j}\, d_k) \vdash \alpha'(i) \;\; s.t.\;\; j+1 \geq i, i > 0, j, k \geq 0$$

and the indexes j are pairwise distinct.

The following example illustrates the definition.

Example 5.2.3 *Given the sequence $\alpha' = (v = 1 \wedge w = 2).z > 8.(x = 7 \wedge y = 5)$, and the structured store $d = true \cdot (v = 1) \cdot (w = 2) \cdot z > 10 \cdot x = 7 \cdot r = 3 \cdot (y = 5 \wedge q < 3)$. We can see that:*

- $\sqcup_{0\leq k\leq 2}\, d_k \vdash \alpha'(1)$ *since it holds that* $(true \wedge v = 1 \wedge w = 2) \vdash (v = 1 \wedge w = 2)$
- $\sqcup_{0\leq k\leq 3}\, d_k \vdash \alpha'(2)$ *since it holds that* $(true \wedge v = 1 \wedge w = 2 \wedge z > 10) \vdash z > 8$
- $\sqcup_{0\leq k\leq 6}\, d_k \vdash \alpha'(3)$ *since it holds that* $(true \wedge v = 1 \wedge w = 2 \wedge z > 10 \wedge x = 7 \wedge r = 3 \wedge y = 5 \wedge q < 3) \vdash (x = 7 \wedge y = 5)$

Therefore, $d \vdash_\tau \alpha'$.

Now we are ready to define the notion of trace inclusion, i.e., when a utcc trace is mimicked by a tccp trace.

Definición 5.2.4 *Given (α,α') sequences of constraints computed by a utcc program, and d the structured store computed by a tccp program. Then, (α,α') is included in d, written $(\alpha,\alpha') \sim_\tau d$, iff $d \vdash_\tau \alpha'$.*

To guarantee the correctness of our method, we shall prove the correctness of the synchronization and transformation processes.

Lemma 5.2.5 (Correctness of Synchronization) *Consider a* utcc *program U of the form $U := U_D.U_R$ where U_D is the declaration set and U_R is the process that initiates the execution of U. Let st^i the store computed by the ith iteration of the function instant on the program U. Let $U_R^1 = follows(U_R, st^0)$, where st^0 is the initial store and $U_R^i = follows(U_R^{i-1}, st^{i-1})$. Let $io_{utcc}(U) = (\alpha, \alpha')$. Then, $st^1 \vdash \alpha'(1)$, $st^2 \vdash \alpha'(2)$, …, $st^n \vdash \alpha'(n)$ s.t.:*

- *$st^1 = instant(U_R^1, true)$ and,*
- *$st^n = instant(U_R^n, true)$ s.t. $n > 1$*

The following theorem states that a given utcc trace is included in the trace generated by the equivalent tccp program, i.e., by the tccp program obtained by the transformation.

Theorem 5.2.6 (Correctness of Transformation) *Consider an* utcc *program U. Let T the* tccp *program resulting of transforming U, $\tau_P(U) = T$. Given $io_{utcc}(U) = (\alpha, \alpha')$, and $io_{tccp}(T) = d$. Then, $(\alpha,\alpha') \sim_\tau d$.*

Lemma 5.2.5 (Correctness of Synchronization). Consider a utcc program U of the form $U := U_D.U_R$ where U_D is the declaration set and U_R is the process that initiates the execution of U. Let st^i the store computed by the ith iteration of the function *instant* on the program U. Let $U_R^1 = follows(U_R, st^0)$, where st^0 is the initial store and $U_R^i = follows(U_R^{i-1}, st^{i-1})$. Let $io_{utcc}(U) = (\alpha, \alpha')$. Then, $st^1 \vdash \alpha'(1)$, $st^2 \vdash \alpha'(2)$, …, $st^n \vdash \alpha'(n)$ s.t.:

- $st^1 = instant(U_R^1, true)$ and,
- $st^n = instant(U_R^n, true)$ *s.t.* $n > 1$

Proof. Each store st^i computed by the *instant* function at each time instant i must entail the store $\alpha'(i)$ of the sequence (α, α') generated by U. Note that the computed store is used as the guard in the choice agent of the clock declaration `utcc_clock`, thus it determines when $Syn \doteq \texttt{ok}$, in other words, when to proceed to the next time interval. We have to prove that the resting points (i.e., the output of the observables), are well determined by these computed stores. To this end, we proceed by structural induction on the form of the utcc processes and by induction on the length of the traces. We proceed case by case with the assumption that the agent considered is the last agent to be executed in the interleaving computation of the parallel processes in a time interval. This assumption is safe thanks to the deterministic nature of the languages: we can choose any order for the execution of parallel agents to reach the same resting point. st will denote the store computed by *instant* up to such execution point in the current time instant whereas $\alpha(i)'$ will denote the store computed by the utcc process up to such execution point in the *i-th* interval time.

Case 1 : $U_R = \mathsf{skip}$. By definition, $st^1 = \mathit{instant}(\mathsf{skip}, st) = st$. Moreover, $io_{utcc}(\mathsf{skip}) = (\alpha, \alpha)$. By induction hypothesis, $st \vdash \alpha(1)$. Then, $st^1 \vdash \alpha(1) \cup \mathit{true}$.

Case 2 : $U_R = \mathsf{tell}(c)$. By definition, $st^1 = \mathit{instant}(\mathsf{tell}(c), st) = st \cup c$. Let $io_{utcc}(\mathsf{tell}(c)) = (\alpha, \alpha')$ where $\alpha(1) = \alpha(1)' \wedge c$. By induction hypothesis; $st \vdash \alpha(1)'$. Then, $st^1 \vdash \alpha(1)$ since $c \vdash c$.

Case 3 : $U_R = (\mathsf{local}\,\vec{x}; c)\,A$. By definition, $st^1 = \mathit{instant}((\mathsf{local}\,\vec{x}; c)\,A, st) = \mathit{instant}(A[\vec{x}'\backslash\vec{x}], c \wedge \exists\vec{x}\,st) = c' \wedge \exists\,\vec{x}st$. Let $io_{utcc}((\mathsf{local}\,\vec{x}; c)\,A) = (\alpha, \alpha')$ where $\alpha(1)' = \exists\vec{x}\,\alpha(1) \wedge d^1$ s.t. $\alpha(1) \vdash c$ and d^1 is the store computed by A renamed. By induction hypothesis; $st \vdash \alpha(1)$ and $c' \vdash d^1$. Then, $\exists\vec{x}\,st \vdash \exists\vec{x}\,\alpha(1)$ and $st^1 \vdash \alpha(1)'$.

Case 4 : $U_R = A \parallel B$. By definition, $st^1 = \mathit{instant}(A \parallel B, st) = \mathit{instant}(A, st) \wedge \mathit{instant}(B, st) = st_A \wedge st_B$. Let $io_{utcc}(A \parallel B) = io_{utcc}(A) \cap io_{utcc}(B)$ where $\cap$, as in [44], represents the intersection between two sequences of constraints, the quiescent points present in both sequences. Let $io_{utcc}(A) = (\alpha, \alpha')$ and $io_{utcc}(B) = (\beta, \beta')$ where $\alpha = c_1.c_2\ldots$, $\alpha' = c_1'.c_2'\ldots$, $\beta = s_1.s_2\ldots$ and $\beta' = s_1'.s_2'\ldots$. Then, $io_{utcc}(A \parallel B) = (c_1 \cap s_1.c_2 \cap s_2\ldots, c_1' \cap s_1'.c_2' \cap s_2'\ldots$. By induction hypothesis we have that $st_A \vdash c_1'$ and $st_B \vdash s_1'$. Then, $st^1 \vdash \alpha(1)' \cup \beta(1)' = c_1' \wedge s_1'$.

Case 5 : $U_R = \mathsf{next}\,A$. By definition, $st^1 = \mathit{instant}(\mathsf{next}\,A, st) = st$. Let $(\beta, \beta') = (s_1.s_2\ldots, s_1'.s_2'\ldots)$ the sequence computed in the time instant previously to the execution of the next process and $io_{utcc}(A) = (\alpha, \alpha')$. Then $io_{utcc}(\mathsf{next}\,A) = (\beta \cup \alpha, \beta' \cup \alpha')$. By induction hypothesis we have that $st \vdash s_1'$. Then, $st^1 \vdash \beta(1)' = s_1'$.

Case 6 : $U_R = \mathsf{unless}\,c\,\mathsf{next}\,A$. By definition, $st^1 = \mathit{instant}(\mathsf{unless}\,c\,\mathsf{next}\,A, st) = st$. Let $(\beta,\beta') = (s_1.s_2.\ldots.s_n, s_1'.s_2'.\ldots.s_n')$ the sequence computed in the time instant previously to the execution of the next process, where the final store of this sequence ($\beta(n)'$ such that n is the current time instant) does not entail c, and $io_{utcc}(A) = (\alpha, \alpha')$, then $io_{utcc}(\mathsf{next}\,A) = (\beta \cup \alpha, \beta' \cup \alpha')$. By induction hypothesis we have that $st \vdash s_1'$. Then, $st^1 \vdash \beta(1)' = s_1'$.

Case 7 : $U_R = !A$. By definition, $st^1 = \mathit{instant}(!A, st) = \mathit{instant}(A, st) = st_A$. The replication process means $A \parallel \mathsf{next}A \parallel \mathsf{next}^2A \parallel \ldots$. Let $io_{utcc}(A) =$

(α, α'). Then, $io_{utcc}(\mathsf{next}\, A) = (\beta \cup \alpha, \beta' \cup \alpha')$, $io_{utcc}(\mathsf{next}^2 A) = (\gamma \cup \alpha, \gamma' \cup \alpha')$, and so on, where $\alpha = c_1.c_2 \ldots$, $\alpha' = c'_1.c'_2 \ldots$, $\beta = s_1.s_2 \ldots$, $\beta' = s'_1.s'_2 \ldots$, $\gamma = g_1.g_2 \ldots$ and $\gamma' = g'_1.g'_2 \ldots$. Then, $io_{utcc}(!A) = (\alpha, \alpha') \cap (\beta \cup \alpha, \beta' \cup \alpha') \cap (\gamma \cup \alpha, \gamma' \cup \alpha') \cap \ldots$. By induction hypothesis we have that $st_A \vdash c'_1$. Then, $st^1 \vdash \alpha(1)' = c'_1$.

Case 8 : $U_R = (\mathsf{abs}\, \vec{x}; c)\, A$. The utcc process abs means $A\, [\vec{y} \backslash \vec{x}] \parallel (\mathsf{abs}\, \vec{x}\, ; c \wedge \vec{x} \neq \vec{y})\, A$, when exist a sequence of terms $\vec{y}$ s.t. $c[\vec{y} \backslash \vec{x}]$ is entailed by the given store. By definition, $st^1 = instant\, ((\mathsf{abs}\, \vec{x};\, c)\, A, st) = \bigwedge_{z \in \mathbb{N}} instant\, (A\, [\vec{y_z} \backslash \vec{x}], st) = st \wedge st^1_A \wedge st^2_A \wedge \ldots \wedge st^z_A$ where z is the number of times that the given store st entails the constraint $c[\vec{y_n} \backslash \vec{x}]$ (with $1 \leq n \leq z$), and st^n_A is the store computed by A each time. Let s_i (s at time instant i) the given store. Let $io_{utcc}(A[\vec{y_z} \backslash \vec{x}]) = (\alpha_z, \alpha'_z)$. Then, we can obtain that $io_{utcc}((\mathsf{abs}\, \vec{x}; c)\, A) = (s_i \wedge \alpha_1, s_i \wedge \alpha'_1) \cap (s_i \wedge \alpha_2, s_i \wedge \alpha'_2) \cap \ldots \cap (s_i \wedge \alpha_z, s_i \wedge \alpha'_z)$. By induction hypothesis we have that $st \vdash s_i$, and $st^n_A \vdash \alpha_n(1)'$. Then, $st^1 \vdash s_i \wedge \alpha_1(1)' \wedge \alpha_2(1)' \wedge \ldots \wedge \alpha_n(1)'$.

Case 9 : $U_R = P = A$. By definition, $st^1 = instant(A, st) = st_A$. The process $P = A$ implies that $io_{utcc}(P) = io_{utcc}(A) = (\alpha, \alpha')$ with the sequences $\alpha = c_1.c_2 \ldots$ and $\alpha' = c'_1.c'_2 \ldots$. By induction hypothesis we have that $st_A \vdash c'_1$. Then, $st^1 \vdash \alpha(1)' = c'_1$.

□

Theorem 5.2.6 (Correctness of Transformation). Consider an utcc program U. Let T the tccp program resulting of transforming U, $\tau_P(U) = T$. Given $io_{utcc}(U) = (\alpha, \alpha')$, and $io_{tccp}(T) = d$. Then, $(\alpha,\alpha') \sim_\tau d$.

Proof. We prove that the trace computed by the utcc program U, of the form $U := U_D.U_R$ where U_D is the declaration set and U_R is the process that initiates the execution of U, is included in the trace computed by the tccp program T, of the form $T := T_D.T_A$ where T_D is the declaration set and T_A is the agent that initiates the execution of T, which is generated by means of the function τ_P, namely $T = \tau_P(U)$. To this end, we proceed by structural induction on the form of the utcc processes and by induction on the length of the traces. In particular, given $\mathtt{U}_R$ and T_A, such that $T_A = \tau_A(U_R)$, we prove the inclusion of the trace computed by the utcc processes U_R in the trace computed by its corresponding tccp agent T_A. Let $\alpha = c_1.c_2 \ldots$, $\alpha' = c'_1.c'_2 \ldots$, $\beta = s_1.s_2 \ldots$ and $\beta' = s'_1.s'_2 \ldots$ sequences of stores. Let $d = d_0 \cdot d_1 \cdot d_2 \cdot \ldots$ a structured store. We assume that both languages use the same constraint system.

Case 1 : $U_R = \mathsf{skip}$, then $T_A = \mathsf{skip}$. By definition these two operators do not add any information, producing the empty store ($true$). Let c_i (c at time instant i) and d the given stores in the execution of both process, respectively. Then, $io_{utcc}(\mathsf{skip}) = (c_i, c_i)$ and $io_{tccp}(\mathsf{skip}) = d$. By induction hypothesis; $d \vdash c_i$. Then, $io_{utcc}(\mathsf{skip}) \sim_\tau io_{tccp}(\mathsf{skip})$.

Case 2 : $U_R = \mathsf{tell}(c)$, then $T_A = \mathsf{tell}(c)$. The utcc process tell adds the constraint c in a given store s_i (s at time instant i). Then, $io_{utcc}(\mathsf{tell}(c)) = (s_i \wedge c, s_i \wedge c)$. The tell tccp agent adds the constraint c in a given store d_j (d at time instant j). Then, $io_{tccp}(\mathsf{tell}(c)) = d$ such that $d = d_j \cdot c$. Let $d_\sqcup = (\sqcup_{0 \leq k \leq j} d_k) = d_j$ and $d'_\sqcup = (\sqcup_{0 \leq k \leq j+1} d_k) = d_j \wedge c$. By induction hypothesis; $d_j \vdash s_i$ then $d'_\sqcup \vdash s_i \wedge c$ since $c \vdash c$. Then, $io_{utcc}(\mathsf{tell}(c)) \sim_\tau io_{tccp}(\mathsf{tell}(c))$.

Case 3 : $U_R = (\mathsf{local}\, \vec{x}; c)\, A$, then $T_A = \exists^c \vec{x}\, (A')$ where $A' = \tau_A(A)$. The hiding operator of tccp, $\exists$, has the same function that the local of utcc. Intuitively, they set the variables in $\vec{x}$ to be local in A' and A, respectively, moreover, the local information produced by A' and A must be hidden from the main operator. Usually, these two operators ($\exists$ and local) evolve renaming $\vec{x}$ by fresh variables, i.e., $\vec{x}'$. By induction hypothesis we have that $io_{utcc}(A) \sim_\tau io_{tccp}(A')$. Then, $io_{utcc}(A[\vec{x}' \backslash \vec{x}]) \sim_\tau io_{tccp}(A'[\vec{x}' \backslash \vec{x}])$ under the same constraint system.

Case 4 : $U_R = A \parallel B$, then $T_A = (A' \parallel B')$ where $A' = \tau_A(A)$ and $B' = \tau_A(B)$. As in the previous case, these two parallel operators have the same goal. Let $io_{utcc}(A \parallel B) = io_{utcc}(A) \cap io_{utcc}(B)$ where $\cap$, as in [44], represents the intersection between two sequences of constraints, the quiescent points present in both sequences. Let $io_{utcc}(A) = (\alpha, \alpha')$ and $io_{utcc}(B) = (\beta, \beta')$. Then $io_{utcc}(A \parallel B) = (c_1 \cap s_1 . c_2 \cap s_2 \ldots, c'_1 \cap s'_1 . c'_2 \cap s'_2 \ldots$. Let $io_{tccp}(A') = d^{A'} = d \cdot d^{A'_1} \cdot d^{A'_2} \cdot \ldots$ and $io_{tccp}(B') = d^{B'} = d \cdot d^{B'_1} \cdot d^{B'_2} \cdot \ldots$ where d is the input structured store, $d^{A'_1} \cdot d^{A'_2} \cdot \ldots$ the structured store computed by A' and $d^{B'_1} \cdot d^{B'_2} \cdot \ldots$ the structured store computed by B'. Then, $io_{tccp}(A' \parallel B') = d^{A'} \sqcup d^{B'} = d \wedge d \cdot d^{A'_1} \wedge d^{B'_1} \cdot d^{A'_2} \wedge d^{B'_2} \cdot \ldots$. By induction hypothesis we have that $io_{utcc}(A) \sim_\tau io_{tccp}(A')$ and $io_{utcc}(B) \sim_\tau io_{tccp}(B')$ then, we can deduce that $io_{utcc}(A \parallel B) \sim_\tau io_{tccp}(A' \parallel B')$.

Case 5 : $U_R = \mathsf{next}\, A$, then $T_A = \mathsf{ask}(Syn \doteq \mathtt{ok}) \rightarrow A'$ where $A' = \tau_A(A)$. Let (β, β') the sequence computed before the execution of the next process and $io_{utcc}(A) = (\alpha, \alpha')$, then $io_{utcc}(\mathsf{next}\, A) = (\beta \cup \alpha, \beta' \cup \alpha')$. Let d the structured store computed until the time instant in which the constraint $Syn \doteq \mathtt{ok}$ hold and $io_{tccp}(A') = d'$ where d' is the structured store computed by A'. By Lemma 1 we know that d corresponds to the resting point. Then $io_{tccp}(\mathsf{ask}(Syn \doteq \mathtt{ok}) \rightarrow A') = d \cdot d'$. By induction hypothesis we have that $(\beta, \beta') \sim_\tau d$ and $(\alpha, \alpha') \sim_\tau d'$. Then $(\beta \cup \alpha, \beta' \cup \alpha') \sim_\tau d \cdot d'$. Then, $io_{utcc}(\mathsf{next}\, A) \sim_\tau io_{tccp}(\mathsf{ask}(Syn \doteq \mathtt{ok}) \rightarrow A')$.

Case 6 : $U_R = \mathsf{unless}\, c\, \mathsf{next}\, A$, then $T_A = \mathsf{ask}(Syn \doteq \mathtt{ok}) \rightarrow \mathsf{now}\, c\, \mathsf{then\, skip\, else}\, A'$ where $A' = \tau_A(A)$. This case is very similar to the previous case. Let (β, β') be the sequence computed before the execution of the next process, which means that the final store of this sequence ($\beta'(n)$ such that n is the current time instant) does not entail c. Moreover, $io_{utcc}(A) = (\alpha, \alpha')$, then $io_{utcc}(\mathsf{unless}\, c\, \mathsf{next}\, A) = (\beta \cup \alpha, \beta' \cup \alpha')$. Let d be the structured store computed until the time instant in which the constraint $Syn \doteq \mathtt{ok}$ does hold. By Lemma 1 we know that d corresponds to the current resting point. The constraint c of the conditional agent does not hold in d and $io_{tccp}(A') = d'$. Then, $io_{tccp}(\mathsf{ask}(Syn \doteq \mathtt{ok}) \rightarrow \mathsf{now}\, c\, \mathsf{then\, skip\, else}\, A') = d \cdot d'$. By induction hypothesis we have that $(\beta, \beta') \sim_\tau d$ and $(\alpha, \alpha') \sim_\tau d'$. Moreover, $(\beta \cup \alpha, \beta' \cup \alpha') \sim_\tau d \cdot d'$. In case that the constraint c is entailed by $\beta'(n)$ and d we can see that both process, unless and now, evolve into skip. Therefore, $io_{utcc}(\mathsf{unless}\, c\, \mathsf{next}\, A) \sim_\tau io_{tccp}(\mathsf{ask}(Syn \doteq \mathtt{ok}) \rightarrow \mathsf{now}\, c\, \mathsf{then\, skip\, else}\, A')$.

Case 7 : $U_R = !A$, then $T_A = (A' \parallel \mathsf{aux}_i)$ where $A' = \tau_A(A)$ and aux_i :- $\mathsf{ask}(Syn \doteq \mathtt{ok}) \rightarrow (A' \parallel \mathsf{aux}_i)$. The replication process means $A \parallel \mathsf{next} A \parallel \mathsf{next}^2 A \parallel \ldots$. Let $io_{utcc}(A) = (\alpha, \alpha')$. Then we can obtain that $io_{utcc}(\mathsf{next}\, A) = (\beta \cup \alpha, \beta' \cup \alpha')$, $io_{utcc}(\mathsf{next}^2 A) = (\gamma \cup \alpha, \gamma' \cup \alpha')$, and so on, where $\gamma = g_1.g_2 \ldots$ and $\gamma' = g'_1.g'_2 \ldots$. We have that $io_{utcc}(!A) = (\alpha, \alpha') \cap (\beta \cup \alpha, \beta' \cup \alpha') \cap (\gamma \cup \alpha \cup \gamma' \cup \alpha') \cap \ldots$. Let $io_{tccp}(A') = d$ and $io_{tccp}(\mathsf{aux}_i) = io_{tccp}(\mathsf{ask}(Syn \doteq \mathtt{ok}) \rightarrow (A' \parallel \mathsf{aux}_i))$. We get that $io_{tccp}(A' \parallel \mathsf{aux}_i) = d \sqcup d'.d \sqcup d''.d \sqcup \ldots$, where d^n $(n \geq 1)$ is the structured store computed until the time instant in which the constraint $Syn \doteq \mathtt{ok}$ in the iteration n, does hold. By Lemma 1, d^n corresponds to the resting point of the current time interval. By induction hypothesis we have that $io_{utcc}(A) \sim_\tau io_{tccp}(A')$, $(\beta, \beta') \sim_\tau d'$, $(\gamma, \gamma') \sim_\tau d''$, etc. Therefore, we can deduce that $((\alpha, \alpha') \cap (\beta \cup \alpha, \beta' \cup \alpha') \cap (\gamma \cup \alpha, \gamma' \cup \alpha') \cap \ldots) \sim_\tau d \sqcup d'.d \sqcup d''.d \sqcup \ldots$ and $io_{utcc}(!A) \sim_\tau io_{tccp}(A' \parallel \mathsf{aux}_i)$.

Case 8 : $U_R = (\mathsf{abs}\, \vec{x}; c)\, A$, then $T_A = \exists\, \vec{x}, S\, (\mathtt{abs}_m(\vec{x}, S) \parallel \mathsf{tell}(\mathtt{subst}_m(S)))$ with:

```
abs_i($\vec{x}, S$):-$\exists\, \vec{t}, S'$ (now ($c[\vec{t}\backslash\vec{x}] \wedge \neg$(find($S, \{\vec{t}\backslash\vec{x}\}$)))
                    then ($A'[\vec{t}\backslash\vec{x}]$ ||
                          tell($S = [\{\vec{t}\backslash\vec{x}\} \mid S']$) ||
                          abs_i($\vec{x}, S$))
                    else skip).   and $A' = \tau_A(A)$
```

The utcc process abs means $A[\vec{t}\backslash\vec{x}] \parallel (\mathsf{abs}\, \vec{x}; c \wedge \vec{x} \neq \vec{t})\, A$, when exist a sequence of terms $\vec{t}$ s.t. $c[\vec{t}\backslash\vec{x}]$ is entailed by the given store. Let s_i (s at time instant i) the given store. Let $io_{utcc}(A[\vec{t_j}\backslash\vec{x}]) = (\alpha_j, \alpha'_j)$ where j is the number of times that A is executed with different $\vec{t}$ $(s_i \vdash c[\vec{t_j}\backslash\vec{x}])$. Then, we can obtain that $io_{utcc}((\mathsf{abs}\,\vec{x};c)\,A) = (s_i \wedge \alpha_1, s_i \wedge \alpha'_1) \cap (s_i \wedge \alpha_2, s_i \wedge \alpha'_2) \cap \ldots \cap (s_i \wedge \alpha_j, s_i \wedge \alpha'_j)$.

$io_{tccp}(\exists\,\vec{x}, S\,(\mathsf{abs}_m(\vec{x},S) \parallel \mathsf{tell}(\mathtt{subst}_m(S)))) = io_{tccp}(\mathsf{abs}_m\,(\vec{x},S)\,[\{\vec{x}' \setminus \vec{x}\}, \{S''\backslash S\}]) \sqcup io_{tccp}(\mathsf{tell}(\mathtt{subst}_m(S))[\{\vec{x}'\backslash\vec{x}\},\{S''\backslash S\}]) = io_{tccp}(\mathsf{abs}_m(\vec{x}',S''))\sqcup io_{tccp}(\mathsf{tell}(\mathtt{subst}_m(S'')))$. $\mathsf{abs}(\vec{x}',S'')$ executes $\exists\,\vec{t}, S'(\mathsf{now}(c[\vec{t}\backslash\vec{x}'] \wedge \neg(\mathsf{find}(S'', \{\vec{t}\backslash\vec{x}'\})))\,\mathsf{then}\,(A'[\vec{t}\backslash\vec{x}'] \parallel \mathsf{tell}(S'' = [\{\vec{t}\backslash\vec{x}'\} \mid S']) \parallel \mathsf{abs}(\vec{x}',S''))$, many times as sequences of $\vec{t}$ can be found s.t. the constraint $c[\vec{t}\backslash\vec{x}'] \wedge \neg(\mathsf{find}(S,\{\vec{t}\backslash\vec{x}'\}))$ is entailed by the given store. In case that the constraint does not hold, the process ends (skip). Then, $io_{tccp}(\mathsf{abs}(\vec{x}',S'')) = io_{tccp}(\mathsf{now}(c[\vec{t}\backslash\vec{x}'] \wedge \neg(\mathsf{find}(S'',\{\vec{t}\backslash\vec{x}'\})))\,\mathsf{then}\,(A'[\vec{t}\backslash\vec{x}'] \parallel \mathsf{tell}(S''=[\{\vec{t}\backslash\vec{x}'\}\mid S''']) \parallel \mathsf{abs}(\vec{x}',S''))$ $\mathsf{else\ skip})$. Then, taking into account those cases when the constraint of the conditional agent does hold, $io_{tccp}(\mathsf{abs}(\vec{x}',S'')) = (io_{tccp}(A'[\vec{t}\backslash\vec{x}']) \sqcup io_{tccp}(\mathsf{tell}(S''=[\{\vec{t}\backslash\vec{x}'\}\mid S'''])) \sqcup io_{tccp}(\mathsf{abs}(\vec{x}',S''))) = d$ s.t. $d = ((d_k\cdot st^1\cdot d^1)\wedge(d_k\cdot st^1\cdot S''=[\{\vec{t}\backslash\vec{x}'\}|S''']) \wedge (((d_k\cdot st^2\cdot d^2)\wedge(d_k\cdot st^2\cdot S'''=[\{\vec{t'}\backslash\vec{x}'\}|S'''']))\wedge\ldots\wedge((d_k\cdot st^n\cdot d^n)\wedge(d_k\cdot st^n\cdot S'^n = [\{\vec{t}^n\backslash\vec{x}'\}|S''^n])\wedge true)))$ where d_k is the given store (d at time instant k), st is the structured store computed so far, d^n is the structured store computed by $A'[\vec{t}^n\backslash\vec{x}']$ ($io_{tccp}(A'[\vec{t}^n\backslash\vec{x}']) = d^n$), and the superindex n (≥ 1) is the number of times that A' is executed with different $\vec{t}$. Then, $d = d_k\cdot st^1\cdot d^1 \wedge S'' = [\{\vec{t}\backslash\vec{x}'\}|S''']\cdot st^2\cdot d^2\wedge S'''=[\{\vec{t'}\backslash\vec{x}'\}|S'''']\cdot\ldots\cdot st^n\cdot d^n\wedge S'^n=[\{\vec{t}^n\backslash\vec{x}'\}|S''^n]$. Moreover, $io_{tccp}(\mathsf{tell}(\mathtt{subst}_m(S''))) = c$ s.t. $c = c_t\cdot\mathtt{subst}_m(S'') = d_k$ since the tell and the procedure call agents consume one time unit. By induction hypothesis we have that $io_{utcc}(A[\vec{t_j}\backslash\vec{x}]) \sim_\tau io_{tccp}(A'[\vec{t}^n\backslash\vec{x}']) = (\alpha_j,\alpha'_j) \sim_\tau d^n$ and $s_i \sim_\tau d_k$. Then, $io_{utcc}((\mathsf{abs}\,\vec{x};c)\,A) \sim_\tau io_{tccp}(\exists\,\vec{x},S\,(\mathsf{abs}_i(\vec{x},S) \parallel \mathsf{tell}(\mathtt{subst}_i(S))))$. Note that when the constraint c does not hold both processes evolve to skip.

Case 9 : $U_R = P = A$, then $T_A = p$ where $p : -A'$, such that $A' = \tau_A(A)$. The process $P = A$ implies that $io_{utcc}(P) = io_{utcc}(A) = (\alpha,\alpha')$. The procedure call agent p, following its operational semantics, executes the agent A' with a given store d_j (d at time instant j). Then $io_{tccp}(p) = d$, such that $d = d_j\cdot d^{A'}$ where $d^{A'}$ is the structured store computed by A'. By induction hypothesis we have that $io_{utcc}(A) \sim_\tau io_{tccp}(A')$. Then, $(\alpha,\alpha') \sim_\tau d^{A'}$ therefore $(\alpha,\alpha') \sim_\tau d$. Then, $io_{utcc}(P = A) \sim_\tau io_{tccp}(p)$.

□

5.3. Concluding remarks

In this chapter, we have presented how the tccp language embeds the utcc language. We have defined a transformation that translates a given utcc program into a tccp program simulating the behavior of the input program.

We have proven that the proposed transformation is correct so that tools defined for tccp can be reused for utcc. Mainly due to the fact that it is necessary to define the utcc abstraction operator in terms of existing tccp operators, the program resulting from the transformation is bigger than the original utcc program. Nevertheless, the size increase is not too hard.

We have proven that the proposed transformation is correct.

6
A tccp interpreter

The tccpInterpreter system is the result of the implementation in Maude of the tccp formalism, i.e., the language operational semantics plus a specific constraint solver. The tool takes as input the specification of a tccp program and simulates its behavior following the semantics of the language. tccpInterpreter consists of approximately 1320 lines of code divided in six Maude modules. Each module models one or more of the entities of tccp: agents, constraints, programs, the store, the underlying constraint system, the operational semantics, etc. Maude allows us to implement a constraint solver for the language or to use an existing one to handle constraints.[1] The interested reader can consult [12, 13] for a more detailed documentation about Maude.

6.1. Syntactic objects

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

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

The skip agent is encoded as:

```
skip :  -> TccpAgent .
```

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

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

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

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

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

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

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

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

The hiding agent is encoded by using the constructor symbol `exists` followed by a list of variables (term of sort `TccpVariableList`), and the corresponding agent.

```
op exists__ :  TccpVariableList TccpAgent -> TccpAgent .
```

The procedure call agent is identified by using the constructor symbol {__}, which contains the procedure's name (term of sort `TccpConstant`) and the parameters of the call (term of sort `TccpExpressionList`).

```
op {__} :  TccpConstant TccpExpressionList -> TccpAgent .
```

The system models all the agents appearing in Figure 2.6, including those introduced in [30].

A tccp program is modeled by using the operator:

```
op {_._} :  TccpDeclarationSet TccpAgent -> TccpProgram .
```

where the sort `TccpDeclarationSet` represents the set of procedure declarations of the program and the sort `TccpAgent` represents the agent that initiates the execution.

A tccp declaration is modeled by using the operator:

```
op {__=def_} :  TccpConstant TccpVariableList TccpAgent ->
                TccpDeclaration .
```

where the sort `TccpConstant` represents the declaration's name, the sort `TccpVaria bleList` represents the declaration's parameters and the sort `TccpAgent` represents the agent describing the declaration's behavior.

6.2. The Operational Semantics

The operational semantics of tccp are encoded in Maude as transitions over configurations by means of Maude rules. For readability, we label each rule with an identifier.

Let us first introduce how we represent the store and the structured store. We denote the store (a term of sort `TccpStore`) as a set of constraints (terms of sort `TccpConstraint`) separated by the symbol `,,` (for example, `('X > 5,,'X < 10)`). The constructor symbol `empty` represents the empty store.

Then, a structured store is represented by a store together with a natural number (between braces). The natural number represents the current time instant (for example, `(('X > 5,,'X < 10) {2})`) is the store at time instant 2). Formally,

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

We use the arrow `=>` to separate each store in the sequence representing a structured store. We use the sort `TccpStrStoreList`) to represent such sequence.

A configuration (state) of the system is encoded by using a term of the form `<` $P1$`,`$P2$`,`$P3$`,`$P4$ `>` $\{P5\}$, where $P1$ represents the given tccp program, $P2$ represents the structured store, $P3$ contains the list of variables present in the store, $P4$ is a boolean constraint used to control the execution of certain agents, and $P5$ is a given threshold (a natural number that states the amount of time units that we can establish for the execution of the given program). In case that the threshold is non-instantiated, the system may run infinite computations. Formally,

```
op < _,_,_,_ > : TccpProgram TccpStrStoreList TccpVariableList Bool
                 Nat -> TccpConfig .
```

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

1. the given constraint `Bl` must be `true`,
2. the store `TpSt` represents all the information stored in `SS` up to the current time instant,
3. `TpSt` satisfies the constraint `CtBl` of the conditional tccp agent. This is checked by means of the function `consultTccpStore (TpSt , CtBl)`, and
4. `Ag1'` is the result of executing `Ag1` with the current store.

```
crl [now-true]:<{DcSt.now (CtBl) then Ag1 else Ag2},SS,VrLt,Bl>{Nt}
               => <{DcSt.Ag1'},SS1,VrLt1,Bl1>{Nt}
    if Bl == true ∧
       TpSt := returnGlobalStoreFromStructuredStoreList (SS) ∧
       consultTccpStore (TpSt , CtBl) == ctrue ∧
       <{DcSt.now (CtBl) then Ag1 else Ag2},SS,VrLt,Bl>{Nt} =>
       <{DcSt.Ag1'},SS1,VrLt1,Bl1>{Nt} .
```

`consultTccpStore (TpSt , CtBl)` gets as input the store `TpSt` and the boolean constraint `CtBl` and returns `ctrue` when the store entails the given constraint or `cfalse` otherwise. When the condition holds, we reach the configuration `<{DcSt.Ag1'},SS1,VrLt1,Bl1>{Nt + 1}` resulting of executing the agent in the `then` branch of the conditional agent.

The conditional rule describing the case when the constraint of the conditional agent does not hold is shown below.

```
crl [now-false]:<{DcSt.now (CtBl) then Ag1 else Ag2},SS,VrLt,Bl>{Nt}
               => <{DcSt.Ag2'},SS1,VrLt1,Bl1>{Nt}
    if Bl == true ∧
       TpSt := returnGlobalStoreFromStructuredStoreList (SS) ∧
       consultTccpStore (TpSt , CtBl) == cfalse ∧
       <{DcSt.now (CtBl) then Ag1 else Ag2},SS,VrLt,Bl>{Nt} =>
       <{DcSt.Ag2'},SS1,VrLt1,Bl1>{Nt} .
```

In this case, `consultTccpStore (TpSt , CtBl)` returns `cfalse` then the agent `Ag2` is executed.

The following code excerpt describes the rules modeling the semantics of the choice agent. The rule `ask-true` specifies the case when a choice agent with a single branch can be executed. In this case, the agent `ask(CtBl) -> Ag` evolves to a configuration containing the original declarations set `DcSt`, the agent to be executed `Ag`, the structured store `SS1` (`SS => empty{t+1}`, where `t` is the current time instant) resulting from updating the given structured store `SS` with the empty store (`empty`), the given list of variables and `false`. The conditional rule states that the agent is executed only when the given constraint `Bl` is `true` and the store `TpSt` satisfies the constraint `CtBl` of the agent, checked by means of `consultTccpStore (TpSt , CtBl) == ctrue`.

```
crl [ask-true]:<{DcSt.ask(CtBl) -> Ag},SS,VrLt,Bl>{Nt} =>
               <{DcSt.Ag},SS1,VrLt,false>{Nt}
    if Bl == true ∧
       TpSt := returnGlobalStoreFromStructuredStoreList (SS) ∧
       consultTccpStore (TpSt , CtBl) == ctrue .
```

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

```
crl [choice-true]:<{DcSt.(ask(CtBl) -> Ag + AgCh)},SS,VrLt,Bl>{Nt}
                 => <{DcSt.Ag},SS1,VrLt,false>{Nt}
   if Bl == true ∧
      TpSt := returnGlobalStoreFromStructuredStoreList (SS) ∧
      consultTccpStore (TpSt , CtBl) == ctrue .
```

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

```
crl [choice-false]:<{DcSt.AgCh},SS,VrLt,Bl>{Nt} =>
                   <{DcSt.AgCh},SS1,VrLt,false>{Nt}
   if Bl == true ∧
      TpSt := returnGlobalStoreFromStructuredStoreList (SS) ∧
      consultTccpStore (TpSt , AgCh) == cfalse .
```

The parallel agent is specified as the following conditional rule, where it is reached the configuration resulting of executing, at the same time, the agents `Ag1` and `Ag2`. The execution of `Ag1` produces `Ag1'` and the structured store `SS1`, whereas the execution of `Ag2` produces `Ag2'` and the structured store `SS2`. Then, the resulting configuration contains the parallel composition of both `Ag1'` and `Ag2'`, and the structured store resulting of joining the structured stores: `(SS1 ∧ SS2)`:

```
crl [parallel]:<{DcSt.(Ag1 || Ag2)},SS,VrLt,Bl>{Nt} =>
               <{DcSt.(Ag1' || Ag2')},(SS1 ∧ SS2),VrLt2,true>{Nt}
   if Bl == true ∧
      <{DcSt.Ag1},SS,VrLt,Bl>{Nt} =>
      <{DcSt.Ag1'},SS1,VrLt1,Bl1>{Nt} ∧
      <{DcSt.Ag2},SS,VrLt1,Bl>{Nt} =>
      <{DcSt.Ag2'},SS2,VrLt2,Bl2>{Nt} .
```

The hiding agent is specified as the following conditional rule. It proceeds by renaming its associated agent `Ag`. By using the auxiliary functions `getFreshVar ListFromVarList` and `replaceVarListInAgent`, we recover in `VrLt2` a list of fresh variables and in `Ag'` the renamed agent `Ag`, respectively. Then, the hiding reaches the configuration resulting of executing the renamed agent `Ag'`. The execution of `Ag'` produces `Ag''` and the structured store `SS1`:

```
crl [hiding]:<{DcSt.exists VrLt Ag},SS,VrLt1,Bl>{Nt} =>
           <{DcSt.Ag''},SS1,VrLt3,Bl1>{Nt}
    if Bl == true ∧
       VrLt2 := getFreshVarListFromVarList (VrLt , VrLt1) ∧
       Ag' := replaceVarListInAgent (VrLt , VrLt2 , Ag) ∧
       <{DcSt.Ag'},SS,(VrLt1,VrLt2),Bl>{Nt} =>
       <{DcSt.Ag''},SS1,VrLt3,Bl1>{Nt} .
```

To simulate the execution of a program, we define the constructor symbol `runs`:

```
op runs : TccpProgram Nat -> TccpConfig .
```

and the equation:

```
eq [program] : runs (DcSt.TpAg,Nt) =
               <{DcSt.TpAg},(empty {0}),nil,true>{Nt} .
```

The function `runs` takes as argument the program to be executed and a natural number that states the threshold for the execution of the program. The equation label with `program` takes, in the lhs, the function `runs` with its parameters and translated it, in the rhs, in an initial configuration containing the given program, the empty store, the empty list and `true`.

6.3. The underlying constraint solver

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

We model the constraints representing terms and streams by using the sorts `TccpTerm` and `TccpStream`, respectively. We use the Maude operators:

```
op [__] : TccpConstant TccpExpressionList -> TccpTerm .
```

to represent a term which is specified by its name (a constant) and its arguments (an expression list), and:

```
op [_|_] : TccpExpressionList TccpVariable -> TccpStream .
```

to represent the content of a stream which is specified by a list of values and the current tail (a variable).

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

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

Currently, `TccpArithmetic` includes floating-point numbers and variables.

The following operators represent the *sum*, *rest*, *multiplication* and *division* of two arithmetic terms (`TccpArithmetic`) returning another arithmetic term, respectively:

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

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

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

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

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

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

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

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

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

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

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

6.4. Running the interpreter

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

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

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

The following example shows the output of Maude when reducing a tccp agent.

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

TCCP-INTERPRETER is the main module of the tccpInterpreter.

The example shows that the given expression is an agent of the language. We can also use the command *rew* expression to explore the possible behavior of a tccp program. For example:

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

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

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

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

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

6.4.1. Illustrative example

Here we describe an example of use of the tccpInterpreter system. In Figure 6.1 we show the specification in tccp of a part of a microwave oven controller that we have borrowed from [17]. To make the description clearer we show a labeled version of the declaration. Labels appear within braces { }:

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

Figure 6.1: 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 in ln8. In case the condition holds, the process forces (with the tell agent in `lt13`) the microwave to be turned-off in the following time instant. Moreover, an error signal must be emitted (agent `lt11`). If the condition does not hold, then the system emits (via another tell agent `lt15`) a signal of *no error* that will be available in the store at the following time instant. These signals may be captured by other processes, thus it can be seen that the store allows the synchronization of processes. Finally, the procedure call agent `microwave_error(D,B,E)` models the recursion of the system.

By using the following command in the Maude console, once loaded the tccpInterpreter, the system simulates the behavior of the given declaration D[2].

```
Maude> search < {D.'microwave_error(['open|'_],['on|'_],['no|'_])},
                empty{0},nil,true>{2} =>* < {D.Ag},St,Vl,Bl>{2} .
```

The first term specifies the configuration, composed by the declaration D, the procedure call agent `'microwave_error (['open|'_],['on|'_],['no|'_])`, the empty store at time instant 0, the empty list and `true`. The proof consists in reaching the second term that specifies the configuration containing D, an agent `Ag`, the structure store `St`, the list of variables and the boolean constraint `true`. By using the non-instantiated variables `Ag`, `St`, `Vl` and `Bl` 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 control the computation steps, we can use the Maude debugging feature [12] to capture each step of the computation.

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

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

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

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

6.5. Experimental results

In this section, we illustrate how it is possible to analyze a protocol specified in tccp by using the interpreter and some capabilities from the specification language Maude. Given the specification of a tccp program we can simulate the behavior of such program following the semantics of the language and, taking advantage of the Maude model-checking feature, we can also carry out certain kind of reachability analysis.

The underlying constraint system. To refine the verification process, we use the auxiliary function `getGlobalStoreFromStrStoreList` which, given a structured store, returns a unique store with all the information stored so far. This transformation facilitates the action of consulting the store. Moreover, we implement some auxiliary functions in the underlying constraint system. These functions are `getStreamOfTrmKnowInStore`, `getExpFromStream` and `consultContent`. The first one returns the stream that contains, in a given store, the private knowledge of a given principal (whose identifier is given in the term `know`). The second returns the value (`TpEx`) that is currently labeled as `secret` in a given stream (in this case `TpVr1`, that models the private knowledge of `'b`). Finally, the function `consultContent` checks if the values of a given stream, recovered from the given store, contain the given expression (`TpEx`).

6.5.1. Analyzing the Needham-Schroeder protocol

Let us show an excerpt of the execution of a trace that simulates the correct behavior of the Needham-Schroeder protocol. Consider the tccp program modeling the specification of the Needham-Schroeder protocol presented in Chapter 4 and whose initial agent is `(init('a,'b) || (resp('b) || environment('i)))`. For readability, we use the variable DS instead of the whole code specifying the set of declarations of the considered program presented in Chapter 4. It can be seen that, the initial agent initiates a communication of the participant `'a` with `'b`. To run the interpreter, we must invoke the following instruction.

```
search runs ({DS .  (init('a,'b)||(resp('b)||environment('i)))} , 22)
  =>* < TpPg , StL , VrLt , Bl > {Nt} such that
    TpSt := getGlobalStoreFromStrStoreList (StL) ∧
    TpVr := getStreamOfTrmKnowInStore (TpSt , ['know ('a,'_)]) ∧
```

```
TpVr1 := getStreamOfTrmKnowInStore (TpSt , ['know ('b,'_)]) ∧
TpVr2 := getStreamOfTrmKnowInStore (TpSt , ['know ('i,'_)]) ∧
TpEx := getExpFromStream (TpVr1 , ['secret ('_)] , TpSt) ∧
consultContent (TpVr , TpEx , TpSt) == true' ∧
consultContent (TpVr2 , TpEx , TpSt) == false' .
```

The term in the left hand side of the symbol =>* is the initial state in the execution of the tccp program. The right hand side of the symbol specifies the final state of the execution. The symbol =>* searches for a proof consisting of none, one, or more steps that reaches the final state. The final state which is composed by some variables following the structure for configurations presented above. `TpPg` represents the given program, `StL` represents the resulting structured store, `VrLt` represents the corresponding list of variables, `Bl` is a boolean constraint and `Nt` represents the given threshold. With the command `such that`, we filter the expected result. In this case, we establish that the resulting structured store `StL` must contain 1) the term `secret/1` in the private knowledge of the responder `'b`, which means that the protocol ended; 2) the value, labeled as `secret` by `'b`, in the private knowledge of the initiator `'a`; and finally 3) the value, labeled as `secret` by `'b` must be known just by `'a`.

As one can see below, the tccpInterpreter computes one solution at state `60`. We know that the protocol has been completed since the expected nonce has been declared `secret`. Thus, `'a` is able to complete the protocol with the responder `'b` at time instant 21. Since we are dealing with structured stores, we can observe which information has been added at each time instant.

```
Solution 1 (state 60)
states: 61 rewrites: 324834 in 620ms cpu (627ms real) (523925 rewrites/second)
StL --> (empty{0}) =>
((('A :=' 'a),,('B :=' 'b),,('B' :=' 'b),,('I :=' 'i)){1}) =>
((['know('A,'SAi)],,('NAi :=' 'nNAi)){2}) =>
((['chn(['msg(['enc(['k('B)],['cons('A,['cons('NAi)])])])],'Si1)],,
  ('SAi :=' [['a('B)],['n('NAi)] | 'SAi1])){3}) => (empty{4}) =>
((['know('I,'SI)],,('Si1 :=' 'S),,('S :=' 'ok),,
('M :=' ['msg(['enc(['k('B)],['cons('A,['cons('NAi)])])])])){5}) =>
((['rcv('M,'S1)],,('I' :=' 'I)){6}) => (empty{7}) =>
((['know('B','SBr)],,('Ar :=' 'A),,('NAr :=' 'NAi),,('NBr :=' 'nNBr),,
('S1 :=' 'Sr1),,('Sr1 :=' 'ok)){8}) =>
((['chn(['msg(['enc(['k('Ar)],['cons('NAr,['cons('NBr)])])])],'Sr2)],,
  ('SBr :=' [['a('Ar)],['n('NAr)],['n('NBr)] | 'SBr1])){9}) =>
(empty{10}) =>
((('M' :=' ['msg(['enc(['k('Ar)],['cons('NAr,['cons('NBr)])])])]),,
  ('S' :=' 'ok),,('SI :=' 'SI'),,('Sr2 :=' 'S')){11}) =>
((['rcv('M','S1')],,('I'' :=' 'I)){12}) => (empty{13}) =>
((('NBi :=' 'NBr),,('S1' :=' 'Si2),,('Si2 :=' 'ok)){14}) =>
```

```
((['chn(['msg(['enc(['k('B)],['cons('NBi)])])],'Si3)],,
  ('SAi1 :=' [['n('NBi)] | 'SAi2]))){15}) => (empty{16}) =>
((('M'' :=' ['msg(['enc(['k('B)],['cons('NBi)])])]),,('S'' :=' 'ok),,
  ('SI :=' 'SI''),,('Si3 :=' 'S''))){17}) =>
((['rcv('M'','S1'')],,('I''' :=' 'I)){18}) => (empty{19}) =>
((('S1'' :=' 'Sr3),,('Sr3 :=' 'ok)){20}) =>
('SBr1 :=' [['secret('NBr)] | 'SBr2]){21}
```

We have shown how to interpret the output for an execution. Let us now demonstrate how to look for an attack in the protocol. Similarly to the first case, we have to specify an initial and final state. The idea is to introduce a *bad* final state, i.e., a state where an attack has occurred. In this case, the final state says that both the honest principal `'a` and the intruder `'i` know the secret nonce generated by `'b`.

```
search runs ({DS .  (init('a,'i) || (resp('i) || (environment('i) ||
                    ask(true)-> (init ('i,'b) || resp('b)))))} , 26)
  =>* < StL > such that
    TpSt := getGlobalStoreFromStrStoreList (StL) ∧
    TpVr := getStreamOfTrmKnowInStore (TpSt , ['know ('a,'_)]) ∧
    TpVr1 := getStreamOfTrmKnowInStore (TpSt , ['know ('b,'_)]) ∧
    TpVr2 := getStreamOfTrmKnowInStore (TpSt , ['know ('i,'_)]) ∧
    TpEx := getExpFromStream (TpVr1 , ['secret ('_)] , TpSt) ∧
    consultContent (TpVr , TpEx , TpSt) == true' ∧
    consultContent (TpVr2 , TpEx , TpSt) == true' .
```

We can see that at time instant `25`, the responder `'b` stores that the nonce generated by him is secret, thus the protocol run has finished. We can also see that during the execution, it has been stored the knowledge gained by the different participants of the protocol. The initiator `'a` has also been completed the protocol (to her knowledge). Moreover, note that `'a` thinks, from the time instant `3`, that the messages she will receive come from `'i` , which is true but from the time instant `11`, `'b` thinks that he is communicating with `'a`, which is false.

```
Solution 1 (state 592)
states: 593 rewrites: 259361436 in 591320ms cpu (591321ms real)
                     (438614 rewrites/second)
StL --> (empty{0}) =>
((('A :=' 'a),,('B :=' 'i),,('B' :=' 'i),,('I :=' 'i)){1}) =>
((['know('A,'SAi)],,('A' :=' 'i),,('B'' :=' 'b),,('B''' :=' 'b),,
  ('NAi :=' 'nNAi)){2}) =>
((['chn(['msg(['enc(['k('B)],['cons('A,['cons('NAi)])])])],'Si1)],,
  ['know('A','SAi')],,('NAi' :=' 'nNAi'),,
  ('SAi :=' [['a('B)],['n('NAi)] | 'SAi1])){3}) =>
(((['chn(['msg(['enc(['k('B'')],['cons('A',['cons('NAi')])])])],'Si1')],,
  ('SAi' :=' [['a('B'')],['n('NAi')] | 'SAi1'])){4}) =>
((('M :=' ['msg(['enc(['k('B)],['cons('A,['cons('NAi)])])])]),,
  ('S :=' 'ok),,('SI :=' 'SAi'),,('Si1 :=' 'S)){5}) =>
((('C :=' ['cons('A,['cons('NAi)])]),,('I' :=' 'I)){6}) =>
((('SAi1' :=' [['cnt('C)] | 'Si]),,('X :=' 'B'')){7}) =>
((['rcv(['msg(['enc(['k('X)],'C)])],'S1)],,('S' :=' 'ok),,
  ('M' :=' ['msg(['enc(['k('B'')],['cons('A',['cons('NAi')])])])]),,
  ('SI' :=' 'SAi'),,('Si1' :=' 'S')){8}) =>
((('I'' :=' 'I),,('Si :=' [['blk('M')] | 'Si'])){9}) =>
((['know('B''','SBr')],,('Ar' :=' 'A),,('NAr' :=' 'NAi),,
  ('NBr' :=' 'nNBr'),,('S1 :=' 'Sr1'),,('Sr1' :=' 'ok)){10}) =>
((['chn(['msg(['enc(['k('Ar')],['cons('NAr',['cons('NBr')])])])],'Sr2')],,
  ('SBr' :=' [['a('Ar')],['n('NAr')],['n('NBr')] | 'SBr1'])){11}) =>
  (empty{12}) =>
((('M'' :=' ['msg(['enc(['k('Ar')],['cons('NAr',['cons('NBr')])])])]),,
  ('S'' :=' 'ok),,('SI'' :=' 'SAi'),,('Sr2' :=' 'S'')){13}) =>
((['rcv('M'','S1')],,('I''' :=' 'I)){14}) => (empty{15}) =>
((('NBi :=' 'NBr'),,('S1' :=' 'Si2),,('Si2 :=' 'ok)){16}) =>
((['chn(['msg(['enc(['k('B)],['cons('NBi)])])],'Si3)],,
  ('SAi1 :=' [['n('NBi)] | 'SAi2])){17}) => (empty{18}) =>
((('M''' :=' ['msg(['enc(['k('B)],['cons('NBi)])])]),,('S''' :=' 'ok),,
  ('SI''' :=' 'SAi'),,('Si3 :=' 'S''')){19}) =>
((('C' :=' ['cons('NBi)]),,('I'''' :=' 'I)){20}) =>
((('Si' :=' [['cnt('C')] | 'Si''']),,('X' :=' 'B'')){21}) =>
((['rcv(['msg(['enc(['k('X')],'C')])],'S1'')]){22}) => (empty{23}) =>
((('S1'' :=' 'Sr3'),,('Sr3' :=' 'ok)){24}) =>
('SBr1' :=' [['secret('NBr')] | 'SBr2']){25}
```

Apart from the solution to the query, Maude shows up the time spent during the computation of each solution and the number of rewrites performed. In the first case, the result is given in `627ms` after rewriting 324834 steps. For the second case, the system spent almost `10 minutes` after 259361436 rewrites. Note that, due to the generality of the model, that does not restricts to the detection of a single attack but is general for all the Dolev-Yao attacks, we have to explore more possible execution paths for the protocol. We think that this performance is also caused due to the non-determinism of some of the parts of the interpreter.

6.5.2. Analyzing the Otway-Rees protocol

Let us show an excerpt of the execution of a trace that simulates the correct behavior of the Otway-Rees protocol. Consider the tccp program modeling the specification of the Otway-Rees protocol presented in Chapter 4 and whose initial agent is `(init('a,'b,'s) || (resp('b,'s) || || (server('s) || environment('i))))`. For readability, we use the variable DS instead of the whole code specifying the set of declarations of the considered program. It can be seen that, the initial agent initiates a communication of the participant `'a` with `'b` via a trusted third party `'s`. To run the interpreter, we must invoke the following instruction.

```
search runs ({DS . (init('a,'b,'s) || (resp('b,'s) || (server('s) ||
                   environment('i))))} , 28)
  =>* < TpPg , StL , VrLt , Bl > {Nt} such that
    TpSt := getGlobalStoreFromStrStoreList (StL) ∧
    TpVr := getStreamOfTrmKnowInStore (TpSt , ['know ('a,'_)]) ∧
    TpVr1 := getStreamOfTrmKnowInStore (TpSt , ['know ('b,'_)]) ∧
    TpVr2 := getStreamOfTrmKnowInStore (TpSt , ['know ('s,'_)]) ∧
    TpVr3 := getStreamOfTrmKnowInStore (TpSt , ['know ('i,'_)]) ∧
    TpEx := getExpFromStream (TpVr2 , ['pk ('_)] , TpSt) ∧
    consultContent (TpVr , TpEx , TpSt) == true' ∧
    consultContent (TpVr1 , TpEx , TpSt) == true' ∧
    consultContent (TpVr3 , TpEx , TpSt) == false' .
```

In this case, we establish that the resulting structured store `StL` must contain 1) the term `pk/1` in the private knowledge of the server `'s`, which means that the server has generated the secret key; 2) the value, labeled as `pk` by `'s`, in the private knowledge of the initiator `'a` and the responder `'b`; and finally 3) the value, labeled as `pk` by `'s` must be unknown by `'i`.

Following, we show the trace corresponding to the correct behavior of the protocol. We can see at time instants 21 and 27 the expected key, unknown by the intruder, stored in the private knowledge of the responder `'b` and the initiator `'a`, respectively.

```
Solution 1 (state 91)
states: 92 rewrites: 2371443 in 7770ms cpu (7769ms real)
                   (305205 rewrites/second)
StL --> (empty{0}) =>
((('A :=' 'a),,('B :=' 'b),,('B' :=' 'b),,('I :=' 'i),,('S :=' 's),,
  ('S' :=' 's),,('S'' :=' 's)){1}) =>
((['know('A,'SAi)],,('NAi :=' 'nNAi),,('Ni :=' 'nNi)){2}) =>
((['chn(['msg(['enc('plain,['cons('Ni,['cons('A,['cons('B)])])])],
              ['enc(['sk('A,'S)],['cons('NAi,['cons('Ni,['cons('A,
                                    ['cons('B)])])])])])],'S1i)],,
  ('SAi :=' [['a('B)],['a('S)],['n('Ni)],['n('NAi)] | 'SA1i])){3}) =>
(empty{4}) =>
((['know('I,'SI)],,('M :='['msg(['enc('plain,['cons('Ni,['cons('A,
                                              ['cons('B)])])])],
                          ['enc(['sk('A,'S)],['cons('NAi,['cons('Ni,
                                ['cons('A,['cons('B)])])])])])]),,
  ('S''' :=' 'ok),,('S1i :=' 'S''')){5}) =>
((['rcv('M,'S1)],,('I' :=' 'I)){6}) => (empty{7}) =>
((['know('B','SBr)],,('Ar :=' 'A),,('NBr :=' 'nNBr),,('Nr :=' 'Ni),,
  ('Enc1 :=' ['enc(['sk('A,'S)],['cons('NAi,['cons('Ni,['cons('A,
                                     ['cons('B)])])])])]),,
  ('S1 :=' 'S1r),,('S1r :=' 'ok)){8}) =>
((['chn(['msg(['enc('plain,['cons('Nr,['cons('Ar,['cons('B')])])])],
         'Enc1,['enc(['sk('B','S')],['cons('NBr,['cons('Nr,
                  ['cons('Ar,['cons('B')])])])])])])],'S2r)],,
  ('SBr :=' [['a('Ar)],['a('S')],['n('Nr)],['n('NBr)] | 'SB1r])){9}) =>
(empty{10}) =>
((('M' :=' ['msg(['enc('plain,['cons('Nr,['cons('Ar,['cons('B')])])])],
           'Enc1,['enc(['sk('B','S')],['cons('NBr,['cons('Nr,
                   ['cons('Ar,['cons('B')])])])])])])]),,
  ('S'''' :=' 'ok),,('S2r :=' 'S''''),,('SI :=' 'SI')){11}) =>
((['rcv('M','S1')],,('I'' :=' 'I)){12}) => (empty{13}) =>
((['know('S'','SSs)],,('A :=' 'A),,('As :=' 'A),,('B :=' 'B'),,
  ('Bs :=' 'B'),,('KABs :=' 'nKABs),,('NAs :=' 'NAi),,('NBs :=' 'NBr),,
  ('Ni :=' 'Ni),,('Ns :=' 'Ni),,('S1' :=' 'S1s),,('S1s :=' 'ok)){14}) =>
((['chn(['msg(['enc('plain,['cons('Ns)])],
     ['enc(['sk('As,'S'')],['cons('NAs,['cons('KABs)])])],
     ['enc(['sk('Bs,'S'')],['cons('NBs,['cons('KABs)])])])])],'S2s)],,
  ('SSs :=' [['a('As)],['a('Bs)],['n('Ns)],['n('NAs)],['n('NBs)],
          ['pk('KABs)] | 'SS1s])){15}) => (empty{16}) =>
((('M'' :=' ['msg(['enc('plain,['cons('Ns)])],
          ['enc(['sk('As,'S'')],['cons('NAs,['cons('KABs)])])],
          ['enc(['sk('Bs,'S'')],['cons('NBs,['cons('KABs)])])])]),,
  ('S''''' :=' 'ok),,('S2s :=' 'S'''''),,('SI :=' 'SI'')){17}) =>
((['rcv('M'','S1'')],,('I''' :=' 'I)){18}) => (empty{19}) =>
((('Enc2 :=' ['enc(['sk('As,'S'')],['cons('NAs,['cons('KABs)])])]),,
  ('KABr :=' 'KABs),,('S1'' :=' 'S3r),,('S3r :=' 'ok)){20}) =>
((['chn(['msg(['enc('plain,['cons('Nr)])],'Enc2)],'S4r)],,
  ('SB1r :=' [['pk('KABr)] | 'SB2r])){21}) => (empty{22}) =>
((('M''' :=' ['msg(['enc('plain,['cons('Nr)])],'Enc2)]),,
  ('S'''''' :=' 'ok),,('S4r :=' 'S''''''),,('SI :=' 'SI''')){23}) =>
((['rcv('M''','S1''')],,('I'''' :=' 'I)){24}) => (empty{25}) =>
((('KABi :=' 'KABs),,('S1''' :=' 'S2i),,('S2i :=' 'ok)){26}) =>
('SA1i :=' [['pk('KABi)] | 'SA2i]){27}
```

In the following, we show the trace corresponding to the typing attack discovered in the Otway-Rees protocol. We can see at time instant 11, the principal 'a stores in her private knowledge the expected key, but this time the key is a sequence of terms. By using the auxiliary function `isSequenceOfTerm` that returns `true'` if the given argument is a sequence of terms, we can detect the vulnerability.

```
search runs ({DS . (init('a,'b,'s) || (resp('b,'s) || (server('s) ||
                    environment('i))))} , 28)
  =>* < TpPg , StL , VrLt , Bl > {Nt} such that
    TpSt := getGlobalStoreFromStrStoreList (StL) ∧
    TpVr := getStreamOfTrmKnowInStore (TpSt , ['know ('a,'_)]) ∧
    TpEx := getExpFromStream (TpVr , ['pk ('_)] , TpSt) ∧
    isSequenceOfTerm (TpEx) == true' .

Solution 1 (state 31)
states: 32 rewrites: 319156 in 1380ms cpu (1378ms real)
                    (231257 rewrites/second)
StL --> (empty{0}) =>
((('A :=' 'a),,('B :=' 'b),,('B' :=' 'b),,('I :=' 'i),,('S :=' 's),,
  ('S' :=' 's),,('S'' :=' 's)){1}) =>
((['know('A,'SAi)],,('NAi :=' 'nNAi),,('Ni :=' 'nNi)){2}) =>
((['chn(['msg(['enc('plain,['cons('Ni,['cons('A,['cons('B)])])])],
              ['enc(['sk('A,'S)],['cons('NAi,['cons('Ni,['cons('A,
                                      ['cons('B)])])])])])])],'S1i)],,
  ('SAi :=' [['a('B)],['a('S)],['n('Ni)],['n('NAi)] | 'SA1i])){3}) =>
(empty{4}) =>
(('M :=' ['msg(['enc('plain,['cons('Ni,['cons('A,['cons('B)])])])],
               ['enc(['sk('A,'S)],['cons('NAi,['cons('Ni,['cons('A,
                                      ['cons('B)])])])])])]),,
  (['know('I,'SI)],,('S''' :=' 'ok),,('S1i :=' 'S'''))){5}) =>
((('E1 :=' ['enc(['sk('A,'S)],['cons('NAi,['cons('Ni,['cons('A,
                                      ['cons('B)])])])])]),,
  ('C :=' 'Ni),,('I' :=' 'I),,('K :=' ['cons('Ni,['cons('A,
                                      ['cons('B)])])]),,
  ('N# :=' ['cons('A,['cons('B)])]),,
  ('N#' :=' ['enc(['sk('A,'S)],['cons('NAi,['cons('Ni,['cons('A,
                                      ['cons('B)])])])])])])){6}) =>
(('SI :=' [['cnt('C)],['pk('K)],'E1 | 'Si]){7}) =>
(['rcv(['msg(['enc('plain,['cons('C)])],'E1)],'S1)]{8}) => (empty{9}) =>
((('KABi :=' ['cons('Ni,['cons('A,['cons('B)])])]),,('S1 :=' 'S2i),,
  ('S2i :=' 'ok)){10}) => ('SA1i :=' [['pk('KABi)] | 'SA2i]){11}
```

6.6. Concluding remarks

In this chapter, we have presented the tccpInterpreter system, an interpreter for the tccp language implemented in Maude. We have described how the Maude language allows us to model the syntax and the operational semantics of tccp, as well as the underlying constraint system. To our knowledge, this is the first tccp interpreter.

By using several examples, we have described the functionalities of the tool. We have show how the tccpInterpreter allows us to simulate the execution of a given tccp program and to carry out certain reachability proofs over tccp specifications. We have also presented the verification process carried out in the analysis of the security protocols presented in Chapter 4.

7
Conclusions and Future Work

In this thesis, we have shown how tccp can be used as a suitable language to verify security protocols. We have taken advantage of the underlying constraint system by introducing several functions that allow us to carry out and refine the verification process. This allows us to improve the compactness and clarity of the model. Many of the defined components in our model, in particular the environment, the specified constraints and the auxiliary functions can be reused for the analysis of other protocols.

We have presented a sound transformation from utcc processes into tccp specifications. The two languages belong to the concurrent constraint paradigm, but they have very different features which make the transformation difficult. They both can be used to specify protocols. The transformation shows that tccp is expressive enough to model utcc processes. In particular, by means of a synchronization mechanism based on a shared stream that acts as a clock, the explicit notion of time of utcc can be simulated. Moreover, the new abstraction operation can be expressed in terms of parameters passing among calls. The transformation can be automatized, which would allow us to reuse the tools defined for the tccp language, such as the recent tccp interpreter.

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. To our knowledge, there was no adequate and public implementation of tccp so far. 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 a perspicuous formulation in the task of specifying transition systems. It presents a rich notation supporting formal specification and implementation of concurrent systems. In this work, we demonstrate the feasibility and the interest of formalizing the behavior of tccp with the Maude language. This interpreter allows us to explore the particular features of tccp and its behavior (maximal parallelism and the underlying constraint system). 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.

Finally, by using the tool, we have shown how we can check safety properties on tccp programs to detect vulnerabilities that may lead to unexpected states. Specifically, we have detected the traces describing the attacks discovered in the Needham-Schroeder protocol and the Otway-Rees protocol.

The tccpInterpreter is publicly available at the following address:
`http://www.dsic.upv.es/~alescaylle/tccp.html`.

We plan to extend the specification of the `environment`, increasing actions an attacker can perform, for modeling other kind of protocols. We plan to implement the transformation, that translates a utcc program into a tccp program, and to study the relation of the tccp language with other concurrent languages such as Linda. We plan to extend the tccpInterpreter in several ways. To improve the interface of the system we plan to construct a graphical web interface. We also plan to improve the performance of the interpreter and to study how to adapt the model-checking technique existing for tccp programs [17] in the Maude rewriting-based framework.

Bibliography

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

[2] M. Alpuente, M.M. Gallardo, E. Pimentel, and A. Villanueva. A semantic framework for the abstract model checking of tccp programs, 2005.

[3] D. Basin and G. Denker. Maude versus Haskell: an Experimental Comparison in Security Protocol Analysis. In Kokichi Futatsugi, editor, *Electronic Notes in Theoretical Computer Science*, volume 36, Amsterdam, 2001. Elsevier Science Publishers.

[4] G. Bella and S. Bistarelli. Soft constraint programming to analysing security protocols, 2004.

[5] J. A. Bergstra and J. W. Klop. Algebra of Communicating Processes with Abstraction. *Theor. Comput. Sci.*, 37:77–121, 1985.

[6] C. Bodei. *Security Issues in Process Calculi.* PhD thesis, Dipartimento di Informatica, Universitadi Pisa, January 2000.

[7] F. S. de Boer, M. Gabbrielli, and M. C. Meo. A Timed Concurrent Constraint Language. *Information and Computation*, 161(1):45–83, 2000.

[8] C. Boyd. Hidden Assumptions in Cryptographic Protocols. *IEE Proceedings*, 6(137):433–436, November 1990.

[9] C. Boyd and A. Mathuria. *Protocols for Authentication and Key Establishment.* Springer-Verlag, Berlin Heidelberg, 2003.

[10] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: a declarative language for real-time programming. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 178–188, New York, NY, USA, 1987. ACM.

[11] J. A. Clark and J. L. Jacob. A survey of authentication protocol literature. Technical report, Defence Evaluation Research Agency, 1997.

[12] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott. *All About Maude - A High-Performance Logical Framework,*

How to Specify, Program and Verify Systems in Rewriting Logic, volume 4350 of *Lecture Notes in Computer Science.* Springer, 2007.

[13] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott. Maude Web Site, 2009. `http://maude.csl.sri.com/`.

[14] R. Corin, S. Etalle, P. H. Hartel, and A. Mader. Timed model checking of security protocols. In *FMSE '04: Proceedings of the 2004 ACM workshop on Formal methods in security engineering*, pages 23–32, New York, NY, USA, 2004. ACM.

[15] G. Denker, J. Meseguer, and C. Talcott. Protocol Specification and Analysis in Maude. In N. Heintze and J. Wing, editors, *Proceedings of Workshop on Formal Methods and Security Protocols*, Indianapolis, Indiana, June 1998.

[16] D. Dolev and A. C. Yao. On the Security of Public Key Protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, March 1983.

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

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

[19] V. Gupta, V. A. Saraswat, and R. Jagadeesan. Foundations of Timed Concurrent Constraint Programming. In *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 71–80, Paris, France, July 1994. IEEE Computer Society Press.

[20] V. Gupta, V. A. Saraswat, and R. Jagadeesan. Hybrid cc, hybrid automata and program verification, 1996.

[21] V. Gupta, V. A. Saraswat, and P. Struss. A model of a photocopier paper path. In *Proceedings of the 2nd IJCAI Workshop on Engineering Problems for Qualitative Reasoning*, August 1995.

[22] V. Gupta, V.A. Saraswat, and R. Jagadeesan. Timed Default Concurrent Constraint Programming. *Journal of Symbolic Computation*, 22(5–6):475–520, December 1996.

[23] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[24] D. Harel and A. Pnueli. On the development of reactive systems, 1985.

[25] S. Haridi, P. Van Roy, P. Brand, and C. Schulte. Programming Languages for Distributed Applications, 1998.

[26] C. A. R. Hoare. Communicating sequential processes, 1978.

[27] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 111–119, New York, NY, USA, 1987. ACM.

[28] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, September 1991.

[29] A. Lescaylle and A. Villanueva. Using tccp for the Specification and Verification of Communication Protocols. In *Proceedings of the 16th International Workshop on Functional and (Constraint) Logic Programming (WFLP'07)*, 2007.

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

[31] A. Lescaylle and A. Villanueva. The tccp Interpreter. *Electronic Notes in Theoretical Computer Science*, to appear, 2009.

[32] A. Lescaylle and A. Villanueva. The typing attack detected in the Otway-Rees Protocol. Technical report, DSIC, Univeridad Politécnica de Valencia, 2009. Available at http://www.dsic.upv.es/˜villanue/techrep-FORTE09.pdf.

[33] L. Logrippo, M. Faci, and M. Haj-Hussein. An introduction to LOTOS: learning by examples, 1992.

[34] G. Lowe. Breaking and Fixing the Needham-Schroeder public-key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 147–166. Springer Verlag, 1996.

[35] Z. Manna and A. Pnueli. *Temporal verification of reactive systems: safety.* Springer-Verlag New York, Inc., New York, NY, USA, 1995.

[36] F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In *In CONCUR. LNCS 630*, pages 550–564. Springer-Verlag, 1992.

[37] N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework, January 2004.

[38] C. Meadows. Formal Verification of Cryptographic Protocols: A Survey. In *ASIACRYPT: Advances in Cryptology – ASIACRYPT: International Conference on the Theory and Application of Cryptology.* LNCS, Springer-Verlag, 1994.

[39] C. Meadows. The NRL Protocol Analyzer: An Overview. *Journal of Logic Programming*, 26(2):113–131, 1996.

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

[41] R. Needham and M. Schroeder. Using Encryption for Authentification in Large Networks of Computers. *Communications of the ACM*, 21(12):993–999, December 1978.

[42] M. Nielsen, C. Palamidessi, and Valencia F.D. Temporal Concurrent Constraint Programming: Denotation, Logic and Applications. *Nordic Journal of Computing*, 9(1):145–188, 2002.

[43] M. Nielsen, C. Palamidessi, and F. D. Valencia. On the expressive power of temporal concurrent constraint programming languages. In *PPDP '02: Proceedings of the 4th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 156–167, New York, NY, USA, 2002. ACM.

[44] 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 8–19, New York, NY, USA, 2008. ACM.

[45] C. Olarte and F. D. Valencia. Universal concurrent constraint programing: symbolic semantics and applications to security. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, pages 145–150, New York, NY, USA, 2008. ACM.

[46] D. Otway and O. Rees. Efficient and timely mutual authentication. *ACM Operating System Review*, 21(1):8–10, 1987.

[47] V. A. Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie-Mellon University, Cambridge, MA, January 1989.

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

[49] E. Shapiro. The family of concurrent logic programming languages, 1989.

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

[51] P. Syverson and C. Meadows. Formal requirements for key distribution protocols. In *Proceedings of Eurocrypt '94*, pages 320–331. Springer-Verlag, 1994.

[52] S. Tini. On the Expressiveness of Timed Concurrent Constraint Programming. In *Electronics Notes in Theoretical Computer Science.* Electronics, 1999.

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

[54] A. Verdejo and N. Marti-Oliet. Implementing CCS in Maude 2. In *Proceedings Fourth International Workshop on Rewriting Logic and its Applications, WRLA 2002*, pages 239–257. Elsevier, 2002.

A
Papers developed from this thesis

- **Internationals**
  - Lescaylle, A. and Villanueva, A.
    Automated verification of security protocols in tccp. Submitted to the 19th International Workshop on Functional and (Constraint) Logic Programming (WFLP'10).
    Madrid (Spain) 2010
  - Lescaylle, A. and Villanueva, A.
    Bridging the gap between two Concurrent Constraint Languages. Submitted to the 19th International Workshop on Functional and (Constraint) Logic Programming (WFLP'10).
    Madrid (Spain) 2010
  - Lescaylle, A. and Villanueva, A.
    A tool for Generating a Symbolic Representation of tccp executions. Proceedings of the 17th International Workshop on Functional and (Constraint) Logic Programming (WFLP'08). Electronic Notes in Theoretical Computer Science, Volume 246, Pages 131-145, August 2009.
    Siena (Italy) 2008
  - Lescaylle, A. and Villanueva, A.
    Using tccp for the Specification and Verification of Communication Protocols. Proceedings of the 16th International Workshop on Functional and (Constraint) Logic Programming (WFLP'07).
    Paris (France) 2007
- **Nationals**
  - Lescaylle, A. and Villanueva, A.
    The tccp Interpreter. IX Jornadas sobre Programación y Lenguajes (PROLE'09). Electronic Notes in Theoretical Computer Science, Volume to appear.
    San Sebastián (Spain) 2009.

  - Lescaylle, A. and Villanueva, A.
    Using tccp for the Specification of Communication Protocols. Actas de las VII Jornadas sobre Programación y Lenguajes (PROLE'07).
    Zaragoza (Spain) 2007

- **Technical reports (unpublished)**

  - Lescaylle, A. and Villanueva, A.
    Bridging the gap between two Concurrent Constraint Languages. Submitted to the 19th International Workshop on Functional and (Constraint) Logic Programming (WFLP'10).
    DSIC, Univeridad Politécnica de Valencia. 2009

- **Others**

  - Lescaylle, A. and Villanueva, A.
    Implementing tccp in Maude. Reunión de la Red Maude 2009.
    Málaga (Spain) 2009.
  - Lescaylle, A. and Villanueva, A.
    Authentication Protocol Analysis in a Timed Concurrent Constraint Language. XVII Jornadas de Concurrencia y Sistemas Distribuidos (JCSD 2009)
    Sagunto (Spain) 2009.

B
The tccpTranslator framework

In Figure B.1, we show the sketch of the transformation from utcc into tccp. The tccpTranslator framework is composed by two principal modules, the first is related to the encoding process and the second specifies the synchronization process between both languages, since the resulting tccp agents must be executed following the notion of time of utcc. tccpTranslator gets from a file the specification of the given utcc program which, by using the auxiliary function τ_P, is mapped in a tccp program, the result of the encoding. τ_P invokes the auxiliary functions τ_A and `synchronization` to complete the translation. τ_A generates a tccp agent from a given utcc process, whereas `synchronization` takes as input an utcc process and generates a tccp agent who captures the passing of time, from the evolution of the given utcc process, in tccp. `synchronization` uses the auxiliary functions `instant` and `follows` to achieve its goal. `instant` and `follows`, given an utcc process and an initial store, return the store generated by the given process and the process to be executed in the following time instant, respectively. We explain each function in more detail below.

The body of the function controlling the framework, tccpTranslator, is shown below. $\mathtt{u}_p$ is instantiated to the given utcc program, which is obtained using the auxiliary function `read`. The variable $\mathtt{t}_p$, whose content is finally returned,

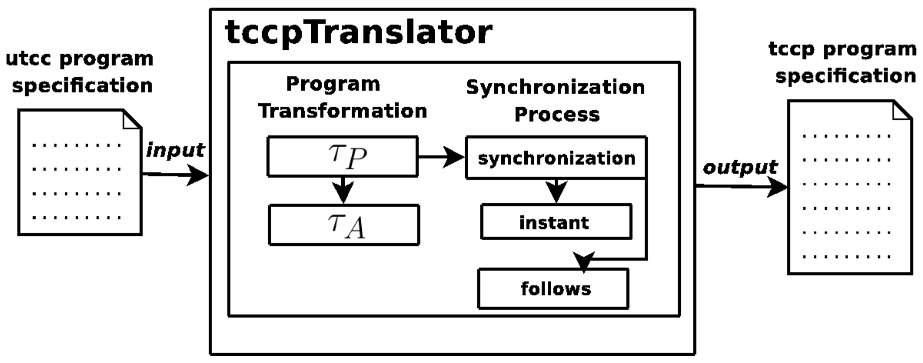

Figure B.1: The tccpTranslator framework

is instantiated to the tccp program embedding the given utcc program. The auxiliary function τ_P carries out the embedding process.

```
tccpProgram tccpTranslator()
    $u_p$ : utccProgram;
    $t_p$ : tccpProgram;
    utccProgramfile : file;
    $u_p$ = read(utccProgramfile);
    $t_p$ = $\tau_P(u_p)$;
    return $t_p$;
end;
```

τ_P, given an utcc program, returns a tccp program that embeds the input program. The embedding process (the general idea) contains the following steps: 1) to convert the set of utcc declarations of processes in a set of tccp declarations; 2) to convert the utcc process to be executed in a tccp agent. To complete the first step, the variable $\mathtt{u}_{ds}$ is instantiated to the set of utcc declarations, where the function `declarations` extracts them from the given program. To complete the second step, $\mathtt{u}_{pr}$ is instantiated to the utcc process to be executed, where the auxiliary function `process` extracts it from the given program. The `while` loop is part of the first step, in which each utcc declaration is translated in a tccp declaration which is add it to the set of declarations of the tccp program that is being built. The variable $\mathtt{u}_d$ is instantiated to an utcc declaration from the set $\mathtt{u}_{ds}$. Then, by using functions `name` and `body`, variables $\mathtt{u}_{dn}$ and $\mathtt{u}'_{pr}$ are instantiated to the name and the process of the current declaration, respectively. $\mathtt{t}'_a$ is instantiated to the tccp agent, generated by the function τ_A, encoding the given utcc process. Thus, the tccp declaration can be created. Therefore, by using the function `updateDelarationSet`, the set of tccp declarations stored in $\mathtt{t}_{ds}$, is updated. Once the set of utcc declarations is processed, the utcc process recovered in the second step is encoded to the corresponding tccp agent. $\mathtt{t}_a$ is instantiated to the resulting tccp agent, which is composed by the parallel composition of the tccp agent returned by τ_A, a tell agent and the tccp agent returned by `synchronization`. The tell agent and the function `synchronization` are used to achieve the time synchronization between utcc and tccp. Note that the notion of time of utcc (explicit) and tccp (implicit) differ. A time unit in utcc (the observable transition) corresponds with several time unit in tccp, so we need to specify a mechanism that emits a signal when the tccp agents, obtained by the encoding process, must be executed following the behaviors of their original utcc processes. The variable *Syn* is used to emits such signal. The value `wait` means that the agents to be executed in the following time instant, according to the execution of the original utcc processes, must wait. `ok` means that such agents must be executed. The tell agent initialises the variable *Syn* to `wait`.

`synchronization` contains in its body the mechanism to update the variable Syn. Finally, the variable $\mathtt{t}_p$ is instantiated to the generated tccp program ($\mathtt{t}_{ds}.\mathtt{t}_a$) and returned.

```
tccpProgram $\tau_P$(input $u_p$ : utccProgram)
    $u_{ds}$ : utccDeclarationSet;
    $u_{pr}$ : utccProcess;
    cursor : integer;
    $t_p$ : tccpProgram;
    $t_{ds}$ : tccpDeclarationSet;
    $t_a$ : tccpAgent;
    $Syn$,$C_T$ : stream;
    $u_{ds}$ = declarations($u_p$);
    $u_{pr}$ = process($u_p$);
    cursor = 0;
    while ($u_{ds}$[cursor] $\neq$ $Null$) do
          $u_d$ : utccDeclaration;
          $u_{dn}$ : utccDeclarationName;
          $u'_{pr}$ : utccProcess;
          $t'_a$ : tccpAgent;
          $u_d$ = $u_{ds}$[cursor];
          $u_{dn}$ = name($u_d$);
          $u'_{pr}$ = body($u_d$);
          $t'_a$ = $\tau_A$($u'_{pr}$,$t_{ds}$);
          updateDelarationSet($t_{ds}$,$u_{dn}$ :- $t'_a$.);
          $inc$(cursor);
    end while;
    $t_a$ = ($\tau_A$($u_{pr}$,$t_{ds}$) $\|$ tell($Syn = $[wait | $C_T$]) $\|$
         synchronization($u_{pr}$,$Syn$));
    $t_p$ = $t_{ds}$.$t_a$;
    return $t_p$;
end;
```

The heart of the function τ_A is specified in a block `case` where each branch contains the instructions to transform an utcc process. The first four cases: the skip, tell, local and $\|$have a direct transformations in tccp; they are: the skip, tell, $\exists$ and $\|$agents, respectively. As we need to encode the input utcc process, we recursively process all its sub-processes. For instance, the process A of the local constructor is processed recursively. The units delay next and unless are encoded in tccp by using the choice agent shown in each case. The constraint $Syn \doteq \mathtt{ok}$ consults whether the current value of Syn is instantiated to `ok`. If the constraint holds (which means that the program can reach the next time instant, according with utcc) then $\tau_A(A,\mathrm{t}_{ds})$ is executed in the first case and, if the constraint c of the conditional agent does not hold, in the second case. To model the replication of utcc in tccp we need to specify a fresh declaration, aux_i, with recursion. Such declaration is added in the given declaration set of the tccp program being built. The choice agent, in the declaration, will execute the transformation of A in parallel with the call to itself (aux_i) when the constraint $Syn \doteq \mathtt{ok}$ holds. Then, the tccp agent resulting from the replication process, stored in $\mathtt{t}_a$, is the parallel composition of the transformation of A and the procedure call aux_i. The translation of the parametric ask is the most complicated due to its behavior. The difficulties arise in the fresh declaration abs_i since the resulting tccp agent is $\exists\, \vec{x}, S\, (\mathsf{abs}_i(\vec{x}, S) \parallel \mathsf{tell}(\mathtt{subst}_i(S)))$. The local variable S is instantiated to the stream containing the substitutions applied so far. $\mathtt{subst}_i\mathtt{/1}$ is a term whose argument, a stream, contains such substitutions. Thus, in abs_i, if the constraint $c[\vec{t}\backslash\vec{x}]$ holds and S does not contain the substitution $\{\vec{t}\backslash\vec{x}\}$ then, in parallel, the transformation of A with the free occurrences of x_i replaced with t_i, the updating of the stream S with the found substitution to avoid executing $A[\vec{t}\backslash\vec{x}]$ again and the procedure call $\mathsf{abs}_i(\vec{x}, S)$, allowing other replacements of $\vec{x}$ in A, are executed; otherwise the process ends, skip. The last case models a process of the form $Name = Process$, since it was previously translated in a tccp declaration, it is traduced in a procedure call agent. τ_A ends returning, in $\mathtt{t}_a$, the generated tccp agent.

```
tccpAgent $\tau_A$($input$ u$_{pr}$ : utccProcess,
               $input/output$ t$_{ds}$ : tccpDeclarationSet)
 t$_a$ : tccpAgent;
 case u$_{pr}$ of
  skip:  t$_a$ = skip;
  tell($c$):  t$_a$ = tell($c$);
  (local $\vec{x}$;$c$) $A$:  t$_a$ = $\exists^c \vec{x}\, (\tau_A(A,\mathrm{t}_{ds}))$;
  $A \parallel B$:  t$_a$ = $(\tau_A(A,\mathrm{t}_{ds}) \parallel \tau_A(B,\mathrm{t}_{ds}))$;
  next $A$:  t$_a$ = ask($Syn \doteq$ ok) $\longrightarrow \tau_A(A,\mathrm{t}_{ds})$;
  unless $c$ next $A$:
    t$_a$ = ask($Syn \doteq$ ok) $\longrightarrow$ now $c$ then skip else $\tau_A(A,\mathrm{t}_{ds})$;
  !$A$:  updateDelarationSet(t$_{ds}$,
         aux$_i$:- ask($Syn \doteq$ ok)$\longrightarrow$  ($\tau_A(A,\mathrm{t}_{ds}) \parallel$ aux$_i$).);
      t$_a$ = ($\tau_A(A,\mathrm{t}_{ds}) \parallel$ aux$_i$);
  (abs $\vec{x}$;$c$) $A$:  updateDelarationSet(t$_{ds}$,
         abs$_i(\vec{x}, S)$:- $\exists\, \vec{t}, S'$ (now($c[\vec{t}\backslash\vec{x}] \wedge \neg$(find($S$, $\{\vec{t}\backslash\vec{x}\}$)))
                                   then ($\tau_A(A[\vec{t}\backslash\vec{x}],\mathrm{t}_{ds}) \parallel$
                                         (tell($S = [\{\vec{t}\backslash\vec{x}\} \mid S']$) $\parallel$
                                          abs$_i(\vec{x}, S)$))
                                   else skip).);
```

```
        t_a = $\exists \vec{x}, S\,(\mathtt{abs}_i(\vec{x}, S) \parallel \mathsf{tell}(\mathtt{subst}_i(S)))$;
  $A = P$:    $a$;
 end case;
 return t_a;
end;
```

The function `synchronization` returns a tccp agent who handles the passing of time of utcc in tccp to achieve the synchronization, on the time, between both languages. This resulting agent, $\mathtt{t}_a$, handles the values of the given variable *Syn*, i.e., when it must be instantiated to *ok*. This situation represents the case when the remaining process, of the given utcc program, must be executed in the following time instant. Thus, the corresponding tccp agent must be executed. In the `while` loop, we calculate, by using the function `instant`, the store generated by the execution of the given utcc process (d in Figure 2.4). The calculated store is compared with the store calculated previously to stop or continue with the generation of clocks process. If both stores are equal then generation process is stopped; otherwise the process continues by generating the resulting tccp agent. The case when the calculated store is equal to the store calculate previously takes place when the given utcc process is the process modeling the replication. Then, to ensure the termination of this process we carry out such comparison. In the block `else`, $\mathtt{t}_a$ is updated with the parallel composition of the procedure call `utcc_clock` whose declaration contains the mechanism to update *Syn*, its first parameter is the store calculated recently and the second is the given stream *Syn*. By using the function `follows`, we may process in each step of the loop the utcc process that must be executed in the current utcc time instant.

```
tccpAgent synchronization(input u_pr : utccProcess,
                          input Syn : stream)
 u'_pr : utccProcess;
 cnt : integer;
 store[] : set_of_store;
 t_a : tccpAgent;
 u'_pr = u_pr;
 cnt = 1;
 t_a = skip;
 while (u'_pr $\neq$ skip) do
        store[cnt] = instant(u'_pr, true);
        if store[cnt] == store[cnt - 1]
        then u'_pr = skip;
        else t_a = (t_a $\parallel$ utcc_clock(store[cnt], Syn));
             u'_pr = follows(u'_pr, store[cnt]);
        ++ cnt;
 end while;
 return t_a;
end;
```

instant calculates the store generated by a given utcc process, u_{pr}. The function is specified in a block case whose possible actions calculate the store generated by u_{pr} following its semantics rules. The most peculiar case is given in the process abs since to calculate the resulting store, first the constraint c must be entailed by the given store *str*. Then, the instant of the "new" A (A with the free occurrences of x_i replaced by y_i) and the "new" abs (in order to run another possible substitution) are calculated, respectively. The resulting store is the conjunction of both results.

```
store instant($input$ $u_{pr}$ : utccProcess,
              $input/output$ $str$ : store)
 st : store;
 case $u_{pr}$ of
  skip:  st = $str$;
  tell($c$):  st = $str \cup c$;
  (local $\vec{x}$;$c$) $A$:  st = instant($A[\vec{x}'\backslash\vec{x}]$,$c \wedge \exists\vec{x} str$);
      //where $\vec{x}'$ is a sequence of fresh variables
  $A \parallel B$:  st = (instant($A$,$str$) $\wedge$ instant($B$,$str$));
  next $A$:  st = $str$;
  unless $c$ next $A$:  st = $str$;
  !$A$:  st = instant($A$,$str$);
  (abs $\vec{x}$;$c$) $A$:
      if $str \vdash c[\vec{y}\backslash\vec{x}]$ then
         st$_1$,st$_2$ : store;
         st$_1$ = instant($A[\vec{y}\backslash\vec{x}]$,$str$);
         st$_2$ = instant((abs $\vec{x}$;$c \wedge \vec{x} \neq \vec{y}$) $A$,$str$);
         st = (st$_1$ $\wedge$ st$_2$);
      else st = $true$;
  $A = P$:  st = instant($P$,$str$);
 end case;
 return st;
end;
```

The function follows obtains the utcc process that must be executed in the following time instant. It simulates the execution of such process following the functionality of the function F, used in the specification of the semantics of utcc, which determines the "future" of the given process, see [45] for more detail. Similar to instant, it is specified in a block case that obtains the process to be executed, depending on the evolution of the given utcc process, u_{pr}.

```
utccProcess follows(input u_pr : utccProcess,
                    input store : store)
 u'_pr : utccProcess;
 case u_pr of
  skip:  u'_pr = skip;
  tell(c):  u'_pr = skip;
  (local $\vec{x}$;c) A:  u'_pr = (local $\vec{x}$) follows(A,$store \wedge c$);
  $A \parallel B$:  (follows(A,store) $\parallel$ follows(B,store));
  next A:  u'_pr = A;
  unless c next A:  if $store \vdash c$ then u'_pr = skip
                    else u'_pr = A;
  !A:  u'_pr = !A;
  (abs $\vec{x}$;c) A:  u'_pr = (abs $\vec{x}$;c) follows(A,$store \wedge c$);
  $A = P$:  follows(P,store);
 end case;
 return u'_pr;
end;
```

Finally, the tccp declaration utcc_clock, modeling the clock, updates the given variable Syn to ok (tell(Syn = [ok | Sn])) when the store, resulting from the execution of the generated tccp program, entails the given store in the variable $Store$ (the constraint of the choice agent holds). Then, in the following time instant, the current value of Syn is instantiated to wait avoiding the execution of agents that must be executed in a different time instant.

```
utcc_clock(Store,Syn) :-
  ask(Store)→ ∃Sn,Sn_1((tell(Syn = [ok | Sn]) ||
                        ask(true)→ tell(Syn = [wait | Sn_1]))).
```