Types for Secure Pattern Matching with Local Knowledge in Universal Concurrent Constraint Programming

Thomas Hildebrandt[1] and Hugo A. López[1]

IT University of Copenhagen
Programming, Logic and Semantics Group
Rued Langgaards Vej 7, DK-2300 Copenhagen S, Denmark
{hilde,lopez}@itu.dk

Abstract. The fundamental primitives of Concurrent Constraint Programming (CCP), *tell* and *ask*, respectively adds knowledge to and infers knowledge from a shared constraint store. These features, and the elegant use of the constraint system to represent the abilities of attackers, make concurrent constraint programming and timed CCP (tcc) interesting candidates for modeling and reasoning about security protocols. However, they lack primitives for the communication of secrets (or local names as in the π -calculus) between agents. The recently proposed *universal* tcc (utcc) introduces a universally quantified ask operation that makes it possible to infer knowledge which is local to other agents. However, it allows agents to guess knowledge even if it is encrypted or communicated on secret channels, simply by quantifying over both the encryption key (or channel) and the message simultaneously. We present a secure utcc ($\texttt{utcc}_s$) based on: (i) a simple type system for constraints allowing to distinguish between restricted (secure) and non-restricted (universally quantifiable) variables in constraints, and (ii) a generalization of the universally quantified ask operation to allow the assumption of local knowledge. We illustrate the use of the $\texttt{utcc}_s$ calculus with examples on communication of local names (as in the π -calculus) and for giving semantics to secure pattern matching in a prototypical security language.
Key words: Concurrent Constraint Programming, Process Calculi, Type systems, Mobility, Security.

1 Introduction

A number of variants of process calculi and logical approaches have been proposed for the analysis of security protocols, including [2,6,5,8,3,11,4,14]. The approaches have generally two features in common: The first is the use of some kind of logical inference/pattern matching/unification to represent the ability of attackers and principals to infer what has been communicated, and from that knowledge construct new messages. The second is a way of representing and communicating local knowledge (such as keys or nonces in security protocols).

The combination of these two features calls for some means to control the ability to infer knowledge which is supposed to be inaccessible, e.g. a message encrypted by a key unknown to the attacker or the key itself. Typically, this takes the form of a restriction

on the rules for inference of knowledge/pattern matching, designed particularly for the considered setting of security protocols. Sometimes the restriction is enforced by the language, as e.g. in [4], however in many cases the restriction must be maintained in the specification of the attacker and the protocol under analysis.

In the present paper we propose a more general solution to representing this kind of restriction. Even though we believe that the solution is broadly applicable, in this paper we focus on the setting of concurrent constraint programming (CCP). This is due to the fact that our work was directly triggered by the interesting recent proposal of the calculus of *universal* timed concurrent constraint programming (utcc)[14], which extends timed concurrent constraint programming [16] to include a universally quantified abstraction (ask) operation. Intuitively, the new operation added in utcc, written $(\lambda\ \vec{x}; \mathsf{c})\ P$, spans a copy of the residual process $P[\vec{t}/\vec{x}]$ for all possible inferences of $\mathsf{c}[\vec{t}/\vec{x}]$. This adds the ability to extend the scope of local knowledge which is not possible in CCP [9]. In particular it was illustrated in [14] how to model a notion of *link mobility* as found in the pi-calculus and to use the universal abstraction operator for communication of messages in security protocols.

However, the universal quantification in utcc is completely unrestricted. This means that in the proposed representations of link mobility and security protocols in utcc, every agent may guess channel names and encrypted values by universal quantification. It is thus necessary to enforce a restriction on the allowed processes to make sure that this is not possible.

As a general solution for making exactly such restrictions, we propose a simple type system for constraints used as patterns in abstractions, which essentially allow to distinguish between universally abstractable and secure variables in predicates. We also propose a novel notion of *abstraction under local knowledge*, which gives a general way to model that a process (principal) knows a key and can use it to decrypt a message encrypted with this key without revealing the key.

We exemplify the type system on π calculus-like mobility of local names and for giving semantics to a novel security protocol language called Security Protocol Concurrent Constraint Programming language (SPCCP), combining the best features of the the Security CCP (SCCP) language proposed by Olarte and Valencia [14] and the Security Protocol Language (SPL) by Crazzolara and Winskel [6].

The foregoing document is divided as follows: Section 2 provides preliminary information and necessary definitions about constraint systems and the concurrent constraint family of programming languages. Section 3 introduces the type system for utcc the new abstraction rule over local knowledge, as well as termination and subject-reduction results over the type system proposed. In Section 4 we give more details on the use of the utcc with secure patterns. Finally, concluding remarks and future work are described in Section 5.

2 Preliminaries

This section provides the interested reader the main concepts of Temporal Concurrent Constraint Programming (tcc) and its universal extension (utcc), following the presentation of [14].

In CCP-based calculi all the (partial) information is *monotonically* accumulated in a so-called *store*. The store keeps the knowledge about the system in terms of *constraints*, or statements defining the possible values a variable can take (e.g., $x + y \geq 42$). Concurrent agents (i.e., processes) that are part of the system interact with each other using the store as a shared communication medium. They have two basic capabilities over the store, represented by *tell* and *ask* operations. While the former *adds* a piece of information about the system, the latter *queries* the store to determine if some piece of information can be inferred from its current content. Tell operations can act concurrently refining the information in the store while asks can serve as a general synchronization mechanism, that will be blocked if there is not enough information into the store to answer its query.

A fundamental notion in CCP-based calculi is that of a *constraint system*. Basically, a constraint system provides a signature from which syntactically denotable objects in the language called *constraints* can be constructed, and an entailment relation ($\Vdash$) specifying interdependencies among such constraints. More precisely,

Definition 1 (Constraint System). *A constraint system is a pair $CS = (\Sigma, \Delta)$ where Σ is a signature of function (*F*) and predicate (*P*) symbols, and Δ is a decidable theory over Σ (i.e., a decidable set of sentences over Σ with at least one model). The underlying language $\mathcal{L}$ of (Σ, Δ) contains the symbols $\neg, \wedge, \Rightarrow, \exists$ denoting logical negation, conjunction, implication, existential quantification.* Constants*, such as* `true` *and* `false` *denote the usual always true and always false values, respectively.* Constraints*, denoted by* $\mathsf{c}, \mathsf{d}, \ldots$ *are first-order formulae over $\mathcal{L}$. We say that* c entails d *in Δ, written* $\mathsf{c} \Vdash_\Delta \mathsf{d}$ *(or just* $\mathsf{c} \Vdash \mathsf{d}$ *when no confusion arises), if* $\mathsf{c} \Rightarrow \mathsf{d}$ *is true in all models of Δ. For operational reasons we shall require $\Vdash$ to be decidable.*

`tcc` arises as the extension of CCP for timed-systems: Including the notion of discrete time intervals (time units), a computation can be described as the interaction of a `tcc` process with the environment: At the instant i a `tcc` process P receives the store c as an initial stimulus, and when it reaches a quiescent point, it outputs d as the resulting constraint store with a residual process Q that will be executed in the instant $i+1$. Here it is where one of the most important differences between ccp and `tcc` resides, as whilst the refinement of c during the execution of P at i is monotonic, d is not necessarily a refinement of c (that is, constraints can be forgotten).

Definition 2 (`tcc` process syntax). *Processes $P, Q, \ldots \in Proc$ are built from constraints $c \in \mathcal{C}$ and variables $x \in \mathcal{V}$ in the underlying constraint system by the following syntax.*

$$P, Q \ldots ::= \mathbf{skip} \mid \mathbf{tell}\,(\mathsf{c}) \mid \mathbf{when}\,\mathsf{c}\,\mathbf{do}\,P \mid P \parallel Q \mid (\mathbf{local}\,\vec{x}; c)P \mid \mathbf{next}\,(P) \mid \mathbf{unless}\,c\,\mathbf{next}\,(P) \mid\, !P$$

Intuitively, the process $\mathbf{skip}$ does nothing, $\mathbf{tell}\,(\mathsf{c})$ adds a new constraint c into the store, while $\mathbf{when}\,\mathsf{c}\,\mathbf{do}\,P$ asks if c is present into the store in order to execute P. $(\mathbf{local}\,\vec{x}; c)P$ binds a set of variables $\vec{x}$ in P by defining their existence under the constraint c. The operators associated with time allow the process to go one time unit in the future ($\mathbf{next}\,(P)$) or to define time-outs: if at the current time unit it is not possible to

entail the constraint c then the process $\mathbf{unless}\,\mathsf{c}\,\mathbf{next}\,P$ will execute P at the next time unit. We will often use $\mathbf{next}^n\,(P)$ as a shorter version of $\mathbf{next}\,(\mathbf{next}\,(\ldots\mathbf{next}\,(P)))$ n-times. Finally, $P \parallel Q$ denotes the usual parallel execution and $!P$ denotes timed replication; that is, $!P = P \parallel \mathbf{next}\,(!P)$ executes P at the current time and replicates its behaviour over the next time period.

utcc [14] is an extension of the tcc calculus with a general *ask* defining a model of synchronization. While in tcc an ask $\mathbf{when}\,\mathsf{c}\,\mathbf{do}\,P$ is blocked if there is not enough information to entail c from the store, utcc inspires its synchronization mechanism on the notion of abstraction in functional programming languages. $(\lambda\,\vec{x};\mathsf{c})\;P$ can be seen as the dual version of $(\mathbf{local}\,\vec{x};\mathsf{c})\;P$ in which the variables are *abstracted* with respect to the constraint c and the process P. The operational semantics provides the intuitions on how utcc processes interact. In principle, a configuration is represented by the tuple $\langle P, \mathsf{c}\rangle$ where P denotes a set of processes and c a constraint store. P can evolve to a further process P' during an *internal transition* ($\rightarrow$) where the constraint store c is monotonically refined, or can execute an *observable transition* ($\Longrightarrow$), producing the result of the future function of P and the constraint store d. The set of operational rules is presented in Figure 1, where $\langle P, \mathsf{c}\rangle$ denotes a configuration, and $F(P)$ denotes the *future function of* P.

Definition 3 (Structural Congruence). *Structural congruence (denoted by $\equiv$) is defined for* utcc *by the axioms: (i) $P \equiv Q$ if they are α-equivalent. (ii) $P \parallel \mathbf{skip} \equiv P$. (iii) $P \parallel Q \equiv Q \parallel P$. (iv) $P \parallel (Q \parallel R) \equiv (P \parallel Q) \parallel R$. (v) $(\mathbf{local}\,\vec{x};\mathsf{c})\;\mathbf{skip} \equiv \mathbf{skip}$. (vi) $P \parallel (\mathbf{local}\,\vec{x};\mathsf{c})\;Q \equiv (\mathbf{local}\,\vec{x};\mathsf{c})\;(P \parallel Q)$ if $\vec{x} \notin fv(P)$. (vii) $\langle P, c\rangle \equiv \langle Q, c\rangle$ iff $P \equiv Q$.*

Intuitively, the operational rules of utcc behaves almost in the same way as its counterpart in tcc, excepting by the general treatment of asks in utcc. Here we will describe the operational consequence of this change, we refer to [14] for further details on the operational semantics. Rule $\mathsf{R_A}$ describes the behavior of the abstraction $(\lambda\,\vec{x};\mathsf{c})\;P$: a configuration here considers two stores, being c and d *local* and *global* stores respectively. If d entails $\mathsf{c}[\tilde{\mathsf{t}}/\tilde{\mathsf{x}}]$ then $P[\vec{t}/\vec{x}]$ is executed. Moreover, the abstraction persists in time, allowing any other process to match with $\vec{x}$ in P while no other replacements of $\vec{x}$ with $\vec{t}$ will occur, as d is augmented with a constraint disallowing this. The notion of *local information* can be evidenced in $\mathsf{R_L}$, considering a process $P = (\mathbf{local}\,\vec{x};\mathsf{c})\;Q$, we have to consider: (i) that the information about $\vec{x}$ locally for P subsumes any other information present for the same set of variables in the global store; therefore, $\vec{x}$ is hidden by the use of an existential quantifier over $\tilde{\mathsf{x}}$ in d. (ii) that the information about $\vec{x}$ that P can produce after the reduction is still local, so we hide it by existentially quantifying $\vec{x}$ in c' before publishing it to the global store. After the reduction, c' will be the new local store of the evolution of internal processes. Finally, observable behaviour is described by $\mathsf{R_o}$: after having used the internal transitions in a process P to evolve to a process Q with a quiescent-point (in which no more information can be added/inferred), the reduction will continue by executing the future function of Q with the resulting constraint store.

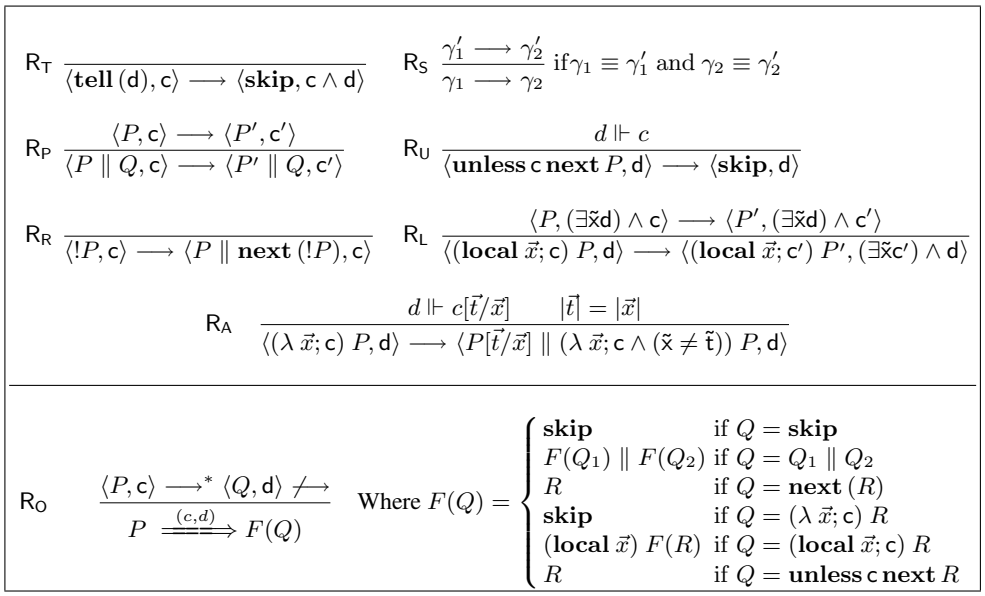

Fig. 1. Transition System for `utcc`: Internal and Observable transitions

3 `utcc` and Secure Pattern Matching

As described in Sec. 2, one of the main advantages of `utcc` with respect to `tcc` is that the universal abstraction operator allows for substitution of constraints for variables in processes. The extension has been proposed for the treatment of *mobile links* as present in the π -calculus [12] and pattern matching in modeling of security protocols. Below we will give two motivating examples for why a more refined abstraction operator is needed for modeling mobile *local* links and secret keys.

3.1 Motivating a refined universal abstraction in `utcc`

Our first example refers to the π calculus-like mobility of local links. Consider the common scenario where a process P sends a request to a service offered by a process Q and includes in the request a local link on which it expects the reply. This can be modeled in `utcc` using a constraint system $CS = (\Sigma, \Delta)$ where Σ includes the predicates req, rep, and res, and the constant 0. The processes P and Q are defined as

$$P = (\mathbf{local}\ z)\ \big(\mathbf{tell}\,(\mathsf{req(z)})\ \parallel (\lambda\, y; \mathsf{rep(z,y)})\ \mathbf{next}\ (\mathbf{tell}\,(\mathsf{res(y)}))\big)$$

and

$$Q = (\lambda\, x; \mathsf{req(x)})\ \mathbf{tell}\,(\mathsf{rep(x,0)})$$

The predicates req and rep are used for the request and reply respectively, and the predicate res is used to report the result (and successful termination of P). The local operator is used to create a local variable z representing the local link.

The intention is that only the processes P and Q can synchronize via the local link z. However, the generality of abstraction in utcc makes it possible to violate this intention: Another process $E = (\lambda\, x, y; \mathsf{rep}(\mathsf{x}, \mathsf{y}))\ \mathbf{skip}$ in parallel with the processes P and Q given above would be able to guess the link z (as well as the result) from the reply.

It is instructive to see how this could be avoided using the π-calculus, where the two processes could be modeled by

$$P = (\nu z)\big(\overline{\mathsf{req}}\langle z\rangle \parallel z\langle y\rangle.\overline{\mathsf{res}}\langle y\rangle\big) \quad \text{and} \quad Q = \mathsf{req}(x).\overline{x}\langle 0\rangle$$

In this case, the z and y are used differently in receiving the reply: The z is used as the communication channel and y is the binder for the received name. Another process in parallel would not be able to guess the channel z. As we will see below, our proposed type system for patterns allows to introduce this kind of distinction between the uses of variables in predicates.

Our second motivating example is from modeling of security protocols, where as pointed out in [4] it it should be impossible for an agent to abstract variables if a one-way function has been applied to it. Consider a unary predicate o (used for output of messages to the network) and an encryption function $\mathsf{enc}(\mathsf{m}, \mathsf{k})$ which represents the encryption of the variable m with the key k. A process P that sends out a local message n encrypted by a local key k can be represented by $P = (\mathbf{local}\ k, n)\ \mathbf{tell}\,(\mathsf{o}(\mathsf{enc}(\mathsf{n}, \mathsf{k})))$. However, in utcc a spy process defined as $S = (\lambda\, x, z; \mathsf{o}(\mathsf{enc}(\mathsf{x}, \mathsf{z})))\ \mathbf{tell}\,(\mathsf{o}(\mathsf{x}) \wedge \mathsf{o}(\mathsf{z}))$, will succeed in retrieving and publishing both the key and the encrypted message.

As for the π-like channels, our proposed type system for patterns will allow us to rule out universal abstraction of variables to which a one-way function has been applied. Further, to be able to allow abstraction of the message when the key is locally known, we propose a novel kind of abstraction assuming local knowledge, which generalizes the universal abstraction of utcc.

3.2 Types for secure abstraction patterns in utcc

Based on the two motivating examples above, we argue that there are basically two sorts of arguments in functions and predicates: the ones that can be universally quantifiable, which means that one would be able to use the abstraction operator for a variable in that argument in order to find a possible matching, and the ones that are not.

We will thus divide the arguments of predicates and functions in two sorts and write $\mathsf{P}(\tilde{\mathsf{t}}; \tilde{\mathsf{t}}')$ and $\mathsf{f}(\tilde{\mathsf{t}}; \tilde{\mathsf{t}}')$ for respectively the predicate P and function f where both $\vec{t}$ and $\vec{t}'$ are tuples of terms over the function signature F, and $\vec{t}$ denotes the restricted arguments and $\vec{t}'$ the unrestricted ones. We assume that both arguments of the equality predicate are restricted. If a predicate or function has either only restricted or unrestricted parameters and the sort is clear from the context, we will simply write $\mathsf{P}(\tilde{\mathsf{t}})$ and $\mathsf{f}(\tilde{\mathsf{t}})$.

The sorted predicates allow us to use a binary predicate $\mathsf{piout}(\mathsf{x}; \mathsf{y})$ representing the π-like communication of y (the object) on the channel x (the subject). By defining that

the subject is a restricted argument and the object an unrestricted argument we obtain the required asymmetry in the roles of the variables. The type rules for patterns should then forbid the abstraction $(\lambda\ x; \mathsf{piout}(\mathsf{x}, \mathsf{y}))\ P$, as it would allow us to identify all channels (also channels not known to us) containing a particular message y. However, they should *allow* the abstraction $(\lambda\ y; \mathsf{piout}(\mathsf{x}, \mathsf{y}))\ P$, reflecting that we can compute the possible messages on a channel x known to us. That is, we want to capture that if we *know* the values of the restricted variables, then we may abstract (i.e. compute all possible matches for) the unrestricted variables.

Similarly, sorted functions allow us to represent semantically that some functions are *one-way* functions such as the function $\mathsf{enc}(\mathsf{k}, \mathsf{m})$ described above for encrypting the message m by the key k. Sorting both arguments as restricted will ensure that e.g. the abstractions $(\lambda\, \vec{x}; \mathsf{o}(\mathsf{enc}(\mathsf{k}, \mathsf{m})))\ P$ will be forbidden for any non-empty $\vec{x} \subseteq \{\mathsf{k}, \mathsf{m}\}$ Thus, even if the single argument of the o predicate is unrestricted (i.e. we can abstract all messages available on the network) then we can not compute the inverse of the encryption function. We may have functions for which an inverse is assumed to exist, such as a function $\mathsf{tup}_2(\mathsf{x}, \mathsf{y})$ for making a pair of x and y. In that case it makes sense to allow abstractions over the two arguments by sorting them as unrestricted.

In general, patterns may be a conjunction of several predicates and thus variables may occur both restricted and unrestricted in the same pattern. An example of this is the abstraction $(\lambda\, y, z; \mathsf{c})\ P$, where $\mathsf{c} = \mathsf{piout}(\mathsf{y}, \mathsf{z}) \wedge \mathsf{piout}(\mathsf{x}, \mathsf{y})$. We argue that this pattern should be *allowed*, since it is possible first to match the unrestricted y in $\mathsf{piout}(\mathsf{x}, \mathsf{y})$ and then subsequently, for the given y, match the unrestricted z in $\mathsf{piout}(\mathsf{y}, \mathsf{z})$. Note that it is not enough simply to require the abstracted variables to occur unrestricted: Both variables x and y appear unrestricted in the abstraction $(\lambda\, x, y; \mathsf{piout}(\mathsf{x}, \mathsf{y}) \wedge \mathsf{piout}(\mathsf{y}, \mathsf{x}))\ P$, but neither of the two basic constraints can be matched without abstracting a restricted variable. As solution we define a set of type rules for constraints used as patterns in abstractions which capture that there exists an order of the basic constraints in which the first occurrence of each variable is unrestricted.

To allow abstractions in cases where the inverse key of the encryption is known we add a new rule $\mathsf{R}_{\mathsf{A}\rightarrow}$ given in Equation 1 in addition to the SOS rules pictured in Figure 1. $\mathsf{R}_{\mathsf{A}\rightarrow}$ allows for abstractions using constraints of the form $\mathsf{c} \Rightarrow \mathsf{c}'$, that is, assuming local knowledge c and a global store d, one can infer c'. The idea is to infer c' using c but without publishing it permanently to the store, as captured by the following operational rule:

$$\mathsf{R}_{\mathsf{A}\rightarrow}\quad \frac{\mathsf{d} \wedge \mathsf{c} \Vdash \mathsf{c}'[\tilde{\mathsf{t}}/\tilde{\mathsf{x}}] \quad |\tilde{\mathsf{t}}| = |\tilde{\mathsf{x}}| \quad \mathsf{d} \wedge \mathsf{c} \Vdash \mathtt{false} \Rightarrow \mathsf{d} \Vdash \mathtt{false}}{\langle (\lambda\, \vec{x}; \mathsf{c} \Rightarrow \mathsf{c}')\ P, \mathsf{d}\rangle \longrightarrow \langle P[\vec{t}/\vec{x}] \parallel (\lambda\, \vec{x}; \mathsf{c} \Rightarrow (\mathsf{c}' \wedge (\tilde{\mathsf{x}} \neq \tilde{\mathsf{t}}))\ P, \mathsf{d}\rangle} \tag{1}$$

The condition $\mathsf{d} \wedge \mathsf{c} \Vdash \mathtt{false} \Rightarrow \mathsf{d} \Vdash \mathtt{false}$ ensures that local assumptions do not make the store inconsistent when combining with the constraint store.

The typing rules for secure patterns and processes are defined in Figure 2. For simplicity we assume patterns are simply conjunction of predicates applied to terms over the function signature. The typing rules use an environment $\Gamma = \Gamma^R; \Gamma^U$, where Γ^R is the set of names used restricted and Γ^U is the set of names used unrestricted. When the distinction does not matter we simply write Γ. We employ three inductively defined

$$\mathsf{T}_{\mathsf{pred}} \quad \frac{}{\Gamma^R;\Gamma^U \vdash \mathsf{P}(\vec{t};\vec{t'}) : pat}\ \Gamma^R = var(\vec{t}) \cup res(\vec{t'}) \text{ and } \Gamma^U = unr(\vec{t'})\backslash\Gamma^R$$

$$\mathsf{T}_{\mathsf{assoc}} \ \frac{\Gamma \vdash \mathsf{c}_1 \wedge (\mathsf{c}_2 \wedge \mathsf{c}_3) : pat}{\Gamma \vdash (\mathsf{c}_1 \wedge \mathsf{c}_2) \wedge \mathsf{c}_3 : pat} \qquad \mathsf{T}_{\mathsf{commute}} \ \frac{\Gamma \vdash \mathsf{c}_1 \wedge \mathsf{c}_2 : pat}{\Gamma \vdash \mathsf{c}_2 \wedge \mathsf{c}_1 : pat}$$

$$\mathsf{T}_{\mathsf{comb}} \quad \frac{\Gamma_1^R;\Gamma_1^U \vdash \mathsf{c}_1 : pat \quad \Gamma_2^R;\Gamma_2^U \vdash \mathsf{c}_2 : pat}{\Gamma^R;\Gamma^U \vdash \mathsf{c}_1 \wedge \mathsf{c}_2 : pat}\ \Gamma^R = (\Gamma_1^R \cup \Gamma_2^R)\backslash\Gamma_1^U \text{ and } \Gamma^U = (\Gamma_1^U \cup \Gamma_2^U)\backslash\Gamma_1^R$$

$$\mathsf{T}_{\mathsf{skip}} \ \frac{}{\vdash \mathbf{skip} : sec} \qquad \mathsf{T}_{\mathsf{tell}} \ \frac{}{\vdash \mathbf{tell}\,(\mathsf{c}) : sec}$$

$$\mathsf{T}_{\mathsf{par}} \ \frac{\vdash P : sec \quad \vdash Q : sec}{\vdash P \parallel Q : sec} \qquad \mathsf{T}_{\mathsf{next}} \ \frac{\vdash P : sec}{\vdash \mathbf{next}\,(P) : sec}$$

$$\mathsf{T}_{\mathsf{bang}} \ \frac{\vdash P : sec}{\vdash !P : sec} \qquad \mathsf{T}_{\mathsf{unls}} \ \frac{\vdash P : sec}{\vdash \mathbf{unless}\,\mathsf{c}\,\mathbf{next}\,P : sec}$$

$$\mathsf{T}_{\mathsf{abs}} \ \frac{\vdash P : sec \quad \Gamma^R;\Gamma^U \vdash \mathsf{c} : pat}{\vdash (\lambda\,\vec{x};\mathsf{d} \implies \mathsf{c})\ P : sec}\ \vec{x} \subseteq dom(\Gamma^U)\backslash fv(d) \qquad \mathsf{T}_{\mathsf{loc}} \ \frac{\vdash P : sec}{\vdash (\mathbf{local}\,\vec{x};\mathsf{c})\ P : sec}$$

Fig. 2. Typing rules for secure patterns and processes

functions on terms over the function signature: $unr(t)$, $res(t)$, and $var(t)$ yielding respectively the variables appearing unrestricted in t according to the sorting, the variables appearing restricted in t, and all variables appearing in t. We extend the functions to vectors of terms by $unr(\vec{t}) = \cup_{1 \le i \le |\vec{t}|} unr(t_i)$ (and similarly for res and var). Formally, the functions are given by $unr(x) = res(x) = var(x) = \{x\}$ for any variable x, and $unr(f(\vec{t};\vec{t'})) = unr(\vec{t'})$, $res(f(\vec{t};\vec{t'})) = res(\vec{t})$, and $var(f(\vec{t};\vec{t'})) = var(\vec{t}) \cup var(\vec{t'})$. Note that obviously $var(t) = res(t) \cup unr(t)$ but also that $res(t) \cap unr(t)$ may be non-empty, i.e. a variable may appear both restricted and non-restricted.

The rule $\mathsf{T}_{\mathsf{Pred}}$ captures that all variables in $\vec{t}$ as well as the variables occurring restricted in $\vec{t'}$ in the predicate $\mathsf{P}(\vec{t};\vec{t'})$ are restricted. The rest of the variables are unrestricted. The rules $\mathsf{T}_{\mathsf{asoc}}$ and $\mathsf{T}_{\mathsf{commute}}$ allow us to change the ordering of the basic constraints. Finally, the rule $\mathsf{T}_{\mathsf{comb}}$ identifies the restricted and unrestricted variables in the joint pattern $\mathsf{c}_1 \wedge \mathsf{c}_2$ assuming that c_1 is matched first. That is, a variable is restricted if it appears restricted in either of the sub patterns c_1 and c_2 and not unrestricted in c_1. (If it appears unrestricted in c_1 it will be instantiated if c_1 is matched first, and thus it is allowed to appear restricted in c_2). Dually, the unrestricted variables in the joint pattern $\mathsf{c}_1 \wedge \mathsf{c}_2$ are the variables that appear unrestricted in either of the sub patterns c_1 and c_2, and do not appear restricted in c_1.

The objective of the type system is to determine the secure patterns, therefore typing rules over processes are rather simple. The only non-trivial rule is the rule $\mathsf{T}_{\mathsf{abs}}$ for

abstractions, which ensure that c is a valid pattern such that the abstracted variables are unrestricted, and no variables in the local d are abstracted.

Theorem 1 (Termination of type checking). *For any process P the type-checking process terminates.*

Proof. (Sketch) Follows from the fact that there are only finitely many permutations of basic constraints (predicates) in a pattern.

The following lemmas are used to prove subject reduction.

Lemma 1 (Constraint substitution does not affect pattern typing). *Given $\Gamma^R; \Gamma^U \vdash \mathsf{c} : pat$ and t and x, then $\Gamma^{R'}; \Gamma^{U'} \vdash \mathsf{c}[t/x] : pat$ and $\Gamma^U \setminus (fv(t) \cup \{x\}) \subseteq \Gamma^{U'} \setminus (fv(t) \cup \{x\})$.*

Proof. (Outline) The proof proceeds by induction on the type inference of $\Gamma^R; \Gamma^U \vdash \mathsf{c} : pat$.

Lemma 2 (Constraint substitution does not affect process typing). *Given a typing judgment $\vdash P' : sec$ then $\vdash P'[t/x] : sec$.*

Proof. (Outline) The proof proceeds by induction on the type inference of $\vdash P' : sec$

Lemma 3 (Structural equivalence preserves typing). *Given P, Q processes, if $P \equiv Q$ and $\vdash P : sec$, then $\vdash Q : sec$.*

Proof. The proof proceeds by trivial case analysis over the structural congruence rules in Definition 3.

Next we check that secure processes can not be made insecure during an internal transition step.

Lemma 4. *If $\langle P, \mathsf{c}\rangle \longrightarrow \langle Q, \mathsf{d}\rangle$ and $\vdash P : sec$, then $\vdash Q : sec$.*

Proof. (Outline) The proof proceeds by induction on the depth of the inference $\langle P, \mathsf{c}\rangle \longrightarrow \langle Q, \mathsf{d}\rangle$ and using the definition of $\vdash P : sec$.

Finally, we show that if a process P is well-typed, it can not perform any internal steps, and its future is defined then the future of P is also well-typed.

Lemma 5. *For all $\vdash P : sec$, if $F(P)$ is defined and $\exists \mathsf{d}.\langle P, \mathsf{d}\rangle \not\longrightarrow$ then $\vdash F(P) : sec$.*

Proof. (Outline) The proof proceed by induction in the definition of $F(P)$.

We now have all the ingredients to prove subject reduction.

Theorem 2 (Subject-reduction). *If $P \stackrel{(c,d)}{\Longrightarrow} Q$ and $\vdash P : sec$, then $\vdash Q : sec$.*

Proof. Assume $P \stackrel{(c,d)}{\Longrightarrow} Q$ and $\vdash P : sec$, then by rule $\mathsf{R_o}$ we get that $\langle P, \mathsf{c}\rangle \longrightarrow^n \langle Q', \mathsf{d}\rangle \not\longrightarrow$ and $Q = F(Q')$. We proceed by induction in n.

In the base case where $n = 0$, we have that $Q' = P$ and $\mathsf{c} = \mathsf{d}$. It follows from lemma 5 that $\vdash F(Q') : sec$.

For the induction step, assume $\langle P, \mathsf{c}\rangle \longrightarrow^1 \langle P', \mathsf{c}'\rangle \longrightarrow^n \langle Q', \mathsf{d}\rangle \not\longrightarrow$. Then $\vdash P' : sec$ by lemma 4 and thus we get by induction that $\vdash F(Q') : sec$.

4 Applications

This section illustrates the use of the type system with some examples in mobility and security. First, let us return to the π calculus example.

We assume the syntactic sugar $x\langle y\rangle$ stands for the binary predicate $\mathsf{piout}(\mathsf{x};\mathsf{y})$ and represents the use of the (restricted) channel x with the (unrestricted) message y. The following type inference show that we can quantify over either x or y for the pattern $y\langle x\rangle \wedge x\langle y\rangle$:

$$\frac{\dfrac{}{x; y \vdash x\langle y\rangle : pat}\,\mathsf{T}_{\mathsf{pred}} \quad \dfrac{}{y; x \vdash y\langle x\rangle : pat}\,\mathsf{T}_{\mathsf{pred}}}{x; y \vdash x\langle y\rangle \wedge y\langle x\rangle : pat}\,\mathsf{T}_{\mathsf{comb}}$$

The way to read the first inference is that we can abstract y if we know x. Conversely, a second inference from the same pattern can lead to a typing of the form $y; x \vdash y\langle x\rangle \wedge x\langle y\rangle : pat$, capturing the fact that one can abstract x if we know y. However, note that we can not infer $\epsilon; x, y \vdash x\langle y\rangle \wedge y\langle x\rangle : pat$, and thus we are not allowed to simultaneously quantify over x and y.

To illustrate the application of $\mathtt{utcc}_s$ in the security domain, we follow the lines of the Security Protocol Language (SPL) [6] and SCCP [13] to define a specification language for security protocols that we have called the Security Protocol Concurrent Constraint Programming (SPCCP) language. The SPCCP embeds $\mathtt{utcc}_s$ in a syntax suitable for defining security protocols, capturing process specifications with respect to input and output events over a global network. The SPCCP language combines the best ideas from SPL and SCCP by having a simple notion of pattern matching as in SPL and using the constraint system to model the attackers ability to combine and split messages as in SCCP. Hereto we add the new concept of *pattern matching under local knowledge*, which allow us to syntactically guarantee that only message parts inferable from the available keys are extracted, which can not be guaranteed in SPL nor in SCCP.

Definition 4 **(SPCCP).** *The Secure Concurrent Constraint Programming language* SCCP *[13] is redefined by the following grammar:*

$$\begin{array}{lrl}
\textit{Values} & \textit{v,v'} = & x \mid k \\
\textit{Keys} & k = & pub(x) \mid priv(x) \mid sym(x) \\
\textit{Messages and patterns} & \textit{M,N} = & v \mid (M_1, \ldots, M_n) \mid \{M\}_k \\
\textit{Processes} & R = & \mathbf{nil} \mid \mathbf{local}(x)\,\mathbf{in}\,R \mid \mathbf{out}(M)\,.R \\
 & & \mid \mathbf{in}_\forall \vec{x}[N]_{\vec{k}}.R \mid\, !R \mid R \parallel R
\end{array} ,$$

where x range over a set of variables and the subscript $\vec{k}$ in $\mathbf{in}_\forall \vec{x}[N]_{\vec{k}}.R$ is a set of keys.

We define the semantics of SPCCP by giving a translation into $\mathtt{utcc}_s$ with a security constraint system given by the signature Σ with a single (unrestricted) unary predicate $\mathsf{o}(\mathsf{t})$ used for message output, and function symbols $\mathsf{F} = \{\mathsf{enc}, \mathsf{pub}, \mathsf{priv}, \mathsf{sym}, \mathsf{tup_n}\}$, and entailment relation given in Fig. 3 inspired on the requirements stated by Dolev and Yao in [7].

$$\mathsf{E}_{\mathsf{k-dec}}\ \frac{\mathsf{c} \Vdash \mathsf{o}(\mathsf{k}^{-1}(\mathsf{x}))\quad \mathsf{c} \Vdash \mathsf{o}(\mathsf{enc}(\mathsf{k}(\mathsf{x}),\mathsf{m}))}{\mathsf{c} \Vdash \mathsf{o}(\mathsf{m})}\text{, for } \mathsf{k} \in \{sym, pub\},\ sym^{-1} = sym,\ \text{and } pub^{-1} = priv$$

$$\mathsf{E}_{\mathsf{enc}}\ \frac{\mathsf{c} \Vdash \mathsf{o}(\mathsf{x}) \qquad \mathsf{c} \Vdash \mathsf{o}(\mathsf{y})}{\mathsf{c} \Vdash \mathsf{o}(\mathsf{enc}(\mathsf{x},\mathsf{y}))} \qquad \mathsf{E}_{\mathsf{k-key}}\ \frac{\mathsf{c} \Vdash \mathsf{o}(\mathsf{x})}{\mathsf{c} \Vdash \mathsf{o}(\mathsf{k}(\mathsf{x}))}\text{, for } \mathsf{k} \in \{sym, pub, priv\}$$

$$\mathsf{E}_{\mathsf{tup_n}}\ \frac{\mathsf{c} \Vdash \mathsf{o}(\mathsf{i}_1) \quad \ldots \quad \mathsf{c} \Vdash \mathsf{o}(\mathsf{i}_\mathsf{n})}{\mathsf{c} \Vdash \mathsf{o}(\mathsf{tup_n}(\mathsf{i}_1,\ldots,\mathsf{i}_\mathsf{n}))} \qquad \mathsf{E}_{\mathsf{proj}}\ \frac{\mathsf{c} \Vdash \mathsf{o}(\mathsf{tup_n}(\mathsf{i}_1,\ldots,\mathsf{i}_\mathsf{n}))}{\mathsf{c} \Vdash \mathsf{o}(\mathsf{i_j})}\quad j \in \{1,\ldots,n\}$$

Fig. 3. Entailment relation for a security constraint system.

The binary function enc takes two unrestricted arguments: a key and a message. The key is intended to be either a symmetric, private, or public key generated by the (restricted) unary functions $sym(x)$, $priv(x)$, or $pub(x)$ respectively. Letting $\mathsf{k} \in \{pub, priv, sym\}$ and defining $sym^{-1} = sym$, and $pub^{-1} = priv$, the entailment rule scheme $\mathsf{E}_{\mathsf{k-dec}}$ for decryption expresses how enc acts as symmetric or asymmetric encryption. The n-ary (unrestricted) tupling functions tup_n allow to create n-ary tuples, from which the individual elements can be projected as expressed by the entailment rule $\mathsf{E}_{\mathsf{proj}}$. As usual, the rules $\mathsf{E}_{\mathsf{enc}}$, $\mathsf{E}_{\mathsf{k-key}}$, and $\mathsf{E}_{\mathsf{tup_n}}$ express that the output of any function of known output values can be inferred.

The messages/patterns of SPCCP are mapped to the terms generated by the corresponding function symbols and variables in the security constraint system, using the usual notation $(M_1,\ldots,M_n)$ for n-tuples and $\{M\}_k$ for $\mathsf{enc}(\mathsf{k}, M)$. For a message M of SPCCP let $v(M)$ denote the set of variables in M. For a set of values $\vec{v} = \{v_1, v_2, \ldots, v_i\}$ let $o(\vec{v})$ be short for $o(v_1) \wedge o(v_2) \wedge \ldots \wedge o(v_i)$, and in particular $o(\emptyset) = \mathtt{true}$.

We are now ready to define the encoding of SPCCP in $\mathtt{utcc}_s$.

Definition 5 (SPCCP **encoding).**

$$[\![R]\!] : \begin{cases} \mathbf{skip} & \text{if } R = \mathbf{nil} \\ (\mathbf{local}\, x)\ [\![R']\!]_{\mathtt{utcc}} & \text{if } R = \mathbf{local}(x)\,\mathbf{in}\,R' \\ \mathbf{tell}\,(\mathsf{o}(\mathsf{M}))\ \parallel\ \mathbf{next}\,([\![R']\!]_{\mathtt{utcc}}) & \text{if } R = \mathbf{out}(M)\,.R' \\ (\lambda\,\vec{x}; \mathsf{o}(\tilde{\mathsf{k}}) \Rightarrow \mathsf{o}(\mathsf{N}) \wedge \mathsf{o}(\tilde{\mathsf{x}}))\ \mathbf{next}\,([\![R']\!]_{\mathtt{utcc}}) & \text{if } R = \mathbf{in}_\forall \vec{x}[N]_{\vec{k}}.R' \\ !\,[\![R']\!]_{\mathtt{utcc}} & \text{if } R = !R' \\ [\![R']\!]_{\mathtt{utcc}} \parallel [\![R'']\!]_{\mathtt{utcc}} & \text{if } R = R' \parallel R'' \end{cases}$$

We will focus on outlining process constructions for pattern matching and network output. The remaining process constructions are mapped directly to the corresponding construct in $\mathtt{utcc}_s$.

$\mathbf{out}(M).\,R$ adds the constraint $\mathsf{o}(\mathsf{M})$ to the constraint store and subsequently in the next time period behaves as (the encoding of) R.

SPCCP differs from SCCP in the treatment of keys and the input operation: $\mathsf{priv}(\mathsf{x})$, $\mathsf{pub}(\mathsf{x})$, and $\mathsf{sym}(\mathsf{x})$ yields respectively the private, public and symmetric key from generator x. The input operator written as $\mathbf{in}_\forall \vec{x}[N]_{\vec{k}}.P$ should be read as "for all possible

messages $\vec{m}$ (available under the assumption of knowing the keys $\vec{k}$) such that $N[\vec{m}/\vec{x}]$ is available as message at the network evolve into $P[\vec{m}/\vec{x}]$". Intuitively, the idea is to check if $\vec{m}$ is available as knowledge assuming locally that the keys in $\vec{k}$ are available as knowledge, and if so, bind the variables in P occurring in the pattern N with the corresponding values in $\vec{m}$. The pattern matching resembles the pattern matching construct in SPL. The key difference is that it proceed for all possible matches, and that we employ the new rule for for universal abstraction under local knowledge introduced in the previous section to allow the use of private keys as local information to perform the decryption of messages. Note that we also require that all the abstracted values can be inferred as output. This guarantees that secret values are not abstracted, and result in well-typedness of the encoding.

Proposition 1 (SPCCP **maps to well-typed** $\texttt{utcc}_s$ **processes).** *For any* SPCCP *process* P, $\vdash [\![P]\!] : sec$.

4.1 Protocols

In Fig. 4 below we recall the protocol steps of the Needham-Schröeder-Lowe protocol [10] (herewith referred as NSL) used as example in [6].

$$
\begin{array}{l}
(1)\ A \rightarrow B\ :\ \{m, A\}_{pub(B)} \\
(2)\ B \rightarrow A\ :\ \{m, n, B\}_{pub(A)} \\
(3)\ A \rightarrow B\ :\ \{n\}_{pub(B)}
\end{array}
$$

Fig. 4. Needham-Schröeder-Lowe protocol with public-key encryption

The NSL protocol describes the interaction between agents A and B. First A sends to B a nonce along its agent name, encrypted with B's public key. Then B decrypts the message with his own private key extracting A's nonce. Next, B sends a message to A containing the proof of reception along with a fresh name encrypted under A's public key. Finally, A decrypts B's message and sends to B the name challenge received in the previous message encrypted with B's public key. The SPCCP version of the protocol is given in Fig. 5.

SPCCP share some similarities with the approaches in LYSA^{NS} [4], SCCP , and the SPL calculus. Particularly, observe that there is no need to explicitly define the communication channels in which agents are transmitting messages. The underlying model acts as an open network in which every actor can access all the messages posted provided that he has the proper keys to decrypt its the message. We assume a disclosure of public keys for every agent, while the private keys are kept secret for each principal. The key difference between the approach in SPCCP to the approaches in SPL and SCCP is that the abstraction of the contents of a message encrypted with a key is only allowed if one possesses the corresponding key for decryption. This is similar to the approach in the LYSA^{NS} calculus [4], except that we employ the constraint system and local knowledge instead of tailoring the pattern matching with a notion of key pairs.

$$
\begin{aligned}
Init(A,B,k_A,p_B) \;&= \mathbf{new}(m)\,\mathbf{out}(\{m,A\}_{p_B}).\\
&\quad\ \mathbf{in}_\forall x[\{m,x,B\}_{pub(k_A)}]_{priv(k_A)}.\\
&\quad\ \mathbf{out}(\{x\}_{p_B}).\,\mathbf{nil}\\[2ex]
Resp(A,B,k_B,p_A) \;&= \mathbf{in}_\forall y[\{y,A\}_{pub(k_B)}]_{priv(k_B)}.\\
&\quad\ \mathbf{new}(n)\,\mathbf{out}(\{y,n,B\}_{p_A}).\\
&\quad\ \mathbf{in}_\forall[\{n\}_{pub(k_B)}]_{priv(k_B)}.\mathbf{nil}\\[2ex]
System(A,B) \;&= \mathbf{new}(k_A)\,\mathbf{new}(k_B)\,(Init(A,B,k_A,pub(k_B))\\
&\quad\ \parallel Resp(A,B,k_B,pub(k_A)))
\end{aligned}
$$

Fig. 5. NSL protocol in SPCCP

The following specification exemplifies the translation into $\mathtt{utcc}_s$:

$$
\begin{aligned}
Init(A,B,k_A,p_B) \;&= (\mathbf{local}\ m)\ \mathbf{tell}\,(\mathsf{o}(\{\langle \mathsf{m},\mathsf{A}\rangle\}_{\mathsf{p_B}})\\
&\quad\ \parallel \mathbf{next}\,(((\lambda\, x;\mathsf{o}(\mathsf{priv}(\mathsf{k_A}))\Rightarrow(\mathsf{o}(\{\mathsf{m},\mathsf{x},\mathsf{B}\}_{\mathsf{pub(k_A)}})\wedge\mathsf{o}(\mathsf{x})))\ )\\
&\quad\ \parallel\mathbf{next}\,(\mathbf{tell}\,(\mathsf{o}(\{\mathsf{x}\}_{\mathsf{pub_B}}))\parallel\mathbf{next}\,(\mathbf{skip})))\\[2ex]
Resp(A,B,k_B,p_A) \;&= (\lambda\, y;\mathsf{o}(\mathsf{priv}(\mathsf{k_B}))\Rightarrow(\mathsf{o}(\{\mathsf{y},\mathsf{A}\}_{\mathsf{pub(k_B)}})\wedge\mathsf{o}(\mathsf{y})))\\
&\quad\ \parallel \mathbf{next}\,(((\mathbf{local}\ n)\ \mathbf{tell}\,(\mathsf{o}(\{\mathsf{y},\mathsf{n},\mathsf{B}\}_{\mathsf{pA}})))\\
&\quad\ \parallel\mathbf{next}\,((\lambda\,\emptyset;\mathsf{o}(\mathsf{priv}(\mathsf{k_B}))\Rightarrow(\mathsf{o}(\{\mathsf{n}\}_{\mathsf{pub(k_B)}})))\ \parallel\mathbf{next}\,(\mathbf{skip})))\\[2ex]
System(A,B) \;&= (\mathbf{local}\ k_A)\ (\mathbf{local}\ k_B)\ Init(A,B,k_A,pub(k_B))\\
&\quad\ \parallel Resp(A,B,k_B,pub(k_A))
\end{aligned}
$$

5 Conclusions and Future Work

We have illustrated that the introduction of universal quantification to CCP for modeling mobile communication and security protocols introduce the problem that information which should be local can be obtained by universal quantification. As a way to remedy the problems we have proposed a simple type system for constraints used as patterns in abstractions which allows us to guarantee semantically that e.g. channel names and encrypted values are only extracted by agents that are able to infer the channel or non-encrypted value from the store. Furthermore, we proposed a novel kind of abstraction allowing abstraction under the assumption of local knowledge. The latter can be applied to infer the plain text of encrypted messages under the assumption of knowledge of the key, without adding the key permanently to the global store. We exemplified the type system by examples of mobility of local links (in the context of the π -calculus) and provided a new language for security protocols combining the key features of the Security CCP (SCCP) language and the SPL calculus, but adding the ability to syntactically constraining the ability to decrypting secret values inspired by the LYSA^{NS} calculus.

The present work is only in its first stage. However, we believe that the proposed distinction between variables that can be universally quantified and variables that can not is an elegant way to remedy the problems we have illustrated connected to the universal quantification to CCP. A next step will be to perform a detailed investigation

of the proposed new variant of the SCCP calculus and applications to model security protocols. In particular, we plan to investigate the application of the analysis techniques for SCCP , SPL and LYSA^{NS} to the SPCCP language.

It is important to remark the importance of the current proposal with respect to other analysis techniques for security protocols. In [3], a framework for the analysis of secrecy properties is proposed with logic programming as its underlying mechanism. The specification language follows the line of the equational theory presented in the Applied π -calculus [1], encoding constructor and destructor functions by means of deduction rules in the framework. Here, pattern-matching is being used to encode the abilities of an attacker to abstract away information from the facts present in the store. Given that the attacker can apply the set of rules in a given specification, the correctness of the analysis relies on the power we give on the inference system. For instance, a rule $\mathsf{attaker}(\mathsf{sign}(\mathsf{m},\mathsf{sk})) \rightarrow \mathsf{attacker}(\mathsf{sk})$ could be specified and the attacker would be able to extract away the secret key from a signature. We believe that a type system similar to the one proposed in this paper can be applied here to limit the extra expressive power of the rule-based approach by allowing only to abstract only variables over unrestricted parts of the predicates, ruling out the example given above by declaring sk a restricted variable over $\mathsf{sign}(\mathsf{m},\mathsf{sk})$. Similar considerations can be applied to other systems that base their analysis on pattern-matching techniques, like the extended strand-space approach in [5] and Miller's linear logic approach for security protocols [11].

As also pointed out in the text the local operator of utcc does not really correspond to the generation of new names in nominal calculi. This has already been noticed by Palamidessi et al. [15], where a logical characterization of name restriction using the existential quantifier does not ensure uniqueness in the fragment of the π -calculus with mismatch. The same occurs in utcc: a process $(\mathbf{local}\ x)\ (\mathbf{local}\ y)\ P$ can hide both x and y from the store, but the current logical formulation does not ensure the uniqueness of x and y, as one may wish when dealing with nonces for security protocols. We leave for future work to study variants of the local operator ensuring uniqueness.

Acknowlegments The authors would like to thank the anonymous reviewers for their suggestions for improvement of this paper. We would also like to thank Jorge A. Pérez and Frank Valencia for their comments on earlier versions of this document. This research has been partially supported by the Trustworthy Pervasive Healthcare Services (TrustCare) project. Danish Research Agency, Grant # 2106-07-0019 (www.TrustCare.eu)

References

1. M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 104–115, New York, NY, USA, 2001. ACM Press.
2. M. Abadi and A. D. Gordon. A Calculus for Cryptographic Protocols: The SPi Calculus. *Inf. Comput.*, 148(1):1–70, 1999.
3. B. Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society.
4. M. Buchholtz, H. Riis Nielson, and F. Nielson. A calculus for control flow analysis of security protocols. *International Journal of Information Security*, 2(3):145–167, 2004.

5. R. Corin and S. Etalle. An improved constraint-based system for the verification of security protocols. In M. V. Hermenegildo and G. Puebla, editors, *9th Int. Static Analysis Symp. (SAS)*, volume LNCS 2477, pages 326–341, Madrid, Spain, Sep 2002. Springer-Verlag, Berlin.
6. F. Crazzolara and G. Winskel. Events in security protocols. In *ACM Conference on Computer and Communications Security*, pages 96–105, 2001.
7. D. Dolev and A. C. Yao. On the security of public key protocols. Technical report, Dept. of Computer Science, Stanford University, Stanford, CA, USA, 1981.
8. M. Fiore and M. Abadi. Computing symbolic models for verifying cryptographic protocols. *Proc. 14th IEEE Computer Security Foundations Workshop*, pages 160–173, 2001.
9. H. A. López, J. A. Pérez, C. Palamidessi, C. Rueda, and F. D. Valencia. A declarative framework for security: Secure concurrent constraint programming. In *22th International Conference on Logic Programming (ICLP'06)*, volume 4079 of *Lecture Notes in Computer Science*. Springer, 2006.
10. G. Lowe. An attack on the needham-schroeder public-key authentication protocol. *Inf. Process. Lett.*, 56(3):131–133, 1995.
11. D. Miller. Encryption as an abstract data type: An Extended Abstract. In *Foundations of Computer Security (FCS)*, volume 84 of *Electronic Notes in Theoretical Computer Science*, pages 3–15. Springer-Verlag, 2003.
12. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. *Journal of Information and Computation*, 100:1–77, September 1992.
13. C. A. Olarte and F. D. Valencia. The expressivity of universal timed ccp. In *10th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, Valencia, Spain, July 2008. ACM Press.
14. C. A. Olarte and F. D. Valencia. Universal concurrent constraint programming: Symbolic semantics and applications to security. In *23rd Annual ACM Symposium on Applied Computing (2008)*, 2008.
15. C. Palamidessi, V. Saraswat, F. Valencia, and B. Victor. On the Expressiveness of Linearity vs Persistence in the Asychronous Pi-Calculus. In *Proceedings of the 21st Annual IEEE Symposium on Logic in Computer Science*, pages 59–68. IEEE Computer Society Washington, DC, USA, 2006.
16. V. Saraswat, R. Jagadeesan, and V. Gupta. Foundations of timed concurrent constraint programming. In *Logic in Computer Science, 1994. LICS'94. Proceedings., Symposium on*, pages 71–80, 1994.