Formal Models for Trustworthy Process and Service Oriented Systems

HUGO ANDRÉS LÓPEZ ACOSTA
lopez@itu.dk

Project report to qualify for the degree of
Master of Science in Information Technologies

Advisor
THOMAS T. HILDEBRANDT

IT University of Copenhagen
Programming Logic and Semantics
Master Programme on Software Development and Technology
Copenhagen
2009

Abstract

This report studies *process calculi* and its connections with service oriented systems. Despite of being such a young trend, service oriented systems have become an important area of research in the last decade, and several trends have been derived. In particular, we center our studies in the relationships between two dichotomies: the global/local views of services, and the imperative/declarative way used for the specification of services. On the one hand, the level of abstraction used for modelling processes plays an important role: either we describe an interaction scenario from a global viewpoint (choreography) or we describe the system as the composition of the local behaviours of each participant (orchestration). On the other hand, descriptions can have imperative or declarative flavors: In an imperative approach, we explicitly define the control flow of commands, whereas in a declarative approach the focus is drifted to the specification of the set of conditions processes should fulfill in order to be considered correct. Even if these two trends address similar concerns, we find that they have evolved rather independently from each other.

The ultimate goals of this thesis are: (i) to provide formal models for process and service oriented systems, possibly combining the different views described above, and (ii) to provide support for the reasoning about correctness and security of a service oriented system, thereby raising its level of trustworthiness. Specifically, this project centers its research within the area of *process calculi*, and its connections with *type systems* and *specification logics* as ways to provide such correctness guarantees. The report can be divided into two main contributions:

- First, we start by using a model of coordination defined in terms of agents that interact over a global store consulting and imposing constraints. The model, known as Concurrent Constraint Programming (CCP), allows for asynchronous communication, abstraction of data, deadlocks and a limited model of mobile behaviour. A recent extension to CCP, known as Universal Timed CCP (`utcc`) introduces the possibility of universally quantify over predicates in the constraint store. We propose a simple type system for constraints used as patterns in process abstractions, which essentially allows us to distinguish between universally abstractable information and secure (non-leakable information) 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.

- Second, we relate the work on Concurrent Constraint Programming to other models of coordination. We describe initial results on the definition of a formal framework for the declarative analysis of services. We shall exploit `utcc`, to give a declarative interpretation to the language defined by Honda, Vasconcelos, and Kubo (henceforth referred to as

HVK). This way, services can be analyzed in a declarative framework where time is defined explicitly. We begin by proposing an encoding of the HVK language into utcc and studying its correctness. We then move to the timed setting, and propose $\mathsf{HVK}^{\mathsf{T}}$, a timed extension of HVK. The extended language explicitly includes information on session duration, allows for declarative preconditions within session establishment constructs, and features a construct for session abortion. We then discuss how the encoding of HVK into utcc straightforwardly extends to $\mathsf{HVK}^{\mathsf{T}}$.

Additionally, we present initial efforts in relating logics and coordination models. We aim at leveraging the trustworthiness of a process model by providing a methodology for the specification and verification of structured communications. As recently presented, choreographies and orchestrations are operationally correspondent, and one can either use an choreography to generate its end-point projections, or to take a set of end-point specifications and describe their respective choreography. We present a way to describe properties over global specifications. Starting with an extension of Hennesy-Milner logic, we introduce a proof system that allows for verification of properties among participants in a choreography. With such a logic, one can see the state of a choreography as a formula, and one can check for satisfaction of desirable properties. Some examples of important properties on service specifications are drawn, and we provide hints on how this work can be extended towards a full verification framework for services, closing the gap between logics for choreography and their correspondent parts over an end-point projection.

Contents

List of Tables

List of Figures

1 Introduction

This report studies *process calculi* and its connections with service oriented systems. In particular, we explore the use of *type systems* and *specification logics* as ways to provide guarantees about the trustworthiness of a system. Formally, the report counts as my Master's thesis and it describes core points of my research during the first two years of my Ph.D.

1.1 Motivation

As recently pointed out by the ICT theme of EU Seventh Framework Programme (FP7), the need of trustworthy and pervasive services infrastructure is considered one of the three mayor challenges in ICT for the next ten years. The "future internet" proposes questions in terms of scalability, mobility, flexibility, security, trust and robustness to the $\rangle 30$ years old current Internet architecture. A vast landscape of application and ever-changing requirements and environments must be supported, and new ways of interaction must be devised, coping with safety and reliability in their coordination methods.

The line of research investigating such questions has been constantly expanding since the early nineties, both combining approaches from the academia and the industry. As result of such efforts, its been normally hard to differentiate between similar derived fields, like Business Processes, Workflow technologies and Service Oriented Computing. A Business Process is the set of steps executed in order to fulfill a (business) goal. Business processes have always been at the hearth of companies interests, and the obvious goal has been to develop better, cheaper, and faster processes, incrementing the profit of the company. Workflows came as an initial response for the need of proper descriptions of business processes, providing a framework for the specification and automation of processes by means of activities respecting a business logic. They aim at integrating coarse-grained components and have a single place where the business logic is specified. Furthermore, Service Oriented Computing (SOC) opens a new different horizon by distributing the places where the business logic is defined: now, small process units (services) can be shared between different organizations, so each of them can fulfill their business goals by reusing and outsourcing services.

One of the most important aspects when modelling services relate to the notion of *trustworthiness*. Here we consider trustworthiness as the set of guarantees that one can evidence from a system, both globally for all participants involved in a service composition, or locally as the guarantee that a single service must comply. A *safe* system is one in which a

property consider harmful for the life of the system would never happen, like for instance the disclosure of the private credentials of the manager to a thief. Local guarantees relate to the causal ordering in which events occurred in a system, like for instance the relation between the payment of a good and its posterior delivery.

Despite of being such a young trend, different but interrelated views for the analysis of service oriented systems have been proposed. We can enclose such approaches in two dichotomies: global/local views of services, and imperative/declarative specifications. In the first dichotomy, either we describe the system as the exchange of messages between different participants, or we consider the system as the composition of the local behaviours of each participant. In this first view, known as *choreography*, we consider the system as a whole, taking care only of the interfaces that participants use when interacting to the outside world. In the second view, known as *orchestration*, we model the system as perceived by the eyes of each participant, sending and receiving messages but not knowing which other actors are present in a communication.

The second dichotomy here considered refers to the approach used to construct the models. Descriptions can have imperative or declarative flavors: In an imperative approach, we explicitly define the control flow of commands. Typical representatives of this approach are based on process calculi, and come with behavioral equivalences and type disciplines as their main analytic tools. On the contrary, in a declarative approach the focus drifts to the specification of the set of conditions processes should fulfill in order to be considered correct. Even if these two trends address similar concerns, we find that they have evolved rather independently from each other.

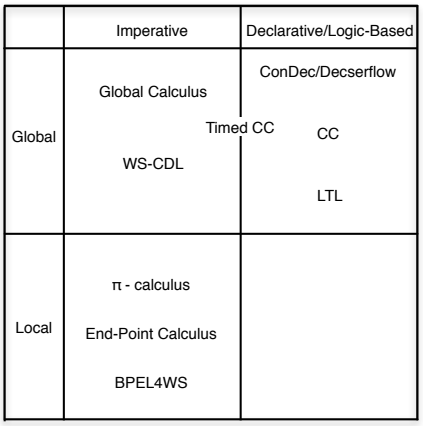

Figure 1.1: Views and approaches in service modelling

Figure 1.1 presents representative languages and their relation to the above mentioned characteristics. Starting with the π-calculus as foundational language, it supplied designers of service oriented systems with rich theories (type systems, behavioural equivalences, static analysis, specification logics) to study the behaviour of workflows [PW05]. This approach, purely imperative in the way workflows were represented, evolved later in more mature extensions of calculi specially defined for choreography and orchestration. The Global calculus [CHY07] originates from Choreography Description Language (CDL), a web service description language developed by W3C WS-CDL working group. The end-

point calculus is a typed π-calculus describing orchestrations and the causal relations between messages. Other approaches have followed a similar evolution, such as for example [LPT07, BBC+06, HVK98, VCS08]. An imperative approach has also been used for industry languages and de-facto standards for the specification and execution of services, such as IBM's Web Service Flow Language [L+01], Microsoft's XLANG [Tha01] and BPEL4WS [ACD+03].

On the declarative side, languages such as ConDec [PvdA06] and DecSerFlow [vdAP06], base their approaches on the specification of systems as the composition of pattern templates for Linear time Temporal Logic (LTL) formulas [MP92]. Similarly, industry-tailored languages such as the Process Matrix [NPS05], have made use of the flexibility provided by a logical formalization of workflows to derive flexible and adaptative service oriented systems [LHM08].

The remaining items in Figure 1.1 refers to the Concurrent Constraint (CC) family of languages [Sar93] and its timed extensions. We can see CC languages part of the declarative approaches for the analysis of choreographies: Differently from the classical approach where a value is assigned to each system variable *(store-as-valuation)*, in CC languages the store represents a *constraint* on the possible values of variables at one point in the life of the system. This allows us to consider both the declarative flavour of logics and the execution of processes both in a single framework: The satisfaction of a formula allows the system to proceed, and the execution/inhibition of a process in the interaction is only defined by the amount of information available in the store. Timed extensions of the CC family refine the notion of store-as-constraint, describing the system as sequences of input-output stimuli between a set of processes and a store. These extensions give us enough modelling power to express declarative and imperative information in the same framework, as we will see further in this document.

This thesis has as main objectives to explore the differences between current approaches in service-oriented systems, start bridging the gap between different views in the specification of services, and to provide current specification languages with new techniques for increase the trustworthiness of a system. The approach here taken involves several independent but interrelated efforts at advancing current process calculi languages with verification techniques where properties about a process can be verified. Particularly, the two verification techniques here studied are:

- Type systems: These are tractable syntactic methods for proving the existence/absence of certain process behaviour. Types enrich process specifications by giving the guarantee that, if respected the type discipline, then some good property will always be respected, as for instance that the private key of a participant will never be disclosed under a communication protocol.

- Specification Logics: Is the connection that a given process specification can have with a given logical counterpart. Depending on the logic used, we can analyse properties of the system such as that a given action would happen in the next evolution of the system, or that a given property will eventually hold. Two main exponents of this approach are the Hennesy-Milner logic [HM80] and temporal logics (branching [Eme91] or linear time [MP92]).

1.2 Contributions

This qualification report presents the results a compilation of the following two research papers:

1. Types for Secure Pattern Matching with Local Knowledge in Universal Concurrent Constraint Programming. Joint work with T. Hildebrandt. In *Proceedings of the International Conference on Logic Programming (ICLP)*, volume 5649 of Lecture Notes in Computer Science, pages 417–431 [HL09].

2. Towards a Unified Framework for Declarative Structured Communications. Joint work with C. Olarte, and J. A. Pérez. In ETAPS satellite workshop on *Programming Language Approaches to Concurrency and Communication-cEntric Software: (PLACES'09)*, Electronic Notes of Theoretical Computer Science [LOP09].

Additionally, the current report draws initial ideas on further topics of research during the second part of my PhD. The first steps towards an specification logic for choreographies is presented in chapter 5.

1.3 Document Structure

The document is structured as follows: In the next chapter we present a brief description about calculi for concurrent communication systems. Such work puts together the preliminaries sections of [HL09] and [LOP09]. We make particular emphasis on the foundations of calculi for the specification of service-oriented systems, such as the π-calculus, and CCP.

In chapter 3 starts our quest for trustworthy methods by introducing an initial type system for the family of CC-languages, in particular, a recent extension of timed CC with a universal abstraction operator [OV08a]. We propose a type system for constraints used as patterns in process abstractions, which essentially allows us to distinguish between public information and secure (non-leakable information) inside 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. The published version of such work can be found at [HL09].

Chapter 4 provides a mapping between different calculi for the specification of services: We shall exploit universal CCP, to give a declarative interpretation to the of structured communications defined by Honda, Vasconcelos, and Kubo. This way, structured communications can be analyzed in a declarative framework where time is defined explicitly. We begin by proposing an encoding of such language into universal CCP and studying its correctness. We then move to the timed setting, and propose a timed extension of the language of structured communications. The language explicitly includes information on session duration, allows for declarative preconditions within session establishment constructs, and features a construct for session abortion. Finally, we make use of the

connections of Universal CCP with Linear Temporal Logic to provide interesting analysis in terms of constraint templates used in declarative analyses of services. The published version of the work can be found at [LOP09].

In Chapter 5 we report initial steps towards a methodology for the specification and verification of structured communications. As recently presented, choreographies and orchestrations are operationally correspondent, and one can either use an choreography to generate its end-point projections, or to take a set of end-point specifications and describe their respective choreography. We present a way to describe properties over global specifications. Starting with an extension of Hennesy-Milner logic, we introduce a proof system that allows for verification of properties among participants in a choreography. With such a logic, one can see the state of a choreography as a formula, and one can check for satisfaction of desirable properties. Some examples of important properties on structured communications are drawn, and we provide hints on how this work can be extended towards a full verification framework for structured communications, closing the gap between logics for choreography and their correspondent parts over an end-point projection.

Finally, in Chapter 6 we present overall concluding remarks, as well as pointing out to principal directions derived from this work in the second part of my PhD.

2 Preliminaries

Process calculi (also known as *process algebras*) are formalisms devised for the description and analysis of the behavior of *concurrent systems*; i.e., systems consisting of multiple computing agents (*processes*) that interact with each other. As such, the goal of a process calculus is to provide a *rigorous framework* where complex systems can be accurately analyzed, including *reasoning techniques* to verify their essential properties. In this section we discuss some basic principles on process calculi, including several issues that distinguish them from other formal models for concurrency and the main approaches to give meaning to processes.

The nature and features of concurrent systems occurring in the real world makes it difficult the task of finding a canonical model in which *every* system can be accurately represented. In fact, even in the context of a restricted field (say, distributed systems) a wide variety of different phenomena, occurring at different levels, can be recognized. The goal is then to identify a set of common set of *underlying principles* in the systems of interest, and to define suitable operators that capture them in a precise way. In other words, a process of *abstraction* is required to define meaningful calculi in the simplest possible way.

Process calculi are then *abstract* specification languages for concurrent systems. This implies that models of systems abstract from real but unimportant details that do not contribute in essential system interactions. Abstraction not only allow designers to better understand the core of a system, but it also turns out to be necessary for an effective use of reasoning techniques associated to the calculus.

In addition, process calculi follow a *compositional* approach for systems description. This implies that a process calculus model of a system is given in terms of models representing its subsystems. This favors an appropriate abstraction of the main components of the systems and, more importantly, allows to explicitly reason about the *interactions* among the identified subsystems. As we will see later, each calculus assumes a particular abstraction criteria over systems, which will have influence on the level of compositionality models will exhibit.

Process calculi also pay special attention to *economy*. There are few process constructors, each one with a distinct and fundamental role in capturing the behavior of systems. A reduced number of constructors in the language helps to maintain the theory underlying the calculus tractable as well as stimulates a precise definition of the abstraction criteria that the calculus intends to express.

Let us illustrate the interplay of the above issues by introducing one of the most representative process calculus for mobility.

2.1 A Process Calculus for Mobile Systems

The π-calculus [Mil99, SW01], was proposed by Milner, Parrow and Walker in the early 90's for the analysis of mobile, distributed systems. The ability of representing *link mobility* is one of the main advances of the π-calculus with respect CCS (Calculus for Communicating Systems) [Mil95], its immediate predecessor. In the π-calculus, the description of mobile systems and their interactions is based on the notion of *name*. In principle, a process (an abstraction of a mobile agent) should be capable of evolving in many different ways, but always maintaining its identity during the whole computation. In addition, a process should be capable of identifying other related processes. In the π-calculus a name also denotes a *communication channel*, in such a way that communication among two processes is possible provided that they share the same channel. As a consequence, in the π-calculus a name abstracts the identity of processes in an interaction by considering the communication channel each process is related to.

In the π-calculus, process capabilities are abstracted as *atomic actions*. They come in two main flavors:

- $x(z)$, representing the *reception* (or reading) of the datum z on the channel x. z is then ready for any subsequent computations.
- $\overline{x}\langle d\rangle$, denoting the *transmission* of a datum d over the channel x.

Actions (denoted by α) are used in the context of *processes* that are constructed by the following syntax:

$$P, Q, \ldots ::= \mathbf{0} \mid \sum_{i \in I} \alpha_i.P_i \mid P \parallel Q \mid !P \mid (\nu x)\, .P\,.$$

Some intuitions underlying the behavior of these processes follow.

- Process $\mathbf{0}$ represents the process that does *nothing*. It is meant to be the basis of more complex processes.
- The *interaction* of processes P and Q is represented by their *parallel composition* $P \parallel Q$. In addition to the individual actions of each process, their communication is possible, provided that they *synchronize* on a channel, as illustrated in the following example.

  $$R = x(y).\overline{y}\langle z\rangle.\mathbf{0} \parallel \overline{x}\langle w\rangle.\mathbf{0}$$

  Here, R represents the interaction of two processes sharing a channel x. The transmission of w through x is complemented by its reception, which involves recognizing w as y. This is regarded as an atomic computational step. Afterwards, a datum z is sent, using the received name w as communication channel. Notice that in the context of R, there is no partner for w in its attempt of transmitting z.
- $\sum_{i \in I} \alpha_i.P_i$, usually known as a *summation* process, represents a choice on the involved P_i's, depending on the capabilities represented by each α_i. Only when any

such processes is ready to interact with another one, a *choice* among all the possible interaction options takes place. For instance, in the process

$$(x(y).\overline{z}\langle y\rangle.\mathbf{0} + z(y).\mathbf{0} + x(w).w(z).\mathbf{0}) \parallel \overline{x}\langle r\rangle.\mathbf{0}$$

the first and third components of the sum are ready to interact with $\overline{x}\langle r\rangle.\mathbf{0}$. Depending on the choice, different resulting processes are possible. For instance, if the third component is selected, the resulting interaction would lead to the process $r(z).\mathbf{0}$.

- Process $!P$ represents the *infinite execution* (or *replication*) of process P. There will be an infinite number of copies of P executing: $!P = P \parallel P \parallel P \parallel \ldots$.
- Process $(\nu x)\ .P$ is meant to describe *restricted* names. Name x is said to be *local* to P and is only visible to it. A disciplined use of restricted names is crucial in delimiting communication.

The π-calculus is thus a language based in a few simple, yet powerful, abstractions. In addition to the above-mentioned abstraction of name as communication channels that can be transmitted, in the π-calculus the behavior of mobile systems is reduced to a few representative phenomena: synchronization on shared channels, infinite behavior and restricted communication. The compositional nature of the calculus is elegantly defined by the parallel composition operator, which is the basis for representing interactions among processes and the construction of models.

2.1.1 Meaning of Processes

Endowing process terms with a formal meaning is crucial in order to analyze process behavior. A process language can have several semantic interpretations. In fact, the combination of two or more approaches is a common practice, since for instance, an approach can be more appropriate for intuitive understanding of processes whereas other can be more suitable for mathematical proofs. This is usually the case of Operational Semantics and Denotational/Algebraic ones. The use of several semantics motivates a legitimate question, that of determining whether different semantics are equivalent to each other. Lets illustrate the use of a semantics with the introduction of an operational semantics for the π-calculus.

Operational Semantics An operational semantics interprets a process term by using *transitions* that define computational steps. A common practice is to capture the state of the system by means of *configurations*, succinct structures that, in addition to a process term, may include other relevant information. Transitions are usually *labelled* by the actions that originate evolution between configurations. This is commonly denoted as $P \xrightarrow{a} Q$, meaning that process P performs action a and then behaves as process Q. Operational semantics are then defined by a set of *(reduction) rules* that formally define the features of the relation $\xrightarrow{a}$. The set of reduction rules that constitute the operational behavior of a calculus is also known as its *labelled transition system* (or *LTS*).

As an example, consider the rule that formalizes the communication of interacting processes in the π-calculus, informally discussed in the previous section:

$$x(y).P \parallel \overline{x}\langle z\rangle.Q \xrightarrow{\tau} P\{z/y\} \parallel Q.$$

In this (labelled) rule, $P\{z/y\}$ denotes the syntactic replacement of all occurrences of the name y with the name z in the context of process P. An excerpt of the transition rules for the late π-calculus is presented in Figure 2.1.

$$\mathsf{P}_{\mathsf{output}}\ \frac{}{x(y).P \xrightarrow{x(y)} P} \qquad \mathsf{P}_{\mathsf{input}}\ \frac{}{\overline{x}\langle z\rangle.P \xrightarrow{\overline{x}\langle z\rangle} P}$$

$$\mathsf{P}_{\mathsf{synch}}\ \frac{P \xrightarrow{x(y)} P' \qquad Q \xrightarrow{\overline{x}\langle z\rangle} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'\{z/y\}} \qquad \mathsf{P}_{\mathsf{struct}}\ \frac{P \equiv P' \quad P' \xrightarrow{\alpha} Q' \quad Q' \equiv Q}{P \xrightarrow{\alpha} Q}$$

Table 2.1: Operational semantics of the π-calculus (excerpt)

2.2 Concurrent Constraint Programming

This section provides the interested reader the main concepts of Concurrent Constraint Programming (CCP), Temporal Concurrent Constraint Programming (`tcc`) and its universal extension (`utcc`), following the presentation of [OV08a].

Concurrent Constraint Programming (CCP) was first introduced by Vijay Saraswat in [Sar93] as a rich family of programming languages where (partial) information plays a fundamental in the computation and control of concurrent programs. 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., $x+y \geq 42$). It also defines an *entailment* relation "$\Vdash$" specifying interdependencies among constraints. Intuitively, $c \Vdash d$ means that the information in d can be deduced from that in c (as in, e.g., $x \geq 42 \Vdash x \geq 0$). 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 `tt` and `ff` 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.

Timed concurrent constraint programming (tcc) [SJG94] extends CCP for modeling reactive systems. In tcc, time is conceptually divided into *time units* (or discrete *time intervals*). In a particular time unit, a tcc process P gets an input (i.e. a constraint) c from the environment, it executes with this input as the initial *store*, and when it reaches its resting point, it *outputs* the resulting store d to the environment. The resting point determines also a residual process Q which is then executed in the next time unit. 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 interval 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.

$$\begin{aligned} P, Q \ldots \quad ::= \quad & \mathbf{skip} \;\mid\; \mathbf{tell}\,(\mathtt{c}) \;\mid\; \mathbf{when}\ \mathtt{c}\ \mathbf{do}\ P \;\mid\; P \parallel Q \;\mid\; (\mathbf{local}\ \vec{x}; c)P \;\mid \\ & \mathbf{next}\,(P) \;\mid\; \mathbf{unless}\ c\ \mathbf{next}\,(P) \;\mid\; !P \end{aligned}$$

Intuitively, the process **skip** does nothing, **tell** (c) adds a new constraint c into the store, while **when** c **do** P asks if c is present into the store in order to execute P. A process (**local** $\vec{x}$; c) P *binds* the variables $\vec{x}$ in P by declaring them private to P under a constraint c. If $\mathtt{c} = \mathtt{tt}$, we write (**local** $\vec{x}$) P instead of (**local** $\vec{x}$; tt) P. The operators associated with time allow the process to go one time unit in the future (**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 **unless** c **next** P will execute P at the next time unit. We will often use $\mathbf{next}^n\,(P)$ as a shorter version of **next** (**next** $(\ldots$ **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 [OV08a] is an extension of the tcc calculus with a general *ask* defining a model of synchronization. While in tcc an ask **when** c **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. ($\lambda\ \vec{x}$; c) P can be seen as the dual version of (**local** $\vec{x}$; c) P in which the variables are *abstracted* with respect to the constraint c and the process P. A process $Q = (\lambda\ \vec{x}; \mathtt{c})\ P$ binds the variables $\vec{x}$ in P and c. It executes $P[\vec{t}/\vec{x}]$ for every term $\vec{t}$ s.t. the current store entails an admissible substitution over $\mathtt{c}[\tilde{\mathtt{t}}/\tilde{\mathtt{x}}]$. The substitution $[\vec{t}/\vec{x}]$ is admissible if $|\vec{x}| = |\vec{t}|$ and no x_i in $\vec{x}$ occurs in $\vec{t}$. Furthermore, Q evolves into **skip** at the end of the time unit, i.e., abstractions are not persistent when passing from one time unit to the next one.

The operational semantics provides the intuitions on how utcc processes interact. In principle, a configuration is represented by the tuple $\langle P, \mathtt{c}\rangle$ where P denotes a set of processes and c a constraint store. P can evolve to a further process P' during an *internal transition* ($\longrightarrow$) 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 2.1, where $\langle P, \mathtt{c}\rangle$ denotes a configuration, and $F(Q)$ denotes the *future function of process* Q.

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 $P \equiv Q$.

$$\mathsf{R_T}\ \frac{}{\langle \mathbf{tell}\,(\mathsf{d}), \mathsf{c}\rangle \longrightarrow \langle \mathbf{skip}, \mathsf{c} \wedge \mathsf{d}\rangle} \qquad \mathsf{R_S}\ \frac{\gamma_1' \longrightarrow \gamma_2'}{\gamma_1 \longrightarrow \gamma_2}\ \text{if} \gamma_1 \equiv \gamma_1' \text{ and } \gamma_2 \equiv \gamma_2'$$

$$\mathsf{R_P}\ \frac{\langle P, \mathsf{c}\rangle \longrightarrow \langle P', \mathsf{c}'\rangle}{\langle P \parallel Q, \mathsf{c}\rangle \longrightarrow \langle P' \parallel Q, \mathsf{c}'\rangle} \qquad \mathsf{R_U}\ \frac{d \Vdash c}{\langle \mathbf{unless}\,\mathsf{c}\,\mathbf{next}\,P, \mathsf{d}\rangle \longrightarrow \langle \mathbf{skip}, \mathsf{d}\rangle}$$

$$\mathsf{R_R}\ \frac{}{\langle !P, \mathsf{c}\rangle \longrightarrow \langle P \parallel \mathbf{next}\,(!P), \mathsf{c}\rangle} \qquad \mathsf{R_L}\ \frac{\langle P, (\exists \tilde{\mathsf{x}}\mathsf{d}) \wedge \mathsf{c}\rangle \longrightarrow \langle P', (\exists \tilde{\mathsf{x}}\mathsf{d}) \wedge \mathsf{c}'\rangle}{\langle (\mathbf{local}\ \vec{x}; \mathsf{c})\ P, \mathsf{d}\rangle \longrightarrow \langle (\mathbf{local}\ \vec{x}; \mathsf{c}')\ P', (\exists \tilde{\mathsf{x}}\mathsf{c}') \wedge \mathsf{d}\rangle}$$

$$\mathsf{R_A}\ \frac{d \Vdash c[\vec{t}/\vec{x}] \qquad [\vec{t}/\vec{x}] \text{ is admissible}}{\langle (\lambda\ \vec{x}; \mathsf{c})\ P, \mathsf{d}\rangle \longrightarrow \langle P[\vec{t}/\vec{x}] \parallel (\lambda\ \vec{x}; \mathsf{c} \wedge (\tilde{\mathsf{x}} \neq \tilde{\mathsf{t}}))\ P, \mathsf{d}\rangle}$$

$$\mathsf{R_O}\ \frac{\langle P, \mathsf{c}\rangle \longrightarrow^* \langle Q, \mathsf{d}\rangle \not\longrightarrow}{P \xRightarrow{(c,d)} F(Q)} \qquad \text{Where } F(Q) = \begin{cases} \mathbf{skip} & \text{if } Q = \mathbf{skip} \\ F(Q_1) \parallel F(Q_2) & \text{if } Q = Q_1 \parallel Q_2 \\ R & \text{if } Q = \mathbf{next}\,(R) \\ \mathbf{skip} & \text{if } Q = (\lambda\ \vec{x}; \mathsf{c})\ R \\ (\mathbf{local}\ \vec{x})\ F(R) & \text{if } Q = (\mathbf{local}\ \vec{x}; \mathsf{c})\ R \\ R & \text{if } Q = \mathbf{unless}\,\mathsf{c}\,\mathbf{next}\,R \end{cases}$$

Figure 2.1: Transition System for `utcc`: Internal and Observable transitions

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 [OV08a] 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.

`utcc` provides a number of reasoning techniques: First, `utcc` processes can be represented as partial closure operators (i.e. idempotent and extensive functions). Also, for a significant fragment of the calculus, the input-output behavior of a process P can be retrieved

from the set of fixed points of its associated closure operator [OV08b]. Second, utcc processes can be characterized as First-order Linear-time Temporal Logic (FLTL) formulas [MP92]. This declarative view of the processes allows for the use of the well-established verification techniques from FLTL to reason about utcc processes.

Definition 4 (Output Behavior)**.** Let $s = c_1.c_2....c_n$ be a sequence of constraints. If $P = P_1 \xrightarrow{(\mathtt{tt},c_1)} P_2 \xrightarrow{(\mathtt{tt},c_2)} \dots P_n \xrightarrow{(\mathtt{tt},c_n)} P_{n+1} \equiv_u Q$ we shall write $P \xrightarrow{s}{}^{*} Q$. If $s = c_1.c_2.c_3...$ is an infinite sequence, we omit Q in $P \xrightarrow{s}{}^{*} Q$. The *output behavior* of P is defined as $o(P) = \{s \mid P \xrightarrow{s}{}^{*}\}$. If $o(P) = o(Q)$ we shall write $P \sim^{o} Q$. Furthermore, if $P \xrightarrow{s} Q$ and s is unimportant we simply write $P \longrightarrow^{*} Q$.

Logic Correspondence. Remarkably, in addition to this operational view, utcc processes admit a declarative interpretation based on Pnueli's first-order linear-time temporal logic (FLTL) [MP92]. This is formalized by the encoding below, which maps utcc processes into FLTL formulas.

Definition 5. Let $\mathsf{TL}[\![\cdot]\!]$ a map from utcc processes to FLTL formulas given by:

$$
\begin{array}{lcllcl}
\mathsf{TL}[\![\mathbf{skip}]\!] & = & \mathtt{tt} & \mathsf{TL}[\![\mathbf{tell}(c)]\!] & = & c \\
\mathsf{TL}[\![P \parallel Q]\!] & = & \mathsf{TL}[\![P]\!] \wedge \mathsf{TL}[\![Q]\!] & \mathsf{TL}[\![(\lambda\ \vec{y}; \mathsf{c})\ P]\!] & = & \forall \vec{y}(c \Rightarrow \mathsf{TL}[\![P]\!]) \\
\mathsf{TL}[\![(\mathbf{local}\, \vec{x}; c)\, P]\!] & = & \exists \vec{x}(c \wedge \mathsf{TL}[\![P]\!]) & \mathsf{TL}[\![\mathbf{next}\, P]\!] & = & \circ \mathsf{TL}[\![P]\!] \\
\mathsf{TL}[\![\mathbf{unless}\ c\ \mathbf{next}\, P]\!] & = & c \vee \circ \mathsf{TL}[\![P]\!] & \mathsf{TL}[\![!\, P]\!] & = & \Box \mathsf{TL}[\![P]\!]
\end{array}
$$

Modalities $\circ F$ and $\Box F$ represent that F holds *next* and *always*, respectively. We use the *eventual* modality $\Diamond F$ as an abbreviation of $\neg\Box\neg F$.

The following theorem relates the operational view of processes with their logic interpretation.

Theorem 1 (Logic correspondence [OV08a])**.** Let $\mathsf{TL}[\![\cdot]\!]$ be as in Definition 5, P a utcc process and $s = c_1.c_2.c_3...$ an infinite sequence of constraints s.t. $P \xrightarrow{s}{}^{*}$. For every constraint d, it holds that: $\mathsf{TL}[\![P]\!] \Vdash \Diamond d$ iff there exists $i \geq 1$ s.t. $c_i \Vdash d$.

Recall that an observable transition $P \xrightarrow{(c,c')} Q$ is obtained from a finite sequence of internal transitions (rule $\mathrm{R_O}$). We notice that there exist processes that may produce infinitely many internal transitions and as such, they cannot exhibit an observable transition; an example is $(\lambda\ x; \mathsf{c}(\mathsf{x}))\ \mathbf{tell}(c(x+1))$. The utcc processes considered in this paper are *well-terminated*, i.e., they never produce an infinite number of internal transitions during a time unit. Notice also that in the Theorem 1 the process P is assumed to be able to output a constraint c_i for all time-unit $i \geq 1$. Therefore, P must be a well-terminated process.

3 Types for Secure Pattern Matching with Local Knowledge in Universal CCP

3.1 Introduction

A number of variants of process calculi and logical approaches have been proposed for the analysis of security protocols, including [AG99, CW01, CE02, FA01, Bla01, Mil03, BRNN04, OV08a]. 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 [BRNN04], 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`)[OV08a], which extends timed concurrent constraint programming [SJG94] to include a universally quantified abstraction (ask) operation. Intuitively, the new operation added in `utcc`, written $(\lambda\ \vec{x}; \mathtt{c})\ P$, spans a copy of the residual process $P[\vec{t}/\vec{x}]$ for all possible inferences of $\mathtt{c}[\vec{t}/\vec{x}]$. This adds the ability to extend the scope of local knowledge which is not possible in CCP [LPP+06]. In particular it was illustrated in [OV08a] 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 [OV08a] and the Security Protocol Language (SPL) by Crazzolara and Winskel [CW01].

The foregoing document is divided as follows: Section 3.2 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 3.3 we give more details on the use of the `utcc` with secure patterns. Finally, concluding remarks and future work are described in Section 3.4. This chapter presupposes previous knowledge on CCP and `utcc`: definitions can be found in Section 2.2.

3.2 `utcc` and Secure Pattern Matching

As described in Sec. 2.2, 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 [MPW92] 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.

3.2.1 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

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

and

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

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 $E = (\lambda\ x, y; \mathsf{rep(x, y)})$ **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 [BRNN04] 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 $\mathsf{enc(m, k)}$ 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 $P = (\mathbf{local}\ k, n)\ \mathbf{tell}\,(\mathsf{o(enc(n, k))})$. However, in `utcc` a spy process defined as $S = (\lambda\ x, z; \mathsf{o(enc(x, z))})\ \mathbf{tell}\,(\mathsf{o(x)} \wedge \mathsf{o(z)})$, 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`.

3.2.2 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 $\mathsf{piout(x; y)}$ 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 3.1 in addition to the SOS rules pictured in Figure 2.1. $\mathsf{R}_{\mathsf{A}\rightarrow}$ 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:

$$\mathsf{R}_{\mathsf{A}\rightarrow}\quad \frac{\mathsf{d} \wedge \mathsf{c} \Vdash \mathsf{c}'[\tilde{\mathsf{t}}/\tilde{\mathsf{x}}] \quad |\tilde{\mathsf{t}}| = |\tilde{\mathsf{x}}| \quad \mathsf{d} \wedge \mathsf{c} \Vdash \mathtt{ff} \Rightarrow \mathsf{d} \Vdash \mathtt{ff}}{\langle (\lambda\ \vec{x}; \mathsf{c} \Rightarrow \mathsf{c}')\ P, \mathsf{d}\rangle \longrightarrow \langle P[\vec{t}/\vec{x}] \parallel (\lambda\ \vec{x}; \mathsf{c} \Rightarrow (\mathsf{c}' \wedge (\tilde{\mathsf{x}} \neq \tilde{\mathsf{t}}))\ P, \mathsf{d}\rangle} \tag{3.1}$$

The condition $\mathsf{d} \wedge \mathsf{c} \Vdash \mathtt{ff} \Rightarrow \mathsf{d} \Vdash \mathtt{ff}$ 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 3.1. 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 functions on terms over the function signature: $unr(t)$, $res(t)$, and $var(t)$ yielding respectively the variables

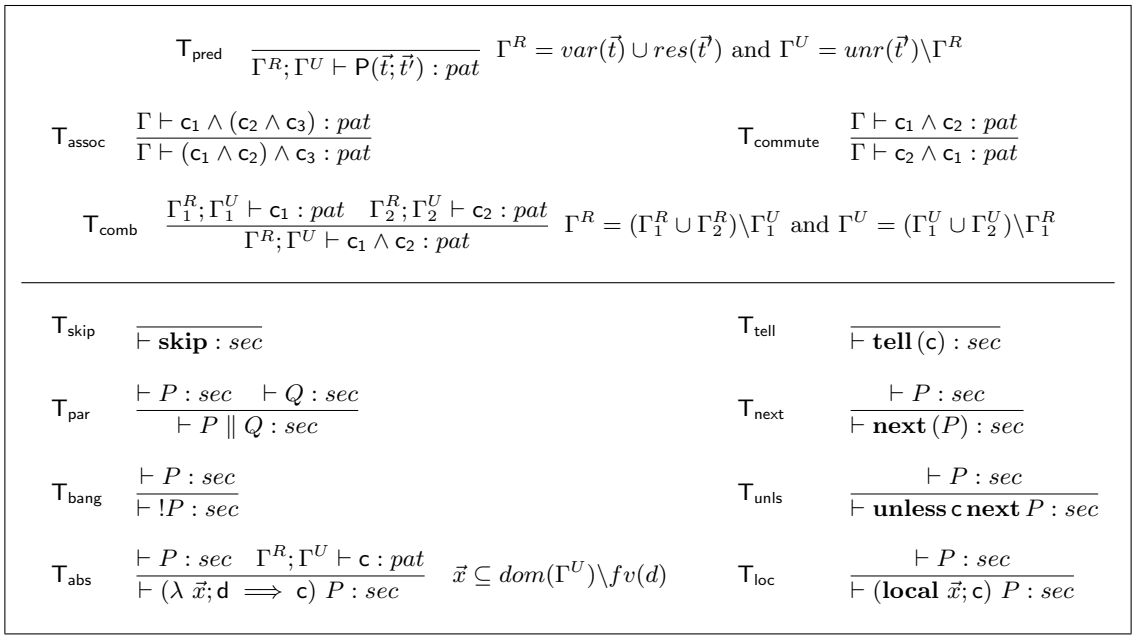

Figure 3.1: Typing rules for secure patterns and processes

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 $unr(\vec{t}) = \cup_{1 \leq i \leq |\vec{t}|} unr(t_i)$ (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 2 (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 3 (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 $\vdash F(Q') : sec$.

□

3.3 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 $\mathsf{piout(x;y)}$ 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 $y\langle x\rangle \wedge x\langle y\rangle$:

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

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) [CW01] and SCCP [OV08b] 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 6 (SPCCP)**.** The Secure Concurrent Constraint Programming language SCCP [OV08b] is redefined by the following grammar:

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

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

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

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

Figure 3.2: Entailment relation for a security constraint system.

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. 3.2 inspired on the requirements stated by Dolev and Yao in [DY81].

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 $o(\emptyset) = \mathtt{tt}$.

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

Definition 7 (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 $\mathtt{utcc}_s$: **nil**, $R \parallel R'$ and $!R$ have the usual meaning of inaction, parallel composition and replication in process calculi; $\mathbf{out}(M).R$ 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: priv(x), pub(x), and sym(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-typeness of the encoding.

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

3.3.1 Protocols

In Fig. 3.3 below we recall the protocol steps of the Needham-Schröeder-Lowe protocol [Low95] (herewith referred as NSL) used as example in [CW01].

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

Figure 3.3: 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. 3.4.

SPCCP share some similarities with the approaches in LySA^{NS} [BRNN04], 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 [BRNN04], except that we employ the constraint system and local knowledge instead of tailoring the pattern matching with a notion of key pairs.

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

Figure 3.4: NSL protocol in SPCCP

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

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

3.4 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 univer-

sal 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 [Bla01], 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 [AF01], 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 $\mathsf{attacker}(\mathsf{sign}(\mathsf{m},\mathsf{sk})) \rightarrow \mathsf{attacker}(\mathsf{sk})$ 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 [CE02] and Miller's linear logic approach for security protocols [Mil03].

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. [PSVV06], 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 (**local** x) (**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.

4 Towards a Unified Framework for Declarative Structured Communications

4.1 Introduction

Motivation. From the viewpoint of *reasoning techniques*, two main trends in modeling in Service Oriented Computing (SOC) can be singled out. On the one hand, an *operational approach* focuses on how process interactions can lead to correct configurations. Typical representatives of this approach are based on process calculi and Petri nets (see, e.g., [vdA98, BBC+06, LVMR07, LPT07]), and count with behavioral equivalences and type disciplines as main analytic tools. On the other hand, in a *declarative approach* the focus is on the set of conditions components should fulfill in order to be considered correct, rather than on the complete specification of the control flows within process activities (see, e.g., [vdAP06, PvdA06]). Even if these two trends address similar concerns, we find that they have evolved rather independently from each other.

The quest for a unified approach in which operational and declarative techniques can harmoniously converge is therefore a legitimate research direction. In this paper we shall argue that Concurrent Constraint Programming (CCP) [Sar93] can serve as a foundation for such an approach. Indeed, the unified framework for operational and logic techniques that CCP provides can be fruitfully exploited for analysis in SOC, possibly in conjunction with other techniques such as type systems. Below we briefly introduce the CCP model and then elaborate on how it can shed light on a particular issue: the analysis of structured communications.

CCP [Sar93] is a well-established model for concurrency where processes interact with each other by *telling* and *asking* for pieces of information (*constraints*) in a shared medium, the *store*. While the former operation simply adds a given constraint to the store (thus making it available for other processes), the latter allows for rich, parameterizable forms of process synchronization. Interaction is thus inherently *asynchronous*, and can be related to a broadcast-like communication discipline, as opposed to the point-to-point discipline enforced by formalisms such as the π-calculus [SW01]. In CCP, the information in the store grows monotonically, as constraints cannot be removed. This condition is relaxed in *timed* extensions of CCP (e.g., [SJG94, NPV02]), where processes evolve along a series of *discrete time units*. Although each unit contains its own store, information is not automatically transferred from one unit to another. In this paper we shall adopt a CCP process language that is timed in this sense.

In addition to the traditional operational view of process calculi, CCP enjoys a *declarative*

nature that distinguishes it from other models of concurrency: CCP programs can be seen, at the same time, as computing agents and as logic formulas [Sar93, NPV02, OV08b], i.e., they can be read and understood as logical specifications. Hence, CCP-based languages are suitable for *both* the specification and verification of programs. In the CCP language used in this paper, processes can be interpreted as linear-time temporal logic formulas; we shall exploit this correspondence to verify properties of our models.

This Work. We describe initial results on the definition of a formal framework for the declarative analysis of structured communications. We shall exploit `utcc` [OV08a], a timed CCP process calculus, to give a declarative interpretation to the language defined by Honda, Vasconcelos, and Kubo in [HVK98] (henceforth referred to as HVK). This way, structured communications can be analyzed in a declarative framework where time is defined explicitly. We begin by proposing an encoding of the HVK language into `utcc` and studying its correctness. We then move to the timed setting, and propose $\mathsf{HVK}^{\mathsf{T}}$, a timed extension of HVK. The extended language explicitly includes information on session duration, allows for declarative preconditions within session establishment constructs, and features a construct for session abortion. We then discuss how the encoding of HVK into `utcc` straightforwardly extends to $\mathsf{HVK}^{\mathsf{T}}$.

A Compelling Example. We now give intuitions on how a declarative approach could be useful in the analysis of structured communications. Consider the ATM example from [HVK98, Sect. 4.1], given below:

$$
\begin{array}{lcl}
ATM(a,b) & = & \textbf{accept}\ a(k)\ \textbf{in}\ k![id]; \\
& & k \rhd \left\{ \begin{array}{l}
deposit : \textbf{request}\ b(h)\ \textbf{in} \\
\quad k?(amt)\ \textbf{in}\ h \lhd deposit; \\
\quad h![id, amt]; ATM(a,b) \\
\parallel withdraw : \textbf{request}\ b(h)\ \textbf{in} \\
\quad k?(amt)\ \textbf{in}\ h \lhd withdraw; h![id, amt]; \\
\quad h \rhd \left\{ \begin{array}{l} success : k \lhd dispense; k![amt]; ATM(a,b) \\ \parallel failure : k \lhd overdraft; ATM(a,b) \end{array} \right\} \\
\parallel balance : \textbf{request}\ b(h)\ \textbf{in}\ h \lhd balance; h?(amt)\ \textbf{in} \\
\quad k![amt]; ATM(a,b)
\end{array} \right\}
\end{array}
$$

Table 4.1: ATM process specification

Here, an ATM has established two sessions: the first one with a user, sharing session k over service a, and the second one with the bank, sharing session h over service b. The ATM offers `deposit`, `balance`, and `withdraw` operations. When executing a withdraw, if there is not enough money in the account, then an *overdraft* message appears to the user. It is interesting to analyze what occurs when this scenario is extended to consider a card reader that acts as a malicious interface between the user and the ATM. The user communicates his personal data with the reader using the service r, which will be kept by the reader after the first withdraw operation to continue withdrawing money without the authorization of the user. A greedy card reader could even withdraw repeatedly until causing an overdraft (labelled "over"), as expressed below:

$$
\begin{array}{lcl}
\mathit{Reader} & = & \mathbf{accept}\ r(k')\ \mathbf{in}\ k'?(id)\,\mathbf{in} \\
& & \quad \mathbf{request}\ a(k)\ \mathbf{in}\ k![id]; \\
& & \qquad k' \rhd \left\{ \begin{array}{l} \mathit{withdraw} : k'?(amt)\,\mathbf{in} \\ \quad k \lhd \mathit{withdraw}; k![amt]; \\ \quad\ k \rhd \{\mathit{dispense} : k' \lhd \mathit{dispense}; k![amt]; R(k, amt) \parallel \mathit{over} : Q\} \end{array} \right\} \\
R(j, x) & = & \mathbf{def}\ R'\ \mathbf{in}\ k \lhd \mathit{withdraw}; j![x]; j \rhd \{\mathit{dispense} : j?(amt)\,\mathbf{in}\ R' \parallel \mathit{over} : Q\} \\
\mathit{User} & = & \mathbf{request}\ r(k')\ \mathbf{in}\ k'![myId]; \\
& & \quad k' \lhd \mathit{withdraw}; k'![58]; \quad k' \rhd \{\mathit{dispense} : k'?(amt)\,\mathbf{in}\ P \parallel \mathit{over} : Q\}
\end{array}
$$

By creating sessions between them, the card reader *Reader* is able to receive the user's information, and to use it later by attempting a session establishment with the bank. Following authentication steps (not modeled above), the card reader allows the user to obtain the requested amount. Additional withdrawing transactions between the reader and the bank are defined by the recursive process R. In the specification above, the process Q can be assumed to send a message (through a session with the bank) representing the fact that the account has run out of money: $Q = k_{bank}![\underline{0}]; \mathbf{inact}$.

Even in this simple scenario, the combination of operational and declarative reasoning techniques may come in handy to reason about the possible states of the system. Indeed, while an operational approach can be used to describe an operational description of the compromised ATM above, the declarative approach can complement such a description by offering declarative insights regarding its evolution. For instance, assuming Q as above, one could show that a `utcc` specification of the ATM example satisfies the linear temporal logic formula $\Diamond\, \mathtt{out}(k_{bank}, 0)$, which intuitively means that in presence of a malicious card reader the user's bank account will eventually reach an overdraft status.

Related Work. One approach to combine the declarative flavor of constraints and process calculi techniques is represented by a number of works that have extended name-passing calculi with some form of partial information (see, e.g., [VP98, DRV98]). The crucial difference between such a strand of work and CCP-based calculi is that the latter offer a tight correspondence with logic, which greatly broadens the spectrum of reasoning techniques at one's disposal. Recent works similar to ours include CC-Pi [BM07] and the calculus for structured communications in [CDC09]. Such languages feature elements that resemble much ideas underlying CCP (especially [BM07]). The main difference between our approach and such works is that we adhere to the use of declarative reasoning techniques based on temporal logic as an effective way of complementing operational reasoning techniques. In [BM07], the reasoning techniques associated to CC-Pi are essentially operational, and used to reason about service-level agreement protocols. In [CDC09], the key for analysis is represented by a type system which provides consistency for session execution, much as in the original approach in [HVK98].

4.2 Preliminaries

4.2.1 A Language for Structured Communication

We begin by introducing HVK, a language for structured communication proposed in [HVK98]. We assume the following conventions: *names* are ranged over by $a, b, \ldots$; *channels* are ranged over by k, k'; *variables* are ranged over by $x, y, \ldots$; *constants* (names, integers, booleans) are ranged over by $c, c', \ldots$; *expressions* (including constants) are ranged over by $e, e', \ldots$; *labels* are ranged over by $l, l', \ldots$; *process variables* are ranged over by $X, Y, \ldots$. Finally, $u, u', \ldots$ denote names and channels. We shall use $\vec{x}$ to denote a sequence (tuple) of variables $x_1 ... x_n$ of length $n = |\vec{x}|$. Notation $\vec{x}$ will be similarly applied to other syntactic entities. The sets of free names/channels/variables/process variables of P, is defined in the standard way, and are respectively denoted by $(\cdot)$, $fc(\cdot)$, $fv(\cdot)$, and $fpv(\cdot)$. Processes without free variables or free channels are called *programs*.

Definition 8 (The HVK language [HVK98])**.** Processes in HVK are built from:

P,Q	::=	**request** $a(k)$ **in** P	Session Request
	|	**accept** $a(k)$ **in** P	Session Acceptance
	|	$k![\vec{e}]$; P	Data Sending
	|	$k?(\vec{x})$ **in** P	Data Reception
	|	$k \lhd l$;P	Label Selection
	|	$k \rhd \{l_1 : P_1 \parallel \cdots \parallel l_n : P_n\}$	Label Branching
	|	**throw** $k[k']$; P	Channel Sending
	|	**catch** $k(k')$ **in** P	Channel Reception
	|	**if** e **then** P **else** Q	Conditional Statement
	|	P | Q	Parallel Composition
	|	**inact**	Inaction
	|	(νu)P	Hiding
	|	**def** D **in** P	Recursion
	|	$X[\vec{e}\vec{k}]$	Process Variables
D	::=	$X_1(x_1k_1) = P_1$ **and** $\cdots$ **and** $X_n(x_nk_n) = P_n$	Declaration for Recursion

Operational Semantics of HVK. The operational semantics of HVK is given by the reduction relation $\longrightarrow_h$ which is the smallest relation on processes generated by the rules in Figure 4.1. In Rule Str, the structural congruence $\equiv_h$ is the smallest relation satisfying : 1) $P \equiv_h Q$ if they differ only by a renaming of bound variables (alpha-conversion). 2) $P \mid \mathbf{inact} \equiv_h P$, $P \mid Q \equiv_h Q \mid P$, $(P \mid Q) \mid R \equiv_h P \mid (Q \mid R)$. 3) $(\nu u)\mathbf{inact} \equiv_h \mathbf{inact}$, $(\nu u u')P \equiv_h (\nu u' u)P$, $(\nu u)(P \mid Q) \equiv_h (\nu u)P \mid Q$ if $x \notin fv(Q)$, $(\nu u)(\mathbf{def}\ D\ \mathbf{in}\ P) \equiv_h (\mathbf{def}\ D\ \mathbf{in}\ ((\nu u)P))$ if $u \notin fv(D)$. 4) $(\mathbf{def}\ D\ \mathbf{in}\ P) \mid Q \equiv_h \mathbf{def}\ D\ \mathbf{in}\ (P \mid Q)$ if $fpv(D) \cap fpv(Q) = \emptyset$. 5) $\mathbf{def}\ D\ \mathbf{in}\ (\mathbf{def}\ D'\ \mathbf{in}\ P) \equiv_h \mathbf{def}\ D$ and $D'\ \mathbf{in}\ P$ if $fpv(D) \cap fpv(D') = \emptyset$.

Let us give some intuitions about the language constructs and the rules in Figure 4.1. The central idea in HVK is the notion of a *session*, i.e., a series of reciprocal interactions between two parties, possibly with branching, delegation and recursion, which serves as an abstraction unit for describing structured communication. Each session has associated a

Link	**request** $a(k)$ **in** Q | **accept** $a(k)$ **in** $P \longrightarrow_h (\nu k)(P \mid Q)$
Com	$(k![\vec{e}]; P) \mid (k?(\vec{x})\,\mathbf{in}\,Q) \longrightarrow_h P \mid Q[\vec{c}/\vec{x}]$ if $e \downarrow \vec{c}$
Label	$k \lhd l_i; P \mid k \rhd \{l_1 : P_1 \parallel \cdots \parallel l_n : P_n\} \longrightarrow_h P \mid Pi\ (1 \leq i \leq n)$
Pass	**throw** $k[k']; P$ | **catch** $k(k')$ **in** $Q \longrightarrow_h P \mid Q$
If1	**if** e **then** P **else** $Q \longrightarrow_h P\ (e \downarrow \mathtt{tt})$
If2	**if** e **then** P **else** $Q \longrightarrow_h Q\ (e \downarrow \mathtt{ff})$
Def	**def** D **in** $(X[\vec{e}\,\vec{k}] \mid Q) \longrightarrow_h$ **def** D **in** $(P[\vec{c}/\vec{x}] \mid Q)\ (e \downarrow \vec{c}, X(\vec{x}\vec{k}) = P \in D)$
Scop	$P \longrightarrow_h P'$ implies $(\nu u)P \longrightarrow_h (\nu u)P'$
Par	$P \longrightarrow_h P'$ implies $P \mid Q \longrightarrow_h P' \mid Q$
Str	If $P \equiv_h P'$ and $P' \longrightarrow_h Q'$ and $Q' \equiv_h Q$ then $P \longrightarrow_h Q$

Figure 4.1: Reduction Semantics of HVK ($\longrightarrow_h$)[HVK98].

specific port, or *channel*. Channels are generated at session initialization; communications inside the session take place on the same channel.

More precisely, sessions are initialized by a process of the form **request** $a(k)$ **in** Q | **accept** $a(k)$ **in** P. In this case, there is a request, on name a, for the initiation of a session and the generation of a fresh channel. This request is matched by an accepting process on a, which generates a new channel k, thus allowing P and Q to communicate each other. This is the intuition behind rule Link. Three kinds of atomic interactions are available in the language: sending (including name passing), branching, and channel passing (also referred to as delegation). Those actions are described by rules Com, Label, and Pass, respectively. In the case of Com, the expression $\vec{e}$ is sent on the port (session channel) k. Process $k?(\vec{x})\,\mathbf{in}\,Q$ then receives such a data and executes $Q[\vec{c}/\vec{x}]$, where $\vec{c}$ is the result of evaluating the expression $\vec{e}$. The case of Pass is similar but considering that in the constructs **throw** $k[k']; P$ and **catch** $k(k')$ **in** Q, only session names can be transmitted. In the case of Label, the process $k \lhd l_i; P$ selects one label and then the corresponding process P_i is executed. The other rules are self-explanatory.

For the sake of simplicity, and without loss of generality (due to rule 5 of $\equiv_h$), in the sequel we shall assume programs of the form **def** D **in** P where there are not procedure definitions in P.

4.2.2 utcc's Derived Constructs.

From a programming language perspective, variables $\vec{x}$ in $(\lambda\ \vec{x}; \mathsf{c})\ P$ can be seen as the formal parameters of P. This way, *recursive definitions* of the form

$$X(\vec{x}) \stackrel{\mathtt{def}}{=} P$$

can be encoded in utcc as:

$$\mathcal{R}[\![X(\vec{x}) \stackrel{\mathtt{def}}{=} P]\!] =! (\lambda\ \vec{x}; \mathsf{call_x}(\tilde{\mathsf{x}}))\ \widehat{P} \tag{4.1}$$

where $\mathsf{call_x}$ is an uninterpreted predicate (a constraint) of arity $|\vec{x}|$. Process $\widehat{P}$ is obtained from P by replacing recursive calls of the form $X(\vec{t})$ with $\mathbf{tell}\,(\mathsf{call_x}(\tilde{\mathsf{t}}))$. Similarly, calls of the form $X(\vec{t})$ in other processes are replaced with $\mathbf{tell}\,(\mathsf{call_x}(\tilde{\mathsf{t}}))$.

Let out be an uninterpreted predicate. One could attempt at representing the actions of sending and receiving as in a name-passing calculus (say, $k![\vec{e}]$ and $k?(\vec{x})\,\mathbf{in}\ P$, resp.) with the utcc processes $\mathbf{tell}(\mathtt{out}(k, \vec{e}))$ and $(\lambda\ \vec{x}; \mathtt{out}(\mathsf{k}, \tilde{\mathsf{x}}))\ P$, respectively. Nevertheless, since these processes are not automatically transferred from one time unit to the next one, they will disappear right after the current time unit, even if they do not interact. To cope with this kind of behavior, we shall define versions of $(\lambda\ \vec{x}; \mathsf{c})\ P$ and $\mathbf{tell}(c)$ processes that are *persistent in time*. More precisely, we shall use the process $(\underline{\mathbf{wait}}\ \vec{x}; c)\ \mathbf{do}\ P$, which transfers itself from one time unit to the next one until, for some $\vec{t}$, $c[\vec{t}/\vec{x}]$ is entailed by the current store. Intuitively, the process behaves like an input that is active until interacting with an output. When this occurs, the process outputs the constraint $\overline{c}[\vec{t}/\vec{x}]$, as a way of acknowledging the successful read of c. When $|\vec{x}| = 0$, we shall write $\mathbf{whenever}\ c\ \mathbf{do}\ P$ instead of $(\underline{\mathbf{wait}}\ \vec{x}; c)\ \mathbf{do}\ P$. Similarly, we define $\underline{\mathbf{tell}}(c)$ for the persistent output of c until some process "reads" c. These processes can be expressed in the basic utcc syntax as follows (in all cases, we assume $stop, go \notin fv(c)$):

$$\begin{array}{lcl}
\underline{\mathbf{tell}}(c) & \stackrel{\mathtt{def}}{=} & (\mathbf{local}\, go, stop)\,(\ \mathbf{tell}(\mathtt{out}'(go)) \parallel\! ! \,\mathbf{when}\ \mathtt{out}'(go)\ \mathbf{do}\ \mathbf{tell}(c) \parallel \\
 & & \quad !\,\mathbf{unless}\ \mathtt{out}'(stop)\ \mathbf{next}\,\mathbf{tell}(\mathtt{out}'(go)) \parallel \\
 & & \quad !\,\mathbf{when}\ \overline{c}\ \mathbf{do}\,!\,\mathbf{tell}(\mathtt{out}'(stop))) \\
(\mathbf{wait}\ \vec{x}; c)\ \mathbf{do}\ P & \stackrel{\mathtt{def}}{=} & (\mathbf{local}\, stop, go)\,(\ \mathbf{tell}(\mathtt{out}'(go)) \parallel\! ! \,\mathbf{unless}\ \mathtt{out}'(stop)\ \mathbf{next}\,\mathbf{tell}(\mathtt{out}'(go)) \\
 & & \quad \parallel\! ! \,(\lambda\ \vec{x}; \mathsf{c} \wedge \mathtt{out}'(\mathtt{go}))\ (P \parallel\! ! \,\mathbf{tell}(\mathtt{out}'(stop))) \\
(\underline{\mathbf{wait}}\ \vec{x}; c)\ \mathbf{do}\ P & \stackrel{\mathtt{def}}{=} & (\mathbf{wait}\ \vec{x}; c)\ \mathbf{do}\ (P \parallel \mathbf{tell}(\overline{c}))
\end{array}$$

Notice that once a pair of processes $\underline{\mathbf{tell}}$ and $\underline{\mathbf{wait}}$ interact, their continuation in the next time unit is a process able to output only a constraint of the form $\exists_x\, \mathtt{out}'(x)$ (e.g., $\exists_{stop}(\mathtt{out}'(stop))$). We define the following equivalence relation that allows us to abstract from these processes.

Definition 9 (Observables)**.** Let $\sim^o$ be the output equivalent relation in Definition 4. We say that P and Q are observable equivalent, notation $P \sim^{obs} Q$, if $P \parallel\! ! \,\mathbf{tell}(\exists_x \mathtt{out}'(x)) \sim^o Q \parallel\! ! \,\mathbf{tell}(\exists_x \mathtt{out}'(x))$.

Using the previous equivalence relation, we can show the following.

Proposition 2. Assume that $c(\vec{x})$ is a predicate symbol of arity $|\vec{x}|$.

1. If $d \not\Vdash c[\vec{t}/\vec{x}]$ for any $\vec{t}$ then $(\underline{\mathbf{wait}}\ \vec{x}; c)$ **do** $P \stackrel{(d,d)}{\Longrightarrow} (\underline{\mathbf{wait}}\ \vec{x}; c)$ **do** P.
2. If $P \equiv_u \underline{\mathbf{tell}}(c(\vec{t})) \parallel (\underline{\mathbf{wait}}\ \vec{x}; c(\vec{x}))$ **do next** Q then $P \Longrightarrow\!\!\sim^{obs} Q[\vec{t}/\vec{x}]$.

4.3 A Declarative Interpretation for Structured Communications

The encoding $[\![\cdot]\!]$ from HVK into `utcc` is defined in Table 4.4. Two noteworthy aspects when considering such a translation are *determinacy* and *timed behavior*. Concerning determinacy, it is of uttermost importance to recall that while `utcc` is a deterministic language, HVK processes may exhibit non-deterministic behavior. Moreover, while HVK is a synchronous language, whereas `utcc` is asynchronous. Consider, for instance, the HVK process:

$$P = k![\vec{e}]; Q_1 \mid k![\vec{e'}]; Q_2 \mid k?(\vec{x})\ \mathbf{in}\ Q_3$$

Process P can have two possible transitions, and evolve into $k![\vec{e'}]; Q_2 \mid\ Q_3[\vec{e}/\vec{x}]$ or into $k![\vec{e}]; Q_1 \mid\ Q_3[\vec{e'}/\vec{x}]$. In both cases, there is an output that cannot interact with the input $k?(\vec{x})\ \mathbf{in}\ Q_3$. In `utcc`, inputs are represented by abstractions which are persistent during a time unit. As a result, in the encoding of P we shall observe that *both* outputs react with the same input, i.e. that $[\![P]\!] \Longrightarrow [\![Q_3[\vec{e}/\vec{x}]]\!] \parallel [\![Q_3[\vec{e'}/\vec{x}]]\!]$.

As for timed behavior, it is crucial to observe that while HVK is an untimed calculus, `utcc` provides constructs for explicit time. In the encoding we shall advocate a timed interpretation of HVK in which all available synchronizations between processes occur at a given time unit, and the continuations of synchronized processes will be executed in the next time unit. This will prove convenient when showing the operational correspondence between both calculi, as we can relate the observable behavior in `utcc` and the reduction semantics in HVK.

Let us briefly provide some intuitions on $[\![\cdot]\!]$. Consider HVK processes $P = \mathbf{request}\ a(k)\ \mathbf{in}\ P'$ and $Q = \mathbf{accept}\ a(x)\ \mathbf{in}\ Q'$. The encoding of P declares a new variable session k and sends it through the channel a by posting the constraint $\mathtt{req}(a, k)$. Upon reception of the session key (local variable) generated by $[\![P]\!]$, process $[\![Q]\!]$ adds the constraint $\mathtt{acc}(a, k)$ to notify the acceptance of k. They can then synchronize on this constraint, and execute their continuations in the next time unit. The encoding of label selection and branching is similar, and uses constraint $\mathtt{sel}(k, l)$ for synchronization. We use the parallel composition $\prod_{1 \le i \le n}\ \mathbf{when}\ l = l_i\ \mathbf{do\ next}\ [\![P_i]\!]$ to execute the selected choice. Notice that we do not require a non-deterministic choice since the constraints $l = l_i$ are mutually exclusive. As in [HVK98], in the encoding of $\mathbf{if}\ e\ \mathbf{then}\ P\ \mathbf{else}\ Q$ we assume an evaluation function on expressions. Once e is evaluated, $\downarrow e$ is a *constant* boolean value. The encoding of $\mathbf{def}\ D\ \mathbf{in}\ P$ exploits the scheme described in Equation 4.1.

Operational Correspondence. Here we study an operational correspondence property for our encoding. The differences with respect to (a)synchrony and determinacy discussed above will have a direct influence on the correspondence. Intuitively, the encoding falls

$$
\begin{array}{rcl}
[\![\textbf{request } a(k) \textbf{ in } P]\!] & = & (\textbf{local } k)\,(\underline{\textbf{tell}}(\texttt{req}(a,k)) \parallel \textbf{whenever } \texttt{acc}(a,k) \textbf{ do next } [\![P]\!]) \\
[\![\textbf{accept } a(k) \textbf{ in } P]\!] & = & (\underline{\textbf{wait}}\ k; \texttt{req}(a,k))\ \textbf{do}\ (\textbf{tell}(\texttt{acc}(a,k)) \parallel \textbf{next } [\![P]\!]) \\
 & & \\
[\![k![\vec{e}]; P]\!] & = & \underline{\textbf{tell}}(\texttt{out}(k,\vec{e})) \parallel \textbf{whenever } \overline{\texttt{out}(k,\vec{e})} \textbf{ do next } [\![P]\!] \\
[\![k?(\vec{x}) \textbf{ in } P]\!] & = & (\underline{\textbf{wait}}\ \vec{x}; \texttt{out}(k,\vec{x}))\ \textbf{do next } [\![P]\!] \\
 & & \\
[\![k \lhd l; P]\!] & = & \underline{\textbf{tell}}(\texttt{sel}(k,l)) \parallel \textbf{whenever } \overline{\texttt{sel}(k,l)} \textbf{ do next } [\![P]\!] \\
[\![k \rhd \{l_1 : P_1 \parallel \ldots \parallel l_n : P_n\}]\!] & = & (\underline{\textbf{wait}}\ l; \texttt{sel}(k,l))\ \textbf{do} \prod_{1 \le i \le n} \textbf{when } l = l_i \textbf{ do next } [\![P_i]\!] \\
[\![\textbf{throw } k[k']; P]\!] & = & \underline{\textbf{tell}}(\texttt{outk}(k,k')) \parallel \textbf{whenever } \overline{\texttt{outk}(k,k')} \textbf{ do next } [\![P]\!] \\
[\![\textbf{catch } k(k') \textbf{ in } P]\!] & = & \underline{\textbf{whenever}}\ \texttt{outk}(k,k')\ \textbf{do next } [\![P]\!] \\
 & & \\
[\![\textbf{if } e \textbf{ then } P \textbf{ else } Q\ ]\!] & = & \textbf{when } e \downarrow \texttt{tt} \textbf{ do next } [\![P]\!] \parallel \textbf{when } e \downarrow \texttt{ff} \textbf{ do next } [\![Q]\!] \\
[\![P|Q]\!] & = & [\![P]\!] \parallel [\![Q]\!] \\
[\![\textbf{inact}]\!] & = & \textbf{skip} \\
[\![(\nu u)P]\!] & = & (\textbf{local } u)\,[\![P]\!] \\
[\![\textbf{def } D \textbf{ in } P]\!] & = & \prod_{X_i(x_i k_i) \in D} \mathcal{R}[\![X_i(x_i k_i)]\!]\widehat{P}
\end{array}
$$

Table 4.4: Encoding from HVK into `utcc`. $\mathcal{R}[\![\cdot]\!]$ and $\widehat{P}$ are defined in Equation 4.1.

short for HVK programs featuring the kind of non-determinism that results from "uneven pairings" between session requesters/providers, label selection/branching, and inputs/outputs as in the example above.

We thus find it convenient to appeal to the type system of HVK to obtain some basic determinacy of the source terms. Roughly speaking, the type discipline in [HVK98] ensures a correct pairing between actions and co-actions once a session is established. Although the type system guarantees a correct match between (the types of) session requesters and providers, it does not rule out the kind of non-determinism induced by different orders in the pairing of requesters and providers. We shall then require session providers to be always willing to engage into a session. This is, given a channel a, we require that there is at most one **accept** process (possibly replicated) on a that is able to synchronize with every process requesting a session on a. Notice that this requirement is in line with a meaningful class of programs, namely those described by the type discipline developed in [BHY08, BHY01].

Before presenting the operational correspondence, we introduce some auxiliary notions.

Definition 10 (Processes in normal form)**.** We say that a HVK process P is in *normal form* if takes the form **inact** or **def** D **in** $\nu\vec{u}(Q_1 \mid \cdots \mid Q_n)$ where neither the operators "ν" and "$|$" nor process variables occur in the top level of $Q_1, \ldots, Q_n$.

The following proposition states that given a process P we can find a process P' in normal form, such that: either P' is structurally congruent to P, or it results from replacing the process variables at the top level of P with their corresponding definition (using rule Def).

Proposition 3. For all HVK process P there exists P' in normal form s.t. $P \longrightarrow_h^* \equiv_h P'$ only using the rules Def and Str in Figure 4.1.

Proof. Let P be a process of the form **def** D **in** Q where there are no procedure definitions in Q. By repeated applications of the rule DEF, we can show that $P \longrightarrow_h^* P'$ where P' does not have occurrences of processes variables in the top level. Then, we use the rules of the structural congruence to move the local variables to the outermost position and find $P'' \equiv_h P'$ in the desired normal form. □

Notice that the rules of the operational semantics of HVK are given for pairs of processes that can interact with each other. We shall refer to each of those pairs as a *redex*.

Definition 11 (Redex)**.** A *redex* is a pair of complementary processes composed in parallel as in:

$$(1)\mathbf{request}\ a(k)\ \mathbf{in}\ P \mid \mathbf{accept}\ a(k)\ \mathbf{in}\ Q$$
$$(2)k![\vec{e}]; P \mid k?(\vec{x})\,\mathbf{in}\ Q$$
$$(3)\mathbf{throw}\ k[k']; P \mid \mathbf{catch}\ k(k')\ \mathbf{in}\ Q$$
$$(4)k \lhd l; P \mid k \rhd \{l_1 : P_1 \parallel \cdots \parallel l_n : P_n\}$$

Notice that a redex in HVK synchronizes and reduces in a single transition as in $(k![\vec{e}]; P) \mid (k?(\vec{x})\,\mathbf{in}\ Q) \longrightarrow_h P \mid Q[\vec{e}/\vec{x}]$. Nevertheless, in `utcc`, the encoding of the processes above requires several internal transitions for adding the constraint $\mathtt{out}(k, \vec{e})$ to the current store, and for "reading" that constraint by means of $(\underline{\mathbf{wait}}\ \vec{x}; \mathtt{out}(k, \vec{x}))\ \mathbf{do\ next}\ [\![Q]\!]$ to later execute $\mathbf{next}\ [\![Q[\vec{e}/\vec{x}]]\!]$. We shall then establish the operational correspondence between an observable transition of `utcc` (obtained from a finite number of internal transitions) and the following subset of reduction relations over HVK processes:

Definition 12 (Outermost Reductions)**.** Let $P \equiv_h \mathbf{def}\ D\ \mathbf{in}\ \nu\vec{x}(Q_1 \mid \cdots \mid Q_n)$ be an HVK program in normal form. We define the *outermost reduction relation* $P \Longrightarrow_h P'$ as the maximal sequence of reductions $P \longrightarrow_h^* P' \equiv_h \mathbf{def}\ D\ \mathbf{in}\ \nu\vec{x'}(Q'_1 \mid \cdots \mid Q'_n)$ such that for every $i \in \{1, ..n\}$, either

1. $Q_i = \mathbf{if}\ e\ \mathbf{then}\ R_1\ \mathbf{else}\ R_2 \longrightarrow_h R_{1/2} = Q'_i$;
2. for some $j \in \{1, ..n\}$, $Q_i|Q_j$ is a redex such that $Q_i|Q_j \longrightarrow_h \nu\vec{y}(Q'_i|Q'_j)$, with $\vec{y} \subseteq \vec{x'}$;
3. there is no $k \in \{1, ..n\}$ such that $Q_i \mid Q_k$ is a redex and $Q_i \equiv_h Q'_i$.

One may argue that the above-presented definition may rule out some possible reductions in HVK. Returning to the concerns about determinacy, an outermost reduction filters out cases where there are more than one possible reduction for a set of parallel processes (i.e.: the parallel composition of two outputs and one input with the same session key). The use of outermost reductions gives us a subset of possible reductions in HVK that keeps synchronous processes and discard processes that are not going to interact in any way (recall that in the typing discipline of HVK the composition of an input and an output with the same session key will consume the channel used; hence, every other process sending information over the same session will not have any complementary process to synchronize with).

In the sequel we shall thus consider only HVK processes P where for $n \geq 1$, if $P \equiv_h P_1 \Longrightarrow_h P_2 \Longrightarrow_h \cdots \Longrightarrow_h P_n$ and $P \equiv_h P'_1 \Longrightarrow_h P'_2 \Longrightarrow_h \cdots \Longrightarrow_h P'_n$ then $P_i \equiv_h P'_i$ for all $i \in \{1, .., n\}$, i.e., P is a *deterministic* process.

Theorem 4 (Operational Correspondence)**.** Let P, Q be deterministic HVK processes in normal form and R,S be utcc processes. It holds:
1) *Soundness*: If $P \Longrightarrow_h Q$ then, for some R, $[\![P]\!] \Longrightarrow R \sim^{obs} [\![Q]\!]$;
2) *Completeness*: If $[\![P]\!] \Longrightarrow S$ then, for some Q, $P \Longrightarrow_h Q$ and $[\![Q]\!] \sim^{obs} S$.

Proof. Assume that $P \equiv_h \mathbf{def}\ D\ \mathbf{in}\ \nu\vec{x}(Q_1 \mid \cdots \mid Q_n)$ and $Q \equiv_h \mathbf{def}\ D\ \mathbf{in}\ \nu\vec{x'}(Q'_1 \mid \cdots \mid Q'_n)$.

1. *Soundness.* Since $P \Longrightarrow_h Q$ there must exist a sequence of derivations of the form $P \equiv_h P_1 \longrightarrow_h P_2 \longrightarrow_h ... \longrightarrow_h P_n \equiv_h Q$. The proof proceeds by induction on the length of this derivation, with a case analysis on the last applied rule. We then have the following cases:

   (a) **Using the rule If1**. It must be the case that there exists $Q_i \equiv_h \mathbf{if}\ e\ \mathbf{then}\ R_1\ \mathbf{else}\ R_2$ and $Q_i \longrightarrow_h R_1 \equiv_h Q'_i$ and $e \downarrow \mathtt{tt}$. One can easily show that $\mathbf{when}\ e \downarrow \mathtt{tt}\ \mathbf{do\ next}\ [\![Q'_i]\!] \Longrightarrow [\![Q'_i]\!]$.

   (b) **Using the rule If2** Similarly as for If1.

   (c) **Using the rule Link**. It must be the case that there exist i, j such that $Q_i \equiv_h \mathbf{request}\ a(k)\ \mathbf{in}\ Q'_i$ and $Q_j \equiv_h \mathbf{accept}\ a(x)\ \mathbf{in}\ Q'_j$ and then $Q_i \mid Q_j \longrightarrow_h (\nu k)(Q'_i \mid Q'_j)$. We then have a derivation

$$\begin{array}{rcl}
[\![Q_i]\!] \parallel [\![Q_k]\!] & \longrightarrow^* & (\mathbf{local}\, k; c)\,(R'_i \parallel \mathbf{whenever}\ \mathtt{acc}(a,k)\ \mathbf{do\ next}\,[\![Q'_i]\!] \parallel \\
& & \qquad (\underline{\mathbf{wait}}\ k'; \mathtt{req}(a,k'))\ \mathbf{do}\ (\mathbf{tell}(\mathtt{acc}(a,k')) \parallel \mathbf{next}\,([\![Q'_j]\!])) \\
& \longrightarrow^* & (\mathbf{local}\, k; c')\,(R'_i \parallel \mathbf{whenever}\ \mathtt{acc}(a,k)\ \mathbf{do\ next}\,[\![Q'_i]\!] \parallel \\
& & \qquad R'_j \parallel \mathbf{tell}(\mathtt{acc}(a,k)) \parallel \mathbf{next}\,([\![Q'_j[k/k']]\!]) \\
& \longrightarrow^* & (\mathbf{local}\, k; c'')\,(R'_i \parallel R'_j \parallel \mathbf{next}\,[\![Q'_i]\!] \parallel \mathbf{next}\,([\![Q'_j[k/k']]\!]) \not\longrightarrow
\end{array}$$

   where $c = \mathtt{req}(a,k)$, $c' = c \wedge \overline{\mathtt{req}(a,k)}$, $c'' = c' \wedge \mathtt{acc}(a,k) \wedge \overline{\mathtt{acc}(a,k)}$ and R'_i, R'_j are the processes resulting after the interaction of the processes in the parallel composition $\underline{\mathbf{tell}}(\mathtt{req}(a,k)) \parallel (\underline{\mathbf{wait}}\ k'; \mathtt{req}(a,k'))\ \mathbf{do}\ \cdots$, i.e.:

$$\begin{array}{rcl}
R'_i & \equiv_u & (\mathbf{local}\, go, stop; \mathtt{out}'(go) \wedge \mathtt{out}'(stop) \wedge c(\vec{t})) \\
& & \qquad \mathbf{next}\,!\,\mathbf{unless}\ \mathtt{out}'(stop)\ \mathbf{next}\,\mathbf{tell}(\mathtt{out}'(go)) \parallel \mathbf{next}\,!\,\mathbf{tell}(\mathtt{out}'(stop))
\end{array}$$

$$\begin{array}{rcl}
R'_j & \equiv_u & (\mathbf{local}\, stop', go'; \mathtt{out}'(go') \wedge \overline{c}(\vec{t}) \wedge \mathtt{out}'(stop'))\,\mathbf{next}\,!\,\mathbf{tell}(\mathtt{out}'(stop')) \\
& & \quad \parallel \mathbf{next}\,!\,\mathbf{unless}\ \mathtt{out}'(stop')\ \mathbf{next}\,\mathbf{tell}(\mathtt{out}'(go')) \\
& & \quad \parallel (\lambda\ \vec{x}; \mathtt{c} \wedge \mathtt{out}'(\mathtt{go}') \wedge \tilde{\mathtt{x}} \neq \tilde{\mathtt{t}})\ (Q \parallel \mathbf{tell}(\overline{c}(\vec{t})) \parallel !\,\mathbf{tell}(\mathtt{out}'(stop')) \\
& & \quad \parallel \mathbf{next}\,!\,(\lambda\ \vec{x}; \mathtt{c} \wedge \mathtt{out}'(\mathtt{go}'))\ (Q \parallel \mathbf{tell}(\overline{c}(\vec{t})) \parallel !\,\mathbf{tell}(\mathtt{out}'(stop'))
\end{array}$$

   We notice that $R'_i \parallel R'_j \not\longrightarrow$ and it is a process that can only output the constraint $\mathtt{out}'(x)$ where x is a local variable. By appealing to Proposition 2 we conclude $[\![Q_i]\!] \parallel [\![Q_j]\!] \Longrightarrow \sim^{obs} (\mathbf{local}\, k)\,([\![Q'_i]\!] \parallel [\![Q'_j]\!])$.

   (d) The cases using the rules Label and Pass can be proven similarly as the case for link.

2. *Completeness.* Given the encoding and the structure of P, we have a `utcc` process $R = [\![P]\!]$ s.t.

$$R \equiv_u (\mathbf{local}\,\vec{x})\,([\![Q_1]\!] \parallel ... \parallel [\![Q_n]\!])\,.$$

Let $R_i = [\![Q_i]\!]$ for $1 \leq i \leq n$. By an analysis on the structure of R, if $R_i \longrightarrow R'_i$ then it must be the case that either (a) $R_i = \mathbf{when}\ e\ \mathbf{do\ next}\ [\![Q'_i]\!]$ and $R'_i = \mathbf{next}\,[\![Q'_i]\!]$ or (b) $\langle R_i, c\rangle \longrightarrow \langle R'_i, c \wedge d\rangle$ where d is a constraint of the form $\mathtt{req}(\cdot)$, $\mathtt{sel}(\cdot)$, $\mathtt{out}(\cdot)$, or $\mathtt{outk}(\cdot)$. In both cases we shall show that there exists a R''_i such that $R_i \longrightarrow^* R''_i \not\longrightarrow$ such that $Q_i \longrightarrow_h Q'_i$ and $R''_i = \mathbf{next}\,[\![Q'_i]\!]$.

   (a) Assume that $R_i = \mathbf{when}\ e \downarrow \mathtt{tt}\ \mathbf{do\ next}\ [\![Q'_i]\!]$ for some Q'_i. Then it must be the case that $Q_i = \mathbf{if}\ e\ \mathbf{then}\ Q'_i\ \mathbf{else}\ Q''_i$. If $e \downarrow \mathtt{tt}$ we then have $R''_i = \mathbf{next}\,[\![Q'_i]\!]$. The case when $e \downarrow \mathtt{ff}$ is similar by considering $R_i = \mathbf{when}\ e \downarrow \mathtt{ff}\ \mathbf{do}\ Q'_i$.

   (b) Assume now that $\langle R_i, c\rangle \longrightarrow \langle R'_i, c \wedge d\rangle$ where d is of the form $\mathtt{req}(\cdot)$, $\mathtt{sel}(\cdot)$, $\mathtt{out}(\cdot)$ or $\mathtt{outk}(\cdot)$. We proceed by case analysis of the constraint d. Let us consider only the case $d = \exists_k(\mathtt{req}(a,k))$; the cases in which d takes the form $\mathtt{sel}(\cdot)$, $\mathtt{out}(\cdot)$, or $\mathtt{outk}(\cdot)$ are handled similarly. If $d = \exists_k(\mathtt{req}(a,k))$ for some a, then we must have that $Q_i \equiv_h \mathbf{request}\ a(k)\ \mathbf{in}\ Q'_i$ for some i. If there exists j such that $Q_j \equiv_h \mathbf{accept}\ a(x)\ \mathbf{in}\ Q'_j$, one can show a derivation similar to the case of the rule Link in soundness to prove that $R_i \parallel R_j \longrightarrow^* \sim^o (\mathbf{local}\,k)\,(\mathbf{next}\,[\![Q'_i]\!] \parallel \mathbf{next}\,[\![Q'_j]\!])$. If there is no Q_j such that $Q_i \,|Q_j$ forms a redex, then one can show by using (1) in Proposition 2 that $R_i \Longrightarrow \sim^{obs} R_i$.

□

4.4 A Timed Extension of HVK

We now propose an extension to HVK in which a bundled treatment of time is explicit and session closure is considered. More precisely, the HVK$^{\mathsf{T}}$ language arises as the extension of HVK processes (Def. 8) with refined constructs for session request and acceptance, as well as with a construct for session abortion:

Definition 13 (A timed language for sessions)**.** HVK$^{\mathsf{T}}$ processes are given by the following syntax:

P	::=	$\mathbf{request}\ a(k)\ \mathbf{during}\ m\ \mathbf{in}$ P	Timed Session Request
	|	$\mathbf{accept}\ a(k)\ \mathbf{given}\ c\ \mathbf{in}$ P	Declarative Session Acceptance
	|	$\cdots$	{ the other constructs, as in Def. 8 }
	|	$\mathbf{kill}\,c_k$	Session Abortion

The intuition behind these three operators is the following: $\mathbf{request}\ a(k)\ \mathbf{during}\ m\ \mathbf{in}\ P$ will request a session k over the service name a during m time units. Its dual construct is $\mathbf{accept}\ a(k)\ \mathbf{given}\ c\ \mathbf{in}\ P$: it will grant the session key k when requested over the service name a provided by a session and a successful check over the constraint c. Notice that c stands for a precondition for agreement between session request and acceptance. In

c, the duration m of the corresponding session key k can be referenced by means of the variable dur_k. In the encoding we syntactically replace it by the variable corresponding to m. Finally, $\mathbf{kill}\, c_k$ will remove c_k from the valid set of sessions.

$$
\begin{array}{rcl}
[\![\mathbf{request}\ a(k)\ \mathbf{during}\ m\ \mathbf{in}\ P]\!] & = & (\mathbf{local}\, k)\, \underline{\mathbf{tell}}(\mathtt{req}(a,k,m)) \parallel \\
& & \quad \mathbf{whenever}\ \mathtt{acc}(a,k)\ \mathbf{do}\ \mathbf{next}\, (\\
& & \quad \mathbf{tell}(\mathtt{act}(k)) \parallel \\
& & \quad \mathcal{G}_{\mathtt{act}(k)}([\![P]\!]) \parallel \\
& & \quad !_{[m]}\mathbf{unless}\ \mathtt{kill}(k)\ \mathbf{next}\, \mathbf{tell}(\mathtt{act}(k))) \\
[\![\mathbf{accept}\ a(k)\ \mathbf{given}\ c\ \mathbf{in}\ P]\!] & = & (\underline{\mathbf{wait}}\ k; \mathtt{req}(a,k,m) \wedge c[m/dur_k])\ \mathbf{do} \\
& & \quad (\mathbf{tell}(\mathtt{acc}(a,k)) \parallel \mathbf{next}\, \mathcal{G}_{\mathtt{act}(k)}([\![P]\!])) \\
[\![\mathbf{kill}\, k]\!] & = & !\, \mathbf{tell}(\mathtt{kill}(k))
\end{array}
$$

Table 4.5: Encoding of $\mathsf{HVK}^{\mathsf{T}}$. $\mathcal{G}_d(P)$ is in Definition 14.

Adapting the encoding in Table 4.4 to consider $\mathsf{HVK}^{\mathsf{T}}$ processes is remarkably simple (see Table 4.5). Indeed, modifications to the encoding of session request and acceptance are straightforward. The most evident change is the addition of the parameter m within the constraint $\mathtt{req}(a,k,m)$. The duration of the requested session is suitably represented as a bounded replication of the process defining the activation of the session k represented as the constraint $\mathtt{act}(k)$. The execution of the continuation $[\![P]\!]$ is guarded by the constraint $\mathtt{act}(k)$ (i.e. P can be executed only when the session k is valid). Thus, in the encoding we use the function $\mathcal{G}_d(P)$ to denote the process behaving as P when the constraint d can be entailed from the current store, doing nothing otherwise. More precisely:

Definition 14. Let $\mathcal{G} : \mathcal{C} \to \mathit{Procs} \to \mathit{Procs}$ be defined as:

$$
\begin{array}{ll}
\mathcal{G}_d(\mathbf{skip}) & = \mathbf{skip} \\
\mathcal{G}_d(P_1 \parallel P_2) & = \mathcal{G}_d(P_1) \parallel \mathcal{G}_d(P_2) \\
\mathcal{G}_d(\mathbf{tell}(c)) & = \mathbf{when}\ d\ \mathbf{do}\ \mathbf{tell}(c) \\
\mathcal{G}_d(!\, Q) & = !\, \mathcal{G}_d(Q) \\
\mathcal{G}_d(\mathbf{next}\, Q) & = \mathbf{when}\ d\ \mathbf{do}\ \mathbf{next}\, \mathcal{G}_d(Q) \\
\mathcal{G}_d((\lambda\ \vec{x}; \mathtt{c})\ Q) & = (\lambda\ \vec{x}; \mathtt{c})\ \mathcal{G}_d(Q) \quad \text{if } \vec{x} \notin \mathit{fv}(d) \\
\mathcal{G}_d(\mathbf{unless}\ c\ \mathbf{next}\, Q) & = \mathbf{when}\ d\ \mathbf{do}\ \mathbf{unless}\ c\ \mathbf{next}\, \mathcal{G}_d(Q) \\
\mathcal{G}_d((\mathbf{local}\, \vec{x}; c)\, Q) & = (\mathbf{local}\, \vec{x}; c)\, \mathcal{G}_d(Q) \quad \text{if } \vec{x} \notin \mathit{fv}(d)
\end{array}
$$

On the side of session acceptance, the main novelty is the introduction of $c[m/dur_k]$. As explained before, we syntactically replace the variable dur_k by the corresponding duration of the session m. This is a generic way to represent the agreement that should exist between a service provider and a client; for instance, it could be the case that the client is requesting a session longer than what the service provider can or want to grant.

4.4.1 Case Study: Electronic booking

Here we present an example that makes use of the constructs introduced in $\mathsf{HVK}^{\mathsf{T}}$.

Let us consider an electronic booking scenario. On one side, consider a company AC which offers flights directly from its website. On the other side, there is a customer looking for the best offers. In this scenario, the customer establishes a timed session with AC and asks for a flight proposal given a set of constraints (dates allowed, destination, etc.). After receiving an offer from AC, the customer can refine the selection further (e.g. by checking that the prices are below a given threshold) and loops until finding a suitable option, that he will accept by starting the booking phase. One possible $\mathsf{HVK}^{\mathsf{T}}$ specification of this scenario is described in Table 4.6.

$$
\begin{array}{lcl}
\text{Customer} & = & \textbf{request}\ ob(k)\ \textbf{during}\ m\ \textbf{in}\ (k![bookingdata]; Select(k)) \\
\text{Select(k)} & = & k?(offer)\ \textbf{in}\ (\textbf{if}\ (offer.price \leq 1500)\ \textbf{then}\ k \lhd Contract;\ \textbf{else}\ Select(k)\ ) \\
\text{AC} & = & \textbf{accept}\ ob(k)\ \textbf{given}\ dur_k \leq MAX\_TIME\ \textbf{in}\ (k?(bookingData)\ \textbf{in} \\
 & & (\nu u)k![u]; k \rhd \{Contract : \overline{Accept} \parallel Reject : \textbf{kill}\, k\})
\end{array}
$$

Table 4.6: Online booking example with two agents.

In a second stage, the customer uses an online broker to mediate between him and a set of airlines acting as service providers. Let n be the number of service providers, and consider two vectors of fixed length: *Offers*, which contains the list $[Offers_0, \ldots, Offers_i, \ldots, Offers_n]$ of offers received by a customer, and SP, which contains the list of trusted services. First, the customer establishes a session with the broker for a given period m; later on, he/she starts requesting for a flight by providing the details of his/her trip to the broker. On the other side, the broker will look into his pool of trusted service providers for the ones that can supply flights that suit the customer's requirements. All possible offers are transferred back to the customer, who will invoke a local procedure Sel (not specified here) that selects one of the offers by performing an output on name a. Once an offer is selected, the broker will allow a final interaction between the customer and the selected service. He does so by delegating to the customer the session key used previously between him and the chosen service provider. Finally, the broker proceeds to cancel all those sessions concerning the discarded services. An $\mathsf{HVK}^{\mathsf{T}}$ specification of this scenario is given in Table 4.7 where, for the sake of readability, processes denoting post-processing activities are abstracted from the specification.

A notable advantage in using $\mathsf{HVK}^{\mathsf{T}}$ as a modeling language is the possibility of exploiting timed constructs in the specification of service enactment and service cancellation. In the above scenario it is possible to see how $\mathsf{HVK}^{\mathsf{T}}$ allows (i) to effectively take explicit account on the maximal times accepted by the customer: the composition of nested services can take different speeds but the service broker will ensure that customers with low speeds are ruled out of the communication; and (ii) to have a more efficient use of the available resources: since there is not need to maintain interactions with discarded services, the service broker will free those resources by sending kill signals.

$$
\begin{array}{ll}
\text{Customer} = & \textbf{request } ob(k) \textbf{ during } m \textbf{ in } (k![bookingdata]; \\
& k?(n) \textbf{ in } (\\
& \prod_{i \in n} (k?(Offers_i) \textbf{ in } (\\
& Sel(Offers); a?(x) \textbf{ in } k![x]; \\
& \textbf{catch } k(k') \textbf{ in } \\
& k'![PaymentDetails]; \textbf{inact})))) \\
\\
\text{SP} = & \textbf{accept } SP_i(k_i') \textbf{ given } N \leq 300ms \textbf{ in } (\\
& k_i'?(bookingData) \textbf{ in } \\
& k_i'![offer]; \\
& k_i'?(paymentDetails) \textbf{ in inact}) \\
\\
\text{Broker} = & \textbf{accept } ob(k) \textbf{ given } m \leq 500ms \textbf{ in } (\\
& k?(bookingData) \textbf{ in } k![|SP|]; \\
& (\nu u) \prod_{i \in |SP|} (\textbf{request } SP_i(k_i') \textbf{ during } N \textbf{ in } \\
& k_i'![bookingData]; \\
& k_i'?(offer_i) \textbf{ in } (u![offer_i]; \textbf{inact} \parallel S(u,k))) \\
& k?(y) \textbf{ in def } X(Offers, k_1', \ldots, k_n') = P \textbf{ in } \\
& \prod_{i \in |SP|} (\textbf{if } (y = offers_i) \textbf{ then } (\textbf{throw } k[k_i']; PostProc) \textbf{ else } \\
& \textbf{kill } k_i' \parallel P(X - \{offers_i, k_i'\}))) \\
\\
\text{S(u,k)} = & \prod_{i \in |SP|} (u?(offer_i) \textbf{ in inact} \parallel k![offer_i]; \textbf{inact})
\end{array}
$$

Table 4.7: Online booking example with online broker.

4.4.2 Exploiting the Logic Correspondence

To exploit the logic correspondence we can draw inspiration from the *constraint templates* put forward in [PvdA06], a set of LTL formulas that represent desirable/undesirable situations in service management. Such templates are divided in three types: *existence constraints*, that specify the number of executions of an activity; *relation constraints*, that define the relation between two activities to be present in the system; and *negation constraints*, which are essentially the negated versions of relation constraints.

By appealing to Theorem 1, our framework allows for the verification of existence and relation constraints over $\mathsf{HVK}^{\mathsf{T}}$ programs. Assume a $\mathsf{HVK}^{\mathsf{T}}$ program P and let $F = \mathsf{TL}[\![P]\!]$ (i.e., the FLTL formula associated to the `utcc` representation of P). For existence constraints, assume that P defines a service accepting requests on channel a. If the service is eventually active, then it must be the case that $F \Vdash \Diamond \exists_k(\mathtt{acc}(a,k))$ (recall that the encoding of **accept** adds the constraint $\mathtt{acc}(a,k)$ when the session k is accepted). A slight modification to the encoding of **accept** would allow us to take into account the number of accepted sessions and then support the verification of properties such as $F \Vdash \Diamond(N_{sessions}(a) = N)$, informally meaning that the service a has accepted N sessions. This kind of formulas correspond to the existence constraints in [PvdA06, Figure 3.1.a–3.1.c]. Furthermore, making use of the guards associated to ask statements, we can verify relation

constraints as eventual consequences over the system. Take for instance the specification in Table 4.6. Let $\overline{Accept}$ be a process that outputs "ok" through a session h. We then may verify the formula $F \Vdash \exists_u(u.price\langle 1.500 \Rightarrow \texttt{out}(h, ok))$. This is a responded existence constraint describing how the presence of an offer with price less or equal than 1.500 would lead to an acceptance state.

4.5 Concluding Remarks

We have argued for a timed CCP language as a suitable foundation for analyzing structured communications. We have presented an encoding of the language for structured communication in [HVK98] into `utcc`, as well as an extension of such a language that considers explicitly elements of partial information and session duration. To the best of our knowledge, a unified framework where behavioral and declarative techniques converge for the analysis of structured communications has not been proposed before.

Languages for structured communication and CCP process calculi are conceptually very different. We have dealt with some of these differences (notably, determinacy) when stating an operational correspondence property for the declarative interpretation of HVK processes. We believe there are at least two ways of achieving more satisfactory notions of operational correspondence. The first one involves considering extensions of `utcc` with (forms of) non-determinism. This would allow to capture some scenarios of session establishment in which the operational correspondence presented here falls short. The main consequence of adding non-determinism to `utcc` is that the correspondence with FLTL as stated in Theorem 1 would not longer hold. This is mainly because non-deterministic choices cannot be faithfully represented as logical disjunctions (see, e.g., [NPV02]). While a non-deterministic extension to `tcc` with a tight connection with temporal logic has been developed (`ntcc` [NPV02]), it does not provide for representations of mobile links. Exploring whether there exists a CCP language between `ntcc` and `utcc` combining both non-determinism and mobility while providing logic-based reasoning techniques is interesting on its own and appears challenging. The second approach consists in defining a type system for HVK and HVK$^{\mathsf{T}}$ processes better suited to the nature of `utcc` processes. This would imply enriching the original type system in [HVK98] with e.g., stronger typing rules for dealing with session establishment. The definition of such a type system is delicate and needs care, as one would not like to rule out too many processes as a result of too stringent typing rules. An advantage of a type system "tuned" in this way is that one could aim at obtaining a correspondence between well-typed processes and logic formulas, similarly as the given by Theorem 1. In these lines, plans for future work include the investigation of effective mechanisms for the seamless integration of new type disciplines and reasoning techniques based on temporal logic within the elegant framework provided by (timed) CCP languages.

The timed extension to HVK presented here includes notions of time that involve only session engagement processes. A further extension could involve the inclusion of time constraints over input/output actions. Such an extension might be useful to realistically specify scenarios in which factors such as, e.g, network traffic and long-lived transactions, prevent interactions between services from occurring instantaneously. Properties of inter-

est in this case could include, for instance, the guarantee that a given interaction has been fired at a valid time, or that the nested composition of services does not violate a certain time frame. We plan to explore case studies of structured communications involving this kind of timed behavior, and extend/adjust $\mathsf{HVK}^{\mathsf{T}}$ accordingly.

5 A Logic for Choreography

5.1 Introduction

When analysing service oriented systems, either we describe the system as the exchange of messages between different participants, or we consider the system as the composition of the local behaviours of each participant. In this first view, known as *choreography*, we consider the system as a whole, taking care only of the interfaces that participants use when interacting to the outside world. In the second view, known as *orchestration*, we model the system as perceived by the eyes of each participant, sending and receiving messages but not knowing which other actors are present in a communication. A good illustration can be seen in the way a soccer match is planned: the coach has an overall view of the team, and organize how players will interact in each play (the role of a choreography) while each player performs his role by interacting with each of the members of his team by throwing/receiving passes. The way each player synchronize with other members of the team represents the role of an orchestration.

The link between choreographies and orchestration has been proposed in [CHY07]. Here, choreographies and orchestration constitute two interrelated approaches for modelling services. Two languages are proposed: A Global calculus to model choreographies and the End Point calculus to model orchestrations. Additionally, global and local specifications has already been shown operational correspondent under certain conditions, and one can generate an orchestrated model of a choreography by a mapping from the Global calculus to the End Point Calculus (something known as the *End Point Projection (EPP)*).

In this chapter we present a joint work between the author, Marco Carbone and Thomas Hildebrandt, aiming at leveraging the trustworthiness level of a system by providing a clear methodology of specification and verification of structured communications. Our goal is to provide service oriented systems with a *logical characterization*, both from the global perspective or and from their end-point projections. Figure 5.1 illustrates the approach for the specification and verification of service-oriented systems. We can analyse a choreography C either by mapping a specification of global behaviour to a formula ϕ_C describing the interaction between participants, and from here generate $\bigcup_i [\![\phi_i]\!]$, a set of formulas describing the local behaviour of each participant. Similarly, we can start from choreography C and then use an end point projection to generate the parallel composition of the local behaviors of each participant involved in the communication; from here we can study their logical meaning as the set of formulas generated for each participant, closing the verification square.

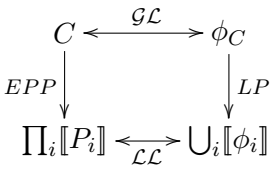

Figure 5.1: Methodology for Service - Oriented Verification

The end point projection between global and local specifications has been previously presented in [CHY07], and their operational correspondence property allow us to move from local to global perspectives and vice versa. Similarly, in a recent work [BHY08] a Hennessy-Milner logic for typed π-calculi is introduced, providing the link between local specifications and logics.

In this document we provide the link between choreographies and a logics (denoted as $\mathcal{GL}$ in Figure 5.1). Starting with an extension of Hennesy-Milner logic [HM80], we provide a proof system that allows for property verification of choreographies. The logic is sound , in terms that a choreography will always reflect a logical state of the system. Furthermore, a final step would conclude a methodology for verification of structured communications. The logic for choreographies should have a correspondent mapping to a logic to reason about end-point projections. This step, denoted by the transformation LP between Global and Local formulas, is left as as further work of the current document.

This chapter is organized as follows: In section 5.2 we describe the global calculus as our reference language, with its syntax and operational semantics. Section 5.3 presents a logical language to express properties about choreographies, giving several examples of its use. Section 5.4 presents the proof system and correctness results while in 5.5 we discuss the future work.

5.2 The Global Calculus

The Global calculus [CHY07] originates from Choreography Description Language (CDL), a web service description language developed by W3C WS-CDL working group. The calculus allows for the description of choreographies as interactions between participants by means of message exchanges. The description of such interactions is centered on the notion of a *session*, in which two interacting parties first establish a private connection and do a series of interactions through that private connection, possibly interleaved with other sessions. More concretely, an interaction between two parties starts by the creation of a fresh session (set of session) channel(s), that later will be used as channels where meaningful interactions take place. Each session is fresh and unique, so each communication activity will be clearly separated from previous interactions by the use of the session. The calculus is equipped with a label transition semantics describing how global descriptions evolve, and a type discipline that describes the structured sequence of message exchanges between participants.

5.2.1 Syntax.

The syntax of the global calculus [CHY06] is given by the following grammar.

$$
\begin{array}{rll}
C ::= & A \to B : a(k).C & \text{(init)} \\
\mid & (\nu k)\, C & \text{(newL)} \\
\mid & A \to B : k\langle e, y\rangle.C & \text{(com)} \\
\mid & (\nu a@A)\, C & \text{(newS)} \\
\mid & A \to B : k[l_i : C_i]_{i \in I} & \text{(choice)} \\
\mid & X^A & \text{(recvar)} \\
\mid & C_1 \mid C_2 & \text{(par)} \\
\mid & \mu X^A.C & \text{(rec)} \\
\mid & \textbf{if } e@A \textbf{ then } C_1 \textbf{ else } C_2 & \text{(cond)} \\
\mid & \mathbf{0} & \text{(inaction)}
\end{array}
$$

$C, C', \ldots$ denote *terms* of the calculus, also called *interactions*; $A, B, C, \ldots$ range over *participants*; $k, k', \ldots$ are *linear channels*; $a, b, c, \ldots$ *shared channels*; $v, w, \ldots$ variables; $X, Y, \ldots$ process variables; $l, l_i, \ldots$ labels for branching; and $e, e', \ldots$ over arithmetic and other first-order expressions.

(init) denotes a session initiation by A via B's service channel a, with fresh session channels k and continuation C. (comm) denotes an in-session communication over a session channel k, where e is an expression. Note that y does not bind in C. (choice) denotes a labelled choice over session channel k and set of labels I. $C_1 \mid C_2$ denotes the parallel product between C_1 and C_2. $(\nu k)\, C$ works the same as the name restriction operator in the π-calculus, binding k in C. Since such a hiding is only generated by session initiation, we assume that a hiding never occurs inside a prefix or a conditional. (cond) is the standard conditional operator ($e@A$ indicates that e is located at participant A). $\mu X^A.C$ denotes recursion, where the variable X^A is bound in C . $\mathbf{0}$ denotes termination. The free and bound session channels and term variables are defined in the usual way. We often omit $\mathbf{0}$ and empty vectors.

5.2.2 Semantics.

We give the operational semantics in terms of a configurations $\langle \sigma, C \rangle$, where σ represents the state of the system and C the choreography actually being executed. σ contains a set of variables labelled by participants. A variable x labelled by participant A is written as $x@A$. The same variable name labelled with different participant names denotes different variables (hence $\sigma@A(x)$ and $\sigma@B(x)$ may differ).

We consider a set of labels $\ell = \{\mathsf{init}\ A \to B\ \mathsf{on}\ a,\ \mathsf{com}\ A \to B\ \mathsf{over}\ s,\ \mathsf{sel}\ A \to B\ \mathsf{over}\ s : l_i\}$ denoting initiation, communication and selection of choreographies between participants A and B, while τ denotes the silent action given by evaluation of expressions.

Structural Congruence. The structural congruence $\equiv$, is defined as the minimal congruence relation on interactions C, such that $\equiv$ is a commutative monoid wrt $\mid$ and $\mathbf{0}$, and satisfies

$$
\begin{array}{rl}
\text{(G-Init)} & (\sigma,\ A \to B : a(k).C) \xrightarrow{\ \mathsf{init}\ A\to B\ \mathsf{on}\ a\ } (\sigma, (\nu k)\ C) \\
\text{(G-Com)} & \sigma' = \sigma[x@B \mapsto v] \ \wedge\ \sigma \vdash e@A \Downarrow v \quad \Rightarrow \quad (\sigma,\ A \to B : k\langle e, x\rangle.C) \xrightarrow{\ \mathsf{com}\ A\to B\ \mathsf{over}\ s\ } (\sigma',\ C) \\
\text{(G-IfT)} & \sigma \vdash e@A \Downarrow \mathtt{tt} \quad \Rightarrow \quad (\sigma,\ \mathbf{if}\ e@A\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ ) \xrightarrow{\ \tau\ } (\sigma, C_1) \\
\text{(G-IfF)} & \sigma \vdash e@A \Downarrow \mathtt{ff} \quad \Rightarrow \quad (\sigma,\ \mathbf{if}\ e@A\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ ) \xrightarrow{\ \tau\ } (\sigma,\ C_2) \\
\text{(G-Choice)} & (\sigma,\ A \to B : k[l_i : C_i]_{i\in I}) \xrightarrow{\ \mathsf{sel}\ A\to B\ \mathsf{over}\ s{:}l_i\ } (\sigma,\ C_i) \\
\text{(G-Par)} & (\sigma,\ C_1) \xrightarrow{\ \ell\ } (\sigma',\ C_1') \quad \Rightarrow \quad (\sigma, C_1 \mid C_2) \xrightarrow{\ \ell\ } (\sigma',\ C_1' \mid C_2) \\
\text{(G-Struct)} & C \equiv C'' \ \wedge\ (\sigma,\ C) \xrightarrow{\ \ell\ } (\sigma',\ C') \ \wedge\ C' \equiv C''' \quad \Rightarrow \quad (\sigma,\ C'') \xrightarrow{\ \ell\ } (\sigma',\ C''') \\
\text{(G-Res)} & (\sigma,\ C) \xrightarrow{\ \ell\ } (\sigma',\ C') \qquad u \in \{a, k\} \quad \Rightarrow \quad (\sigma, (\nu u)\ .C) \xrightarrow{\ \ell\ } (\sigma',\ (\nu u)\ .C')
\end{array}
$$

Table 5.1: Operational Semantics for the Global Calculus

the following rules ($\equiv_\alpha$ denotes alpha equivalence on terms):

$$
\begin{array}{rcl}
C \equiv C' & & \text{if } C \equiv_\alpha C' \\
(\nu v)\ .C_1 | C_2 & \equiv & (\nu v)\ .(C_1 | C_2) \qquad \text{if } v \notin fn(C_2) \\
\mu X^A.C & \equiv & C[\mu X^A.C / X^A]
\end{array}
\tag{5.1}
$$

An action in the semantics is defined using an intuitive notation, $(\sigma, C) \xrightarrow{\ \ell\ } (\sigma', C')$ which says that a choreography C in a state σ (which is the collection of all local states of the participants) executes an action ℓ and evolves into C' with a new state σ' . This idea comes from the small-step semantics given to imperative languages. We will write $C \longrightarrow C'$ when the states σ, σ' and ℓ are irrelevant, and we will use $\longrightarrow^*$ as the transitive closure of $\longrightarrow$. The transition relation $\longrightarrow$ is defined as the minimum relation on pairs state/interaction satisfying the rules of Table 5.1.

Transition (G-Init) describes the evolution of a session initiation: after A initiates a session with B on service channel a, A and B share k locally. This will be denoted by a restriction of the session channel k over the continuation C, as denoted by $(\nu k)\ .C$. (G-Com) describes the main interaction rule of the calculus: the expression e is evaluated into v in the A-portion of the state σ and then assigned to the variable x located at B resulting in the new state $\sigma[x@B \mapsto v]$. (G-Choice) chooses the evolution of a choreography resulting from a labelled choice over a session key k. (G-IfT) and (G-IfF) show the possible paths that a deterministic evolution of a choreography can produce. (G-Par), (G-Res) (G-Rec) and (G-Struct) behave as the standard rules for parallel product, restriction, recursion and structural congruence.

In the sequel, we write $C \downarrow_\ell$ whenever either

- $\ell = \mathsf{init}\ A \to B\ \mathsf{on}\ a$ and $C \equiv (\nu \vec{s})\ (A \to B : a(s).C' \mid C'')$; or
- $\ell = \mathsf{com}\ A \to B\ \mathsf{over}\ s$ and $C \equiv (\nu \vec{s})\ (A \to B : s\langle e, x\rangle.C' \mid C'')$

- $\ell = \mathsf{sel}\ A \to B\ \mathsf{over}\ s : l_i$ and $C \equiv (\nu s)(A \longrightarrow B : s[l_i : C_i']_{i \in I} \mid C'')$

Similarly, we denote with $C \downarrow_\tau$ the evaluation of expressions: $C \downarrow_\tau$ if $C \equiv \mathbf{if}\ e@A\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2$ and $e@A \Downarrow \mathtt{tt}$ or $e@A \Downarrow \mathtt{ff}$.

5.2.3 Type discipline for the Global Calculus

Roughly speaking, the type discipline in [HVK98] ensures a correct pairing between actions and co-actions once a session is established. We use a generalisation of session types [HVK98] for global interactions, first presented in [CHY+09]. The grammar of types follows.

$$\alpha \stackrel{\mathtt{def}}{=} \Sigma_i s \downarrow op_i(\theta_i).\alpha_i \mid \Sigma_i s \uparrow op_i(\theta_i).\alpha_i \mid \alpha_1 \mid \alpha_2 \mid \mathtt{end} \mid \mu \mathbf{t}.\alpha \mid \mathbf{t} \tag{5.2}$$

where $\theta, \theta', \ldots$ range over value types $\mathtt{bool}, \mathtt{string}, \mathtt{int}, \ldots$. $\alpha, \alpha', \ldots$ are session types. $\Sigma_i s \downarrow op_i(\theta_i).\alpha_i$ is a branching input type at session channel s, indicating a process is ready to receive any of the (pairwise distinct) operators in $\{op_i\}$, each with a value of type θ_i; $\Sigma_i s \uparrow op_i(\theta_i).\alpha_i$ describes its co-type: a branching output type at s. Type $\alpha_1 \mid \alpha_2$ is a parallel composition of session types α_1 and α_2. The type $\mathtt{end}$ indicates session termination and is often omitted. We take $\mid$ to be commutative and associative with $\mathtt{end}$. $\mu \mathbf{t}.\alpha$ indicates a recursive type with $\mathbf{t}$ as a type variable. $\mu \mathbf{t}.\alpha$ binds the free occurrences of $\mathbf{t}$ in α. We take an *equi-recursive* view on types, not distinguishing between $\mu \mathbf{t}.\alpha$ and its unfolding $\alpha[\mu \mathbf{t}.\alpha/\mathbf{t}]$.

Session types in the global calculus are used to enforce a linear usage of the resources in the communication between participants. A typing judgment has the form $\Gamma \vdash C : \Delta$, where Γ, Δ are *service type* and *session type* environments, respectively. Typically, Γ contains a set of type assignments of the form $a@A : (\vec{k})\alpha$, which says that a service a located at participant A may be invoked with a fresh $\vec{s}$ followed with a session α. Δ contains types assignments of the form $\vec{k}[A, B] : \alpha$ which says that a vector of session channels $\vec{k}$, all belonging to the same session between participants A and B, has the session type α when seen from the viewpoint of A.

The typing rules are omitted, and we refer to [CHY+09] for the full account of the type discipline. As an example, take a simplified version of the booking scenario in section 4.4.1. Here, the customer establishes a session with the airline company AC using service *ob* and creating session keys k_1, k_2. Once sessions are established, the customer will request the company about a flight offer with his booking data, along the session key k_1. The airline company will process the customer request and will send a reply back with an offer using the session key k_2. The customer will eventually accept the offer, sending back an acknowledgment to the airline company using k_1. The following specification in the global calculus represents the protocol:

$$
\begin{aligned}
C_{OB} = & Cust \to AC : ob(k_1, k_2).\\
& Cust \to AC : k_1\langle booking, x\rangle.\\
& AC \to Cust : k_2\langle offer, y\rangle.\\
& Cust \to AC : k_1\langle accept, z\rangle.\mathbf{0}
\end{aligned} \tag{5.3}
$$

Then, the service type of the airline company at channel *ob* is described as:

$$
(k_1, k_2).k_1 \downarrow booking(\texttt{string}).k_2 \uparrow offer(\texttt{int}).k_1 \downarrow accept(\texttt{int}).\texttt{end} \tag{5.4}
$$

Remark 1. *From now on, we will consider only choreographies that respect the linearity conditions established by the typing discipline of the global calculus. All processes along this chapter are considered to respect the typing discipline in [CHY+09].*

5.3 A Logic for the Global Calculus

5.3.1 Syntax of the Logic.

In this section, we introduce a simple logic for choreography, inspired by the modal logic for session types presented in [BHY08]. The logical language comprises assertions for equality and value/name passing. The grammar of assertions is given in table 5.2.

ϕ	::=	$\langle \ell \rangle \phi$	(action)
	$\mid$	$\exists t.\phi$	(exists)
	$\mid$	$\texttt{tt}$	(true)
	$\mid$	$e_1 = e_2$	(equality)
	$\mid$	$\phi \wedge \chi$	(and)
	$\mid$	$\neg\phi$	(neg)
	$\mid$	$\circ\phi$	(next)
	$\mid$	$\Diamond\phi$	(may)
	$\mid$	$\phi * \chi$	(separation)
	$\mid$	$\phi \mathrel{-\!\!*} \chi$	(wand)
ℓ	::=	$\mathsf{init}\ A \to B\ \mathsf{on}\ a$	(init)
	$\mid$	$\mathsf{com}\ A \to B\ \mathsf{over}\ s$	(com)
	$\mid$	$\mathsf{sel}\ A \to B\ \mathsf{over}\ s : l$	(branch)

Table 5.2: Assertions of Choreography logic

Choreography assertions (ranged over by $\phi, \phi', \chi, \dots$) give a logical interpretation of the global calculus introduced in the previous section. The logic consists of the standard FOL

operators $\wedge$, $\neg$, $\mathtt{tt}$ and $\exists$. In $\exists t.\phi$, the variable t is meant to range over service and session channels, participants, labels and basic placeholders for expressions. Accordingly, it works as a binder in ϕ. In addition to the standard operators, we include an unspecified (decidable) equality on expressions ($e_1 = e_2$) as in [BHY08]. Our operators depend on the labels of the labelled transition system of the global calculus: $\langle \ell \rangle \phi$ represents the execution of a labelled action ℓ followed by the assertion ϕ; $\circ\phi$ and $\Diamond\phi$ denote the standard next and evenutally operators respectively. The spatial operator in $\phi * \chi$ denotes composition of formulae: because of the unique nature of parallel composition in choreographies, we have used the symbol $*$ (as in separation logic) in order to stress the fact that there is no interference between two choreographies running in parallel. $\phi \mathrel{-\!\!*} \chi$ is standardly defined accordingly.

Remark 2 (Derived Operators)**.** *We can get the full account of the logic by deriving the standard set of strong modalities from the above presented operators:*

- $\mathtt{ff} = \neg\mathtt{tt}$,
- $(e_1 \neq e_2) = \neg(e_1 = e_2)$,
- $\phi \vee \chi = \neg(\neg\phi \wedge \neg\chi)$,
- $\phi \Rightarrow \chi = \neg\phi \vee \chi$,
- $\forall x.\phi = \neg\exists x.\neg\phi$,
- $\Box\phi = \neg\Diamond\neg\phi$,
- $[\ell]\phi = \neg\langle\neg\ell\rangle\phi$.

The following examples give an intuition of how these modalities combined with the existential operator $\exists$ allow to express properties of choreographies.

Example 5.1 (Availability)**.** *We would like to express that, given a service requester (known as A in this setting) requesting the service a, there exists another participant in the choreography providing a and A is invoking him. This can be formulated in the logic as*

$$\exists x.\mathit{init}\ A \to x\ \mathit{on}\ a(s) \tag{5.5}$$

Example 5.2 (Service Usage)**.** *We want to ensure that services available are actually used. We can use the dual property for availability to specify that, for a service provider B offering a, there exists a service requester different from B that will invoke a.*

$$\exists x.\mathit{init}\ x \to B\ \mathit{on}\ a(s) \tag{5.6}$$

Example 5.3 (Coupling)**.** *We want to verify what is the level of connectedness between two different participants in a choreography. One way to specify this is to ask for possible services that this two participants are using in between. We can model that in the logic as follows:*

$$\exists x.\mathit{init}\ A \to B\ \mathit{on}\ x(s) \tag{5.7}$$

Example 5.4 (Causality Analysis)**.** *We can use the modal operators of the logic in order to perform studies of the causal properties that our specified choreography can fulfill. For instance, we can specify that given an expression e evaluated to true, then there is an eventual firing of a choreography that satisfies property ϕ_1, moreover, ϕ_2 will never be satisfied. Such a property can be specified as follows:*

$$(e = \mathtt{tt}) \wedge \Diamond(\phi_1) \wedge \Box\neg\phi_2 \tag{5.8}$$

Example 5.5 (Fair Response)**.** *An interesting aspect of our logic is that it allows for the declaration of partial specification properties regarding the interaction of the participants involved in a choreography. Take for instance the interaction diagram in Figure 5.2:*

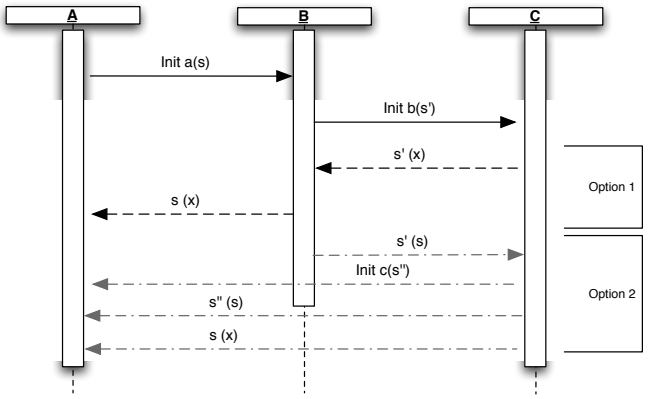

Figure 5.2: Alternatives for service synchronization

There, participant A invokes service b at B's and then B invokes C's service c. At this point, C can send the content of variable x to A in two different ways: either by using those originally established sessions, or by invoking a new service at A's. However, at the end of the either computation path, variable z (located at A's) will contain the value of x. In the global calculus:

$$C \;=\; A \longrightarrow B : b(k).B \longrightarrow C : c(k').\mathbf{if}\ e@C\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2 \qquad \textit{where}$$

$$\begin{aligned} C_1 &= \; C \longrightarrow B : k'\langle x, y_B\rangle.B \longrightarrow A : k\langle y_B, z\rangle \\ C_2 &= \; C \longrightarrow A : a(k'').C \longrightarrow A : k''\langle x, z\rangle \end{aligned}$$

We argue that, under the point of view of A, both options are sufficiently good if, after an initial interaction with B is established, there is an eventual response that binds the variable z as a response. Such a property can be expressed in the logic by the formula:

$$\exists X, k''.\langle \mathit{init}\ A \to B\ \mathit{on}\ a(k)\rangle\ \Diamond\Big(\langle \mathit{com}\ X \to A\ \mathit{over}\ k''\rangle\ \wedge\ z@A = x@C\Big) \tag{5.9}$$

Note that a third option for the protocol above is to use delegation. However, the current version of the global calculus does not feature such an operation and we leave it as future work. □

Remark 3 (Connectedness)**.** *The work in [CHY07] proposes a set of criteria for guaranteeing a safe end-point projection between global and local specifications (note that the*

choreography in the previous example does not respect such properties). Essentially, a valid global specification have to fulfill three different criteria, namely Connectedness, Well-threadedness and Coherence. It is interesting to see that some of this criteria relate to global and local causality relations between the interactions in a choreography, and can be easily formalized as properties in the choreography logic here presented. Below, we consider the notion of connectedness and leave the other cases as future work. Connectedness dictates a global causality principle in interaction. If A initiates any action (say sending messages, assignment, etc) as a result of a previous event (e.g. message reception), then such a preceding event should have taken place at A. In the sequel, let $\mathsf{Interact}(A, B)\phi$ *be a predicate which is true whenever* $\langle \ell \rangle \phi$ *holds for some* ℓ *with an interaction from* A *to* B*. Connectedness can then be specified as follows:*

$$\forall A, B.\Box\Big(\mathsf{Interact}(A, B)\mathtt{tt} \;\Rightarrow\; \exists C.\big(\mathsf{Interact}(A, B)\;\mathsf{Interact}(B, C)\;\mathtt{tt} \vee \mathsf{Interact}(A, B)\;\neg\langle \ell \rangle \mathtt{tt}\big)\Big)$$

□

Note that the above formalization is simpler than the definition given at [CHY06], that is because the logical characterization of choreographies allow us to study interactions in terms of their observable behaviour; further constructs (such as conditional behaviour, recursion, . . .) will derive only more observable behaviour, as presented in Section 5.4.

5.3.2 Semantics of the Logic.

We now give a formal meaning to the assertions introduced above with respect to the semantics of the global calculus introduced in the previous section. In particular we introduce the notion of satisfaction. We write $C \models_\sigma \phi$ whenever an environment σ and a choreography C satisfy a formula ϕ. The relation $\models$ is the minimum relation satisfying the rules given in Table 5.3.

$$\begin{array}{lll}
C \models_\sigma e_1 = e_2 & \text{iff} & \sigma(e_1) = \sigma(e_2) \\
C \models_\sigma \mathtt{tt} & \text{iff} & \mathtt{tt} \\
C \models_\sigma \phi \wedge \chi & \text{iff} & C \models_\sigma \phi \text{ and } C \models_\sigma \chi \\
C \models_\sigma \neg\phi & \text{iff} & C \not\models_\sigma \phi \\
C \models_\sigma \exists t.\phi & \text{iff} & C[w/t] \models \phi \quad (\text{for some appropriate } w) \\
C \models_\sigma \Diamond\phi & \text{iff} & C \longrightarrow^* C' \text{ and } C' \models_\sigma \phi \\
C \models_\sigma (\nu k)\,.\phi & \text{iff} & C \equiv (\nu k)C' \text{ and } C' \models_\sigma \phi \\
C \models_\sigma \circ\phi & \text{iff} & C \longrightarrow C' \text{ and } C' \models_\sigma \phi \\
C \models_\sigma \phi * \chi & \text{iff} & C \equiv C_1 \mid C_2 \text{ s.t. } C_1 \models_\sigma \phi \text{ and } C_2 \models_\sigma \chi \\
C \models_\sigma \langle \ell \rangle \phi & \text{iff} & C \xrightarrow{\ell} C' \text{ and } C' \models_\sigma \phi \\
C \models_\sigma \phi -\!\!* \chi & \text{iff} & \forall C_1 \text{ s.t. } C \mid C_1 \text{ well typed } \wedge\, C_1 \models_\sigma \phi \text{ then } C \mid C_1 \models_\sigma \chi
\end{array}$$

Table 5.3: Assertions of the Choreography Logic

Above, we assume that variables occurring in an expression e are always located e.g. $x@A$. In the $\exists t.\phi$ case, w should be an appropriate value according to the type of t e.g.

a participant if t is a participant placeholder. A formula ϕ is a "logical consequence" of a formula χ if every interpretation that makes ϕ true also makes χ true. In this case one says that χ is logically implied by ϕ ($\phi \Rightarrow \chi$).

Additional Definitions. A formula ϕ is a "logical consequence" of a formula χ if every interpretation that makes ϕ true also makes χ true. In this case one says that χ is logically implied by ϕ ($\phi \Rightarrow \chi$).

Definition 15 (Semantic Equivalence)**.** Let ϕ and χ two formulae in the choreography logic. We say that ϕ is semantically equivalent to χ (denoted by $\phi \equiv_{\models} \chi$) if $\phi \models \chi$ and $\chi \models \phi$ holds.

A sentence is satisfiable if there is some choreography under which it is true. A formula with free variables is said to be satisfied by a choreography if the formula remains true regardless which participants, names/values are assigned to its free variables. A formula ϕ is *valid* if it is true in every choreography, that is $\forall C; C \models \phi$.

5.4 Proof System

Here, the proof system is presented. In order to reason about judgments $C \models \phi$, we propose a proof (or inference) system for assertions of the form $C \vdash \phi$. Intuitively, we want $C \vdash \phi$ to be as approximate as possible to $C \models \phi$ (ideally, they should be equivalent). We write $C \vdash \phi$ for the provability judgement where C is a process and ϕ contains formulae composed from the grammar in Table 5.2.

Definition 16 ($C \vdash \phi$)**.** We say that $C \vdash \phi$ (C *exhibits* ϕ) iff the assertion $C \vdash \phi$ has a proof in the system given in Table 5.4.

$$\mathsf{P}_{\mathsf{init}}\ \frac{C \vdash \phi}{A \longrightarrow B : a(k).C \vdash \langle \mathsf{init}\ A \to B\ \mathsf{on}\ a \rangle \phi} \qquad \mathsf{P}_{\mathsf{par}}\ \frac{\forall_{i \in \{1,2\}} \quad C_i \vdash \phi_i}{C_1 \mid C_2 \vdash \phi_1 * \phi_2}$$

$$\mathsf{P}_{\mathsf{sel}}\ \frac{\forall_{i \in I} \quad C_i \vdash \phi_i}{A \longrightarrow B : k[l_i : C_i]_{i \in I} \vdash \bigwedge_{i \in I} \langle \mathsf{sel}\ A \to B\ \mathsf{over}\ l_i : s \rangle \phi_i} \qquad \mathsf{P}_{\mathsf{res}}\ \frac{C \vdash \phi \qquad x\ \text{is fresh}}{(\nu x)\, C \vdash \nu x.\phi}$$

$$\mathsf{P}_{\mathsf{com}}\ \frac{C \vdash \phi}{A \longrightarrow B : k\langle e, y \rangle.C \vdash \langle \mathsf{com}\ A \to B\ \mathsf{over}\ s \rangle \phi} \qquad \mathsf{P}_{\mathsf{sub}}\ \frac{C \vdash \phi \qquad \phi \Rightarrow \chi}{C \vdash \chi}$$

$$\mathsf{P}_{\mathsf{if}}\ \frac{C_1 \vdash e \Rightarrow \phi \qquad C_2 \vdash \neg e \Rightarrow \phi}{\mathbf{if}\ e\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \vdash\ \circ\, \phi} \qquad \mathsf{P}_{\mathsf{Inact}}\ \frac{-}{\mathbf{0} \vdash \mathtt{tt}}$$

Table 5.4: Proof system for the Global Calculus

Let us now describe some of the inference rules of the proof system. $\mathsf{P}_{\mathsf{inact}}$ and is the standard rule for inaction $\mathsf{P}_{\mathsf{sel}}$ can be explained as follows: suppose we are given a process $P = A \longrightarrow B : k[l_i : C_i]_{i \in I}$, a set of branch labels $\{l_i\}$ (determined by typing) and we are given a proof that each C_i satisfies ϕ_i, then we certainly have a proof saying that every derivation of P should satisfy a guard l_i followed by a formula ϕ_i.

The initiation and interaction rule $\mathsf{P}_{\mathsf{init}}, \mathsf{P}_{\mathsf{com}}$ behave similarly to $\mathsf{P}_{\mathsf{sel}}$: given an initiation/communication process in P and a proof that its continuation satisfies the proof term ϕ, we can derive a proof that P will first exhibit an initiation/communication action followed by ϕ. The conditional rule P_{if} is standard. The subsumption rule $\mathsf{P}_{\mathsf{sub}}$ is the standard consequence rule as found in Hoare logic.

The rules for parallel composition and hiding are represented in $\mathsf{P}_{\mathsf{par}}$ and $\mathsf{P}_{\mathsf{res}}$ respectively, and they do not indicate the behaviour of a given choreography, but hint information about the structure of the process: $\mathsf{P}_{\mathsf{par}}$ juxtaposes the behaviour of two processes and combines their respective formulae by the use of a separation operator, $\mathsf{P}_{\mathsf{res}}$ hides a variable x in a formula ϕ; the intuition is that since $(\nu x)\ C$ is a choreography C with x restricted, then if C proves ϕ and x is a fresh variable, then $(\nu x)\ C$ should satisfy ϕ with hidden x.

We now proceed to prove the soundness of the proof system with respect to the semantics of assertions presented before.

Lemma 6 (Structural equivalence preserves Logical validity)**.** If $C \equiv C'$ and $C \vdash \phi$, then $C' \vdash \phi$

Proof. It follows by trivial case analysis over $\equiv$. □

Theorem 5 (soundness)**.** for any given choreography C, if $C \vdash \phi$, then $C \models \phi$.

Proof. It follows by induction on the derivation of $\vdash$.

- Case $\mathsf{P}_{\mathsf{Inact}}$. We have that $C = \mathbf{0}$, then $\mathbf{0} \vdash \mathtt{tt}$ and $\mathbf{0} \models \mathtt{tt}$ immediately.
- Case $\mathsf{P}_{\mathsf{init}}$. We have that $C = A \longrightarrow B : a(k).C'$ and by $\mathsf{P}_{\mathsf{init}}$ then $\phi = \langle \mathsf{init}\ A \to B\ \mathsf{on}\ a \rangle \phi'$. We have to show that $C \models \langle \mathsf{init}\ A \to B\ \mathsf{on}\ a \rangle \phi'$.

  By definition, $C \models \langle \mathsf{init}\ A \to B\ \mathsf{on}\ a \rangle \phi'$ iff $C \downarrow_{\mathsf{init}\ A \to B\ \mathsf{on}\ a}$ and $\exists C'$ s.t. $C \longrightarrow^* C'$ and $C' \models \phi'$. Obviously, by applying (G-Init) on C we get $C \longrightarrow (\nu k)\ .C'$; by induction hypothesis we have that $C' \models \phi'$, so we can use the assertion of $(\nu x)\ .\phi'$ such that: $C \models (\nu x)\ .\phi'$ iff $C \equiv (\nu v).C'$ and $C' \models \phi(x \mapsto v)$, and by using lemma 6 we have that $(\nu x)\ .\phi \equiv_{\models} \phi$ with fresh x.

  To prove $C \downarrow_{\mathsf{init}\ A \to B\ \mathsf{on}\ a}$, take $\ell = \mathsf{init}\ A \to B\ \mathsf{on}\ a$, then we just have to check that $C \equiv (\nu s)\ .(A \longrightarrow B : a(s)).C' \mid C''$ which we already know by taking $C'' = \mathbf{0}$ and scope extension with (νs) and $C'' = \mathbf{0}$.
- Case $\mathsf{P}_{\mathsf{com}}$. We have that $C = A \longrightarrow B : s\langle e, y \rangle.C'$ and by induction hypothesis we have that $C' \models \phi'$. We have to show that $C \models \langle \mathsf{com}\ A \to B\ \mathsf{over}\ s \rangle \phi'$ given $C \vdash \langle \mathsf{com}\ A \to B\ \mathsf{over}\ s \rangle \phi'$.

  By definition, $C \models \langle \mathsf{com}\ A \to B\ \mathsf{over}\ s \rangle \phi'$ iff $C \downarrow_{\mathsf{com}\ A \to B\ \mathsf{over}\ s}$ and $\exists C'$ such that $C \longrightarrow^* C'$ and $C' \models \phi'$. To prove $C \downarrow_{\mathsf{com}\ A \to B\ \mathsf{over}\ s}$, we have that $C \downarrow_\ell$ if $\ell =$

$\mathsf{com}\ A \to B\ \mathsf{over}\ s$ and $C \equiv (\nu s)(A \to B : s\langle e, x\rangle.C' | C'')$, which is trivially true using $C'' = \mathbf{0}$ and $[y/x]$. Moreover, we can apply (G-COM) to get $C \longrightarrow C'$, and by induction hypothesis, $C' \models \phi'$ so we are done.

- Case $\mathsf{P_{sel}}$. We have that $C = A \to B : k[l_i : C'_i]_{i\in I}$ and $C \vdash \bigwedge_{i\in I}\langle \mathsf{sel}\ A \to B\ \mathsf{over}\ l_i : s\rangle\phi$. Because of $\mathsf{P_{sel}}$ we have that $\forall_{i\in I} C_i \vdash \phi_i$ and by induction hypothesis, $\forall_{i\in I}\, C_i \models \phi_i$. We have to show that $C \models \bigwedge_{i\in I}\langle \mathsf{sel}\ A \to B\ \mathsf{over}\ l_i : s\rangle\phi_i$.

  By definition of the semantics of $\langle\ell\rangle\phi$ and conjunction, then:

$$C \models \bigwedge_{i\in I}\langle \mathsf{sel}\ A \to B\ \mathsf{over}\ l_i : s\rangle\phi_i \text{ iff } \bigwedge_{i\in I}(C \downarrow_{\mathsf{sel}\ A\to B\ \mathsf{over}\ l_i:s} \wedge \exists C' \text{ s.t. } C \longrightarrow^* C' \wedge C' \models \phi_i)$$

  Take $\ell = \mathsf{sel}\ A \to B\ \mathsf{over}\ l_i : s$ for $i \in I$, then $C \models \bigwedge_{i\in I}\langle\ell\rangle\phi_i$ if $\ell = \mathsf{sel}\ A \to B\ \mathsf{over}\ l_i : s$ (trivially true) and if $C \equiv (\nu s)(A \to B : s[l_i : C'_i]_{i\in I} | C'')$ which is equivalent to C up-to substitution $[s/k]$. Moreover, $\exists C'$ s.t. $C \longrightarrow^* C'$ and $C' \models \phi_i$ holds by induction hypothesis, so we are done.

- Case $\mathsf{P_{if}}$. We have that $C = \mathbf{if}\ e\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2$, by induction hypothesis we have that $C_1 \vdash e \Rightarrow \phi$ and $C_2 \vdash \neg e \Rightarrow \phi$. We have to show that $C \models \circ\phi$. Assume a σ s.t. $\sigma \vdash e@A \Downarrow \mathtt{tt}$ (The other case is symmetric), then we get by the use of (G-IFT) with σ and C that $(\sigma, C) \longrightarrow (\sigma, C_1)$. Additionally we have that:

$$\begin{aligned} &C_1 \vdash e@A \Downarrow \mathtt{tt} \Rightarrow \phi \\ &C_1 \vdash \mathtt{ff} \vee \phi \\ &C_1 \vdash \phi \end{aligned}$$

  Then is easy to find that $C \downarrow_\tau$ and that $C_1 = C'$ in $\exists C' \in \{\![\, \phi \,]\!\}$ s.t. $C \longrightarrow C'$, so we are done.

- Case $\mathsf{P_{res}}$. we have that $C = (\nu x)\ .C'$ and by induction hypothesis we have that $C' \vdash \phi$ and with x fresh variable. We have to show that $C \models \phi$ iff $C \equiv (\nu u)\ .C'$ and $C' \in \{\![\, \phi(x \mapsto v) \,]\!\}$. This follows straightforwardly: With x fresh, we know that $C \equiv (\nu u)\ C'[x/u]$. Similarly, from $C' \vdash \phi$ and fresh x and lemma 6 we know that $(\nu u)\ .C' \models \phi(x \mapsto u)$, so we are done.

- Case for $\mathsf{P_{par}}$: Immediate.

□

5.5 Future work

The work here presented constitutes just the first from the initial steps towards a verification framework of structured communications. Our main concerns relate to (i) establish a completeness relation between the choreography logic and its proof system, (ii) the ability of integrate the proof system here presented with other logical frameworks for the specification of sessions, and (iii) the ability of reasoning about partial information within the

framework. In [BHY08], Berger et al. presented proof systems characterizing May/Must testing preorders and bisimilarities over typed π-calculus processes. The connection between types and logics in such system comes in handy to restrict the shape of the processes one might be interested, allowing us to consider such work as a suitable proof system for the calculus of end points. Moreover, a connection between the global logic here presented and the modal logic used in typed π-specifications results necessary, allowing us to project a property satisfied in a choreography into sets of properties in π-calculus specifications. Other improvements to the system proposed include the use of fixed points, essential for describing state-changing loops, and auxiliary axioms describing not only structural properties of a choreography, but also expected results given by the interaction of one or more choreographies.

6 Concluding Remarks

Along this report we have discussed different approaches for leveraging the confidence of a service oriented specification. Instead of providing an unified model, we followed ideas inspired in type systems, concurrency theory and modal logics, gaining a clear understanding on what a service oriented system is composed and the requirements that such specifications should fulfill.

Chapter 3 started explored the trustworthiness of a system in terms of the security of its communications. Given the `utcc` process calculus (introduced in Chapter 2), we showed why the unrestricted abstraction present in the calculus allows for the leakage of secure data in a communication, and we proposed a type system for constraints used as patterns in process abstractions, which essentially allows us to distinguish between public information and secure (non-leakable information) inside predicates. A link between the Concurrent Constraint family of languages and languages for service oriented systems has been drawn in Chapter 4, where a mapping between HVK and `utcc` has been proposed. Such a link allows for a declarative study of structured communications using linear temporal logic, and extends current languages for sessions with explicit constructions for timeouts and session abortion. Finally, Chapter 5 moves away from the tradition of Concurrent Constraint Programming but tackles a similar component in the specification of services: the description of interactions in terms of choreographies. we reported initial steps towards a methodology for the specification and verification of structured communications. We presented a way to describe properties over global specifications. We introduced a proof system that allows for verification of properties among participants in a choreography. With such a logic, one can see the state of a choreography as a formula, and one can check for satisfaction of desirable properties. Some examples of important properties on structured communications are drawn, and we provide hints on how this work can be extended towards a full verification framework for structured communications, closing the gap between logics for choreography and their correspondent parts over an end-point projection.

6.1 Future Directions

The following ideas emerge as directions for future work:

- Logics for Choreography.

  Certainly the work presented in Chapter 5 represents the first and the clearest of in our paths for future work. We want to complete the verification framework for chore-

ographies, expanding the choreography logic with completeness results that allow us to guarantee that each formula in the logic is provable, and project choreographic formulae into sets of specification formulae for its end point projections.

- Nominal Concurrent Constraint Programming.

  One of the long-withstanding goals in the research of CCP languages has been the correct representation of *mobile* behaviour over concurrent constraint programs. When referring to mobile behaviour we can consider either *link mobility*, the ability of the network to reconfigure the connections between nodes, or *process mobility*, the ability to reconfigure the topology of the network. Link mobility has been modelled in `utcc` [OV08a]. In the classical setting, CCP-like calculi have modelled the logical view of a restriction operator as an existential quantifier over a constraint store. By this formulation one can say that a variable x is *private* from the constraint store c as no other process can know the contents of x in $\exists x\ \mathsf{c}$, except the one that imposed the constraint. In [PSVV06] it has been noticed that such a logical characterization of name restriction using the existential quantifier does not ensure uniqueness in the fragment of the π-calculus with mismatch: given $\exists x\, \exists y\, \mathsf{c}(\mathsf{x}, \mathsf{y})$ we cannot say that x is different than y, therefore the freshness of name generation cannot be guaranteed (as previously discussed in Chapter 3).

  Given the importance of freshness and uniqueness conditions when dealing with sessions, we aim for a reformulation of the name hiding on cc-calculi using a different conception. For doing so, we started working in a new variant of cc-calculi, herewith called *Fresh CCP*, to deal with name generation. In fresh CCP, fresh name generation is achieved by a redefinition of the underlying constraint system and the denotational model of CCP with the use of *nominal logic* [Pit03], an extension of first order logic with bundled notions of name swapping and fresh terms.

- Context - Sensitive Services.

  Research in context-sensitive services aims at providing IT-technology operating across and adapting to different contexts, e.g. location, time, activity, personal preferences, history, gender, language/culture or work/private life. The simultaneous emergence of context-sensitive services and the widespread use of mobile devices (phones, music players, navigators etc.) hold both great promise and great challenges. Namely, the promise of ubiquitous computing, and the dual challenges of finding human-computer interaction methods, and advancing development technology for service-oriented architectures to also support the notion of context. Following the research line proposed in the Genie-Jingling project [STG+10], we started work on the development of Context Sensitive type systems. The idea is that dynamic orchestration of services are composed by parametric services. A parametric service is a service oriented system with parametric interfaces and explicit dependencies. The objective of providing such dynamicity within orchestration is to equip current service specifications with versatile ways of composing new services given the resources already deployed in a context. We are currently working in the definition of a calculus for dynamically orchestrated parametric services, and type system for such a language that excludes incorrect dependencies and features progress of well-typed service orchestrations. Moreover, currently we are working on a implementation of a type checker where the theory presented can be tested.

Bibliography

[ACD+03] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, et al. Business process execution language for web services, version 1.1. *Standards proposal by BEA Systems, International Business Machines Corporation, and Microsoft Corporation*, 2003.

[AF01] Martín Abadi and Cédric 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.

[AG99] Martín Abadi and Andrew D. Gordon. A Calculus for Cryptographic Protocols: The SPi Calculus. *Inf. Comput.*, 148(1):1–70, 1999.

[BBC+06] M. Boreale, R. Bruni, L. Caires, R. De Nicola, I. Lanese, M. Loreti, F. Martins, U. Montanari, A. Ravara, and D. Sangiorgi. SCC: a service centered calculus. *Proceedings of WS-FM*, 4184:38–57, 2006.

[BHY01] Martin Berger, Kohei Honda, and Nobuko Yoshida. Sequentiality and the Pi-Calculus. In S. Abramsky, editor, *TLCA 2001*, volume 2044 of *Lecture Notes in Computer Science*, pages 29–45. Springer, Berlin Heidelberg, 2001.

[BHY08] Martin Berger, Kohei Honda, and Nobuko Yoshida. Completeness and logical full abstraction in modal logics for typed mobile processes. In Luca Aceto, editor, *ICALP'08*, number 5126 in LNCS, pages 99–111. Springer-Verlag, Berlin Germany, 2008.

[Bla01] Bruno 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, Los Alamitos (2001).

[BM07] Maria Grazia Buscemi and Ugo Montanari. CC-Pi: A Constraint-Based Language for Specifying Service Level Agreements. *16th European Symposium on Programming (ESOP'07)*, 2007.

[BRNN04] 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.

[CDC09] Mario Coppo and Mariangiola Dezani-Ciancaglini. Structured Communications with Concurrent Constraints. In *TGC'08*, volume 5474 of *LNCS*, pages 104–125. Springer, 2009.

[CE02] 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 2477 of *LNCS*, pages 326–341, Madrid, Spain, Sep 2002. Springer, Heidelberg.

[CHY06] M. Carbone, K. Honda, and N. Yoshida. A calculus of global interaction based on session types. In *2nd Workshop on Developments in Computational Models (DCM)*, ENTCS, 2006.

[CHY07] M. Carbone, K. Honda, and N. Yoshida. Structured communication-centred programming for web services. In *16th European Symposium on Programming (ESOP'2007), held as part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007*, volume 4421 of *Lecture Notes in Computer Science*, pages 2–17, Braga, Portugal, March 24–April 1 2007. Springer, Berlin Heidelberg.

[CHY+09] M. Carbone, K. Honda, N. Yoshida, R. Milner, G. Brown, and S. Ross-Talbot. A Theoretical Basis of Communication-Centred Concurrent Programming. *Web Services Choreography Working Group mailing list, to appear as a WS-CDL working report*, 2009.

[CW01] Federico Crazzolara and Glynn Winskel. Events in security protocols. In *ACM Conference on Computer and Communications Security*, pages 96–105, 2001.

[DRV98] J.F. Diaz, C. Rueda, and F. Valencia. A calculus for concurrent processes with constraints. *CLEI Electronic Journal*, 1(2), 1998.

[DY81] Danny Dolev and Andrew C. Yao. On the security of public key protocols. Technical report, Dept. of Computer Science, Stanford University, Stanford, CA, USA, 1981.

[Eme91] E.A. Emerson. Temporal and modal logic. In *Handbook of theoretical computer science (vol. B)*, page 1072. MIT Press, 1991.

[FA01] M. Fiore and M. Abadi. Computing symbolic models for verifying cryptographic protocols. *Proc. 14th IEEE Computer Security Foundations Workshop*, pages 160–173, 2001.

[HL09] Thomas Hildebrandt and Hugo A. López. Types for Secure Pattern Matching with Local Knowledge in Universal Concurrent Constraint Programming . In P.M. Hill and D.S. Warren, editors, *International Conference on Logic Programming (ICLP)*, volume 5649 of *Lecture Notes in Computer Science*, pages 417–431. Springer, Berlin Heidelberg, 2009.

[HM80] M. Hennessy and R. Milner. On Observing Nondeterminism and Concurrency. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, pages 299–309. Springer-Verlag London, UK, 1980.

[HVK98] K. Honda, V.T. Vasconcelos, and M. Kubo. Language Primitives and Type Discipline for Structured Communication-Based Programming. *Proceedings of the 7th European Symposium on Programming: Programming Languages and Systems*, pages 122–138, 1998.

[L+01] F. Leymann et al. Web services flow language (WSFL 1.0), 2001.

[LHM08] Karen Marie Lyng, Thomas Hildebrandt, and Raghava Rao Mukkamala. The Resultmaker Online Consultant: From Declarative Workflow Management in Practice to LTL. In *In Proc. of 1st International Workshop on Dynamic and Declarative Business Processes (DDBP 2008)*, Munich, Germany, 2008.

[LOP09] Hugo A. López, Carlos Olarte, and Jorge A. Pérez. Towards a Unified Framework for Declarative Structured Communications. In *Programming Language Approaches to Concurrency and Communication-cEntric Software: PLACES'09*, February 2009.

[Low95] Gavin Lowe. An attack on the needham-schroeder public-key authentication protocol. *Inf. Process. Lett.*, 56(3):131–133, 1995.

[LPP+06] Hugo A. López, Jorge A. Pérez, Catuscia Palamidessi, Camilo Rueda, and Frank D. Valencia. A declarative framework for security: Secure concurrent constraint programming. In Sandro Etalle and M. Truszczyński, editors, *22th International Conference on Logic Programming (ICLP'06)*, volume 4079 of *Lecture Notes in Computer Science*. Springer, Heidelberg, 2006.

[LPT07] A. Lapadula, R. Pugliese, and F. Tiezzi. A calculus for orchestration of web services. In *Proc. of 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *Lecture Notes in Computer Science*, pages 33–47. Springer, 2007.

[LVMR07] I. Lanese, V.T. Vasconcelos, F. Martins, and A. Ravara. Disciplining Orchestration and Conversation in Service-Oriented Computing. *Fifth IEEE International Conference on Software Engineering and Formal Methods (SEFM'2007)*, pages 305–314, 2007.

[Mil95] Robin Milner. *Communication and concurrency*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1995.

[Mil99] Robin Milner. *Communicating and Mobile systems. The Pi Calculus*. Cambridge University Press, 1999.

[Mil03] Dale 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, Heidelberg, 2003.

[MP92] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems: Specification*. Springer, 1992.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. *Journal of Information and Computation*, 100:1–77, September 1992.

[NPS05] Anders Kaare Nørgaard, Lars Pedersen, and Peter Strøiman. Method for generating a workflow on a computer, and a computer system adapted for performing the method. Patent, 05 2005. US 6895573.

[NPV02] Mogens Nielsen, Catuscia Palamidessi, and Frank Valencia. Temporal concurrent constraint programming: Denotation, logic and applications. *Nordic J. of Computing*, 2002.

[OV08a] Carlos A. Olarte and Frank D. Valencia. Universal concurrent constraint programming: Symbolic semantics and applications to security. In *23rd Annual ACM Symposium on Applied Computing (2008)*, 2008.

[OV08b] Carlos Alberto Olarte and Frank 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, New York.

[Pit03] A.M. Pitts. Nominal logic, a first order theory of names and binding. *Information and computation*, 186(2):165–193, 2003.

[PSVV06] C. Palamidessi, V. Saraswat, F.D. 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.

[PvdA06] M. Pesic and W.M.P. van der Aalst. A Declarative Approach for Flexible Business Processes Management. *Lecture Notes in Computer Science*, 4103:169, 2006.

[PW05] F. Puhlmann and M. Weske. Using the Pi-Calculus for Formalizing Workflow Patterns. *BPM*, 3649:153–168, 2005.

[Sar93] V. Saraswat. *Concurrent Constraint Programming*. MIT Press, 1993.

[SJG94] Vijay A. Saraswat, Radha Jagadeesan, and Vineet Gupta. Foundations of timed concurrent constraint programming. In *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science (LICS 1994)*, pages 71–80. IEEE Computer Society Press, July 1994.

[STG+10] Jørgen Staunstrup, Frank Tung, Arne Glenstrup, Thomas Hildebrandt, Li Weiping, Lin Huiping, Yu Lian, and Søren Debois. The jingling - genies project. http://sites.google.com/site/jinglinggenie/Home, January 2010.

[SW01] Davide Sangiorgi and David Walker. *PI-Calculus: A Theory of Mobile Processes*. Cambridge University Press, New York, NY, USA, 2001.

[Tha01] S. Thatte. XLANG: web services for business process design, 2001. *Microsoft http://www.gotdotnet.com/team/xml-wspecs/xlang-cl/default.htm*, 2001.

[VCS08] H.T. Vieira, L. Caires, and J.C. Seco. The Conversation Calculus: A Model of Service-Oriented Computation. In *Programming languages and systems: 17th European Symposium on Programming, ESOP 2008, held as part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008: proceedings*, page 269. Springer-Verlag New York Inc, 2008.

[vdA98] W.M.P. van der Aalst. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.

[vdAP06] W.M.P. van der Aalst and M. Pesic. DecSerFlow: Towards a Truly Declarative Service Flow Language. *Lecture Notes in Computer Science*, 4184:1, 2006.

[VP98] Björn Victor and Joachim Parrow. Concurrent constraints in the fusion calculus. In *Proc. of ICALP*, volume 1443 of *LNCS*, pages 455–469. Springer, 1998.