

Towards a Unified Framework for Declarative Structured Communications

Hugo A. López
IT University of Copenhagen
hual@itu.dk

Carlos Olarte
INRIA and LIX, École Polytechnique
colarte@lix.polytechnique.fr

Jorge A. Pérez
Università di Bologna
perez@cs.unibo.it

Abstract

We describe ongoing work on the definition of a formal framework for the declarative analysis of structured communication. By relying on a (timed) concurrent constraint programming language, we show that in addition to the usual behavioral techniques from process calculi, session analysis can elegantly exploit logic based reasoning techniques, all in a single framework. As a preliminary result, we report a concurrent constraint interpretation of a language for structured communication proposed by Honda, Vasconcelos and Kubo. Distinguishing features of our approach are: the possibility of including partial information (constraints) in the session model; the use of explicit time for reasoning about session duration and expiration; a tight correspondence with logic, which formally relates session execution and linear-time temporal logic formulas.

1 Introduction

Motivation. From the viewpoint of *reasoning techniques*, two main trends in modeling Service Oriented Computing (SOC) can be singled out. On the one hand, a *behavioral approach* focuses on how process interactions can lead to correct configurations. Typical representatives of this approach (as, e.g., [8, 1, 7, 14]) are based on process calculi and Petri nets, 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 (e.g. [14]). 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 behavioral and declarative techniques can harmoniously converge is therefore a legitimate research direction. In this paper we shall argue that Concurrent Constraint Programming (CCP) [17] can provide solid foundations for such an approach. Indeed, we find that the unified framework for behavioral and logic techniques that CCP provides can be fruitfully exploited for analysis in SOC, and could complement well other approaches, such as those based on 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 *declarative* sessions.

CCP [17] 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 piece of partial information to the store (thus making it available for other processes), the latter allows for rich, parameterizable forms of process synchronization. Interaction in CCP is thus inherently *asynchronous*, and can be related to a broadcast-like communication discipline, as opposed to the point-to-point communication in process calculi such as the π -calculus [15]. In CCP, the store grows monotonically, i.e., constraints cannot be removed. This condition is relaxed in *timed* extensions of CCP (e.g., [16, 11]), where processes evolve along a series of *discrete time intervals*. Each interval contains its own store and information is not automatically transferred from one interval to another. In this paper we shall use a CCP process language that is timed in this sense.

In addition to the traditional operational view of process calculi, CCP enjoys a *declarative* view that distinguishes it from other models of concurrency: CCP programs can be seen, at the same time, as computing agents and as logic formulas [17, 11, 12], 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 temporal logic formulas; we shall exploit this correspondence to verify properties of structured communications.

This Work. We describe ongoing work on the definition of a formal framework for the declarative analysis of sessions. We shall exploit *utcc*, a timed CCP process calculus [13], to give an alternative interpretation to the language defined by Honda, Vasconcelos and Kubo in [6] (henceforth referred to as HVK). This way, structured communications can be studied in a declarative framework in which time is explicit. We begin by proposing an encoding of the HVK language into *utcc*; such an encoding defines asynchronous session establishment and satisfies a rather standard operational correspondence property. We then move to the timed setting, and propose HVK^T , a timed extension of the HVK language. 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 show that the encoding of HVK into *utcc* straightforwardly extends to HVK^T . Finally, we show the applicability of a declarative characterization of structured communications by relating processes specifications with linear temporal logic (FLTL).

An extended version of this work, including the appendices herewith referred, is available at [9].

A Compelling Example. We now give intuitions on how a declarative approach could be useful in the analysis of sessions. Consider the ATM example from [6, Sect. 4.1]. There, an ATM has established sessions with a user and his/her bank; it allows for *deposit*, *balance*, and *withdraw* operations. In the latter case, if there is not enough money to withdraw, then an *overdraft* message appears to the user. It would be interesting to verify what occurs when the example is extended with a malicious card reader that keeps the user’s sensible information and uses it to continue withdrawing money without his/her authorization. A greedy card reader could even repeatedly withdraw until causing an overdraft.

In this simple scenario, the correspondence between *utcc* and linear temporal logic may come in handy to reason about the possible states for this specification. These can be used not only to describe the operational behavior of the compromised ATM above, but also to provide declarative arguments regarding its the evolution. For instance, assuming a global channel *out* and an overdraft message M_O , one could show that a *utcc* specification of the ATM example satisfies the FLTL formula $\diamond \text{out}(M_O)$, which intuitively means that in the presence of the malicious card reader the user’s bank account will eventually go to overdraft.

Related Work. One approach to combine the declarativeness 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., [18, 4]). 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 [2] and the calculus for structured communication in [3]. Such languages feature elements that resemble much ideas underlying CCP (especially [2]). The main difference between such works and our approach is that the reasoning techniques they feature are different from logic-based ones. In [2], a language for Service-Level Agreement (SLA) is proposed, featuring constructs for name-passing, constraint retraction and soft constraints. There, the reasoning techniques are essentially operational. In [3] a language for sessions featuring constraints is proposed. There, the key for analysis is represented by a type system which provides consistency for session execution, much as in the original approach in [6].

2 Preliminaries

We begin by introducing HVK, the language for structured communication proposed in [6]. We assume the following notational conventions: *names* are ranged over by a, b, \dots ; *channels* are ranged over by k, k' ; *variables* are ranged over by x, y, \dots ; *constants* (names, integers, booleans) are ranged over by

c, c', \dots ; *expressions* (including constants) are ranged over by e, e', \dots ; *labels* are ranged over by l, l', \dots ; *process variables* are ranged over by X, Y, \dots . Finally, u, u', \dots denote names and channels.

Definition 1 (The HVK language [6]). *Processes in HVK are built from:*

P, Q	$::=$	request $a(k)$ in P	<i>Session Request</i>		accept $a(x)$ in P	<i>Session Acceptance</i>
		$k![\vec{e}]; P$	<i>Data Sending</i>		$k?(x)$ in P	<i>Data Reception</i>
		$k \triangleleft l; P$	<i>Label Selection</i>		$k \triangleright \{l_1 : P_1 \parallel \dots \parallel l_n : P_n\}$	<i>Label Branching</i>
		throw $k[k']; P$	<i>Channel Sending</i>		catch $k(k')$ in P	<i>Channel Reception</i>
		if e then P else Q	<i>Conditional Statement</i>		$P \mid Q$	<i>Parallel Composition</i>
		inact	<i>Inaction</i>		$(\nu u)P$	<i>Hiding</i>
		def D in P	<i>Recursion</i>		$X[\vec{e}\vec{k}]$	<i>Process Variables</i>
D	$::=$	$X_1(x_1k_1) = P_1$ and \dots and $X_n(x_nk_n) = P_n$				

Reduction for HVK processes, denoted by \rightarrow , is defined as the smallest relation generated by a set of rules which we omit for space reasons (see [6]). As usual, \rightarrow^* is the reflexive, transitive closure of \rightarrow .

2.1 Timed Concurrent Constraint Programming

Timed concurrent constraint programming (tcc) [16] extends CCP for modeling reactive systems. In tcc , time is conceptually divided into *time units* or *time intervals*. In a particular time interval, a tcc process P gets an input 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 interval. It is worth noticing that the final store is not automatically transferred to the next time unit.

The utcc calculus [13] extends tcc for mobile reactive systems. Here *mobility* is understood as the dynamic reconfiguration of system linkage through communication, much like in the π -calculus [15]. utcc generalizes tcc by considering a *parametric* ask operator of the form $(\mathbf{abs} \vec{x}; c)P$, with the following intuitive meaning: process $P[\vec{t}/\vec{x}]$ is executed for every term \vec{t} such that the current store entails $c[\vec{t}/\vec{x}]$. This process can be viewed as an *abstraction* of the process P on the variables \vec{x} under the constraint (or with the *guard*) c .

utcc provides interesting reasoning techniques: First, utcc processes can be represented as partial closure operators (idempotent and extensive functions). Moreover, 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 [12]. Second, utcc processes can be characterized as FLTL formulas [10]. This declarative view of the processes allows for the use of the well-established verification techniques from FLTL to reason about utcc processes.

Syntax. Processes in utcc are parametric in a *constraint system* [17] which specifies the basic constraints that agents can tell or ask during execution. 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., $x > 42 \vdash x > 0$). The syntax of the language is as follows:

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

with the variables in \vec{x} being pairwise distinct.

A process \mathbf{skip} does nothing; process $\mathbf{tell}(c)$ adds c to the store in the current time interval. A process $Q = (\mathbf{abs} \vec{x}; c)P$ binds the variables \vec{x} in P and c . It executes $P[\vec{t}/\vec{x}]$ for every term \vec{t} such that the current store entails $c[\vec{t}/\vec{x}]$. When the vector of variables \vec{x} is empty, we retrieve the standard tcc ask operator **when** c **do** P . $P \parallel Q$ denotes P and Q running in parallel during the current time interval. A process $(\mathbf{local} \vec{x}; c)P$ binds the variables \vec{x} in P by declaring them private to P under a constraint c . The *unit-delay*

next P executes P in the next time interval. The *time-out unless* c **next** P is also a unit-delay, but P is executed in the next time unit iff c is not entailed by the final store at the current time interval. Finally, the *replication* $!P$ stands for $P \parallel \mathbf{next}P \parallel \mathbf{next}^2P \parallel \dots$, i.e., unboundedly many copies of P but one at a time. We shall use $!_{[n]}P$ to denote $P \parallel \mathbf{next}P \parallel \dots \parallel \mathbf{next}^{n-1}P$.

From a programming language perspective, \vec{x} in $(\mathbf{abs} \vec{x}; c)P$ can be seen as the formal parameters of P . This way, *recursive definitions* of the form $X(\vec{x}) \stackrel{\text{def}}{=} P$ can be encoded in `utcc` as

$$\mathcal{R}[\![X(\vec{x}) \stackrel{\text{def}}{=} P]\!] = !(\mathbf{abs} \vec{x}; \text{call}_x(\vec{x}))\widehat{P} \quad (1)$$

where 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 $\mathbf{tell}(\text{call}_x(\vec{t}))$. Similarly, calls of the form $X(\vec{t})$ in other processes are replaced with $\mathbf{tell}(\text{call}_x(\vec{t}))$.

Operational Semantics. The structural operational semantics (SOS) of `utcc` is given by two transition relations (see [9, Appendix B]): The *internal transition* $\langle P, d \rangle \longrightarrow \langle P', d' \rangle$ informally means “ P with store d reduces, in one internal step, to P' with store d' ”. And the *observable transition* $P \xrightarrow{(c,d)} R$ meaning “ P on input c , reduces in one *time unit* to R and outputs d ”. The latter is obtained from a finite sequence of internal transitions. They realize the operational intuitions given above (see [12] for details).

We shall write $P \Longrightarrow Q$ when $P \xrightarrow{(\text{true},c)} Q$ when c is unimportant. Let $s = c_1.c_2\dots.c_n$ be a sequence of constraints. If $P = P_1 \xrightarrow{(\text{true},c_1)} P_2 \xrightarrow{(\text{true},c_2)} \dots P_n \xrightarrow{(\text{true},c_n)} P_{n+1} = Q$ then we write $P \xrightarrow{s}^* Q$ ($P \Longrightarrow^* Q$, when s is unimportant). The *output behavior* of P is defined as $o(P) = \{s \mid P \xrightarrow{s}^*\}$. We shall write $P \sim^o Q$ if $o(P) = o(Q)$.

Logic correspondence. In the CCP spirit, `utcc` constructs admit a logic interpretation. The encoding below maps `utcc` processes into FLTL formulas. Theorem 1 relates this interpretation with the SOS.

Definition 2. Let $\llbracket \cdot \rrbracket$ be a map from `utcc` processes to FLTL formulas given by:

$$\begin{array}{llll} \llbracket \mathbf{skip} \rrbracket & = \text{true} & \llbracket \mathbf{tell}(c) \rrbracket & = c & \llbracket P \parallel Q \rrbracket & = \llbracket P \rrbracket \wedge \llbracket Q \rrbracket \\ \llbracket (\mathbf{abs} \vec{y}; c)P \rrbracket & = \forall \vec{y}(c \Rightarrow \llbracket P \rrbracket) & \llbracket (\mathbf{local} \vec{x}; c)P \rrbracket & = \exists \vec{x}(c \wedge \llbracket P \rrbracket) & \llbracket \mathbf{next}P \rrbracket & = \circ \llbracket P \rrbracket \\ \llbracket \mathbf{unless} c \mathbf{next}P \rrbracket & = c \vee \circ \llbracket P \rrbracket & \llbracket !P \rrbracket & = \Box \llbracket P \rrbracket & & \end{array}$$

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

Theorem 1 (Logic correspondence [13]). Let $\mathcal{L}[\llbracket \cdot \rrbracket]$ be as in Definition 2 and $s = c_1.c_2.c_3\dots$ such that $P \xrightarrow{s}^*$. For every constraint d , it holds that: $\mathcal{L}[\llbracket P \rrbracket] \vdash \Diamond d$ iff there exists $i \geq 1$ such that $c_i \vdash d$.

Derived Constructs [9]. Let out be an uninterpreted predicate. One could attempt at representing the actions of sending and receiving as in a name-passing calculus (say, $\bar{k}(\vec{e}).P$ and $k(\vec{x}).P$, resp.) with the `utcc` processes $\mathbf{tell}(\text{out}(k, \vec{e}))$ and $(\mathbf{abs} \vec{x}; \text{out}(k, \vec{x}))P$, resp. Nevertheless, these processes are not automatically transferred from one time-unit to the next one. Thus, they will disappear right after the current time unit even if they did not interact. Consequently, we shall use the process $(\mathbf{wait} \vec{x}; c) \mathbf{do} P$ (written $(\mathbf{whenever} c) \mathbf{do} P$ when $|\vec{x}| = 0$) 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, after interacting with an output. When it happens, it outputs the constraint $\bar{c}[\vec{t}/\vec{x}]$ acknowledging the successful read of c . Similarly, we define $\mathbf{tell}(c)$ for the persistent output of c until some process reads c .

3 A Declarative Interpretation for Sessions

Here we present a compositional encoding of HVK into `utcc`. The encoding, given in Table 3, is defined over *well-typed* HVK processes. Let us briefly provide intuitions on it. Consider HVK processes $P =$

$\llbracket \mathbf{request} \ a(k) \ \mathbf{in} \ P \rrbracket$	$=$	$(\mathbf{local} \ k) (\mathbf{tell}(\mathbf{req}(a, k)) \parallel \mathbf{whenever} \ \mathbf{acc}(a, k) \ \mathbf{do} \ \mathbf{next} \ \llbracket P \rrbracket)$
$\llbracket \mathbf{accept} \ a(k) \ \mathbf{in} \ P \rrbracket$	$=$	$(\mathbf{wait} \ k; \mathbf{req}(a, k)) \ \mathbf{do} \ (\mathbf{tell}(\mathbf{acc}(a, k)) \parallel \mathbf{next}(\llbracket P \rrbracket))$
$\llbracket k![\vec{e}]; P \rrbracket$	$=$	$\mathbf{tell}(\mathbf{out}(k, \vec{e})) \parallel \mathbf{whenever} \ \overline{\mathbf{out}(k, \vec{e})} \ \mathbf{do} \ \mathbf{next} \ \llbracket P \rrbracket$
$\llbracket k?(x) \ \mathbf{in} \ P \rrbracket$	$=$	$(\mathbf{wait} \ x; \mathbf{out}(k, x)) \ \mathbf{do} \ \mathbf{next} \ \llbracket P \rrbracket$
$\llbracket k < l; P \rrbracket$	$=$	$\mathbf{tell}(\mathbf{sel}(k, l)) \parallel \mathbf{whenever} \ \overline{\mathbf{sel}(k, l)} \ \mathbf{do} \ \mathbf{next} \ \llbracket P \rrbracket$
$\llbracket k \triangleright \{l_1 : P_1 \parallel \dots \parallel l_n : P_n\} \rrbracket$	$=$	$(\mathbf{wait} \ l; \mathbf{sel}(k, l)) \ \mathbf{do} \ \prod_{1 \leq i \leq n} \mathbf{when} \ l = l_i \ \mathbf{do} \ \mathbf{next} \ \llbracket P_i \rrbracket$
$\llbracket \mathbf{throw} \ k[k']; P \rrbracket$	$=$	$\mathbf{tell}(\mathbf{out}(k, k')) \parallel \mathbf{whenever} \ \overline{\mathbf{out}(k, k')} \ \mathbf{do} \ \mathbf{next} \ \llbracket P \rrbracket$
$\llbracket \mathbf{catch} \ k(k') \ \mathbf{in} \ P \rrbracket$	$=$	$(\mathbf{whenever} \ \overline{\mathbf{out}(k, k')}) \ \mathbf{do} \ \mathbf{next} \ \llbracket P \rrbracket$
$\llbracket \mathbf{if} \ e \ \mathbf{then} \ P \ \mathbf{else} \ Q \rrbracket$	$=$	$\mathbf{when} \ e \downarrow \ \mathbf{true} \ \mathbf{do} \ \mathbf{next} \ \llbracket P \rrbracket \parallel \mathbf{when} \ e \downarrow \ \mathbf{false} \ \mathbf{do} \ \mathbf{next} \ \llbracket Q \rrbracket$
$\llbracket P \parallel Q \rrbracket$	$=$	$\llbracket P \rrbracket \parallel \llbracket Q \rrbracket$
$\llbracket \mathbf{inact} \rrbracket$	$=$	\mathbf{skip}
$\llbracket (vu)P \rrbracket$	$=$	$(\mathbf{local} \ u) \llbracket P \rrbracket$
$\llbracket \mathbf{def} \ D \ \mathbf{in} \ P \rrbracket$	$=$	$\prod_{X_i(x_i, k_i) \in D} \mathcal{R}[\llbracket X_i(x_i, k_i) \rrbracket] \hat{P}$

Table 3: An Encoding from HVK into utcc. $\mathcal{R}[\cdot]$ and \hat{P} are defined in Equation 1.

request $a(k)$ **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 $\mathbf{req}(a, k)$. Once $\llbracket Q \rrbracket$ receives the session key (local variable) generated by $\llbracket P \rrbracket$, it adds the constraint $\mathbf{acc}(a, k)$ to notify the acceptance of k . Then, $\llbracket P \rrbracket$ and $\llbracket Q \rrbracket$ synchronize using this constraint and they execute their continuation in the next time unit. Label selection and branching synchronize on the constraint $\mathbf{sel}(k, l)$. We use the parallel composition $\prod_{1 \leq i \leq n} \mathbf{when} \ l = l_i \ \mathbf{do} \ \mathbf{next} \ \llbracket P_i \rrbracket$ to execute the selected choice. Notice that we do not require a non-deterministic choice since the constraints $l = l_i$ are mutually exclusive [11]. As in [6], in the encoding of **if** e **then** P **else** Q , we assume an evaluation function on expressions. Once e is evaluated, $\downarrow e$ is a *constant* boolean value. The encoding of **def** D **in** P exploits the scheme described in Section 2.1 (Eq.1).

In general, acceptance in HVK is not *persistent* since, e.g., given a single **accept** $a(k)$ **in** P and two or more processes **request** $a(k)$ **in** P' , one of them will be discarded. It is interesting to consider a variant of HVK processes in which session acceptance is persistent over time; this is useful to represent services which are *always* available. For such a variant (that we shall call *persistent* HVK) the following holds:

Theorem 2 (Operational Correspondence). *Suppose $\llbracket \cdot \rrbracket$ is the encoding in Table 3. Let P, P' be well-typed, persistent HVK processes, and let R, S be utcc processes. It holds:*

- 1) Soundness: *If $P \rightarrow P'$ then, for some P'' and R , $P' \longrightarrow^* P''$ and $\llbracket P \rrbracket \Longrightarrow R \sim^o \llbracket P'' \rrbracket$.*
- 2) Completeness: *If $\llbracket P \rrbracket \Longrightarrow^* S$ then, for some P' , $P \rightarrow^* P'$ and $\llbracket P' \rrbracket \sim^o S$.*

3.1 An Extended Language

We now propose HVK^\top , a timed extension to HVK. In HVK^\top , constructs for session request and acceptance are refined with explicit time and a construct for session abortion is considered:

Definition 3 (A timed language for sessions). HVK^\top processes are given by the following syntax:

$P ::=$	request $a(k)$ during m in P	<i>Timed Session Request</i>
	accept $a(k)$ given c in P	<i>Declarative Session Acceptance</i>
	\dots	<i>{ the other constructs, as in Def. 1 }</i>
	kill c_k	<i>Session Abortion</i>

The intuition behind these three operators is the following: **request** $a(k)$ **during** m **in** P will request a session k over the service name a during m time units. Its dual construct is **accept** $a(k)$ **given** c **in** P , which will grant the session key k when requested over the service name a , provided 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, **kill** c_k will remove c_k from the valid set of sessions.

$\llbracket \text{request } a(k) \text{ during } m \text{ in } P \rrbracket$	=	$(\text{local } k) \underline{\text{tell}}(\text{req}(a, k, m)) \parallel$ $\text{whenever } \text{acc}(a, k) \text{ do next } (\underline{\text{tell}}(\text{act}(k)) \parallel \mathcal{G}_{\text{act}(k)}(\llbracket P \rrbracket) \parallel$ $\text{!}_{[m]} \text{unless } \text{kill}(k) \text{ next } \underline{\text{tell}}(\text{act}(k)))$
$\llbracket \text{accept } a(k) \text{ given } c \text{ in } P \rrbracket$	=	$(\underline{\text{wait}} k; \text{req}(a, k, m) \wedge c[m/dur_k]) \text{ do } (\underline{\text{tell}}(\text{acc}(a, k)) \parallel \text{next } \mathcal{G}_{\text{act}(k)}(\llbracket P \rrbracket))$
$\llbracket \text{kill } k \rrbracket$	=	$\text{!tell}(\text{kill}(k))$

Table 4: The Extended Encoding. $\mathcal{G}_d(P)$ is defined in [9, Appendix C]

Adapting the encoding in Table 3 to consider HVK^\top processes is remarkably simple. 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 $\text{req}(a, k, m)$. 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 $\text{act}(k)$. The execution of the continuation $\llbracket P \rrbracket$ is guarded by the constraint $\text{act}(k)$ (i.e., P can be executed only when the session k is valid). In the encoding, we use function $\mathcal{G}_d(P)$ to stand for the process which behaves as P when the constraint d can be entailed from the current store, and that is precluded from execution otherwise; see [9, Appendix C] for details. In 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 requests a session longer than what the service provider is willing to grant.

3.2 Case Study: Electronic booking

Here we present a more involved example in a electronic booking scenario which makes use of the constructs introduced in HVK^\top .

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 a first stage, 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 he finds a suitable option, that he will accept by starting the contracting phase. One possible HVK^\top specification of this scenario is given in Table 5.

In a second stage, the customer uses an online broker to mediate between him and a set of airlines acting as service providers. First, the customer establishes a session with the broker for a given period m ; later on, the customer asks for a flight by providing his request details to the broker. At the other side, the broker will start to look into his pool of trusted service providers for the ones which can supply flights fulfilling the customer needs. All possible offers are transmitted back to the customer, who will select one from the pool of options. After that, the broker will allow the interaction between the customer and the selected service by delegating the session key used to get the offer back to the customer, while doing some post-processing after the session is established. See [9] for an HVK^\top specification of this scenario.

Customer	=	request $ob(k)$ during m in $(k![bookingdata]; Select(k))$
Select(k)	=	$k?(offer)$ in (if $(offer.price \leq 1500)$ then $k \triangleleft Contract$; else $Select(k)$)
AC	=	accept $ob(k)$ given $m \leq MAX_TIME$ in ($k?(bookingData)$ in $(\nu u)k![u]; k \triangleright \{Contract : \overline{Accept} \parallel Reject : kill k\}$)

Table 5: Online booking example with two agents.

3.3 Exploiting the Logic Correspondence

To exploit the logic correspondence we can draw inspiration from the *constraint templates* put forward in [14], 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. Appealing to Theorem 1, our framework allows for the verification of existence and relation constraints over HVK^T programs. Assume a HVK^T program P and let $F = \mathcal{L}[[[P]]]$ (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 $F \vdash \diamond \exists_k (acc(a, k))$ (recall that the encoding of **accept** adds the constraint $acc(a, k)$ 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 [14, 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 5. Let \overline{Accept} be a process that outputs “ok” through a session h . We may then verify the formula $F \vdash \exists_u (u.price < 1.500 \Rightarrow out(h, ok))$. 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 Concluding Remarks

We have argued for a timed CCP language as a suitable foundation for analyzing declarative sessions. Preliminary results presented here include an encoding of the language for structured communication in [6] 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 sessions has not been proposed before.

Our work has not addressed the typed nature of the HVK language. Roughly speaking, the type discipline in [6] ensures a correct “pairing” between complementary components (e.g. session providers and requesters). Our encoding assumes processes to be well-typed with respect to such a discipline. This is because, in our view, declarative techniques should not conflict with operational techniques. Hence, we find it reasonable to assume that a *utcc*-based analysis of sessions takes place once the type system in [6] has ensured a correct pairing. In this initial effort, we have focused on exploring to what extent temporal logic can provide correctness guarantees at the session level. In a later stage, we expect to undertake a thorough study of the interplay between types, constraints, and temporal formulas in the unified framework CCP provides.

Ongoing work includes the development of a proof system—in the lines of [11]— to better support logic-based reasoning. Also, we plan to explore alternative formulations of our encodings. In particular, we would like to determine whether or not they can be expressed in the *monotonic* fragment of *utcc*,

i.e. the fragment without occurrences of **unless** processes. Such a fragment enjoys an extended set of reasoning techniques, including symbolic and denotational semantics [12], as well as static analysis techniques [5]. Hence, having encodings of HVK into such a fragment would further support our claims on the convenience of a CCP-based framework for declarative structured communications.

Acknowledgments. We are grateful to Thomas Hildebrandt and Marco Carbone for insightful discussions on the topics of this paper, and for giving useful comments on previous versions of this document. We are also grateful to the anonymous reviewers for their comments and remarks. The contribution of Olarte and Pérez was initiated during short research visits to the IT University of Copenhagen. They are most grateful to the IT University and to the FIRST PhD Graduate School for funding such visits.

References

- [1] M. Boreale, R. Bruni, L. Caires, R. D. Nicola, I. Lanese, M. Loreti, F. Martins, U. Montanari, A. Ravara, D. Sangiorgi, V. T. Vasconcelos, and G. Zavattaro. Scc: A service centered calculus. In *Proc. of WS-FM*, volume 4184 of *LNCS*, pages 38–57. Springer, 2006.
- [2] M. G. Buscemi and U. Montanari. Cc-pi: A constraint-based language for specifying service level agreements. In *Proc. of ESOP*, volume 4421 of *LNCS*, pages 18–32. Springer, 2007.
- [3] M. Coppo and M. Dezani-Ciancaglini. Structured Communications with Concurrent Constraints. In *Proc. of TGC’08*, LNCS. Springer. To Appear.
- [4] J. F. Díaz, C. Rueda, and F. D. Valencia. Pi+- calculus: A calculus for concurrent processes with constraints. *CLEI Electron. J.*, 1(2), 1998.
- [5] M. Falaschi, C. Olarte, and C. Palamidessi. A framework for abstract interpretation of universal timed concurrent constraint programs. Technical report, LIX, Ecole Polytechnique and Siena University, 2008.
- [6] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *Proc. of ESOP*, volume 1381 of *LNCS*. Springer, 1998.
- [7] I. Lanese, F. Martins, V. T. Vasconcelos, and A. Ravara. Disciplining orchestration and conversation in service-oriented computing. In *Proc. of SEFM*, pages 305–314. IEEE Computer Society, 2007.
- [8] A. Lapadula, R. Pugliese, and F. Tiezzi. A calculus for orchestration of web services. In *Proc. of ESOP*, volume 4421 of *LNCS*, pages 33–47. Springer, 2007.
- [9] H. A. López, C. Olarte, and J. A. Pérez. Towards a unified framework for declarative structured communications. Working Draft Available at http://www.itu.dk/~hual/PersonalSite/Publications_files/sessions-extended.pdf, February 2009.
- [10] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer, 1991.
- [11] M. Nielsen, C. Palamidessi, and F. D. Valencia. Temporal concurrent constraint programming: Denotation, logic and applications. *Nord. J. Comput.*, 9(1):145–188, 2002.
- [12] C. Olarte and F. D. Valencia. The expressivity of universal timed ccp: undecidability of monadic ftl and closure operators for security. In *Proc. of PPDP*, pages 8–19. ACM, 2008.
- [13] C. Olarte and F. D. Valencia. Universal concurrent constraint programming: symbolic semantics and applications to security. In *Proc. of SAC*, pages 145–150. ACM, 2008.
- [14] M. Pesic and W. M. P. van der Aalst. A declarative approach for flexible business processes management. In *BPM’06 Workshops*, volume 4103 of *LNCS*, pages 169–180. Springer, 2006.
- [15] D. Sangiorgi and D. Walker. *The π -calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.
- [16] V. Saraswat, R. Jagadeesan, and V. Gupta. Foundations of timed concurrent constraint programming. In *Proc. of LICS*, pages 71–80. IEEE Computer Society, 1994.
- [17] V. A. Saraswat. *Concurrent Constraint Programming*. MIT Press, 1993.
- [18] B. Victor and J. Parrow. Concurrent constraints in the fusion calculus. In *Proc. of ICALP*, volume 1443 of *LNCS*, pages 455–469. Springer, 1998.