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
University of Bologna
perez@cs.unibo.it

In this paper we aim at describing a unified framework for the declarative analysis of structured communications. By relying on a (timed) concurrent constraint programming language, we show that in addition to the usual operational techniques from process calculi, the analysis of structured communications can elegantly exploit logic-based reasoning techniques. In this work, we present a concurrent constraint interpretation of the language for structured communications 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 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., [19, 3, 9, 10]), 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., [20, 15]). 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) [18] 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 [18] 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 [16]. 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., [17, 12]), where processes evolve along a series of *discrete time intervals*. Although each interval contains its own store, information is not automatically transferred from one interval to another. In this paper we shall adopt a CCP process language that is timed in this sense.

This is a preliminary version of a paper
that will appear in Electronic Proceedings
in Theoretical Computer Science.

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 [18, 12, 13], 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`, a timed CCP process calculus [14], to give a declarative interpretation to the language defined by Honda, Vasconcelos, and Kubo in [8] (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 HVK$^{\mathsf{T}}$, 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$^{\mathsf{T}}$.

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 [8, Sect. 4.1]. There, 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 no 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 an interface between the user and the ATM. Suppose the card reader is malicious in that it keeps the user's sensible information after a withdraw operation, and uses it to continue withdrawing money without his/her authorization. A greedy card reader could even withdraw repeatedly until causing an overdraft, as expressed below:

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

By creating sessions between them, the card reader *Reader* is able to receive the user's information, and to use it later to attempt 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, 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: $Q = k_{bank}![\underline{0}]; \textbf{inact}$.

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 specification. 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., [21, 7]). The crucial difference between such a strand of work and CCP-based calculi is that the latter offers 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 [4] and the calculus for structured communications in [5]. Such languages feature elements that resemble much ideas underlying CCP (especially [4]). 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 [4], the reasoning techniques associated to CC-Pi are essentially operational, and used in the context of service-level agreement scenarios. In [5], the key for analysis is represented by a type system which provides consistency for session execution, much as in the original approach in [8].

2 Preliminaries

2.1 A Language for Structured Communication

We begin by introducing HVK, the language for structured communication proposed in [8]. 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. The sets of free names/channels/variables/process variables of P, is defined in the standard way, and respectively denoted by $fn(\cdot)$,$fc(\cdot)$,$fv(\cdot)$ and $fpv(\cdot)$. Processes without free variables or free channels are called *programs*.

Definition 1 (The HVK language [8])**.** *Processes in* HVK *are built from:*

P,Q	::=	**request** $a(k)$ **in** P	*Session Request*	**accept** $a(x)$ **in** P	*Session Acceptance*
	|	$k![\vec{e}];P$	*Data Sending*	$k?(x)$ **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*	$P \mid Q$	*Parallel Composition*
	|	**inact**	*Inaction*	$(\nu u)P$	*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 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) $(\nu u)\mathbf{inact} \equiv_h \mathbf{inact}$, $(\nu uu)P \equiv_h (\nu u)P$, $(\nu uu')P \equiv_h (\nu u'u)P$, $(\nu u)(P \mid Q) \equiv_h (\nu u)P \mid Q$ 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) **def** D **in** (**def** D' **in** P) $\equiv_h$ **def** D and D' **in** P if $fpv(D) \cap fpv(D') = \emptyset$.

LINK	**accept** $a(x)$ **in** $P \mid$ **request** $a(k)$ **in** $Q \longrightarrow_h (\nu k)(P \mid Q)$
COM	$(k![\vec{e}];P) \mid (k?(x)$ **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** $k[k'];P \mid$ **catch** $k(k')$ **in** $Q \longrightarrow_h P \mid Q$
IF1	**if** e **then** P **else** $Q \longrightarrow_h P$ $(e \downarrow \texttt{true})$
IF2	**if** e **then** P **else** $Q \longrightarrow_h Q$ $(e \downarrow \texttt{false})$
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 1: Reduction Relation for HVK ($\longrightarrow_h$)[8].

Let us give an intuition about the language constructs and of the rules in Figure 1. The central idea in HVK is the notion of *session*, i.e., a series of reciprocal interactions between two parties, possibly with branching and recursion, which serves as an abstraction unit for describing structured communication. Each session has associated a 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 **accept** $a(x)$ **in** $P \mid$ **request** $a(k)$ **in** Q. 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 $k?(x)$ **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.

2.2 Timed Concurrent Constraint Programming

Timed concurrent constraint programming (`tcc`) [17] 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 (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 interval. It is worth noticing that the final store is not automatically transferred to the next time unit.

The `utcc` calculus [14] extends `tcc` for reactive systems featuring mobility. Here *mobility* is understood as the dynamic reconfiguration of system linkage through communication, much like in the π-calculus [16]. `utcc` generalizes `tcc` by considering a *parametric* ask operator of the form $(\textbf{abs}\ \vec{x};c)\,P$, 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 seen as an *abstraction* of the process P on the variables $\vec{x}$ under the constraint (or with the *guard*) c.

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 [13]. Second, utcc processes can be characterized as First-order Linear-time Temporal Logic (FLTL) formulas [11]. 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* [18] 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:

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

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

A process **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 $\vec{t}$ s.t. the current store entails $c[\vec{t}/\vec{x}]$. 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. $P \parallel Q$ 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* $\mathbf{next}\,P$ executes P in the next time interval. The *time-out* $\mathbf{unless}\ c\ \mathbf{next}\,P$ 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* $!P$ means $P \parallel \mathbf{next}\,P \parallel \mathbf{next}^2 P \parallel \ldots$, i.e., unboundedly many copies of P but one at a time. We shall use $!_{[n]}P$ to denote *bounded replication*, i.e., $P \parallel \mathbf{next}\,P \parallel \ldots \parallel \mathbf{next}^{n-1}P$.

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

$$\mathscr{R}[\![X(\vec{x}) \stackrel{\mathtt{def}}{=} P]\!] = !\,(\mathbf{abs}\,\vec{x}; call_x(\vec{x}))\,\widehat{P} \tag{1}$$

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

Operational Semantics. The operational semantics considers *transitions* between process-store *configurations* $\langle P,c\rangle$ with stores represented as constraints and processes quotiented by the structural congruence $\equiv_u$ defined below. We shall use $\gamma,\gamma',\ldots$ to range over configurations.

The semantics is given in terms of an *internal* and an *observable* transition relation; both are given in Figure 2. 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' ". We sometimes abuse of notation by writing $P \longrightarrow P'$ when d,d' are unimportant. The *observable transition* $P \stackrel{(c,d)}{\Longrightarrow} R$ means "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.

In rule R_S, the structural congruence $\equiv_u$ is the smallest congruence satisfying: 1) $P \equiv_u Q$ if they differ only by a renaming of bound variables (alpha-conversion). 2) $P \parallel \mathbf{skip} \equiv_u P$. 3) $P \parallel Q \equiv_u Q \parallel P$, $P \parallel (Q \parallel$

$$\mathrm{R_T}\ \frac{}{\langle \mathbf{tell}(c), d\rangle \longrightarrow \langle \mathbf{skip}, d \wedge c\rangle} \qquad \mathrm{R_P}\ \frac{\langle P, c\rangle \longrightarrow \langle P', d\rangle}{\langle P \parallel Q, c\rangle \longrightarrow \langle P' \parallel Q, d\rangle} \qquad \mathrm{R_U}\ \frac{d \vdash c}{\langle \mathbf{unless}\ c\ \mathbf{next}\ P, d\rangle \longrightarrow \langle \mathbf{skip}, d\rangle}$$

$$\mathrm{R_L}\ \frac{\langle P, c \wedge (\exists \vec{x} d)\rangle \longrightarrow \langle P', c' \wedge (\exists \vec{x} d)\rangle}{\langle (\mathbf{local}\, \vec{x}; c)\, P, d\rangle \longrightarrow \langle (\mathbf{local}\, \vec{x}; c')\, P', d \wedge \exists \vec{x} c'\rangle} \qquad \mathrm{R_A}\ \frac{d \vdash c[\vec{t}/\vec{x}] \qquad |\vec{t}| = |\vec{x}|}{\langle (\mathbf{abs}\, \vec{x}; c)\, P, d\rangle \longrightarrow \langle P[\vec{t}/\vec{x}] \parallel (\mathbf{abs}\, \vec{x}; c \wedge \vec{x} \neq \vec{t})\, P, d\rangle}$$

$$\mathrm{R_S}\ \frac{\gamma_1 \longrightarrow \gamma_2}{\gamma_1' \longrightarrow \gamma_2'}\ \text{if } \gamma_1 \equiv_u \gamma_1' \text{ and } \gamma_2 \equiv_u \gamma_2' \qquad \mathrm{R_R}\ \frac{}{\langle !\, P, d\rangle \longrightarrow \langle P \parallel \mathbf{next}\, !\, P, d\rangle}$$

$$\mathrm{R_O}\ \frac{\langle P, c\rangle \longrightarrow^* \langle Q, d\rangle \not\longrightarrow}{P \xrightarrow{(c,d)} F(Q)}\ \text{where}\ \ \mathrm{F(P)} = \begin{cases} \mathbf{skip} & \text{if } P = \mathbf{skip} \text{ or } P = (\mathbf{abs}\, \vec{x}; c)\, Q \\ F(P_1) \parallel F(P_2) & \text{if } P = P_1 \parallel P_2 \\ (\mathbf{local}\, \vec{x})\, F(Q) & \text{if } P = (\mathbf{local}\, \vec{x}; c)\, Q \\ Q & \text{if } P = \mathbf{next}\, Q \text{ or } P = \mathbf{unless}\ c\ \mathbf{next}\, Q \end{cases}$$

Figure 2: Operational Semantics for utcc. In $\mathrm{R_A}$, $\vec{x} \neq \vec{t}$ ($\vec{x}$ syntactically different from $\vec{t}$) denotes $\bigvee_{1 \leq i \leq |\vec{x}|} x_i \neq t_i$. If $|\vec{x}| = 0$, $\vec{x} \neq \vec{t}$ is defined as false.

$R) \equiv_u (P \parallel Q) \parallel R$. 4) $P \parallel (\mathbf{local}\, \vec{x}; c)\, Q \equiv_u (\mathbf{local}\, \vec{x}; c)\, (P \parallel Q)$ if $\vec{x} \notin fv(P)$. 5) $(\mathbf{local}\, \vec{x}; c)\, (\mathbf{local}\, \vec{y}; d)\, P \equiv_u (\mathbf{local}\, \vec{x}; \vec{y}\, ; c \wedge d)\, P$ if $\vec{x} \cap \vec{y} = \emptyset$ and $\vec{y} \notin fv(c)$. Extend $\equiv_u$ by decreeing that $\langle P, c\rangle \equiv_u \langle Q, c\rangle$ iff $P \equiv_u Q$.

Definition 2 (Output Behavior)**.** *Let $s = c_1.c_2....c_n$ be a sequence of constraints. If $P = P_1 \xrightarrow{(\mathtt{true}, c_1)} P_2 \xrightarrow{(\mathtt{true}, c_2)} \ldots P_n \xrightarrow{(\mathtt{true}, c_n)} P_{n+1} \equiv_u Q$ we shall write $P \xrightarrow{s}{}^* Q$. 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 temporal logic. This is formalized by encoding below, which maps utcc processes into FLTL formulas.

Definition 3. *Let $\mathsf{TL}[\![\cdot]\!]$ a map from utcc processes to FLTL formulas given by:*

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

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 [14])**.** *Let $\mathsf{TL}[\![\cdot]\!]$ be as in Definition 3 and $s = c_1.c_2.c_3....$ 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. $c_i \vdash d$.*

Derived Constructs. 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 $k?(x)\,\mathbf{in}\, P$, resp.) with the utcc processes $\mathbf{tell}(\mathtt{out}(k, \vec{e}))$ and $(\mathbf{abs}\, \vec{x}; \mathtt{out}(k, \vec{x}))\, P$, 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 (**abs** $\vec{x};c)\,P$ and **tell**(c) processes that are *persistent in time*. More precisely, we shall use process (**wait** $\vec{x};c$) **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 $\bar{c}[\vec{t}/\vec{x}]$, as a way of acknowledging the successful read of c. When $|\vec{x}|=0$, we shall write **whenever** c **do** P instead of (**wait** $\vec{x};c$) **do** P. Similarly, we define **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 $stop, go \notin fv(c)$):

$$
\begin{array}{lcll}
\textbf{tell}(c) & \stackrel{\texttt{def}}{=} & (\textbf{local}\, go, stop) & \textbf{tell}(\texttt{out}'(go)) \parallel !\,\textbf{when}\ \texttt{out}'(go)\ \textbf{do}\ \textbf{tell}(c) \parallel \\
 & & & !\,\textbf{unless}\ \texttt{out}'(stop)\ \textbf{next}\,\textbf{tell}(\texttt{out}'(go)) \parallel \\
 & & & !\,\textbf{when}\ \bar{c}\ \textbf{do}\ !\,\textbf{tell}(\texttt{out}'(stop)) \\
(\textbf{wait}\ \vec{x};c)\ \textbf{do}\ P & \stackrel{\texttt{def}}{=} & (\textbf{local}\, stop, go)\,(& \textbf{tell}(\texttt{out}'(go)) \\
 & & & \parallel !\,\textbf{unless}\ \texttt{out}'(stop)\ \textbf{next}\,\textbf{tell}(\texttt{out}'(go)) \\
 & & & \parallel !\,(\textbf{abs}\ \vec{x};c \wedge \texttt{out}'(go))\,(P \parallel !\,\textbf{tell}(\texttt{out}'(stop))) \\
(\underline{\textbf{wait}}\ \vec{x};c)\ \textbf{do}\ P & \stackrel{\texttt{def}}{=} & (\textbf{wait}\ \vec{x};c)\ \textbf{do}\ (P \parallel \textbf{tell}(\bar{c})) &
\end{array}
$$

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

Definition 4 (Observables)**.** *Let $\sim^o$ be the output equivalent relation in Definition 2. We say that P and Q are observable equivalent, notation $P \sim^{obs} Q$, if $P \parallel !\,\textbf{tell}(\exists_x \texttt{out}'(x)) \sim^o Q \parallel !\,\textbf{tell}(\exists_x \texttt{out}'(x))$.*

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

Proposition 1. *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* (**wait** $\vec{x};c$) **do** $P \stackrel{(d,d)}{\Longrightarrow}$ (**wait** $\vec{x};c$) **do** P.
2. *If $P \equiv_u$ **tell**$(c(\vec{t})) \parallel$ (**wait** $\vec{x};c(\vec{x})$) **do next** Q then $P \Longrightarrow \sim^{obs} Q[\vec{t}/\vec{x}]$.*

3 A Declarative Interpretation for Structured Communications

Here we present a compositional encoding of HVK into utcc. The encoding $[\![\cdot]\!]$ from HVK into utcc is defined in Table 3; let us briefly provide intuitions on it. Consider HVK processes $P = \textbf{request}\ a(k)\ \textbf{in}\ P'$ and $Q = \textbf{accept}\ a(x)\ \textbf{in}\ Q'$. The encoding of P declares a new variable session k and sends it through the channel a by posting the constraint $\texttt{req}(a,k)$. Upon reception of the session key (local variable) generated by $[\![P]\!]$, process $[\![Q]\!]$ adds the constraint $\texttt{acc}(a,k)$ 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 synchronize is similar, and uses constraint $\texttt{sel}(k,l)$ for synchronization. We use the parallel composition $\prod_{1\le i\le n}$ **when** $l = l_i$ **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 [8], 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 Equation 1.

A noteworthy aspect to consider here is that HVK is a synchronous language, whereas utcc is asynchronous. Moreover, there is a difference concerning *determinacy*: while utcc is a deterministic language, HVK processes may exhibit non-deterministic behavior. Consider, for instance, the HVK process

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

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

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

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 $k?(\vec{x})$ **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 $[\![P]\!] \Longrightarrow [\![Q_3[\vec{e}/\vec{x}]]\!] \parallel [\![Q_3[\vec{e'}/\vec{x}]]\!]$.

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 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 [8] 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 [2, 1].

Before presenting the operational correspondence, let us introduce some terminology and auxiliary results.

Definition 5 (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 "$\mid$" 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 2. *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 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 6 (Redex)**.** *A* redex *is a pair of complementary processes composed in parallel as in*

- **request** $a(k)$ **in** $P \mid$ **accept** $a(k)$ **in** Q
- $k![\vec{e}];P \mid k?(\vec{x})$ **in** Q
- $k \lhd l; P \mid k \rhd \{l_1 : P_1 \parallel \cdots \parallel l_n : P_n\}$
- **throw** $k[k'];P \mid$ **catch** $k(k')$ **in** Q.

Notice that a redex in HVK synchronizes and reduces in a single transition as in $(k![\vec{e}];P) \mid (k?(x)$ **in** $Q) \longrightarrow_h P \mid Q[\vec{e}/\vec{x}]$. Nevertheless, in `utcc`, the encoding of the processes above requires two internal transitions: one for adding the constraint $\mathtt{out}(k,\vec{e})$ to the current store, and another one in which the process $(\underline{\textbf{wait}}\ \vec{x};\mathtt{out}(k,\vec{x}))$ **do next** $[\![Q]\!]$ "reads" that constraint to later execute **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 reduction relation over HVK processes:

Definition 7 (Outermost Reductions)**.** *Let $P \equiv_h$* **def** D **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 $P \longrightarrow_h^* P' \equiv_h$* **def** D **in** $\nu\vec{x'}(Q'_1 \mid \cdots \mid Q'_n)$ *such that for every $i \in \{1,..n\}$, either*

1. $Q_i =$ **if** e **then** R_1 **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 $Q_i \equiv_h Q'_i$.*

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 2 (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 $P \equiv_h$ **def** D **in** $\nu\vec{x}(Q_1 \mid \cdots \mid Q_n)$ and $Q \equiv_h$ **def** D **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 $Q_i \equiv_h$ **if** e **then** R_1 **else** R_2 and $Q_i \longrightarrow_h R_1 \equiv_h Q'_i$ and $e \downarrow \mathtt{true}$. One can easily show that **when** $e \downarrow \mathtt{true}$ **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 of the form

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

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.:

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

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 1 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 the rule LINK.

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

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

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 $R_i = \mathbf{when}\ e \downarrow \mathtt{true}\ \mathbf{do\ next}\, [\![Q_i']\!]$ 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 $e \downarrow \mathtt{true}$ we then have $R_i'' = \mathbf{next}\, [\![Q_i']\!]$. The case when $e \downarrow \mathtt{false}$ is similar by considering $R_i = \mathbf{when}\ e \downarrow \mathtt{false}\ \mathbf{do}\ Q_i'$.

(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 $Q_i \mid Q_j$ forms a redex, then one can show that $R_i \Longrightarrow\!\sim^{obs} R_i$.

□

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 $\mathsf{HVK}^{\mathsf{T}}$ language arises as the extension of HVK processes (Def. 1) with refined constructs for session request and acceptance, as well as with a construct for session abortion:

Definition 8 (A timed language for sessions)**.** $\mathsf{HVK}^{\mathsf{T}}$ *processes are given by the following syntax:*

$$
\begin{array}{rcll}
P & ::= & \textbf{request}\ a(k)\ \textbf{during}\ m\ \textbf{in}\ P & \textit{Timed Session Request} \\
 & | & \textbf{accept}\ a(k)\ \textbf{given}\ c\ \textbf{in}\ P & \textit{Declarative Session Acceptance} \\
 & | & \cdots & \{\ \textit{the other constructs, as in Def. 1}\ \} \\
 & | & \textbf{kill}\, c_k & \textit{Session Abortion}
\end{array}
$$

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: 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, **kill** c_k will remove c_k from the valid set of sessions.

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

Table 4: The Extended Encoding. $\mathscr{G}_d(P)$ is in Definition 9.

Adapting the encoding in Table 3 to consider $\mathsf{HVK}^{\mathsf{T}}$ processes is remarkably simple (see Table 4). 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). In the encoding, we use the function $\mathscr{G}_d(P)$ to denote the process behaving as P when the constraint d can be entailed from the current store, doing nothing otherwise. More precisely:

Definition 9. *Let* $\mathscr{G} : \mathscr{C} \to \mathit{Procs} \to \mathit{Procs}$ *be defined as*

$$
\mathscr{G}_d(P) = \begin{cases}
\textbf{skip} & \text{if } P = \textbf{skip} \\
\textbf{when}\ d\ \textbf{do}\ \textbf{tell}(c) & \text{if } P = \textbf{tell}(c) \\
(\textbf{abs}\ \vec{x}; c)\, \mathscr{G}_d(Q) & \text{if } P = (\textbf{abs}\ \vec{x}; c)\, Q \text{ and } \vec{x} \notin fv(d) \\
\mathscr{G}_d(P_1) \parallel \mathscr{G}_d(P_2) & \text{if } P = P_1 \parallel P_2 \\
(\textbf{local}\, \vec{x}; c)\, \mathscr{G}_d(Q) & \text{if } P = (\textbf{local}\, \vec{x}; c)\, Q \text{ and } \vec{x} \notin fv(d) \\
\textbf{when}\ d\ \textbf{do}\ \textbf{next}\, \mathscr{G}_d(Q) & \text{if } P = \textbf{next}\, Q \\
\textbf{when}\ d\ \textbf{do unless}\ c\ \textbf{next}\, \mathscr{G}_d(Q) & \text{if } P = \textbf{unless}\ c\ \textbf{next}\, Q \\
!\, \mathscr{G}_d(Q) & \text{if } P = !\, Q
\end{cases}
$$

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.

4.1 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 5.

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

Table 5: 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. Consider two vectors of fixed length: *Offers*, which contains the list of offers received by a customer, and *SP*, which contains the list of trusted service providers. 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 6 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.

(a) Customer and Service Provider

$$\begin{aligned}\text{Customer} = {} & \textbf{request}\ ob(k)\ \textbf{during}\ m\ \textbf{in}\ (k![bookingdata];\\ & k?(\mathit{Offers})\ \textbf{in}\ (\\ & \mathit{Sel}(\mathit{Offers}); a?(x)\ \textbf{in}\ k![x];\\ & \ \textbf{catch}\ k(k')\ \textbf{in}\\ & \ k'![\mathit{PaymentDetails}]; \textbf{inact}))\end{aligned}$$

$$\begin{aligned}\text{SP} = {} & \textbf{accept}\ SP_i(k_i')\ \textbf{given}\ N \leq 300ms\ \textbf{in}\ (\\ & k_i'?(\mathit{bookingData})\ \textbf{in}\\ & \ k_i'![\mathit{offer}];\\ & \ k_i'?(\mathit{paymentDetails})\ \textbf{in inact})\end{aligned}$$

(b) Online Broker

$$\begin{aligned}\text{Broker} = {} & \textbf{accept}\ ob(k)\ \textbf{given}\ m \leq 500ms\ \textbf{in}\ (\\ & k?(\mathit{bookingData})\ \textbf{in}\\ & (\nu u)\ \prod_{i\in SP}\ (\textbf{request}\ SP_i(k_i')\ \textbf{during}\ N\ \textbf{in}\\ & \ k_i'![\mathit{bookingData}];\\ & \ k_i'?(\mathit{offer}_i)\ \textbf{in}\ (u![\mathit{offer}_i]; \textbf{inact} \parallel S(u,k)))\\ & \ k?(y)\ \textbf{in def}\ P\ \textbf{in}\\ & \quad \textbf{if}\ (y = \mathit{offers}_i)\ \textbf{then}\ (\textbf{throw}\ k[k_i']; \mathit{PostProc})\ \textbf{else}\\ & \qquad \textbf{kill}\, k_i' \parallel P)\end{aligned}$$

$$\text{S(u,k)} = \prod_{i\in SP}\ (u?(\mathit{offer}_i)\ \textbf{in inact} \parallel k![\mathit{offer}_i]; \textbf{inact})$$

Table 6: Online booking example with online broker.

4.2 Exploiting the Logic Correspondence

To exploit the logic correspondence we can draw inspiration from the *constraint templates* put forward in [15], 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 [15, 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 then may verify the formula $F \vdash \exists_u(u.price < 1.500 \Rightarrow \mathtt{out}(h,ok))$. 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.

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 [8] 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 when stating an operational correspondence property for the declarative interpretation of HVK processes. We believe there are at least two ways of achieving

more natural notions of operational correspondence. The first one involves considering a variant of utcc extended 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., [6]). While there is a non-deterministic extension to tcc with a tight connection with temporal logic (ntcc [12]), 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 $\mathsf{HVK}^{\mathsf{T}}$ processes better suited to the nature of utcc processes. This would imply enriching the original type system in [8] 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 of (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 network traffic and long-lived transactions (for instance) prevent interactions between services from occurring instantaneously. Properties of interest 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.

Acknowledgments. We are grateful to Marco Carbone and Thomas Hildebrandt 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 PLACES'09 attendees 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. Berger, K. Honda, and N. Yoshida. Sequentiality and the pi-calculus. In *Proc. of TLCA*, volume 2044 of *LNCS*, pages 29–45. Springer, 2001.

[2] M. Berger, K. Honda, and N. Yoshida. Completeness and logical full abstraction in modal logics for typed mobile processes. In *ICALP'08, Part II*, volume 5126 of *LNCS*, pages 99–111. Springer, 2008.

[3] 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.

[4] 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.

[5] M. Coppo and M. Dezani-Ciancaglini. Structured Communications with Concurrent Constraints. In *Proc. of TGC'08*, LNCS, pages 104–125. Springer, 2009.

[6] F. S. de Boer, M. Gabbrielli, E. Marchiori, and C. Palamidessi. Proving concurrent constraint programs correct. *ACM Trans. Program. Lang. Syst.*, 19(5):685–725, 1997.

[7] J. F. Díaz, C. Rueda, and F. D. Valencia. Pi+- calculus: A calculus for concurrent processes with constraints. *CLEI Electron. J.*, 1(2), 1998.

[8] 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.

[9] 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.

[10] 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.

[11] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer, 1991.

[12] M. Nielsen, C. Palamidessi, and F. D. Valencia. Temporal concurrent constraint programming: Denotation, logic and applications. *Nord. J. Comput.*, 9(1):145–188, 2002.

[13] C. Olarte and F. D. Valencia. The expressivity of universal timed ccp: undecidability of monadic fltl and closure operators for security. In *Proc. of PPDP*, pages 8–19. ACM, 2008.

[14] C. Olarte and F. D. Valencia. Universal concurrent constraint programing: symbolic semantics and applications to security. In *Proc. of SAC*, pages 145–150. ACM, 2008.

[15] 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.

[16] D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

[17] V. Saraswat, R. Jagadeesan, and V. Gupta. Foundations of timed concurrent constraint programming. In *Proc. of LICS*, pages 71–80. IEEE Computer Society, 1994.

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

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

[20] W. M. P. van der Aalst and M. Pesic. DecSerFlow: Towards a Truly Declarative Service Flow Language. In *Proc. of WS-FM*, volume 4184 of *LNCS*, pages 1–23. Springer, 2006.

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