Asynchronous Timed Session Types & Processes

Laura Bocchi
Maurizio Murgia
Vasco Thudichum Vasconcelos
Nobuko Yoshida

University of Kent
University of Kent
University of Lisbon
Imperial College London
Agenda

• Session types ♥ Time

• Synchronous [Bartoletti, Cimoli & Murgia@FORTE’13]

• Multiparty asynchronous [Bocchi, Yang, Yoshida@CONCUR’14]
  • restriction to types/protocols that could be used for type-checking
  • limitations to the expressiveness of the calculus

• Today

• Designing timed protocols: asynchronous timed duality

• Checking timed programs: a time-sensitive calculus & typing system
Time & trouble

• A timed protocol is not correct by definition

\![\text{Int}.?\text{String}]

\![\text{Int}(x \leq 3).?\text{String}(x \leq 2)]

• Usually this is handled by adding some conditions

feasibility [Bocchi, Yang, Yoshida@CONCUR’14], interaction-enabledness (CTA) [Bocchi, Lange, Yoshida@CONCUR’15], compliance [Bartoletti, Cimoli, Murgia@FORTE’16], formation + duality [Bocchi, Murgia, Yoshida, Vasconcelos’18]

\![\text{Int}(x \leq 3).?\text{String}(x \leq 3)]
Duality & progress

• Duality characterises well-behaved systems

\[ S = !\text{Int}(x \leq 1, x) . ?\text{String}(x \leq 2) \quad \overline{S} = ?\text{Int}(y \leq 1, y) . !\text{String}(y \leq 2) \]

Synchronous

\[ S | \overline{S} \xrightarrow{0.4} S | \overline{S} \xrightarrow{\text{Int}} ?\text{String}(x \leq 2) | !\text{String}(y \leq 2) \xrightarrow{2} \text{String} \]
Duality & progress

- Duality characterises well-behaved systems

\[ S = \text{!Int}(x \leq 1, x) . \text{?String}(x \leq 2) \quad \overline{S} = \text{?Int}(y \leq 1, y) . \text{!String}(y \leq 2) \]

Synchronous

\[ S \mid \overline{S} \xrightarrow{0.4} S \mid \overline{S} \xrightarrow{\text{Int}} \text{?String}(x \leq 2) \mid \text{!String}(y \leq 2) \xrightarrow{2} \text{String} \]

Asynchronous

\[ S \mid \overline{S} \xrightarrow{0.4} S \mid \overline{S} \xrightarrow{!\text{Int}} \text{?String}(x \leq 2) \mid \text{?Int}(y \leq 1, y) . \text{!String}(y \leq 2) \xrightarrow{0.6} \text{?Int} \xrightarrow{\text{String}} \text{?String}(x \leq 2) \]

\[ x \quad \begin{array}{c} 2.6 \end{array} \quad y \quad \begin{array}{c} 2 \end{array} \]
Receive & asynchrony (1/2)

\[ S = \mu_t.!(\text{Int}(x \leq 1, x)\cdot ?\text{String}(x \leq 2)\cdot t) \]

```go
func S (a chan<- int, b <-chan string, start time.Time)
{
    for {
        time.Sleep(400 * time.Millisecond)
        t := time.Now()
        a<-10
        fmt.Printf("sent int 10 at time %s\n", t.Sub(start))
        select{
            case c:=<-b :
                t := time.Now()
                fmt.Printf("received string %s at time %s\n", c, t.Sub(start))
            case <-time.After(2 * time.Second):
                fmt.Println("S Failed! String not received within deadline")
        }
    }
}
```
Receive & asynchrony (2/2)

$$S = \mu t. ?\text{Int}(y \leq 1, y).!\text{String}(y \leq 2).t$$

```go
def Sd(a <-chan int, b chan<- string, start time.Time){
    for{
        select{
            case c:=<-a :  
                t := time.Now()
                fmt.Printf("received int %d at time %s\n", c, t.Sub(start))
        }
        time.Sleep(600 * time.Millisecond)
        t := time.Now()
        b<-"hello!"
        fmt.Printf("sent 'hello!' at time %s\n", t.Sub(start))
    }
}
```
Urgent receive semantics

- Urgent receive semantics: messages are received as soon as
  - they are in a channel, and
  - the time constraint of the receiver is satisfied
- Urgent receive semantics yields executions that are
  - a bit more synchronous …
  - … but as asynchronous as when using (common) receive primitives

\[ S | \overline{S} \xrightarrow{0.4} S | \overline{S} \xrightarrow{\text{Int}} ?\text{String}(x \leq 2) | ?\text{Int}(y \leq 1,y).!\text{String}(y \leq 2) \]

\[ 0.6 \xrightarrow{\text{Int}} ?\text{String}(x \leq 2) | !\text{String}(y \leq 2) \xrightarrow{2} !\text{String} \xrightarrow{?}\text{String}(x \leq 2) \]
Urgent receive semantics

• Urgent receive semantics: messages are received as soon as they are in a channel, and the time constraint of the receiver is satisfied
• Urgent receive semantics yields executions that are
  • are a bit more synchronous …
  • … but as asynchronous as when using (common) receive primitives

Type Progress: \((\nu_1, S_1, M_1) \mid (\nu_2, S_2, M_2)\) satisfies progress if any reachable state is either success (end types and empty queues) or allows an action, possibly after some delay.

Theorem (Duality Progress). \((\nu_0, S, \emptyset) \mid (\nu_0, \overline{S}, \emptyset)\) enjoys progress (when using urgent receive semantics).
Subtyping

• Asymmetric as e.g., [Gay&Hole’05][Demangeon&Honda’11] [Chen,Dezani-Ciancaglini&Yoshida’14]

**Definition (Timed Simulation).** Fix $s_1 = (\nu_1, S_1)$ and $s_2 = (\nu_2, S_2)$. A relation $R \in (\forall \times S)^2$ is a timed simulation if $(s_1, s_2) \in R$ implies:

1. $S_1 = \text{end}$ implies $S_2 = \text{end}$
2. $s_1 \xrightarrow{t!m_1} s'_1$ implies $\exists s'_2, m_2 : s_2 \xrightarrow{t!m_2} s'_2$, $(m_2, m_1) \in S$, and $(s'_1, s'_2) \in R$
3. $s_2 \xrightarrow{t?m_2} s'_2$ implies $\exists s'_1, m_1 : s_1 \xrightarrow{t?m_1} s'_1$, $(m_1, m_2) \in S$, and $(s'_1, s'_2) \in R$
4. $s_1 \Rightarrow$ implies $s_2 \Rightarrow$ and $s_2 \Rightarrow$ implies $s_1 \Rightarrow$

\[
\begin{align*}
\text{String}(x = 0) & \prec \text{String}(x \leq 2) \\
\text{String}(x \leq 2) & \prec \text{String}(x = 0) \\
\text{String}(x \leq 2), \text{String}(x = 1) & \nsubseteq \text{String}(x = 0), \text{String}(x = 1) \\
\text{String}(\text{true}) & \nsubseteq \text{String}(\text{true})
\end{align*}
\]
Subtyping

- Asymmetric as e.g., [Gay&Hole’05][Demangeon&Honda’11][Chen,Dezani-Ciancaglini&Yoshida’14]

**Definition (Timed Simulation).** Fix $s_1 = (ν_1, S_1)$ and $s_2 = (ν_2, S_2)$.
A relation $R \in (V \times S)^2$ is a timed simulation if $(s_1, s_2) \in R$ implies:

1. $S_1 = \text{end}$ implies $S_2 = \text{end}$
2. $s_1 \xrightarrow{t!m_1} s'_1$ implies $\exists s'_2, m_2 : s_2 \xrightarrow{t!m_2} s'_2$, $(m_2, m_1) \in S$, and $(s'_1, s'_2) \in R$
3. $s_2 \xrightarrow{t?m_0} s'_2$ implies $\exists s'_1, m_1 : s_1 \xrightarrow{t?m_1} s'_1$, $(m_1, m_2) \in S$, and $(s'_1, s'_2) \in R$
4. $s_1 \xrightarrow{?} \Rightarrow$ implies $s_2 \xrightarrow{?} \Rightarrow$ and $s_2 \xrightarrow{?} \Rightarrow$ implies $s_1 \xrightarrow{?} \Rightarrow$

**Theorem (Safe/Progressing Substitution).** Let $S' <: \overline{S}$ then

1) $(ν_0, S, \emptyset) | (ν_0, S', \emptyset) \preceq (ν_0, S, \emptyset) | (ν_0, \overline{S}, \emptyset)$
2) $(ν_0, S, \emptyset) | (ν_0, S', \emptyset)$ enjoys progress.

In [Bartoletti,Bocchi,Murgia@CONCUR’18] asymmetric refinement does not preserve behaviour/progress (it was “local” and did not assume duality)
Implementing dual types

“An SMTP server SHOULD have a timeout of at least 5 minutes while it is awaiting the next command from the sender” [RFC 5321]

\[
S = \text{?Com}(x < 5, x).S' \\
C = !\text{Com}(y < 5, y).C'
\]

• This protocol can be implemented e.g., in Go, Erlang (timeout pattern), Real-Time Java, ...

```plaintext
select{
  case <-b : \ \ \ \ proceed as S'
  case <-time.After(5 * time.Second): \ \ \ \ explode
}
```

• This protocol cannot be correctly implemented with the calculus in [Bocchi,Yang&Yoshida’14]
Implementing dual types

\[ S = \text{Com}(x < 5, x).S' \quad C = \text{Com}(y < 5, y).C' \]

\[ \text{delay}(4.90).a(b).P'_s | \text{delay}(4.99).\overline{a}(\text{HELO}).P'_c \]

\[ \longrightarrow \ a(b).P'_s | \text{delay}(0.09).\overline{a}(\text{HELO}).P'_c \]

**Wait-freedom** [Bocchi, Yang, Yoshida’14]: the solutions of the constraint of a receive action must be all after any solution of the corresponding send action

\[ S = \text{Com}(x = 5, x).S' \quad C = \text{Com}(y < 5, y).C' \]
Programs

\[ P ::= \overline{a} v . P \]
\[ \quad | \ a < 1 . P \]
\[ \quad | \ \text{if } v \ \text{then } P \ \text{else } P \]
\[ \quad | \ P \mid P \]
\[ \quad | \ 0 \]
\[ \quad | \ \text{def } D \ \text{in } P \]
\[ \quad | \ X(\overrightarrow{a}; \overrightarrow{a}) \]
\[ \quad | \ (\nu a b) \ P \]
\[ \quad | \ a b : h \]

\[ \begin{array}{c}
\text{time-consuming} \\
\text{delay}(\delta).P \\
\text{blocking} \\
\text{blocking with timeout} \\
\end{array} \]

\[ C = !\text{Com}(y < 5, y).C' \]
\[ \text{delay}(x = 4.90).a^0(b).P'_s \]
\[ a^5(b).P'_s \]
Programs

\[ P ::= \overline{a}v \cdot P \]

\[ | \quad a < 1 \cdot P \]

\[ | \quad \text{if } \nu \text{ then } P \text{ else } P \]

\[ | \quad P \mid P \]

\[ | \quad 0 \]

\[ | \quad \text{def } D \text{ in } P \]

\[ | \quad X(\overrightarrow{a}; \overrightarrow{a}) \]

\[ | \quad (\nu ab) \cdot P \]

\[ | \quad ab : h \]

\[ \text{delay}(\delta) \cdot P \]

\[ \text{delay}(x = 4.99) \cdot \overline{a}(\text{HELO}) \cdot P'_c \]

\[ \text{delay}(4.8 \leq x < 5) \cdot \overline{a}(\text{HELO}) \cdot P'_c \]
There are also typing rules...

\[
\frac{\Gamma \vdash b : T \quad \nu \models \delta \quad \Gamma \vdash P \triangleright \Delta, a : (\nu[\lambda \mapsto 0], S)}{\Gamma \vdash ab.P \triangleright \Delta, a : (\nu, !T(\delta, \lambda). S)} [send]
\]

\[
\forall t : \nu + t \models \delta \iff t \leq n
\]
\[
\forall t \leq n : \quad \Gamma, b : T \vdash P \triangleright \Delta + t, a : (\nu + t[\lambda \mapsto 0], S) \quad \Delta \text{ not t-reading}
\]
\[
\Gamma \vdash a^n(b).P \triangleright \Delta, a : (\nu, ?T(\delta, \lambda).S) [rcv]
\]

\[
\forall n \in \delta : \quad \Gamma \vdash \text{delay}(n).P \triangleright \Delta [\text{delay1}]
\]
\[
\Gamma \vdash \text{delay}(\delta).P \triangleright \Delta
\]

\[
\Gamma \vdash P \triangleright \Delta + n \quad \Delta \text{ not n-reading} [\text{delay2}]
\]
\[
\Gamma \vdash \text{delay}(n).P \triangleright \Delta
\]
What is a missed deadline?

\[ S = \text{?Com}(x < 5, x).S' \quad C = \text{!Com}(y < 5, y).C' \]

\[ \text{delay}(4.90).a(b).P'_s \mid \text{delay}(4.99).\overline{a}(\text{HELO}).P'_c \]

\[ \quad \quad \rightarrow \quad a(b).P'_s \mid \text{delay}(0.09).\overline{a}(\text{HELO}).P'_c \]

\[ \quad \quad \rightarrow \quad \text{failed} \mid \text{delay}(0.09).\overline{a}(\text{HELO}).P'_c \]

- Failing semantics:
  - See system’s behaviour beyond failure of some parts (-> error handling)
  - Reveals relationship between \textit{untimed progress} and \textit{time safety}
Programs

\[ P ::= \bar{a}v \cdot P \]
\[ | \quad a \triangleleft l \cdot P \]
\[ | \quad \text{if } v \text{ then } P \text{ else } P \]
\[ | \quad P \mid P \]
\[ | \quad 0 \]
\[ | \quad \text{def } D \text{ in } P \]
\[ | \quad X(\bar{a}; \bar{a}) \]
\[ | \quad (vab) \ P \]
\[ | \quad ab : h \]

\[ \text{time-consuming} \]
\[ | \quad \text{delay}(\delta) \cdot P \]
\[ | \quad a^n(b) \cdot P \]
\[ | \quad a^n \triangleright \{ l_i : P_i \}_{i \in I} \]

\[ \text{run-time} \]
\[ | \quad \text{delay}(n) \cdot P \]
\[ | \quad \text{failed} \]
Subject reduction?

\((\nu ab)(\nu cd) \ a^5(e) \ . \ \bar{d}e.0 \ | \ c^5(e) \ . \ \bar{b}e.0 \ | \ a b : \emptyset \ | \ b a : \emptyset \ | \ c d : \emptyset \ | \ c d : \emptyset\)

\[\rightarrow (\nu ab)(\nu cd)(\text{failed} \ | \ \text{failed} \ | \ a b : \emptyset \ | \ b a : \emptyset \ | \ c d : \emptyset \ | \ c d : \emptyset)\]

Well typed

\[\emptyset \vdash P \triangleright a : (\nu_0, S), \ b : (\nu_0, \bar{S}), \ c : (\nu_0, S), \ d : (\nu_0, \bar{S})\]

\[S = !\text{Int}(x \leq 5, \emptyset). \text{end}\]

Subject reduction does not hold in general
Subject reduction!

**Definition (Live process).** $\hat{P}$ is live if, for each $\hat{P}'$ such that $\hat{P} \rightarrow^* \hat{P}'$:

$$\hat{P}' \equiv (\nu ab)\hat{Q} \land a \in \text{Wait}(\hat{Q}) \implies \exists \hat{Q}': \hat{Q} \rightarrow^* \hat{Q}' \land a \in \text{NEQueue}(\hat{Q}')$$

**Theorem (Subject Reduction).** Let $\text{erase}(P)$ be live. If $\emptyset \vdash P \triangleright \emptyset$ and $P \rightarrow P'$ then $\emptyset \vdash P \triangleright \emptyset$.

**Theorem (Time Safety).** If $\text{erase}(P)$ is live, $\emptyset \vdash P \triangleright \emptyset$ and $P \rightarrow^* P'$ then $P'$ is fail-free.
In summary

• Duality, subtyping, & urgent receive.

   In [Bartoletti, Bocchi, Murgia@CONCUR’18] asymmetric refinement does not preserve behaviour/progress (it was “local” and did not assume duality)

• Dual types cannot be (correctly) implemented with previous work on Multiparty Asynchronous Timed Session Types

   A time-sensitive calculus with: parametric receive, delays with arbitrary but constrained delays, and explicit failures upon timeout

   A typing system for processes (with delegation) that satisfies subject reduction and time safety

   Considerations on the meaning of progress and failure in a timed context
Future work

- Time-sensitive protocol design and implementation EP/N035372/1
- Expressiveness (flexible timing schedules) + run-time adjustments

System:

- Program
- Run-time Instrumentation

(Safe) Program + Run-time Instrumentation
Thank you!