Primitives for Contract-based Synchronization

Massimo Bartoletti
Dipartimento di Matematica e Informatica, Università degli Studi di Cagliari

Roberto Zunino
Dipartimento di Ingegneria e Scienza dell’Informazione, Università degli Studi di Trento
Topic of the talk:

Contract-based computing

Meaningless buzzwords, until one sets up the two crucial concepts:

- Contracts
- Computing — the focus of this talk
What is a contract?

A contract is a document where a promising party commits himself to offer some guarantee, provided that some assumptions are verified.

A set of contracts induces an agreement among the promising parties, when the promises made “match” the conditions required.}

Promise $\Rightarrow$ Behaviour
Contracts – example

Alice has lost her kitten. She prints out a leaflet, and places a reward if the kitten is found.

Alice: “I promise that I will give 50 EU reward to anybody who finds my kitten”

This is a form of unilateral contract: only one party to the contract makes a promise.

SLAs are mostly unilateral contracts.
Contracts – example

Two kids, Alice and Bob, want to share their toys.

Alice: “I promise that I will lend my airplane, provided that I borrow a bike”

Bob: “I promise that I will lend my bike, provided that I borrow an airplane”

This is a form of bilateral contract: exactly two parties make a promise. Note the circularity between the guarantees and the assumptions.
Main decision procedures on contracts:

- Given a set of contracts, do they induce an **agreement** among the promising parties?

- In case an agreement is reached, which are the **duties** of each involved party?

- Given an execution context, whether the agreement has been **violated** or not.

- In case of violation, who is the **responsible**?
So far, we have characterized contracts by some desirable decision procedures they must support.

To characterize contract-based computations, we must also say how contracts can be manipulated.

- We will do that while abstracting from the actual formalism used to model contracts.
- We will also abstract from security issues, e.g.
  - sign a contract in the presence of attackers
  - prevent non-repudiation of a contract
Primitives for contracts

What are the key primitives for contract-based computing?

- Issue a contract
- Upon agreement, establish a session among all (and only) the involved parties
- Check if an obligation is due
- Check if an obligation has been fulfilled
- Join a session (e.g. if you are a judge who has to sanction the responsible of a violation)
Contracts as constraints

I said that we abstract from the actual formalism used to model contracts. This was half-true.

We will only require that contracts are arranged into a constraint system $(D, \vdash)$

- $C \vdash c$ means that a set of contracts $C$ induces an agreement, which prescribes some duty $c$
  
  \[
  \{ \text{kitten, kitten} \rightarrow \text{pay} \} \vdash \text{pay}
  \]

- $C \vdash \bot$ means that a set of contracts $C$ is inconsistent, i.e. $C \vdash c$ for all $c$
Our contract calculus is an extension of CCP, Concurrent Constraint Programming.

To issue a contract \( c \), we just “tell” it, i.e. we add \( c \) to the constraint store.

\[
\text{Alice} = \text{tell} \text{ kitten } \rightarrow \text{ pay}. \ P
\]

The behaviour of a \text{tell} \( c \) is to put \( c \) in parallel with the other processes:

\[
\text{Alice} \rightarrow \text{ kitten } \rightarrow \text{ pay} \ | \ P
\]
How to check if a set of contracts has given rise to an agreement?

We do not provide any primitive to do that, yet we exploit CCP $\text{ask}_c$ to check if a set of contracts induces an agreement which prescribes $c$.

$$\text{Alice} = \text{tell kitten} \rightarrow \text{pay. ask pay. } P$$

If $\text{ask pay}$ passes, then an agreement has been reached – somebody has found the kitten!
Consider a computation involving Alice and Bob:

\[
\begin{align*}
\text{Alice} &= \text{tell kitten } \rightarrow \text{pay. ask pay. } P \\
\text{Bob} &= \text{tell kitten. } Q
\end{align*}
\]

\[
\begin{align*}
\text{Alice} &\mid \text{Bob} \\
\rightarrow^* \\
\text{kitten} &\rightarrow \text{pay} \mid \text{kitten} \mid \text{ask pay. } P \mid Q
\end{align*}
\]
On distinguishing agreements

Now suppose Alice has lost two kittens.

\[
\text{Alice} = \text{tell} \text{ kitten} \rightarrow \text{pay}. \\
\text{tell} \text{ kitten} \rightarrow \text{pay}. \\
(\text{ask pay}.P \mid \text{ask pay}.Q)
\]

Assume that Bob finds only one kitten.

\[
\text{Alice} \mid \text{Bob} \rightarrow^* C \mid P \mid Q
\]
On distinguishing agreements

Now suppose Alice has lost two kittens.

\[
\text{Alice} = \text{tell} \text{ kitten } \rightarrow \text{ pay}. \\
\text{tell} \text{ kitten } \rightarrow \text{ pay}. \\
(\text{ask pay}.P \mid \text{ask pay}.Q)
\]

Assume that Bob finds only one kitten.

\[
\text{Alice} \mid \text{Bob} \rightarrow^* C \mid P \mid Q
\]

Alice pays twice for the same cat!
Sessions

We can use variables $x$ in contracts, to model session IDs.

\[
\text{Alice} = (x)(y) \text{tell} \text{ kitten}(x) \rightarrow \text{pay}(x).
\]

\[
\text{tell kitten}(y) \rightarrow \text{pay}(y).
\]

\[
\text{ask pay}(x).P \mid \text{ask pay}(y).Q
\]

\[
\text{Bob} = (z) \text{tell} \text{ kitten}(z).Z
\]

\[
\text{Carl} = (w) \text{tell} \text{ kitten}(w).W
\]
Asks vs. Sessions

The **ask** primitive does not work with sessions.

Consider e.g. the following computation:

\[
\text{Alice} \mid \text{Bob} \xrightarrow{*} (x)(y)(z) \\
\text{kitten}(x) \rightarrow \text{pay}(x) \mid \\
\text{kitten}(z) \mid \cdots \mid \\
\text{ask pay}(x). \ P \mid \cdots
\]

Here we do *not* reach an agreement, since:

\[
\{ \text{kitten}(x) \rightarrow \text{pay}(x), \text{kitten}(z) \} \not\vdash \text{pay}(x)
\]
To reach agreements within sessions, we extend CCP with a new primitive. We call it \textit{fuse}.

A \textit{fuse}_x c fuses $x$ with a set of variables in the context, in order to entail $c$.

If the entailment of $c$ is eventually reached, $x$ and the other variables are fused to a \textit{fresh name}, which models a fresh session ID.

\[
\{\text{kitten}(x) \rightarrow \text{pay}(x), \text{kitten}(z)\} \not \vdash \text{pay}(x)
\]

\[
\{\text{kitten}(n) \rightarrow \text{pay}(n), \text{kitten}(n)\} \vdash \text{pay}(n)
\]
Alice = (x) tell kitten(x) → pay(x). fuse_x pay(x). P

Bob = (z) tell kitten(z). Q

Alice | Bob →^*

(x)(z) kitten(x) → pay(x) | kitten(z) |

fuse_x pay(x). P | Q →^*

(n) kitten(n) → pay(n) | kitten(n) | P{ n/x } | Q{ n/z }
Circular assume-guarantee

For “circular” contracts we use a peculiar form of implication, called contractual implication, e.g.:

\[ \text{bike} \rightarrow \text{airplane} \]

means that Alice promises to lend her airplane, provided that someone promises to lend his bike.

\[ \text{PCL} = \text{IPC} + \rightarrow \]

\[ \{ \text{bike} \rightarrow \text{airplane}, \text{airplane} \rightarrow \text{bike} \} \vdash_{\text{PCL}} \text{airplane} \]
Alice = (x) tell bike(x) → airplane(x).

fuse_x airplane(x). lendAirplane

Bob = (z) tell airplane(z) → bike(z).

fuse_z bike(z). lendBike

Alice | Bob →∗

(n) bike(n) → airplane(n) | airplane(n) → bike(n) |
lendAirplane{n/x} | lendBike{n/z}
On fusions

A fuse$_x$ chooses a set of variables to fuse with $x$.

Is the choice of variables totally unconstrained?

For instance, do we want Carl to be included in the session between Alice and Bob?

Alice $|$ Bob $|$ Carl

$\rightarrow^*$

$(n)$ lendAirplane$_{n/x}$ $|$ lendBike$_{n/z}$ $|$ Carl$_{n/y}$
A fuse \( x \) chooses a set of variables to fuse with \( x \).

Is the choice of variables totally unconstrained?

For instance, do we want Carl to be included in the session between Alice and Bob?

Alice | Bob | Carl

\[ \rightarrow^* \]

\( n \) lendAirplane\( \{n/x\} \) | lendBike\( \{n/z\} \) | Carl\( \{n/y\} \)

No, since there is no need of involving Carl.
Minimal fusions

The first requirement we put on fusion is **minimality**: only those variables required for the entailment are fused.

\[ C = \{ \text{bike}(x) \rightarrow \text{airplane}(x), \text{airplane}(z) \rightarrow \text{bike}(z) \} \]

Consider the tentative entailment \( C\sigma \vdash \text{bike}(z)\sigma \).

\[ \sigma = \{ n/x \} \]
\[ \sigma = \{ n/x, n/z \} \]
\[ \sigma = \{ n/x, n/z, n/y \} \]
The first requirement we put on fusion is **minimality**: only those variables required for the entailment are fused.

\[ C = \{ \text{bike}(x) \rightarrow \text{airplane}(x), \text{airplane}(z) \rightarrow \text{bike}(z) \} \]

Consider the tentative entailment \( C\sigma \vdash \text{bike}(z)\sigma \).

\[ \sigma = \{ n/x \} \quad \text{no entailment} \]
\[ \sigma = \{ n/x, n/z \} \]
\[ \sigma = \{ n/x, n/z, n/y \} \]
The first requirement we put on fusion is **minimality**: only those variables required for the entailment are fused.

\[
C = \{ \text{bike}(x) \rightarrow \text{airplane}(x), \text{airplane}(z) \rightarrow \text{bike}(z) \}
\]

Consider the tentative entailment \( C\sigma \vdash \text{bike}(z)\sigma \).

\[
\sigma = \{n/x\} \quad \text{no entailment}
\]

\[
\sigma = \{n/x, n/z\} \quad \text{minimal}
\]

\[
\sigma = \{n/x, n/z, n/y\}
\]
The first requirement we put on fusion is **minimality**: only those variables required for the entailment are fused.

\[ C = \{ \text{bike}(x) \rightarrow \text{airplane}(x), \text{airplane}(z) \rightarrow \text{bike}(z) \} \]

Consider the tentative entailment \( C\sigma \vdash \text{bike}(z)\sigma \).

- \( \sigma = \{ n/x \} \) no entailment
- \( \sigma = \{ n/x, n/z \} \) minimal
- \( \sigma = \{ n/x, n/z, n/y \} \) not minimal
Local fusions

The second requirements that we put on fusions is that they can be local.

This just means that one does not need to observe the whole world to decide the minimality of a fusion.

If a subset of the constraints suffices to obtain a minimal fusion (w.r.t. that subset), it is ok.

Thus, we require minimality to hold on a subset of the constraints at hand.
A fusion $\sigma = \{n/\vec{z}\}$ is **local minimal** for $C$ and $c$ if and only if:

$$\exists \ C' \subseteq C : \quad C' \sigma \vdash c \sigma \land \forall \vec{w} \subset \vec{z} : \ C' \{n/\vec{w}\} \vdash c\{n/\vec{w}\}$$
Join

A fuse establishes a session among all the parties involved in an agreement.

But how to join a session?

A $\text{join}_x \ c$ instantiates $x$ to any known name, provided that after doing so $c$ is entailed.

We are quite liberal about joining sessions. Indeed, nobody can prevent a judge from inspecting what happens in an agreement.
A **check** $c$ checks if $c$ is consistent with the set of constraints (as in CCP).

Mixing **join** and **check** we can decide if some promise has not been respected.

Buyer $= (x) \text{tell } \text{send}(x) \rightarrow \text{pay}(x). \text{fuse}_x \text{pay}(x). B$

Seller $= (y) \text{tell } \text{pay}(y) \rightarrow \text{send}(y). \text{fuse}_y \text{send}(y).$

$\tau. \text{tell } \text{sent}(y) + \tau. \text{NoSend}$
A check $c$ checks if $c$ is consistent with the set of constraints (as in CCP).

Mixing join and check we can decide if some promise has not been respected.

\[
\text{Buyer} = (x) \text{tell } \text{send}(x) \implies \text{pay}(x). \text{fuse}_x \text{pay}(x). B
\]

\[
\text{Seller} = (y) \text{tell } \text{pay}(y) \implies \text{send}(y). \text{fuse}_y \text{send}(y).
\]

\[
\tau.\text{tell } \text{sent}(y) + \tau.\text{NoSend}
\]

\[
\text{Judge} = (z) \text{join}_z \text{send}(z). \text{check } \neg\text{sent}(z). \text{JailSeller}
\]
A contract calculus

\[ \pi ::= \tau \mid \text{tell } c \mid \text{ask } c \mid \text{fuse}_x c \mid \text{join}_x c \mid \text{check } c \]

\[ P ::= c \quad \text{constraint} \]

\[ \sum_{i \in I} \pi_i \cdot P_i \quad \text{(guarded) summation} \]

\[ P \mid P \quad \text{parallel composition} \]

\[ (a)P \quad \text{delimitation} \]

\[ X(\vec{a}) \quad \text{constant} \]
Join: how it works

\[
\text{join}_x \ c. P \xrightarrow{\emptyset \vdash J_x c} P
\]

\[P \xrightarrow{(\vec{a})C} P' \quad Q \xrightarrow{(\vec{b})(C' \vdash J_x c)} Q'\]

\[P|Q \xrightarrow{(\vec{a}\vec{b})(C \cup C' \vdash J_x c)} P'|Q'\]

\[P \xrightarrow{(xn\vec{a})(C \vdash J_x c)} P' \quad \text{if} \quad C\sigma \vdash c\sigma, \quad \sigma = \{n/x\}\]

\[P \xrightarrow{\tau} (n\vec{a})P'\sigma\]
Fuse: how it works

\[
\text{fuse}_x c. P \xrightarrow{\emptyset \vdash^F x \ c} P
\]

\[
\begin{align*}
P & \xrightarrow{\vec{a} \ C} P' \\
Q & \xrightarrow{\vec{b} \ (C' \vdash^F x \ c)} Q'
\end{align*}
\]

\[
P| Q \xrightarrow{\vec{a} \vec{b} \ (C \cup C' \vdash^F x \ c)} P'| Q'
\]

\[
P \xrightarrow{(x \vec{y} \vec{a}) \ (C \vdash^F x \ c)} P'
\]

\[
P \xrightarrow{\tau} (n \vec{a}) (P' \sigma)
\]

\[
\text{if } \sigma = \{n/x\vec{y}\} \quad n \text{ fresh}
\]

\[
C \vdash^{loc}_\sigma c
\]

[fuse]

[parfuse]

[closefuse]
Fuse: ICE vs. LICS

\[
P \xrightarrow{(xy\vec{d})(C \vdash^F_x c)} P' \quad C \vdash^\text{loc}_\sigma c
\]
\[
if \quad \sigma = \{n/xy\vec{y}\} \quad n \text{ fresh} \quad [\text{ICE}]
\]

\[
P \xrightarrow{\tau} (n\vec{a})(P'\sigma)
\]

\[
P \xrightarrow{(x\vec{y}\vec{m}\vec{a})(C \vdash^F_x c)} P' \quad C \vdash^\text{loc}_\sigma c
\]
\[
if \quad \sigma(x) = n \text{ fresh} \quad \sigma(\vec{y}) \subseteq n\vec{m} \quad [\text{LICS}]
\]
Principals

\[ c_B = n_B \text{ says } ((x_S \text{ says } \text{send}(x) \rightarrow \text{pay}(x))) \]
\[ c_S = n_S \text{ says } ((y_B \text{ says } \text{pay}(y)) \rightarrow \text{send}(y)) \]

\[ B = (x)(x_S)(n_B) (\text{tell } c_B. \ \text{fuse}_x (n_B \text{ says } \text{pay}(x)). B') \]
\[ S = (y)(y_B)(n_S) (\text{tell } c_S. \ \text{fuse}_y (n_S \text{ says } \text{send}(y)). S') \]

\[ \sigma = \{n/x, n/y, n_S/x_S, n_B/y_B\} \]
Questions?