$$$: $services, $essions and $treams

Roberto Bruni

Dipartimento di Informatica
Università di Pisa

SENSORIA
Theme 1 Meeting
Florence, Italy, January 17–19, 2007

Outline

Outline

A General Theory of Services

The strategy of SENSORIA

Integration of foundational theories, techniques, methods and tools in a pragmatic software engineering approach.

The role of process calculi

A crucial role in the project will be played by formalisms for service description that can lay the mathematical basis for analysing and experimenting with components interactions, and for combining services.

Core calculi (WP 2)

Several calculi have emerged in months 0-12, each based on a small set of primitives that might serve as a basis for formalizing and programming service oriented applications.
Months 13-30 will be used to assess and refine them.

A General Theory of Services

The strategy of SENSORIA

Integration of foundational theories, techniques, methods and tools in a pragmatic software engineering approach.

The role of process calculi

A crucial role in the project will be played by formalisms for service description that can lay the mathematical basis for analysing and experimenting with components interactions, and for combining services.

Core calculi (WP 2)

Several calculi have emerged in months 0-12, each based on a small set of primitives that might serve as a basis for formalizing and programming service oriented applications.
Months 13-30 will be used to assess and refine them.

A General Theory of Services

The strategy of SENSORIA

Integration of foundational theories, techniques, methods and tools in a pragmatic software engineering approach.

The role of process calculi

A crucial role in the project will be played by formalisms for service description that can lay the mathematical basis for analysing and experimenting with components interactions, and for combining services.

Core calculi (WP 2)

Several calculi have emerged in months 0-12, each based on a small set of primitives that might serve as a basis for formalizing and programming service oriented applications.
Months 13-30 will be used to assess and refine them.

This talk

SCC: Service Centered Calculus

- A guided tour of SCC
- Prototypes and work in progress
- Pragmatic evidence that SCC reconcile theory and practice

To keep in mind

- We are dealing with conceptual abstractions
- The syntax does not necessarily reflect implementation details

Example 1

All service instances (serving different requests) can be handled by one service port

Example 2

A session is a logical entity that can be implemented by an additional *sid* parameter carried by all related messaging

This talk

SCC: Service Centered Calculus

- A guided tour of SCC
- Prototypes and work in progress
- Pragmatic evidence that SCC reconcile theory and practice

To keep in mind

- We are dealing with conceptual abstractions
- The syntax does not necessarily reflect implementation details

Example 1

All service instances (serving different requests) can be handled by one service port

Example 2

A session is a logical entity that can be implemented by an additional *sid* parameter carried by all related messaging

This talk

SCC: Service Centered Calculus

- A guided tour of SCC
- Prototypes and work in progress
- Pragmatic evidence that SCC reconcile theory and practice

To keep in mind

- We are dealing with conceptual abstractions
- The syntax does not necessarily reflect implementation details

Example 1

All service instances (serving different requests) can be handled by one service port

Example 2

A session is a logical entity that can be implemented by an additional *sid* parameter carried by all related messaging

Outline

Service Centered Calculus: General Principles

Service definitions

- services can be deployed dynamically
- services expose their protocols
- services can handle multiple requests separately
- services can be shut down and updated

Service invocations

- service calls expose their protocols
- multiple invocations by data streaming

Sessions

- sessions are two-party (service-side + client-side)
- service invocation spawn fresh session parties (locally to each partner)
- communication within sessions is bi-directional
- sessions can be nested
- values can be returned outside sessions (one level up)

Service Centered Calculus: General Principles

Service definitions

- services can be deployed dynamically
- services expose their protocols
- services can handle multiple requests separately
- services can be shut down and updated

Service invocations

- service calls expose their protocols
- multiple invocations by data streaming

Sessions

- sessions are two-party (service-side + client-side)
- service invocation spawn fresh session parties (locally to each partner)
- communication within sessions is bi-directional
- sessions can be nested
- values can be returned outside sessions (one level up)

Service Centered Calculus: General Principles

Service definitions

- services can be deployed dynamically
- services expose their protocols
- services can handle multiple requests separately
- services can be shut down and updated

Service invocations

- service calls expose their protocols
- multiple invocations by data streaming

Sessions

- sessions are two-party (service-side + client-side)
- service invocation spawn fresh session parties (locally to each partner)
- communication within sessions is bi-directional
- sessions can be nested
- values can be returned outside sessions (one level up)

Service Centered Calculus: Session Closing Principles

Session termination

- service definitions expose *generic* termination handlers (processes)
- service invocations expose *specific* termination handlers (service names)
- the local closure of a session activates partner's handler (if any)
- local session termination: autonomous + on partner's request
- session termination cancels all locally nested processes (including service definitions)

Sources of inspiration

We have integrated complementary aspects from

- π-calculus (names handling primitives)
- Orc (pipelining and pruning of activities)
- webπ, cjoin, Sagas (primitives for LRT and compensations)

Service Centered Calculus: Session Closing Principles

Session termination

- service definitions expose *generic* termination handlers (processes)
- service invocations expose *specific* termination handlers (service names)
- the local closure of a session activates partner's handler (if any)
- local session termination: autonomous + on partner's request
- session termination cancels all locally nested processes (including service definitions)

Sources of inspiration

We have integrated complementary aspects from

- π-calculus (names handling primitives)
- Orc (pipelining and pruning of activities)
- webπ, cjoin, Sagas (primitives for LRT and compensations)

Service Definition

Service definition

$$s \Rightarrow (x)P$$

- s is the service name
- x is the formal parameter
- P is the actual implementation of the service.

Examples: Successor and prime teller

$$\texttt{succ} \Rightarrow (x)x + 1$$

Received an integer communicates back its successor.

$$\texttt{prime} \Rightarrow (n)P$$

Received an integer n communicates back the n-th prime number.

Service Definition

Service definition

$$s \Rightarrow (x)P$$

- s is the service name
- x is the formal parameter
- P is the actual implementation of the service.

Examples: Successor and prime teller

$$\texttt{succ} \Rightarrow (x)x + 1$$

Received an integer communicates back its successor.

$$\texttt{prime} \Rightarrow (n)P$$

Received an integer n communicates back the n-th prime number.

Service Invocation

Service invocation

$$s\{(x)P\} \Leftarrow Q$$

- each new value v produced by the client Q will trigger a new invocation of service s (like Orc sequencing $Q > x > P$)
- for each invocation, a suitable instance $P\{^{v}/_{x}\}$ of the process P, implements the client-side protocol

Example: A sample client

$$\texttt{prime}\{(x)(y)\text{return } y\} \Leftarrow 5$$

Shorthand notation

The client side makes no use of the formal parameter x: we abbreviate it as

$$\texttt{prime}\{(-)(y)\text{return } y\} \Leftarrow 5$$

Service Invocation

Service invocation

$$s\{(x)P\} \Leftarrow Q$$

- each new value v produced by the client Q will trigger a new invocation of service s (like Orc sequencing $Q > x > P$)
- for each invocation, a suitable instance $P\{^{v}/_{x}\}$ of the process P, implements the client-side protocol

Example: A sample client

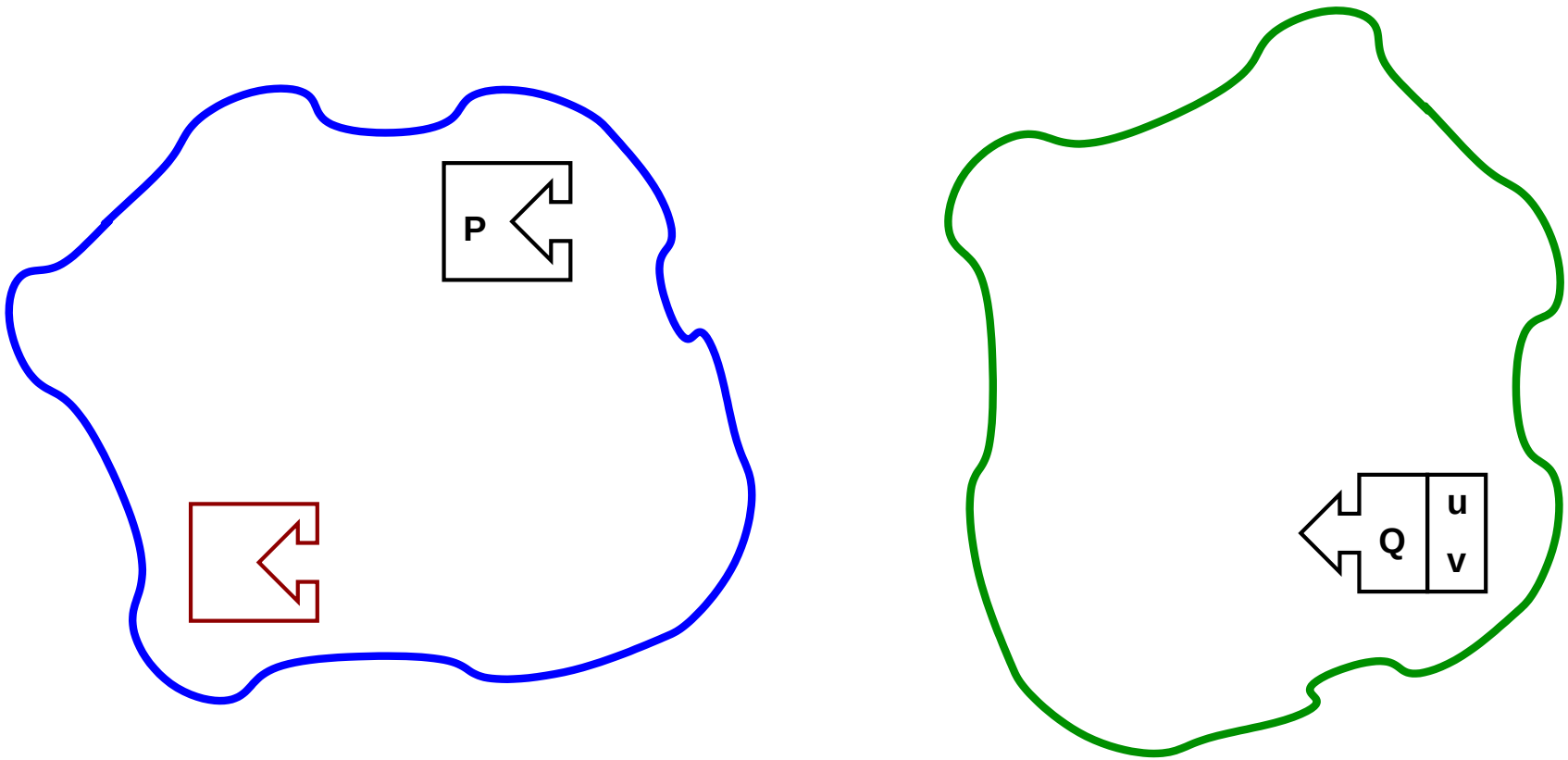
$$\texttt{prime}\{(x)(y)\text{return } y\} \Leftarrow 5$$

Shorthand notation

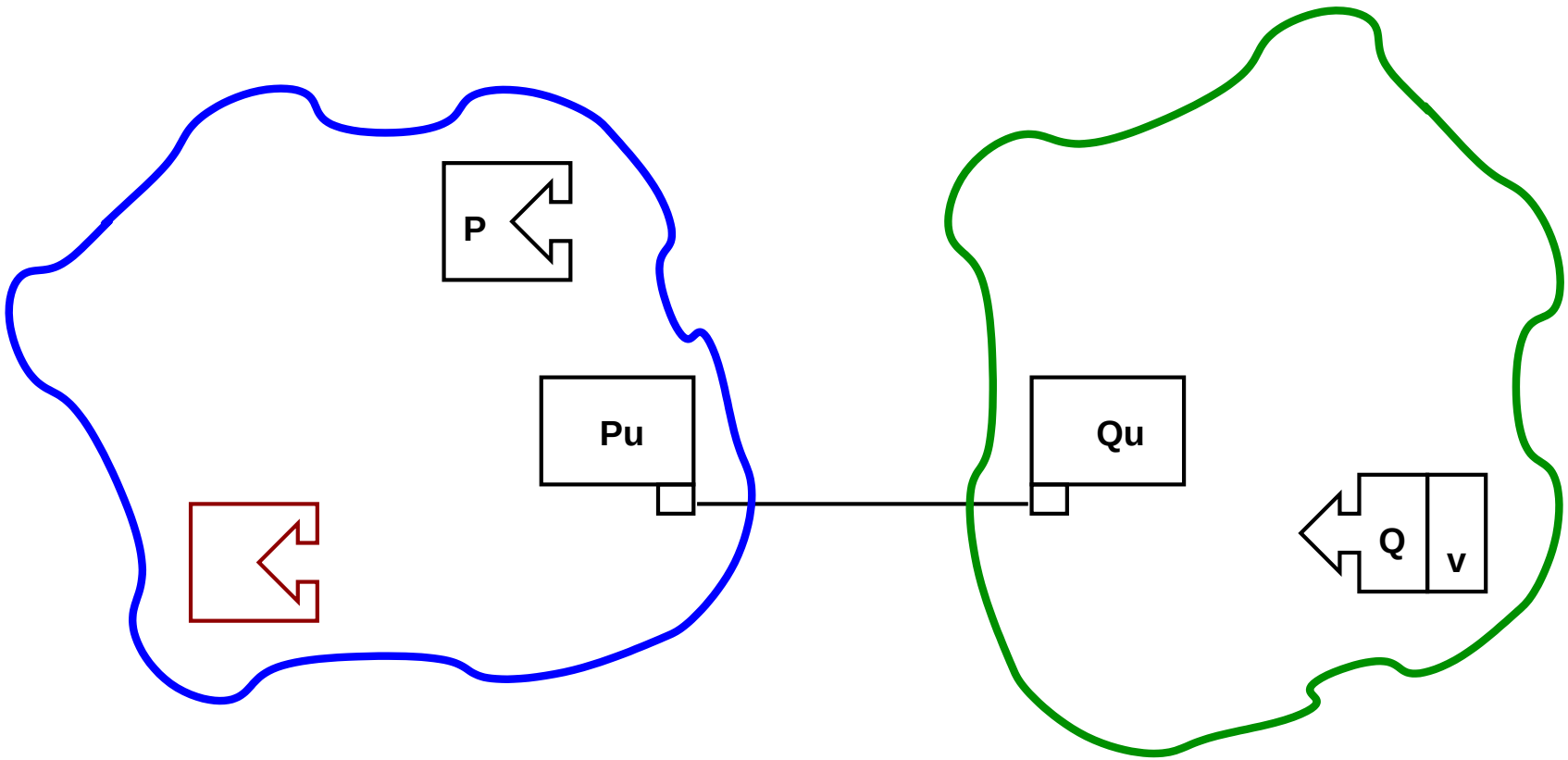
The client side makes no use of the formal parameter x: we abbreviate it as

$$\texttt{prime}\{(-)(y)\text{return } y\} \Leftarrow 5$$

Service Invocation, Graphically



Bidirectional Session, Graphically



Service Activation

Service activation

$$\mathbb{C}[\![\, s \Rightarrow (x)P \,]\!] \mid \mathbb{D}[\![\, s\{(y)P'\} \Leftarrow (Q \mid u.R) \,]\!] \;\rightarrow\; (\nu r)\left(\mathbb{C}[\![\, r \triangleright P\{^{u}/_{x}\} \mid s \Rightarrow (x)P \,]\!] \mid \mathbb{D}[\![\, \overline{r} \triangleright P'\{^{u}/_{y}\} \mid s\{(y)P'\} \Leftarrow (Q \mid R) \,]\!] \right)$$

if r is fresh and u, s not bound by $\mathbb{C}$,$\mathbb{D}$

A service invocation causes activation of a new session:

- dual fresh identifiers, r and $\overline{r}$, name the two sides of the session
- client and service protocols run each at the proper side of the session

Example: Asking for prime numbers

The invocation of service `prime` triggers the session

$$(\nu r)\left(\ldots r \triangleright P\{^{5}/_{n}\} \ldots \mid \ldots \overline{r} \triangleright (z)\mathrm{return}\, z \ldots \right)$$

The client waits for a value from the server (11) to be substituted for z

Service Activation

Service activation

$$\mathbb{C}[\![\, s \Rightarrow (x)P \,]\!] \mid \mathbb{D}[\![\, s\{(y)P'\} \Leftarrow (Q \mid u.R) \,]\!] \;\rightarrow\; (\nu r)\left(\mathbb{C}[\![\, r \rhd P\{^{u}/_{x}\} \mid s \Rightarrow (x)P \,]\!] \mid \mathbb{D}[\![\, \overline{r} \rhd P'\{^{u}/_{y}\} \mid s\{(y)P'\} \Leftarrow (Q \mid R) \,]\!] \right)$$

if r is fresh and u, s not bound by $\mathbb{C}$,$\mathbb{D}$

A service invocation causes activation of a new session:

- dual fresh identifiers, r and $\overline{r}$, name the two sides of the session
- client and service protocols run each at the proper side of the session

Example: Asking for prime numbers

The invocation of service `prime` triggers the session

$$(\nu r)\left(\ldots r \rhd P\{^{5}/_{n}\} \ldots \mid \ldots \overline{r} \rhd (z)\mathrm{return}\, z \ldots \right)$$

The client waits for a value from the server (11) to be substituted for z

Service Activation

Service activation

$$\mathbb{C}[\![\, s \Rightarrow (x)P \,]\!] \mid \mathbb{D}[\![\, s\{(y)P'\} \Leftarrow (Q|u.R) \,]\!] \;\rightarrow\; (\nu r)\left(\mathbb{C}[\![\, r \triangleright P\{^{u}/_{x}\} \mid s \Rightarrow (x)P \,]\!] \mid \mathbb{D}[\![\, \overline{r} \triangleright P'\{^{u}/_{y}\} \mid s\{(y)P'\} \Leftarrow (Q|R) \,]\!] \right)$$

if r is fresh and u, s not bound by $\mathbb{C}$,$\mathbb{D}$

A service invocation causes activation of a new session:

- dual fresh identifiers, r and $\overline{r}$, name the two sides of the session
- client and service protocols run each at the proper side of the session

Example: Asking for prime numbers

The invocation of service `prime` triggers the session

$$(\nu r)\left(\ldots r \triangleright P\{^{5}/_{n}\} \ldots \mid \ldots \overline{r} \triangleright (z)\mathsf{return}\, z \ldots \right)$$

The client waits for a value from the server (11) to be substituted for z

Service Activation

Service activation

$$\begin{array}{l}\mathbb{C}[\![\, s \Rightarrow (x)P \,]\!] \mid \\ \mathbb{D}[\![\, s\{(y)P'\} \Leftarrow (Q|u.R) \,]\!]\end{array} \quad \rightarrow \quad (\nu r)\left(\begin{array}{l}\mathbb{C}[\![\, r \rhd P\{^{u}/_{x}\} \mid s \Rightarrow (x)P \,]\!] \mid \\ \mathbb{D}[\![\, \overline{r} \rhd P'\{^{u}/_{y}\} \mid s\{(y)P'\} \Leftarrow (Q|R) \,]\!]\end{array}\right)$$

if r is fresh and u, s not bound by $\mathbb{C}$,$\mathbb{D}$

A service invocation causes activation of a new session:

- dual fresh identifiers, r and $\overline{r}$, name the two sides of the session
- client and service protocols run each at the proper side of the session

Example: Asking for prime numbers

The invocation of service `prime` triggers the session

$$(\nu r)\big(\,\ldots\, r \rhd P\{^{5}/_{n}\}\,\ldots \mid \ldots\, \overline{r} \rhd (z)\mathsf{return}\, z\,\ldots\,\big)$$

The client waits for a value from the server (11) to be substituted for z

Intra-Session Communication, Graphically

Intra-Session Communication, Graphically

Intra-Session Communication, Graphically

Session Communication

Session communication

$$\begin{array}{l}\mathbb{C}[\![\, r \triangleright (\, P \mid u.Q \,) \,]\!] \mid \\ \mathbb{D}[\![\, \overline{r} \triangleright (\, R \mid (z)S \,) \,]\!]\end{array} \quad \longrightarrow \quad \begin{array}{l}\mathbb{C}[\![\, r \triangleright (\, P \mid Q \,) \,]\!] \mid \\ \mathbb{D}[\![\, \overline{r} \triangleright (\, R \mid S\{^{u}/_{z}\} \,) \,]\!]\end{array}$$

if u, r not bound by $\mathbb{C}$,$\mathbb{D}$

Within sessions, communication is bi-directional, in the sense that the interacting protocols can exchange data in both directions.

Example: 5th prime evaluated and communicated

After the value 11 has been computed, it can be communicated:

$$(\nu r)(\, \ldots r \triangleright 11 \ldots \mid \ldots \overline{r} \triangleright (z)\mathsf{return}\, z \ldots \,)$$

Session Communication

Session communication

$$\begin{array}{l} \mathbb{C}[\![\, r \triangleright (\, P \mid u.Q \,)\,]\!] \mid \\ \mathbb{D}[\![\, \overline{r} \triangleright (\, R \mid (z)S \,)\,]\!] \end{array} \quad \longrightarrow \quad \begin{array}{l} \mathbb{C}[\![\, r \triangleright (\, P \mid Q \,)\,]\!] \mid \\ \mathbb{D}[\![\, \overline{r} \triangleright (\, R \mid S\{^{u}/_{z}\} \,)\,]\!] \end{array}$$

if u, r not bound by $\mathbb{C}$,$\mathbb{D}$

Within sessions, communication is bi-directional, in the sense that the interacting protocols can exchange data in both directions.

Example: 5th prime evaluated and communicated

After the value 11 has been computed, it can be communicated:

$$(\nu r)(\, \ldots\, r \triangleright 11 \ldots \mid \ldots\, \overline{r} \triangleright (z)\mathsf{return}\, z \ldots\,)$$

$$(\nu r)(\, \ldots\, r \triangleright 0 \ldots \mid \ldots\, r \triangleright \mathsf{return}\, 11 \ldots\,)$$

Session Communication

Session communication

$$\begin{array}{l} \mathbb{C}[\![\, r \triangleright (\, P \mid u.Q \,)\,]\!] \mid \\ \mathbb{D}[\![\, \overline{r} \triangleright (\, R \mid (z)S \,)\,]\!] \end{array} \quad \rightarrow \quad \begin{array}{l} \mathbb{C}[\![\, r \triangleright (\, P \mid Q \,)\,]\!] \mid \\ \mathbb{D}[\![\, \overline{r} \triangleright (\, R \mid S\{^{u}/_{z}\} \,)\,]\!] \end{array}$$

$$\text{if } u, r \text{ not bound by } \mathbb{C},\mathbb{D}$$

Within sessions, communication is bi-directional, in the sense that the interacting protocols can exchange data in both directions.

Example: 5th prime evaluated and communicated

After the value 11 has been computed, it can be communicated:

$$(\nu r)(\, \ldots r \triangleright 11 \ldots \mid \ldots \overline{r} \triangleright (z)\text{return } z \ldots\,)$$

$$(\nu r)(\, \ldots r \triangleright 0 \ldots \mid \ldots \overline{r} \triangleright \text{return } 11 \ldots\,)$$

Session Communication

Session communication

$$\begin{array}{l}\mathbb{C}[\![\, r \triangleright (\, P \mid u.Q \,) \,]\!] \mid \\ \mathbb{D}[\![\, \overline{r} \triangleright (\, R \mid (z)S \,) \,]\!]\end{array} \quad \rightarrow \quad \begin{array}{l}\mathbb{C}[\![\, r \triangleright (\, P \mid Q \,) \,]\!] \mid \\ \mathbb{D}[\![\, \overline{r} \triangleright (\, R \mid S\{^{u}/_{z}\} \,) \,]\!]\end{array}$$

$$\text{if } u, r \text{ not bound by } \mathbb{C},\mathbb{D}$$

Within sessions, communication is bi-directional, in the sense that the interacting protocols can exchange data in both directions.

Example: 5th prime evaluated and communicated

After the value 11 has been computed, it can be communicated:

$$(\nu r)\big(\, \ldots r \triangleright 11 \ldots \mid \ldots \overline{r} \triangleright (z)\mathsf{return}\, z \ldots \,\big)$$

$$(\nu r)\big(\, \ldots r \triangleright \mathbf{0} \ldots \mid \ldots \overline{r} \triangleright \mathsf{return}\, 11 \ldots \,\big)$$

Session Communication

Session communication

$$\begin{array}{l}\mathbb{C}[\![\, r \triangleright (\, P \mid u.Q \,) \,]\!] \mid \\ \mathbb{D}[\![\, \overline{r} \triangleright (\, R \mid (z)S \,) \,]\!]\end{array} \quad \rightarrow \quad \begin{array}{l}\mathbb{C}[\![\, r \triangleright (\, P \mid Q \,) \,]\!] \mid \\ \mathbb{D}[\![\, \overline{r} \triangleright (\, R \mid S\{^{u}/_{z}\} \,) \,]\!]\end{array}$$

$$\text{if } u, r \text{ not bound by } \mathbb{C},\mathbb{D}$$

Within sessions, communication is bi-directional, in the sense that the interacting protocols can exchange data in both directions.

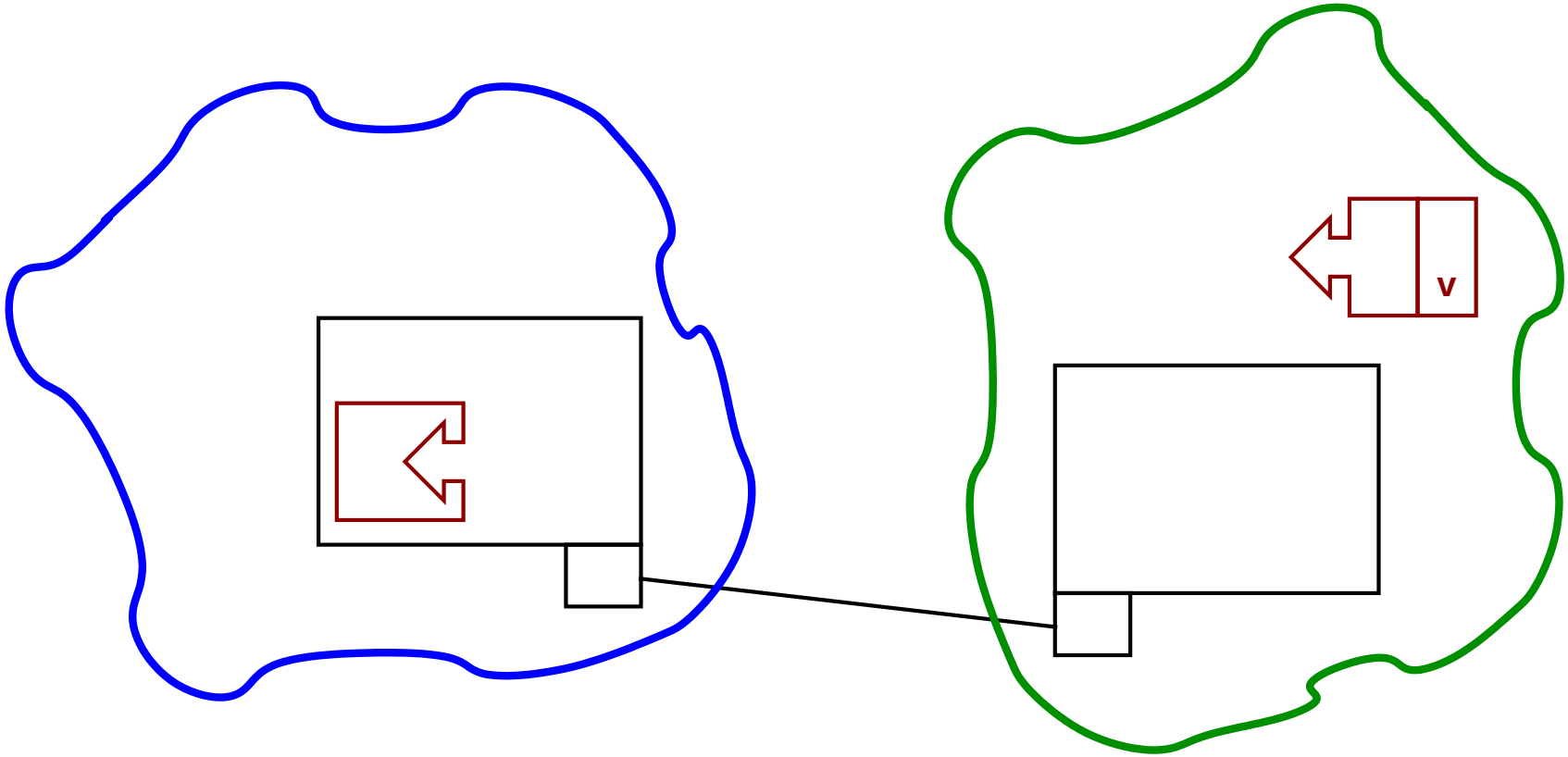
Example: 5th prime evaluated and communicated

After the value 11 has been computed, it can be communicated:

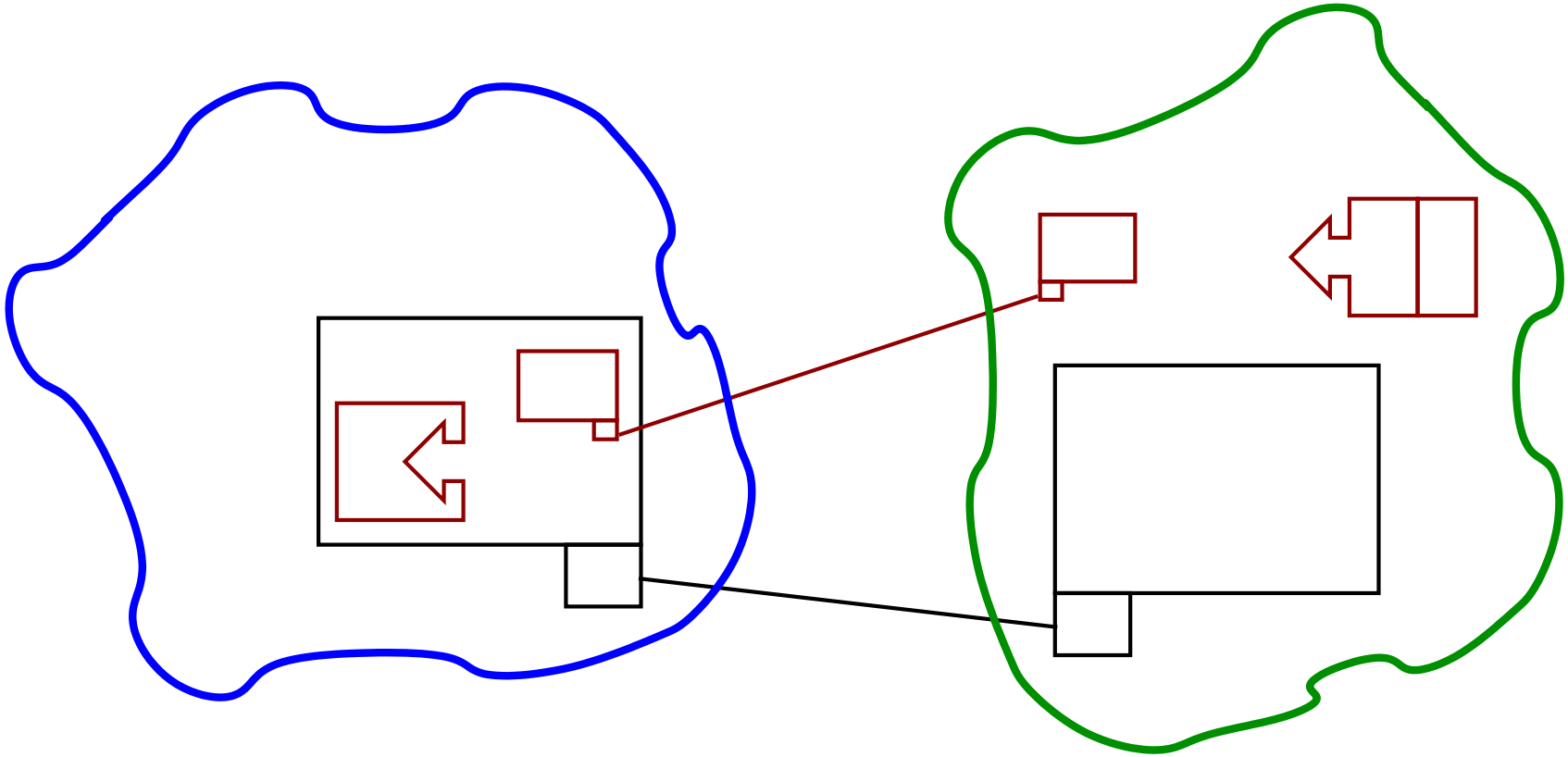
$$(\nu r)\big(\, \ldots r \triangleright 11 \ldots \mid \ldots \overline{r} \triangleright (z)\mathsf{return}\, z \ldots \big)$$

$$(\nu r)\big(\, \ldots r \triangleright \mathbf{0} \ldots \mid \ldots \overline{r} \triangleright \mathsf{return}\, 11 \ldots \big)$$

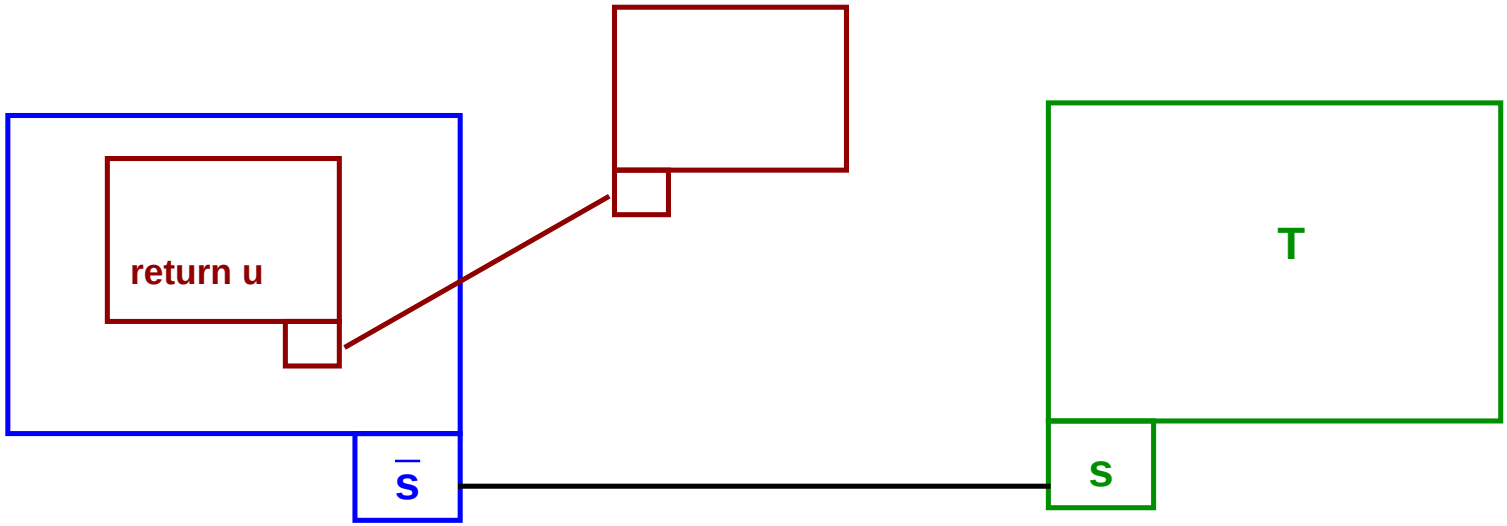
Nested Services and Multi-Sessions, Graphically



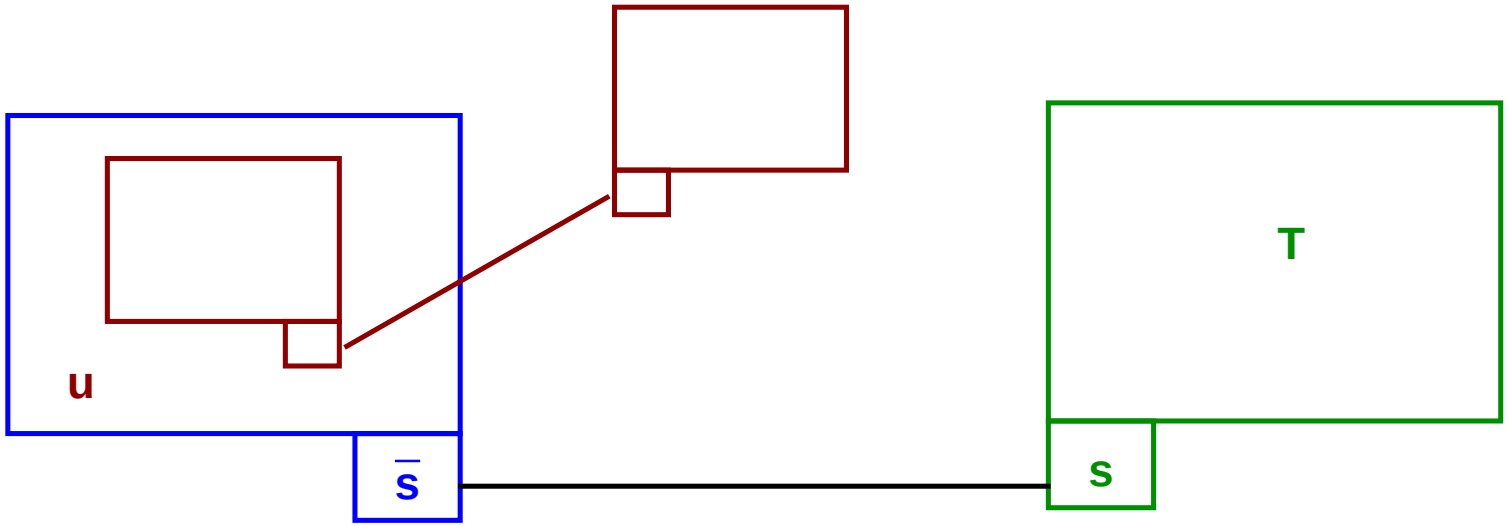
Nested Services and Multi-Sessions, Graphically



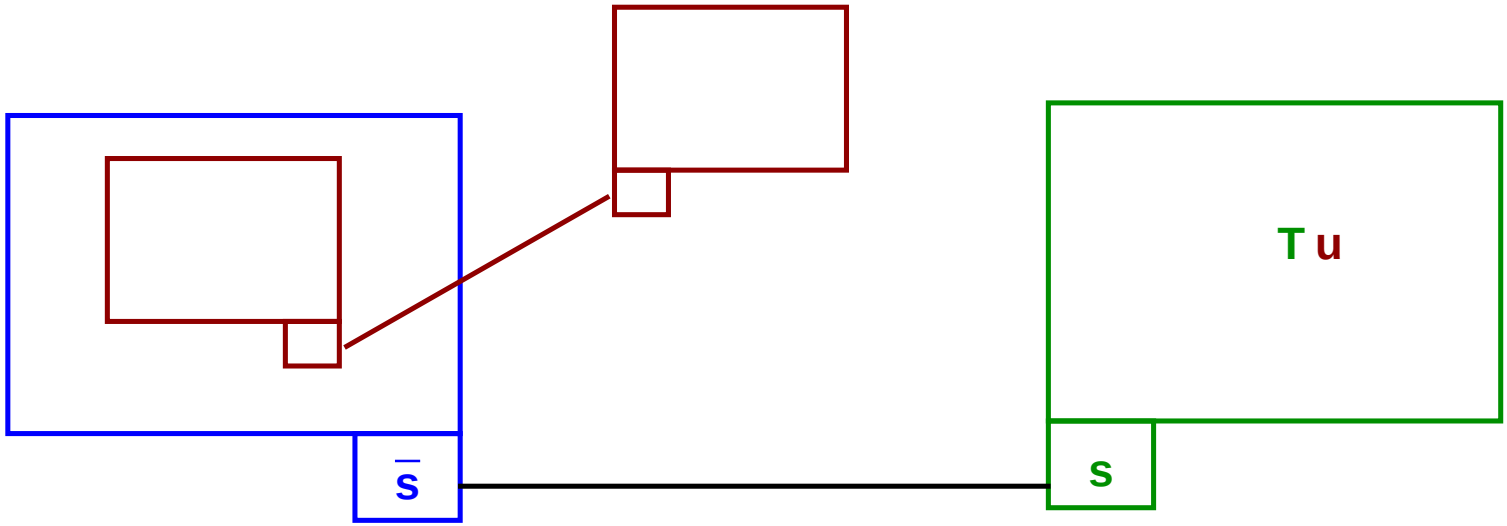
Returning Values, Graphically



Returning Values, Graphically



Returning Values, Graphically



Session Returning Values

Session returning values

$$r \triangleright (P \mid \text{return } u.Q) \rightarrow u \mid r \triangleright (P \mid Q)$$

Values can be returned outside the session to the enclosing environment and used for invoking other services.

Example: Returning the 5th prime number

$$(\nu r)(\ldots r \triangleright \mathbf{0} \ldots \mid \ldots 11 \mid \overline{r} \triangleright \mathbf{0} \ldots)$$

A taste of structural congruence

Terminated protocols are immaterial

$$r \triangleright \mathbf{0} \quad \equiv \quad \mathbf{0}$$

Session Returning Values

Session returning values

$$r \triangleright (P \mid \text{return } u.Q) \;\rightarrow\; u \mid r \triangleright (P \mid Q)$$

Values can be returned outside the session to the enclosing environment and used for invoking other services.

Example: Returning the 5th prime number

$$(\nu r)\big(\ldots r \triangleright \mathbf{0} \ldots \mid \ldots 11 \mid \overline{r} \triangleright \mathbf{0} \ldots \big)$$

A taste of structural congruence

Terminated protocols are immaterial

$$r \triangleright \mathbf{0} \quad \equiv \quad \mathbf{0}$$

Session Returning Values

Session returning values

$$r \triangleright (P \mid \text{return } u.Q) \;\longrightarrow\; u \mid r \triangleright (P \mid Q)$$

Values can be returned outside the session to the enclosing environment and used for invoking other services.

Example: Returning the 5th prime number

$$(\nu r)\big(\ldots r \triangleright \mathbf{0} \ldots \mid \ldots 11 \mid \overline{r} \triangleright \mathbf{0} \ldots \big)$$

A taste of structural congruence

Terminated protocols are immaterial

$$r \triangleright \mathbf{0} \quad \equiv \quad \mathbf{0}$$

Request-Response

A "functional" protocol

A common pattern of service invocation is:

$$s\{(-)(y)\text{return } y\} \Leftarrow P$$

where s is invoked on every value that P produces

Shorthand notation

$$s \Leftarrow P$$

Example: Successor of a prime

We write

$$\texttt{succ} \Leftarrow (\texttt{prime} \Leftarrow 5)$$

instead of $\texttt{succ}\{(-)(w)\text{return } w\} \Leftarrow (\texttt{prime}\{(-)(y)\text{return } y\} \Leftarrow 5)$

Request-Response

A "functional" protocol

A common pattern of service invocation is:

$$s\{(-)(y)\text{return } y\} \Leftarrow P$$

where s is invoked on every value that P produces

Shorthand notation

$$s \Leftarrow P$$

Example: Successor of a prime

We write

$$\texttt{succ} \Leftarrow (\texttt{prime} \Leftarrow 5)$$

instead of $\texttt{succ}\{(-)(w)\text{return } w\} \Leftarrow (\texttt{prime}\{(-)(y)\text{return } y\} \Leftarrow 5)$

Request-Response

A "functional" protocol

A common pattern of service invocation is:

$$s\{(-)(y)\text{return } y\} \Leftarrow P$$

where s is invoked on every value that P produces

Shorthand notation

$$s \Leftarrow P$$

Example: Successor of a prime

We write

$$\texttt{succ} \Leftarrow (\texttt{prime} \Leftarrow 5)$$

instead of $\texttt{succ}\{(-)(w)\text{return } w\} \Leftarrow (\texttt{prime}\{(-)(y)\text{return } y\} \Leftarrow 5)$

One-Way

Vacuous protocol

If no reply is expected from a service, the client can employ a vacuous protocol

$$a\{(-)\mathbf{0}\} \Leftarrow P$$

Shorthand notation

$$a\{\} \Leftarrow P$$

Example: Printing values

A client invokes the service `prime` and then prints the result:

$$\texttt{print}\{\} \Leftarrow (\,\texttt{prime} \Leftarrow 5\,)$$

In this case, the service `print` is invoked with vacuous protocol $(z)\mathbf{0}$

One-Way

Vacuous protocol

If no reply is expected from a service, the client can employ a vacuous protocol

$$a\{(-)\mathbf{0}\} \Leftarrow P$$

Shorthand notation

$$a\{\} \Leftarrow P$$

Example: Printing values

A client invokes the service `prime` and then prints the result:

$$\texttt{print}\{\} \Leftarrow (\,\texttt{prime} \Leftarrow 5\,)$$

In this case, the service `print` is invoked with vacuous protocol $(z)\mathbf{0}$

One-Way

Vacuous protocol

If no reply is expected from a service, the client can employ a vacuous protocol

$$a\{(-)\mathbf{0}\} \Leftarrow P$$

Shorthand notation

$$a\{\} \Leftarrow P$$

Example: Printing values

A client invokes the service `prime` and then prints the result:

$$\texttt{print}\{\} \Leftarrow (\,\texttt{prime} \Leftarrow 5\,)$$

In this case, the service `print` is invoked with vacuous protocol $(z)\mathbf{0}$

PSC Syntax

Grammar

P, Q ::=	$\mathbf{0}$	Nil
	$\mid a.P$	Concretion (pass a to session partner)
	$\mid (x)P$	Abstraction (take from session partner)
	$\mid \mathsf{return}\, a.P$	Return Value (out of current session)
	$\mid s \Rightarrow (x)P$	Service Definition
	$\mid s\{(x)P\} \Leftarrow Q$	Service Invocation
	$\mid r \rhd P$	Session Side
	$\mid P \mid Q$	Parallel Composition
	$\mid (\nu a)P$	New Name

We call it PSC for *persistent session calculus*:

- sessions can be established
- a session can be garbage collected when the protocol has run entirely,
- but sessions can neither be aborted nor closed by one of the parties

PSC Structural Congruence

Axioms

$$
\begin{array}{rcll}
P & \equiv & Q & \text{if } P =_\alpha Q \\
\\
(P \mid Q) \mid R & \equiv & P \mid (Q \mid R) & \\
P \mid Q & \equiv & Q \mid P & \\
P \mid \mathbf{0} & \equiv & P & \\
\\
(\nu x)(\nu y)P & \equiv & (\nu y)(\nu x)P & \\
(\nu x)\mathbf{0} & \equiv & \mathbf{0} & \\
\\
P \mid (\nu x)Q & \equiv & (\nu x)(P \mid Q) & \text{if } x \notin fn(P) \\
r \rhd (\nu x)P & \equiv & (\nu x)(r \rhd P) & \text{if } x \notin \{r, \overline{r}\} \\
s\{(x)P\} \Leftarrow (\nu y)Q & \equiv & (\nu y)(r\{(x)P\} \Leftarrow Q) & \text{if } y \notin fn((x)P) \cup \{r, \overline{r}\} \\
\\
r \rhd \mathbf{0} & \equiv & \mathbf{0} &
\end{array}
$$

PSC Operational Semantics

Active contexts

$$\mathbb{C}, \mathbb{D} \;::=\; [\![\cdot]\!] \;\mid\; \mathbb{C}|P \;\mid\; a\{(x)P\} \Leftarrow \mathbb{C} \;\mid\; a \rhd \mathbb{C} \;\mid\; (\nu a)\mathbb{C}$$

An active context is a process with a hole $[\![\cdot]\!]$ in an active position.
We denote by $\mathbb{C}[\![P]\!]$ the process obtained by filling the hole in $\mathbb{C}$ with P

Reductions

$$\begin{array}{l}\mathbb{C}[\![\, s \Rightarrow (x)P \,]\!] \mid \\ \mathbb{D}[\![\, s\{(y)P'\} \Leftarrow (Q|u.R) \,]\!]\end{array} \;\rightarrow\; (\nu r)\left(\begin{array}{l}\mathbb{C}[\![\, r \rhd P\{^u/_x\} \mid s \Rightarrow (x)P \,]\!] \mid \\ \mathbb{D}[\![\, \overline{r} \rhd P'\{^u/_y\} \mid s\{(y)P'\} \Leftarrow (Q|R) \,]\!]\end{array}\right)$$

if r is fresh and u, s not bound by $\mathbb{C}$,$\mathbb{D}$

$$\mathbb{C}[\![\, r \rhd (P|u.Q) \,]\!] \mid \mathbb{D}[\![\, \overline{r} \rhd (R|(z)S) \,]\!] \;\rightarrow\; \mathbb{C}[\![\, r \rhd (P|Q) \,]\!] \mid \mathbb{D}[\![\, \overline{r} \rhd (R|S\{^u/_z\}) \,]\!]$$

if u, r not bound by $\mathbb{C}$,$\mathbb{D}$

$$r \rhd (P|\mathsf{return}\, u.Q) \;\rightarrow\; u \,|\, r \rhd (P|Q)$$

$$\mathbb{C}[\![P]\!] \;\rightarrow\; \mathbb{C}[\![P']\!] \quad \text{if } P \equiv Q,\; Q \rightarrow Q',\; Q' \equiv P'$$

Room Booking: A more elaborated protocol

Room Booking (service side)

$$bookRoom \Rightarrow (d)\left(\begin{array}{l} avail \Leftarrow d \mid \\ (cs)(\nu\, code)code.(cc)epay\{(-)cc.(i)\mathtt{return}\ i\} \Leftarrow price \Leftarrow cs \end{array}\right)$$

Room Booking (client side)

$$bookRoom\{(-)(r)(select \Leftarrow r \mid (c)\mathtt{myCC}.(cid)\mathtt{return}\ \langle c, cid\rangle)\} \Leftarrow \mathtt{dates}$$

Comments

bookRoom is invoked with the dates *d* for the reservation.
It gets (via local service *avail*) and passes to the client the set of available rooms *r*.
The client interacts with the user (service *select*) and sends the selection *cs*.
A fresh reservation code is sent to the client.
The client sends her credit card number.
The service debits the cost to the credit card via service *epay*. (Note that *price* computes the cost of the chosen room.)
Finally, if everything is ok, the client receives the confirmation id *i* generated by *epay*.

Room Booking: A more elaborated protocol

Room Booking (service side)

$$bookRoom \Rightarrow (d)\left(\begin{array}{l} avail \Leftarrow d \mid \\ (cs)(\nu\, code)code.(cc)epay\{(-)cc.(i)\mathtt{return}\ i\} \Leftarrow price \Leftarrow cs \end{array}\right)$$

Room Booking (client side)

$$bookRoom\{(-)(r)(select \Leftarrow r \mid (c)\mathtt{myCC}.(cid)\mathtt{return}\ \langle c, cid\rangle)\} \Leftarrow \mathtt{dates}$$

Comments

bookRoom is invoked with the dates *d* for the reservation.
It gets (via local service *avail*) and passes to the client the set of available rooms *r*.
The client interacts with the user (service *select*) and sends the selection *cs*.
A fresh reservation code is sent to the client.
The client sends her credit card number.
The service debits the cost to the credit card via service *epay*. (Note that *price* computes the cost of the chosen room.)
Finally, if everything is ok, the client receives the confirmation id *i* generated by *epay*.

Room Booking: A more elaborated protocol

Room Booking (service side)

$$bookRoom \Rightarrow (d)\left(\begin{array}{l} avail \Leftarrow d \mid \\ (cs)(\nu\, code)code.(cc)epay\{(-)cc.(i)\mathtt{return}\ i\} \Leftarrow price \Leftarrow cs \end{array}\right)$$

Room Booking (client side)

$$bookRoom\{(-)(r)(select \Leftarrow r \mid (c)\mathtt{myCC}.(cid)\mathtt{return}\ \langle c, cid\rangle)\} \Leftarrow \mathtt{dates}$$

Comments

bookRoom is invoked with the dates *d* for the reservation.
It gets (via local service *avail*) and passes to the client the set of available rooms *r*.
The client interacts with the user (service *select*) and sends the selection *cs*.
A fresh reservation code is sent to the client.
The client sends her credit card number.
The service debits the cost to the credit card via service *epay*. (Note that *price* computes the cost of the chosen room.)
Finally, if everything is ok, the client receives the confirmation id *i* generated by *epay*.

Room Booking: A more elaborated protocol

Room Booking (service side)

$$bookRoom \Rightarrow (d)\left(\begin{array}{l} avail \Leftarrow d \mid \\ (cs)(\nu\, code)code.(cc)epay\{(-)cc.(i)\mathtt{return}\ i\} \Leftarrow price \Leftarrow cs \end{array}\right)$$

Room Booking (client side)

$$bookRoom\{(-)(r)(select \Leftarrow r \mid (c)\mathtt{myCC}.(cid)\mathtt{return}\ \langle c, cid\rangle)\} \Leftarrow \mathtt{dates}$$

Comments

bookRoom is invoked with the dates *d* for the reservation.
It gets (via local service *avail*) and passes to the client the set of available rooms *r*.
The client interacts with the user (service *select*) and sends the selection *cs*.
A fresh reservation code is sent to the client.
The client sends her credit card number.
The service debits the cost to the credit card via service *epay*. (Note that *price* computes the cost of the chosen room.)
Finally, if everything is ok, the client receives the confirmation id *i* generated by *epay*.

Room Booking: A more elaborated protocol

Room Booking (service side)

$$bookRoom \Rightarrow (d)\left(\begin{array}{l} avail \Leftarrow d \mid \\ (cs)(\nu\, code)code.(cc)epay\{(-)cc.(i)\text{return } i\} \Leftarrow price \Leftarrow cs \end{array}\right)$$

Room Booking (client side)

$$bookRoom\{(-)(r)(select \Leftarrow r \mid (c)\texttt{myCC}.(cid)\text{return } \langle c, cid\rangle)\} \Leftarrow \texttt{dates}$$

Comments

bookRoom is invoked with the dates *d* for the reservation.
It gets (via local service *avail*) and passes to the client the set of available rooms *r*.
The client interacts with the user (service *select*) and sends the selection *cs*.
A fresh reservation code is sent to the client.
The client sends her credit card number.
The service debits the cost to the credit card via service *epay*. (Note that *price* computes the cost of the chosen room.)
Finally, if everything is ok, the client receives the confirmation id *i* generated by *epay*.

Room Booking: A more elaborated protocol

Room Booking (service side)

$$bookRoom \Rightarrow (d)\left(\begin{array}{l} avail \Leftarrow d \mid \\ (cs)(\nu\, code)code.(cc)epay\{(-)cc.(i)\text{return } i\} \Leftarrow price \Leftarrow cs \end{array}\right)$$

Room Booking (client side)

$$bookRoom\{(-)(r)(select \Leftarrow r \mid (c)\texttt{myCC}.(cid)\text{return } \langle c, cid\rangle)\} \Leftarrow \texttt{dates}$$

Comments

bookRoom is invoked with the dates *d* for the reservation.
It gets (via local service *avail*) and passes to the client the set of available rooms *r*.
The client interacts with the user (service *select*) and sends the selection *cs*.
A fresh reservation code is sent to the client.
The client sends her credit card number.
The service debits the cost to the credit card via service *epay*. (Note that *price* computes the cost of the chosen room.)
Finally, if everything is ok, the client receives the confirmation id *i* generated by *epay*.

Room Booking: A more elaborated protocol

Room Booking (service side)

$$bookRoom \Rightarrow (d)\left(\begin{array}{l} avail \Leftarrow d \mid \\ (cs)(\nu\, code)code.(cc)epay\{(-)cc.(i)\text{return } i\} \Leftarrow price \Leftarrow cs \end{array}\right)$$

Room Booking (client side)

$$bookRoom\{(-)(r)(select \Leftarrow r \mid (c)\texttt{myCC}.(cid)\text{return } \langle c, cid\rangle)\} \Leftarrow \texttt{dates}$$

Comments

bookRoom is invoked with the dates *d* for the reservation.
It gets (via local service *avail*) and passes to the client the set of available rooms *r*.
The client interacts with the user (service *select*) and sends the selection *cs*.
A fresh reservation code is sent to the client.
The client sends her credit card number.
The service debits the cost to the credit card via service *epay*. (Note that *price* computes the cost of the chosen room.)
Finally, if everything is ok, the client receives the confirmation id *i* generated by *epay*.

Automotive Scenario: Car Assistance Service

$$\mathsf{BMW} \Rightarrow (pol, cloc, mob)(\nu r)$$
$$r \triangleright \mathsf{select}\{(pol, cloc)(cr, tt, g)\mathsf{return}\,\langle cr, tt, g\rangle\} \Leftarrow \langle pol, cloc\rangle \mid$$
$$\overline{r} \triangleright cr\{(l)(crid)mob \Leftarrow crid\} \Leftarrow tt\{(gloc, gid)S\} \Leftarrow g \Leftarrow cloc$$

$$S \triangleq cloc\ \mathbf{if}\ gid$$
$$\mathbf{then}\,(tid)\,\mathbf{if}\ tid$$
$$\mathbf{then}\,(\mathsf{return}\ gloc \mid mob \Leftarrow tid)$$
$$\mathbf{else}\,(\mathsf{return}\ cloc \mid mob \Leftarrow \mathsf{notg} \mid gid \Leftarrow \mathsf{nott})$$
$$\mathbf{else}\,(\mathsf{return}\ cloc \mid mob \Leftarrow \mathsf{notg})$$

From PSC to SCC - I

WS-FM 2006 Proceedings

- Encoding of lazy λ-calculus in PSC
- Encoding of PSC in π-calculus
  - the vice versa is not easy because of sessioning

Garbage collecting concluded sessions

The client-side and service-side protocols are garbage collected by the structural congruence only when they reduce to $\mathbf{0}$.
But many protocols cannot reduce to $\mathbf{0}$, e.g., those deploying service definitions!

Canceling

Also, one may want to explicit program session termination, for instance in order to implement *cancellation workflow patterns* or Orc's *asymmetric parallel* or to manage *abnormal events*.

From PSC to SCC - I

WS-FM 2006 Proceedings

- Encoding of lazy λ-calculus in PSC
- Encoding of PSC in π-calculus
  - the vice versa is not easy because of sessioning

Garbage collecting concluded sessions

The client-side and service-side protocols are garbage collected by the structural congruence only when they reduce to $\mathbf{0}$.
But many protocols cannot reduce to $\mathbf{0}$, e.g., those deploying service definitions!

Canceling

Also, one may want to explicit program session termination, for instance in order to implement *cancellation workflow patterns* or Orc's *asymmetric parallel* or to manage *abnormal events*.

From PSC to SCC - I

WS-FM 2006 Proceedings

- Encoding of lazy λ-calculus in PSC
- Encoding of PSC in π-calculus
  - the vice versa is not easy because of sessioning

Garbage collecting concluded sessions

The client-side and service-side protocols are garbage collected by the structural congruence only when they reduce to $\mathbf{0}$.
But many protocols cannot reduce to $\mathbf{0}$, e.g., those deploying service definitions!

Canceling

Also, one may want to explicit program session termination, for instance in order to implement *cancellation workflow patterns* or Orc's *asymmetric parallel* or to manage *abnormal events*.

Outline

From PSC to SCC - II

Extending sessions

The idea is that the termination of the session on one side, should be communicated to (termination handler service at) the opposite side.

- A service name k, identifying the so-called *termination handler* service, can be associated to a session: $r \rhd_k P$
- The first time the protocol P running inside the session invokes such a service k, the session is closed

Extending services: A slight asymmetry

- The syntax of clients becomes: $a\{(x)P\} \Leftarrow_k Q$
  (we added the name k of the termination handler service to be associated to the session instantiated on the service-side)
- Services are now specified with the process $a \Rightarrow (x)P : (y)T$
  (an additional protocol $(y)T$ is specified which represents the body of a fresh termination handler service that will be associated to the corresponding session on the client-side).

From PSC to SCC - II

Extending sessions

The idea is that the termination of the session on one side, should be communicated to (termination handler service at) the opposite side.

- A service name k, identifying the so-called *termination handler* service, can be associated to a session: $r \rhd_k P$
- The first time the protocol P running inside the session invokes such a service k, the session is closed

Extending services: A slight asymmetry

- The syntax of clients becomes: $a\{(x)P\} \Leftarrow_k Q$
  (we added the name k of the termination handler service to be associated to the session instantiated on the service-side)
- Services are now specified with the process $a \Rightarrow (x)P : (y)T$
  (an additional protocol $(y)T$ is specified which represents the body of a fresh termination handler service that will be associated to the corresponding session on the client-side).

SCC Syntax

Grammar

$P, Q, T, \ldots$	$::= \quad \mathbf{0}$	Nil
	$\mid a.P$	Concretion
	$\mid (x)P$	Abstraction
	$\mid \text{return } a.P$	Return Value
	$\mid a \Rightarrow (x)P : (y)T$	Service Definition
	$\mid a\{(x)P\} \Leftarrow_k Q$	Service Invocation
	$\mid a \rhd_k P$	Session
	$\mid P \mid Q$	Parallel Composition
	$\mid (\nu a)P$	New Name

A special name close is reserved for the specification of session protocols.

Shorthand notation

We write $a \Rightarrow (x)P$ for $a \Rightarrow (x)P : (y)\mathbf{0}$.
We also omit k in $a\{(x)P\} \Leftarrow_k Q$ and $a \Leftarrow_k Q$ when it is not relevant.

SCC Operational Semantics

$$\mathbb{C}[\![\, s \Rightarrow (x)P : (z)T \,]\!] \mid \mathbb{D}[\![\, s\{(y)P'\} \Leftarrow_k (Q|u.R) \,]\!] \;\rightarrow\; (\nu r, k')\left(\mathbb{C}[\![\, s \Rightarrow (x)P : (z)T \mid r \rhd_k (k' \Rightarrow (z)T\{^k/_{\mathsf{close}}\} \mid P\{^u/_x\}\{^k/_{\mathsf{close}}\}) \,]\!] \mid \mathbb{D}[\![\, \overline{r} \rhd_{k'} P'\{^u/_y\}\{^{k'}/_{\mathsf{close}}\} \mid s\{(y)P'\} \Leftarrow_k (Q|R) \,]\!]\right)$$

if $s \notin tn(\mathbb{D})$, r, k' are fresh and u, s, k not bound by $\mathbb{C}$,$\mathbb{D}$

$$r \rhd_s \mathbb{D}[\![\, s\{(y)P\} \Leftarrow_k (Q|u.R) \,]\!] \;\rightarrow\; s\{\} \Leftarrow_k u$$

if $s \notin tn(\mathbb{D})$ and u, s, k not bound by $\mathbb{D}$

$$\mathbb{C}[\![\, r \rhd_k (P|u.Q) \,]\!] \mid \mathbb{D}[\![\, \overline{r} \rhd_{k'} (R|(z)S) \,]\!] \;\rightarrow\; \mathbb{C}[\![\, r \rhd_k (P|Q) \,]\!] \mid \mathbb{D}[\![\, \overline{r} \rhd_{k'} (S\{^u/_z\} \mid R) \,]\!]$$

if u, r not bound by $\mathbb{C}$,$\mathbb{D}$

$$r \rhd_k (P|\mathsf{return}\, u.Q) \;\rightarrow\; u \mid r \rhd_k (P|Q)$$

$$\mathbb{C}[\![\, P \,]\!] \;\rightarrow\; \mathbb{C}[\![\, P' \,]\!] \text{ if } P \equiv Q,\; Q \rightarrow Q',\; Q' \equiv P'$$

SCC Examples: Closure Protocol

A typical usage of termination handler services is the closure of the current session .

A typical service-side closure protocol

$$s \Rightarrow (x)P' : (y)\mathsf{close}\,\{\} \Leftarrow y$$

A typical client-side closure protocol

$$End \triangleq \mathsf{close}\,\{\} \Leftarrow (end \Rightarrow (x)\mathsf{return}\,x))$$

End is designed to be included in the client-side protocol:

$$(\nu\, end)s\{\,(y)\big(P \mid End\big)\,\} \Leftarrow_{end} v$$

Closing the client-side session will in turn activate the service-side termination handler.

SCC Examples: Closure Protocol

A typical usage of termination handler services is the closure of the current session .

A typical service-side closure protocol

$$s \Rightarrow (x)P' : (y)\mathsf{close}\,\{\} \Leftarrow y$$

A typical client-side closure protocol

$$End \triangleq \mathsf{close}\,\{\} \Leftarrow (end \Rightarrow (x)\mathsf{return}\,x))$$

End is designed to be included in the client-side protocol:

$$(\nu\, end)s\{\,(y)\big(P \mid End\big)\,\} \Leftarrow_{end} v$$

Closing the client-side session will in turn activate the service-side termination handler.

SCC Examples: Closure Protocol

A typical usage of termination handler services is the closure of the current session .

A typical service-side closure protocol

$$s \Rightarrow (x)P' : (y)\mathsf{close}\,\{\} \Leftarrow y$$

A typical client-side closure protocol

$$\mathit{End} \triangleq \mathsf{close}\,\{\} \Leftarrow (\mathit{end} \Rightarrow (x)\mathsf{return}\,x))$$

End is designed to be included in the client-side protocol:

$$(\nu\, \mathit{end})s\{\,(y)\big(P \mid \mathit{End}\big)\,\} \Leftarrow_{\mathit{end}} v$$

Closing the client-side session will in turn activate the service-side termination handler.

Outline

Novelties

What's new?

The main novelty regards session handling mechanisms for the definition of

- *session naming and scoping*
- *structured interaction protocols*
- *service interruption, cancelation and update* (dynamic environment)

In particular

- The protocols to be run within the service-side / client-side session are well-exposed in the syntax of the calculus to favour type checking, service conformance check, service discovery
- While Orc's cancelation is too demanding (it can destroy a wide area computation), SCC has just a local termination that activates a proper handler at the partner site.

And why not just π?

- Higher-level primitives can favour and make more scalable the development of typing systems and proof techniques.

Prototypes

PSC

- Leonardo Mezzina (IMT Lucca), written in C#: translates PSC (extended with data types, conditionals and a library of local sites) to Orc and pi for simulation and static analysis.
- Franco Mazzanti (CNR), written in DHTML, shell, Ada: simulation environment for PSC (extended with expressions).

SCC

Any volunteers?

On-going work

Extensions

- Types
- Channels
- Distribution
- Long-running transactions and compensations
- Delegation
- XML querying
- SLA and QoS

Re-design options

- Multi-party sessioning
- Recursion primitives
- Synchronized termination
- More flexible, usable and realistic streaming of data

Data driven coordination

Observation

If P produces a stream of values then the composition $s \Leftarrow P$ invokes s infinitely many times.

Anomaly

- It is easy to program a service p that can produce unbound streams of values,
- but programming a client-side protocol for collecting all the values returned by p is not possible unless p is slightly re-engineered to address all values to some suitable publishing service.

Other limitations

A general mechanism for an elegant writing of different coordination patterns is frequently needed in practice (sessions and returns are not abstract enough to help structured programming, typing and analysis)

Data driven coordination

Observation

If P produces a stream of values then the composition $s \Leftarrow P$ invokes s infinitely many times.

Anomaly

- It is easy to program a service p that can produce unbound streams of values,
- but programming a client-side protocol for collecting all the values returned by p is not possible unless p is slightly re-engineered to address all values to some suitable publishing service.

Other limitations

A general mechanism for an elegant writing of different coordination patterns is frequently needed in practice (sessions and returns are not abstract enough to help structured programming, typing and analysis)

Data driven coordination

Observation

If P produces a stream of values then the composition $s \Leftarrow P$ invokes s infinitely many times.

Anomaly

- It is easy to program a service p that can produce unbound streams of values,
- but programming a client-side protocol for collecting all the values returned by p is not possible unless p is slightly re-engineered to address all values to some suitable publishing service.

Other limitations

A general mechanism for an elegant writing of different coordination patterns is frequently needed in practice (sessions and returns are not abstract enough to help structured programming, typing and analysis)

PSC Examples: Stream Connection

Conference announcements

For instance, if *eatcs* and *eapls* return streams of conference announcements on the received service name, then

$$emailme\{\} \Leftarrow \left(\begin{array}{ll} & pub \Rightarrow (s)\text{return } s \\ | & eatcs\{\} \Leftarrow pub \\ | & eapls\{\} \Leftarrow pub \end{array} \right)$$

will send you all the announcements collected from *eatcs* and *eapls*.
More concisely, this can be equivalently written as

$$eatcs\{\} \Leftarrow emailme \mid eapls\{\} \Leftarrow emailme.$$

PSC Examples: Stream Connection

Conference announcements

For instance, if *eatcs* and *eapls* return streams of conference announcements on the received service name, then

$$emailme\{\} \Leftarrow \left(\begin{array}{ll} & pub \Rightarrow (s)\text{return } s \\ | & eatcs\{\} \Leftarrow pub \\ | & eapls\{\} \Leftarrow pub \end{array} \right)$$

will send you all the announcements collected from *eatcs* and *eapls*. More concisely, this can be equivalently written as

$$eatcs\{\} \Leftarrow emailme \mid eapls\{\} \Leftarrow emailme.$$

SSCC: Streaming SCC

Orchestration constructs (similar to Orc's buffer object)

- stream P as f in Q: both P and Q execute concurrently
- f is the name of a stream of data from P to Q
- P can write data into the stream via feed $v.P'$ (non blocking, the value is stored in the stream)
- Q can read from the stream via $f(x).Q'$ (blocking, it is possible to read from multiple streams)
- two possible semantics for streams: ordered or unordered

Simplified syntax for services

$$s \Rightarrow P \qquad\qquad s \Leftarrow Q$$

No "streaming" invocation: P and Q are the protocols.

SSCC: Examples

Chaining: Q waits the first n results fed from P before starting

$$P >^n x_1 \ldots x_n > Q \quad \triangleq \quad \text{stream } P \text{ as } f \text{ in } f(x_1).\ldots.f(x_n).Q$$

Invoking a with n parameters and feeding the result.

$$\text{call } a(x_1, \ldots, x_n) \quad \triangleq \quad a \Leftarrow x_1.\ldots.x_n.(y)\text{feed } y$$

Sequencing (in standard SCC is more complicated)

$$\text{call } a(v) >^1 x > \text{call } b(x)$$

Orc pipeline (one instance of Q for each value fed by P)

$$P > x > Q \quad \triangleq \quad \text{stream } P \text{ as } f \text{ in } \text{rec}X.f(x).(Q|X)$$

Merging streams

$$(eatcs \Leftarrow \text{rec}X.(x).\text{feed } x.X | eapls \Leftarrow \text{rec}X.(x).\text{feed } x.X) > y > emailme \Leftarrow y$$

SSCC: Examples

Chaining: Q waits the first n results fed from P before starting

$$P >^n x_1 \ldots x_n > Q \quad \triangleq \quad \text{stream } P \text{ as } f \text{ in } f(x_1).\ldots.f(x_n).Q$$

Invoking a with n parameters and feeding the result.

$$\text{call } a(x_1, \ldots, x_n) \quad \triangleq \quad a \Leftarrow x_1.\ldots.x_n.(y)\text{feed } y$$

Sequencing (in standard SCC is more complicated)

$$\text{call } a(v) >^1 x > \text{call } b(x)$$

Orc pipeline (one instance of Q for each value fed by P)

$$P > x > Q \quad \triangleq \quad \text{stream } P \text{ as } f \text{ in } \text{rec}X.f(x).(Q|X)$$

Merging streams

$$(eatcs \Leftarrow \text{rec}X.(x).\text{feed } x.X | eapls \Leftarrow \text{rec}X.(x).\text{feed } x.X) > y > emailme \Leftarrow y$$

SSCC: Examples

Chaining: Q waits the first n results fed from P before starting

$$P >^n x_1 \ldots x_n > Q \quad \triangleq \quad \text{stream } P \text{ as } f \text{ in } f(x_1).\ldots.f(x_n).Q$$

Invoking a with n parameters and feeding the result.

$$\text{call } a(x_1, \ldots, x_n) \quad \triangleq \quad a \Leftarrow x_1.\ldots.x_n.(y)\text{feed } y$$

Sequencing (in standard SCC is more complicated)

$$\text{call } a(v) >^1 x > \text{call } b(x)$$

Orc pipeline (one instance of Q for each value fed by P)

$$P > x > Q \quad \triangleq \quad \text{stream } P \text{ as } f \text{ in } \text{rec}X.f(x).(Q|X)$$

Merging streams

$$(eatcs \Leftarrow \text{rec}X.(x).\text{feed } x.X | eapls \Leftarrow \text{rec}X.(x).\text{feed } x.X) > y > emailme \Leftarrow y$$

SSCC: Recap

Done

- Simplified service orchestration using more expressive and easy-to-grasp primitives (recursion, streaming, uniform notation)
- Reduction and LTS semantics with correspondence theorem
- Simple type system (for compatibility of deterministic conversations)
- Implementation of all the WP not requiring the close .

Plans

- Improve typing systems (more refined ones)
- Introducing a close primitive

Related work

Other type systems for SCC and its variants are under development (see minutes from yesterday SCC meeting)

SSCC: Recap

Done

- Simplified service orchestration using more expressive and easy-to-grasp primitives (recursion, streaming, uniform notation)
- Reduction and LTS semantics with correspondence theorem
- Simple type system (for compatibility of deterministic conversations)
- Implementation of all the WP not requiring the close .

Plans

- Improve typing systems (more refined ones)
- Introducing a close primitive

Related work

Other type systems for SCC and its variants are under development (see minutes from yesterday SCC meeting)

SSCC: Recap

Done

- Simplified service orchestration using more expressive and easy-to-grasp primitives (recursion, streaming, uniform notation)
- Reduction and LTS semantics with correspondence theorem
- Simple type system (for compatibility of deterministic conversations)
- Implementation of all the WP not requiring the close .

Plans

- Improve typing systems (more refined ones)
- Introducing a close primitive

Related work

Other type systems for SCC and its variants are under development (see minutes from yesterday SCC meeting)

Outline

SCC: From Theory to Practice I

Soccer world champion (June 2006)

$$SWC \Rightarrow (-)\texttt{brasil}$$

When a team becomes the new world champion then the service must be updated!

In PSC there is no way to cancel a definition and replace it with a new one.

By contrast, in SCC we can define the termination handler

$$new \Rightarrow (z)\big(\ SWC \Rightarrow (-)z \mid new\{\} \Leftarrow_{new} (update \Rightarrow (y)\text{return}\, y)\ \big)$$

to be run in parallel with

$$r \rhd_{new} \big(\ SWC \Rightarrow (-)\texttt{brasil} \mid new\{\} \Leftarrow_{new} (update \Rightarrow (y)\text{return}\, y)\ \big)$$

SCC: From Theory to Practice - II

And the winner is...

$$\left(\begin{array}{ll} & update\{\} \Leftarrow \texttt{italy} \\ | & new \Rightarrow (z)(\ SWC \Rightarrow (-)z \mid new\{\} \Leftarrow_{new} (update \Rightarrow (y)\text{return } y)\ ) \\ | & r \rhd_{new} (\ SWC \Rightarrow (-)\texttt{brasil} \mid new\{\} \Leftarrow_{new} (update \Rightarrow (y)\text{return } y)\ ) \end{array}\right)$$

$$\rightarrow \quad (\nu a)\left(\begin{array}{ll} & update\{\} \Leftarrow \mathbf{0} \mid \bar{a} \rhd \mathbf{0} \mid new \Rightarrow (z)(\ ...) \\ | & r \rhd_{new} (\ ... \mid new\{\} \Leftarrow_{new} (\ ... \mid a \rhd \text{return } \texttt{italy}\ )\ ) \end{array}\right)$$

$$\rightarrow \quad (\nu a)\left(\begin{array}{ll} & update\{\} \Leftarrow \mathbf{0} \mid new \Rightarrow (z)(\ ...) \\ | & r \rhd_{new} (\ ... \mid new\{\} \Leftarrow_{new} (\ ... \mid \texttt{italy} \mid a \rhd \mathbf{0}\ )\ ) \end{array}\right)$$

$$\rightarrow \quad (\ update\{\} \Leftarrow \mathbf{0} \mid new \Rightarrow (z)(\ ...)\ \mid new\{\} \Leftarrow_{new} \texttt{italy}\ ) \quad \rightarrow$$

$$(\nu b)\left(\begin{array}{ll} & update\{\} \Leftarrow \mathbf{0} \mid new \Rightarrow (z)(\ ...) \\ | & b \rhd_{new} (\ SWC \Rightarrow (-)\texttt{italy} \mid new\{\} \Leftarrow_{new} (update \Rightarrow (y)\text{return } y)\ ) \\ | & new\{\} \Leftarrow_{new} \mathbf{0} \mid \bar{b} \rhd \mathbf{0} \end{array}\right)$$

SCC: From Theory to Practice - II

And the winner is...

$$\left(\begin{array}{ll} & update\{\} \Leftarrow \texttt{italy} \\ | & new \Rightarrow (z)(\ SWC \Rightarrow (-)z \mid new\{\} \Leftarrow_{new} (update \Rightarrow (y)\text{return } y)\ ) \\ | & r \rhd_{new} (\ SWC \Rightarrow (-)\texttt{brasil} \mid new\{\} \Leftarrow_{new} (update \Rightarrow (y)\text{return } y)\ ) \end{array}\right)$$

$$\rightarrow \quad (\nu a)\left(\begin{array}{ll} & update\{\} \Leftarrow \mathbf{0} \mid \bar{a} \rhd \mathbf{0} \mid new \Rightarrow (z)(\ \ldots) \\ | & r \rhd_{new} (\ \ldots \mid new\{\} \Leftarrow_{new} (\ \ldots \mid a \rhd \text{return } \texttt{italy}\ )\ ) \end{array}\right)$$

$$\rightarrow \quad (\nu a)\left(\begin{array}{ll} & update\{\} \Leftarrow \mathbf{0} \mid new \Rightarrow (z)(\ \ldots) \\ | & r \rhd_{new} (\ \ldots \mid new\{\} \Leftarrow_{new} (\ \ldots \mid \texttt{italy} \mid a \rhd \mathbf{0}\ )\ ) \end{array}\right)$$

$$\rightarrow \quad (\ update\{\} \Leftarrow \mathbf{0} \mid new \Rightarrow (z)(\ \ldots) \mid new\{\} \Leftarrow_{new} \texttt{italy}\ ) \quad \rightarrow$$

$$(\nu b)\left(\begin{array}{ll} & update\{\} \Leftarrow \mathbf{0} \mid new \Rightarrow (z)(\ \ldots) \\ | & b \rhd_{new} (\ SWC \Rightarrow (-)\texttt{italy} \mid new\{\} \Leftarrow_{new} (update \Rightarrow (y)\text{return } y)\ ) \\ | & new\{\} \Leftarrow_{new} \mathbf{0} \mid \bar{b} \rhd \mathbf{0} \end{array}\right)$$

SCC WORKS!

And SSCC might even work better!

SCC WORKS!

And SSCC might even work better!

PSC Examples: Encoding of the lazy λ-calculus

The translation is in the spirit of Milner's π-calculus encoding:

$$
\begin{array}{rcl}
[\![x]\!]_p & = & x\{\} \Leftarrow p \\
[\![\lambda x.M]\!]_p & = & p \Rightarrow (x)(q)[\![M]\!]_q \\
[\![M\,N]\!]_p & = & (\nu m)(\nu n)\left(\begin{array}{ll} & [\![M]\!]_m \\ | & n \Rightarrow (s)[\![N]\!]_s \\ | & m\{(-)p\} \Leftarrow n \end{array}\right)
\end{array}
$$

The more important differences

1. each service invocation opens a new session where the computation can progress (remind that sessions cannot be closed in PSC)
2. all service definitions will remain available even when no further invocation will be possible.

If on one hand, the encoding witnesses the expressive power of PSC, on the other hand, it also motivates the introduction of some mechanism for closing sessions.

Encoding of PSC into π-calculus

The encoding below shows that PSC can be seen as a fragment of the π-calculus.

$$
\begin{aligned}
[\![a\{(x)P\} \Leftarrow Q]\!]_{in,out,ret} &= (\nu z)\big([\![Q]\!]_{in,z,ret} \mid !z(x).(\nu r,\tilde{r})\overline{a}\langle r,\tilde{r},x\rangle.[\![P]\!]_{r,\tilde{r},out} \big) \\
[\![a \Rightarrow (x)P]\!]_{in,out,ret} &= !a(r,\tilde{r},x).([\![P]\!]_{\tilde{r},r,out}) \\
[\![a \rhd P]\!]_{in,out,ret} &= [\![P]\!]_{a,\tilde{a},out} \\
[\![a.P]\!]_{in,out,ret} &= \overline{out}\ a.[\![P]\!]_{in,out,ret} \\
[\![(x)P]\!]_{in,out,ret} &= in(x).[\![P]\!]_{in,out,ret} \\
[\![\text{return } a.P]\!]_{in,out,ret} &= \overline{ret}\ a \mid [\![P]\!]_{in,out,ret} \\
[\![P \mid Q]\!]_{in,out,ret} &= [\![P]\!]_{in,out,ret} \mid [\![Q]\!]_{in,out,ret} \\
[\![(\nu x)P]\!]_{in,out,ret} &= (\nu x)[\![P]\!]_{in,out,ret} \\
[\![\mathbf{0}]\!]_{in,out,ret} &= \mathbf{0}
\end{aligned}
$$

The encoding can hardly be extended to full SCC calculus due to the session interruption mechanism that has no direct couterpart in the π-calculus.

SCC Examples: A blog service - I

Blog

We consider a service that implements a *blog*, i.e. a web page used by a web client to log personal annotations.

Interaction with the Blog

A blog provides two services:

- *get* to read the current contents of the blog
- *set* to modify the contents.

The close-free fragment is not expressive enough to faithfully model such a service because it does not support service update, here needed to update the blog contents.

SCC Examples: A blog service - II

Blog Factory

$$\begin{array}{l} \mathit{newBlog} \Rightarrow (v, \mathit{get}, \mathit{set})\big(\mathit{blog}\{\} \Leftarrow_{\mathit{newBlog}} \langle v, \mathit{get}, \mathit{set}\rangle\big) \mid \\ \mathit{blog} \Rightarrow (v, \mathit{get}, \mathit{set})\big(\quad \mathit{get} \Rightarrow (-)v \mid \\ \qquad\qquad\qquad\qquad\quad \mathrm{close}\,\{\} \Leftarrow (\mathit{set} \Rightarrow (v')\mathrm{return}\,\langle v', \mathit{get}, \mathit{set}\rangle) \quad \big) \end{array}$$

We use the service *newBlog* as a factory of blogs. It receives three names:

- the initial contents v
- the name for the new *get* service
- the name for the new *set* service

Upon invocation, the factory forwards the three received values to the *blog* service which is the responsible for the actual instantiation of the *get* and *set* services.

The update of the blog contents is achieved by invoking the service close which is bound to *newBlog*; this invocation cancels the currently available *get* and *set* services and delegates to *newBlog* the creation of their new instances passing also the updated contents v'.

SCC Examples: A blog service - III

Blog update

The process below installs a wiki page with initial contents v, then it adds some new contents v'.

$$newBlog\{\} \Leftarrow \langle v, get, set\rangle \mid$$
$$set\{\} \Leftarrow \big(concat\{\,(-)v'.get \Leftarrow \bullet \mid (x)\mathrm{return}\,x\,\} \Leftarrow \bullet\big)$$

The service *concat* simply computes the new contents appending v' to the contents v received after service invocation:

$$concat \Rightarrow (-)(y)(z).y \circ z$$

Here $\circ$ denotes justaposition of blog contents.

SCC Examples: Encoding Orc in SCC

SCC as a service orchestration language

To evaluate the expressiveness and usability of SCC as a language for service orchestration, one has to challenge its ability of encoding some frequently used service composition patterns.

Workflow patterns

A library of basic patterns, called the *workflow patterns*, has been identified by van der Aalst et al.

Workflow patterns and Orc

Orc can conveniently model most workflow patterns! [Coordination'06]
Since SCC can encode Orc, then by transitivity SCC can implement van der Aalst's workflow patterns.

SCC Examples: Encoding Orc in SCC

SCC as a service orchestration language

To evaluate the expressiveness and usability of SCC as a language for service orchestration, one has to challenge its ability of encoding some frequently used service composition patterns.

Workflow patterns

A library of basic patterns, called the *workflow patterns*, has been identified by van der Aalst et al.

Workflow patterns and Orc

Orc can conveniently model most workflow patterns! [Coordination'06]
Since SCC can encode Orc, then by transitivity SCC can implement van der Aalst's workflow patterns.

Encoding Orc in SCC - I

While a value is trivially encoded as itself, i.e., $[\![u]\!] = u$, for variables (and thus for actual parameters) we need two different encodings, depending on whether they are passed by name or evaluated.

We distinguish the two encodings by different subscripts:

$$[\![x]\!]_n = x \qquad [\![x]\!]_v = x \Leftarrow \bullet$$

- The evaluation of a variable x is encoded as a request for the current value to the variable manager of x.
- Variable managers are created by both sequential composition and asymmetric parallel composition.

Encoding Orc in SCC - II

$$
\begin{array}{rcl}
[\![E(x) \triangleq P]\!] & = & E \Rightarrow (x)[\![P]\!] \\
[\![a(p)]\!] & = & a \Leftarrow [\![p]\!]_v \\
[\![x(p)]\!] & = & (\nu forw, pub)\big(\ forw\{\} \Leftarrow [\![x]\!]_v \mid \\
 & & \qquad forw \Rightarrow (a)pub\{\} \Leftarrow [\![a(p)]\!] \mid \\
 & & \qquad pub \Rightarrow (y)\mathsf{return}\, y\ \big) \\
[\![E(p)]\!] & = & E \Leftarrow [\![p]\!]_n \\
[\![P|Q]\!] & = & [\![P]\!]|[\![Q]\!] \\
[\![P > x > Q]\!] & = & (\nu z, pub)\big(\ z\{\} \Leftarrow [\![P]\!] \mid \\
 & & \qquad z \Rightarrow (y)(\nu x)(x \Rightarrow (-)y \mid pub\{\} \Leftarrow [\![Q]\!]) \mid \\
 & & \qquad pub \Rightarrow (y)\mathsf{return}\, y\ \big) \\
[\![Q\ \mathbf{where}\, x :\in P]\!] & = & (\nu x, z, w)\big(\ [\![Q]\!] \mid \\
 & & \qquad z \Rightarrow (y)(x \Rightarrow (-)y) \mid \\
 & & \qquad w\{\} \Leftarrow_z \bullet \mid \\
 & & \qquad w \Rightarrow (-)(\mathsf{close}\,\{\} \Leftarrow [\![P]\!])\ \big)
\end{array}
$$

From Orc to SCC: An Example

Emailing news in Orc

Let us consider the Orc expression

$$CNN(d)|BBC(d) > x > email(x)$$

which invokes the news services of both *CNN* and *BBC* asking for news of day d. For each reply it sends an email (to a default address) with the received news. Thus this expression can send from zero up to two emails. The SCC encoding is as follows:

$$\begin{aligned}(\nu z, pub)\big(z\{\} \Leftarrow (CNN \Leftarrow d|BBC \Leftarrow d) \mid \\ z \Rightarrow (y)(\nu x)(x \Rightarrow (-)y \mid pub\{\} \Leftarrow email \Leftarrow x \Leftarrow \bullet) \mid \\ pub \Rightarrow (y)\text{return}\, y\ \big)\end{aligned}$$

We have supposed here to have *CNN*, *BBC* and *email* available as services.

PSC Examples: Recursion - I

Shorthand notation

We presuppose a distinct name $\bullet$ to be used as a unit value.

Clock (service side)

Service invocations can be nested recursively inside a service definition:

$$\texttt{clock} \Rightarrow (-) \left(\begin{array}{l} \texttt{return tick} \\ \mid \quad \texttt{clock}\{\} \Leftarrow \bullet \end{array} \right)$$

Invoked with $\texttt{clock}\{\} \Leftarrow \bullet$, produces an infinite number of `ticks`...
but just on the service-side!

PSC Examples: Recursion - I

Shorthand notation

We presuppose a distinct name $\bullet$ to be used as a unit value.

Clock (service side)

Service invocations can be nested recursively inside a service definition:

$$\texttt{clock} \Rightarrow (-) \left(\begin{array}{l} \texttt{return tick} \\ \mid \quad \texttt{clock}\{\} \Leftarrow \bullet \end{array} \right)$$

Invoked with $\texttt{clock}\{\} \Leftarrow \bullet$, produces an infinite number of `ticks`...
but just on the service-side!

PSC Examples: Recursion - II

Clock (client side)

To produce the `ticks` on a specific location different from the service-side, the service to be invoked can be written as

$$remoteClock \Rightarrow (s) \left(\begin{array}{ll} & s\{\} \Leftarrow \texttt{tick} \\ | & remoteClock\{\} \Leftarrow s \end{array} \right)$$

and a local publishing service

$$pub \Rightarrow (t)\text{return } t$$

must be located where the `ticks` must be produced.
Then invoke the service as below:

$$remoteClock\{\} \Leftarrow pub$$

PSC Examples: Stream Connection - II

Trivial recursion does not work!

One might think to exploit recursion to deploy local receivers of the form $(x)\text{return}\, x$, but the implicit nesting of sessions would cause all such receivers to collect values only from different sessions than the original one.

No Replicator!

Extending the syntax with π-calculus like replicator $!P$:

$$pipe = (-)!(x)\text{return}\, x$$

No Code Passing!

Extending the syntax with $\text{return}\, P.Q$, whose semantics is:

$$r \rhd (R|\text{return}\, P.Q) \;\rightarrow\; P \,|\, r \rhd\; (R|Q)$$

Replication can then be coded as follows:

$$!P = (\nu\, rec)\big(\; rec \Rightarrow (-)(\,\text{return}\, P \,|\, rec\{\} \Leftarrow \bullet\,) \,|\, rec\{\} \Leftarrow \bullet\;\big)$$

PSC Examples: Stream Connection - II

Trivial recursion does not work!

One might think to exploit recursion to deploy local receivers of the form $(x)\text{return}\, x$, but the implicit nesting of sessions would cause all such receivers to collect values only from different sessions than the original one.

No Replicator!

Extending the syntax with π-calculus like replicator $!P$:

$$pipe = (-)!(x)\text{return}\, x$$

No Code Passing!

Extending the syntax with $\text{return}\, P.Q$, whose semantics is:

$$r \rhd (R|\text{return}\, P.Q) \;\rightarrow\; P \,|\, r \rhd\; (R|Q)$$

Replication can then be coded as follows:

$$!P = (\nu\, rec)\big(\; rec \Rightarrow (-)(\,\text{return}\, P \,|\, rec\{\} \Leftarrow \bullet\,) \,|\, rec\{\} \Leftarrow \bullet\;\big)$$

PSC Examples: Stream Connection - II

Trivial recursion does not work!

One might think to exploit recursion to deploy local receivers of the form $(x)\text{return}\, x$, but the implicit nesting of sessions would cause all such receivers to collect values only from different sessions than the original one.

No Replicator!

Extending the syntax with π-calculus like replicator $!P$:

$$pipe = (-)!(x)\text{return}\, x$$

No Code Passing!

Extending the syntax with return $P.Q$, whose semantics is:

$$r \rhd (R|\text{return}\, P.Q) \;\rightarrow\; P \,|\, r \rhd \;(R|Q)$$

Replication can then be coded as follows:

$$!P = (\nu\, rec)(\; rec \Rightarrow (-)(\,\text{return}\, P \,|\, rec\{\} \Leftarrow \bullet\,) \,|\, rec\{\} \Leftarrow \bullet\;)$$

PSC Examples: Stream Connection - III

Conference announcements

For instance, if *EATCS* and *EAPLS* return streams of conference announcements on the received service name, then

$$emailMe\{\} \Leftarrow \left(\begin{array}{ll} & pub \Rightarrow (s)\text{return } s \\ | & EATCS\{\} \Leftarrow pub \\ | & EAPLS\{\} \Leftarrow pub \end{array} \right)$$

will send you all the announcements collected from *EATCS* and *EAPLS*. More concisely, this can be equivalently written as

$$EATCS\{\} \Leftarrow emailMe \mid EAPLS\{\} \Leftarrow emailMe.$$

PSC Examples: Stream Connection - III

Conference announcements

For instance, if *EATCS* and *EAPLS* return streams of conference announcements on the received service name, then

$$emailMe\{\} \Leftarrow \left(\begin{array}{ll} & pub \Rightarrow (s)\text{return } s \\ | & EATCS\{\} \Leftarrow pub \\ | & EAPLS\{\} \Leftarrow pub \end{array} \right)$$

will send you all the announcements collected from *EATCS* and *EAPLS*. More concisely, this can be equivalently written as

$$EATCS\{\} \Leftarrow emailMe \mid EAPLS\{\} \Leftarrow emailMe.$$

PSC Examples: Stream Connection - III

Conference announcements

For instance, if *EATCS* and *EAPLS* return streams of conference announcements on the received service name, then

$$emailMe\{\} \Leftarrow \left(\begin{array}{ll} & pub \Rightarrow (s)\text{return } s \\ | & EATCS\{\} \Leftarrow pub \\ | & EAPLS\{\} \Leftarrow pub \end{array} \right)$$

will send you all the announcements collected from *EATCS* and *EAPLS*. More concisely, this can be equivalently written as

$$EATCS\{\} \Leftarrow emailMe \mid EAPLS\{\} \Leftarrow emailMe.$$

PSC Examples: News Streaming

Programming Pattern

The service *pub* (or alike) can be useful in many applications.

In fact, in PSC *sessions cannot be closed* and therefore recursive invocations on the client-side are nested at increasing depth (while the return instruction can move values only one level up).

News Streaming (client side)

A recursive process that repeatedly invokes service s on value x with publishing service p is shown below:

$$rec \Rightarrow (s,x,p)s\left\{(-)\left(\begin{array}{l}(y)p\{\} \Leftarrow y \\ \mid \quad rec\{\} \Leftarrow \langle s,x,p\rangle\end{array}\right)\right\} \Leftarrow x$$

Sample of invocation of the service *rec*:

$$rec\{\} \Leftarrow \langle ANSA, \bullet, pub\rangle \mid pub \Rightarrow (x)\text{return}\, x$$

that returns the stream of news obtained from the *ANSA* service.