Core Calculi for Service-Oriented Computing

Rocco De Nicola

DSIUF- Università di Firenze

SENSORIA Workshop
Leicester - June 13, 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

We seek for a small set of primitives that might serve as a basis for formalizing and programming service oriented applications over global computers.

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

We seek for a small set of primitives that might serve as a basis for formalizing and programming service oriented applications over global computers.

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

We seek for a small set of primitives that might serve as a basis for formalizing and programming service oriented applications over global computers.

Main Objectives of WP2

- **Provide a foundational understanding** of the Service-Oriented Computing (SOC) paradigm.
- **Develop calculi** for service specifications and analysis that:
  - comply with a service-oriented approach to business modelling;
  - allow for modular description of services;
  - support dynamic, ad-hoc, "just-in-time" composition.
- **Assess the quality of our proposals**, by means of a number of case studies, with respect to:
  - service contracts;
  - service discovery;
  - service composition.

Core Calculi

- Calculi based on process algebras but enhanced with
  - primitives for manipulating semi-structured data (e.g. pattern matching)
  - mechanisms for describing safe client-service (e.g. sessions)
  - operators for composing (possibly unreliable) services
  - techniques for query and discovery of services.
- The outcome will not necessarily be a single core language; there are competing paradigms concerning
  - chosen level of abstraction
  - client-service interaction/coordination
  - primitives for orchestration

Outline

The calculi of the first 18 months

Pre-existing SENSORIA

In the first period a number of existing calculi has been used: CCS, $\pi-$calculus, Join, Ambient, KLAIM, ...

Within SENSORIA

A first suite of calculi has been proposed, each of them aiming at capturing specific issues of SOC.

- SOCK
- COWS
- SCC
- SC
- λ^{req}

It is now needed to assess their relative merits and expressiveness, and look for unifying solutions.

SOCK: a calculus for Service Oriented Computing

Main Aims

- SOCK is a formal calculus which aims at characterizing the basic features of Service Oriented Computing and takes its inspiration from WS-BPEL, a 'de facto' standard for Web Service technology
- C. Guidi, R. Lucchi, R. Gorrieri, N. Busi, G. Zavattaro 4th Int. Conf. on Service-Oriented Computing, LNCS 4294, Springer, 2006

Basic Structure

A three layered structure inspired by the Web services protocol stack.

- The service behaviour calculus provides the primitives for services programming;
- The service engine calculus provides the mechanisms for describing service engines;
- The service system calculus provides the mechanisms for composing/deployng service engines into a system.

COWS: a Calculus for Orchestration of Web Services

Main Aims

- COWS exploits WS-BPEL to drive the design of a foundational calculus to reason on specified services.
- A. Lapadula, R. Pugliese, F. Tiezzi, 16th European Symposium on Programming, LNCS 4421, Springer, 2007

Basic Structure

- Threads of a service can share the store and sessions can be modeled by means of correlation sets.
- Some WS-BPEL constructs, e.g. fault and compensation handlers and flow graphs, do not have direct counterparts; they can be encoded by exploiting COWS operators.
- Borrows ideas from other calculi: asynch. comm., pattern matching, polyadic synchronization, localized input ($L\pi$), delimited kill & protection ($StAC_i$)

SCC: a Sesrvice-Centered Calculus

Main Aims

- A a small set of primitives for programming and orchestrating services. A concept of *session* for client-server interaction.
- M. Boreale, R. Bruni, L. Caires, R. De Nicola, I. Lanese, M. Loreti, F. Martins, U. Montanari, A. Ravara, D. Sangiorgi, V. Vasconcelos, G. Zavattaro, 3rd Int. Workshop on Web Services and Formal Methods, LNCS 4184, pages 38-57, Springer, 2006.

Basic Structure

- Looks for a small set of primitives that might serve as a basis for formalising and programming service oriented applications over global computers
- integrates complementary aspects from π-calculus (naming, hence sessions), Orc (pipelining of activities), webπ, cjoin, Sagas (implicit transactions and compensations)

SC: Signal Calculus

Main Aims

- Event-Based Service Coordination with the goal of providing a semantic based framework to implement higher level languages e.g. WS-CDL, BPEL, SAGA, . . . , and a common programming model for service coordination (choreography)
- G. Ferrari, R. Guanciale, D. Strollo, 4th Int. Conference on Service-Oriented Computing, LNCS 4294, Springer, 2006

Basic Structure

- A process calculus with asynchronous communication based on the Event Notification paradigm
- Uses event topics to describe coordination policies
- Permits dynamic interface publication and modification of service connections.

λ^{req} (or CCC): A Calculus for Call by Contract

Main Aims

- Calculus based on call-by-contract service selection that guarantees security-awareness from the design phase and supports verification techniques for planning sound compositions
- M. Bartoletti, P. Degano, G.L. Ferrari, Tech Report TR-07-02, Dip. Informatica, Univ. Pisa. To appear in Journal of Computer Security, 2007.

Basic Structure

- Selects and configures services, to guarantee that their composition enjoys some desirable properties
- Takes into account non-functional aspects: security and contracts
- Analizes service infrastructure to see which features are needed to guarantee semantic properties on service behaviour

Assessing the Calculi

Different abstraction levels and different aims

SC aims at providing a basic framework (a middleware?) for implementing languages for SOC;

SOCK is an abstraction of BPEL approach but sufficiently close to it to mimic its development process;

COWS starts from BPEL but abstract more from it to get very close to a classical process description language;

SCC starts from the abstract notion of session to develop a calculus along the lines of π-calculus;

λ^{req} is specifically designed for call-by-contract service selection and the advocated paradigm could be used to extend any of the previous formalisms.

Assessing the Calculi

Different abstraction levels and different aims

SC aims at providing a basic framework (a middleware?) for implementing languages for SOC;

SOCK is an abstraction of BPEL approach but sufficiently close to it to mimic its development process;

COWS starts from BPEL but abstract more from it to get very close to a classical process description language;

SCC starts from the abstract notion of session to develop a calculus along the lines of π-calculus;

λ^{req} is specifically designed for call-by-contract service selection and the advocated paradigm could be used to extend any of the previous formalisms.

Assessing the Calculi

Different abstraction levels and different aims

SC aims at providing a basic framework (a middleware?) for implementing languages for SOC;

SOCK is an abstraction of BPEL approach but sufficiently close to it to mimic its development process;

COWS starts from BPEL but abstract more from it to get very close to a classical process description language;

SCC starts from the abstract notion of session to develop a calculus along the lines of π-calculus;

λ^{req} is specifically designed for call-by-contract service selection and the advocated paradigm could be used to extend any of the previous formalisms.

Assessing the Calculi

Different abstraction levels and different aims

SC aims at providing a basic framework (a middleware?) for implementing languages for SOC;

SOCK is an abstraction of BPEL approach but sufficiently close to it to mimic its development process;

COWS starts from BPEL but abstract more from it to get very close to a classical process description language;

SCC starts from the abstract notion of session to develop a calculus along the lines of π-calculus;

λ^{req} is specifically designed for call-by-contract service selection and the advocated paradigm could be used to extend any of the previous formalisms.

Assessing the Calculi

Different abstraction levels and different aims

SC aims at providing a basic framework (a middleware?) for implementing languages for SOC;

SOCK is an abstraction of BPEL approach but sufficiently close to it to mimic its development process;

COWS starts from BPEL but abstract more from it to get very close to a classical process description language;

SCC starts from the abstract notion of session to develop a calculus along the lines of π-calculus;

λ^{req} is specifically designed for call-by-contract service selection and the advocated paradigm could be used to extend any of the previous formalisms.

Assessing the Calculi

Different abstraction levels and different aims

SC aims at providing a basic framework (a middleware?) for implementing languages for SOC;

SOCK is an abstraction of BPEL approach but sufficiently close to it to mimic its development process;

COWS starts from BPEL but abstract more from it to get very close to a classical process description language;

SCC starts from the abstract notion of session to develop a calculus along the lines of π-calculus;

λ^{req} is specifically designed for call-by-contract service selection and the advocated paradigm could be used to extend any of the previous formalisms.

Future Development of the Calculi

Possible future for the Calculi

SC could be used as a basis for developing (in alternative to IMC or together with it) middleware for the implementation of service description languages;

SOCK could be distilled into a process calculus, possibly evaluating its relationships with COWS;

COWS could be refined and contrasted with SCC with the aim of understanding which of them is better suited for qualitative and quantitative analysis of the specified systems;

SCC that in the mean time has given rise to different dialects should be cleaned up and assessed against case studies;

λ^{req} will be part of the service composition deliverable and will be used to extend other calculi.

Future Development of the Calculi

Possible future for the Calculi

SC could be used as a basis for developing (in alternative to IMC or together with it) middleware for the implementation of service description languages;

SOCK could be distilled into a process calculus, possibly evaluating its relationships with COWS;

COWS could be refined and contrasted with SCC with the aim of understanding which of them is better suited for qualitative and quantitative analysis of the specified systems;

SCC that in the mean time has given rise to different dialects should be cleaned up and assessed against case studies;

λ^{req} will be part of the service composition deliverable and will be used to extend other calculi.

Future Development of the Calculi

Possible future for the Calculi

SC could be used as a basis for developing (in alternative to IMC or together with it) middleware for the implementation of service description languages;

SOCK could be distilled into a process calculus, possibly evaluating its relationships with COWS;

COWS could be refined and contrasted with SCC with the aim of understanding which of them is better suited for qualitative and quantitative analysis of the specified systems;

SCC that in the mean time has given rise to different dialects should be cleaned up and assessed against case studies;

λ^{req} will be part of the service composition deliverable and will be used to extend other calculi.

Future Development of the Calculi

Possible future for the Calculi

SC could be used as a basis for developing (in alternative to IMC or together with it) middleware for the implementation of service description languages;

SOCK could be distilled into a process calculus, possibly evaluating its relationships with COWS;

COWS could be refined and contrasted with SCC with the aim of understanding which of them is better suited for qualitative and quantitative analysis of the specified systems;

SCC that in the mean time has given rise to different dialects should be cleaned up and assessed against case studies;

λ^{req} will be part of the service composition deliverable and will be used to extend other calculi.

Future Development of the Calculi

Possible future for the Calculi

SC could be used as a basis for developing (in alternative to IMC or together with it) middleware for the implementation of service description languages;

SOCK could be distilled into a process calculus, possibly evaluating its relationships with COWS;

COWS could be refined and contrasted with SCC with the aim of understanding which of them is better suited for qualitative and quantitative analysis of the specified systems;

SCC that in the mean time has given rise to different dialects should be cleaned up and assessed against case studies;

λ^{req} will be part of the service composition deliverable and will be used to extend other calculi.

Future Development of the Calculi

Possible future for the Calculi

SC could be used as a basis for developing (in alternative to IMC or together with it) middleware for the implementation of service description languages;

SOCK could be distilled into a process calculus, possibly evaluating its relationships with COWS;

COWS could be refined and contrasted with SCC with the aim of understanding which of them is better suited for qualitative and quantitative analysis of the specified systems;

SCC that in the mean time has given rise to different dialects should be cleaned up and assessed against case studies;

λ^{req} will be part of the service composition deliverable and will be used to extend other calculi.

Simplifying the Scenario

We propose to:

- Further study SC within WP6 as a tool for developing middleware for services deployment.
- To study λ^{req} within the task of WP2 dealing with resources handling and services discovery.

This will enable us to concentrate the calculi that appear to be at comparable abstraction level:

- SOCK
- COWS
- SCC

and to better understand their structure.

We would like to end up with a set of basic concepts - primitives - operators that capture basic notion of SOC and could be then grouped/subsetted in different calculi aiming for capturing different approaches to services development and analysis.

Correlations vs Sessions

Correlation Based Calculi

The link between partners (caller and callee) are determined by correlation sets. An instance contains some correlation values and only messages with the right correlation values are received. Two different alternatives are being considered corresponding to COWS and SOCKS, their main difference is that the former is stateless (correlation based on values) and the latter is stateful (correlation based on variables).

Session Based Calculi

In SCC, a session corresponds to a private channel that is instantiated when calling a service: It binds caller and callee and is used for their communication. To manage inter-session communication different mechanisms have been proposed ({C,P,S}-SCC) and are at the moment under evaluation.

Variants of SCC

PSCC: Dataflow oriented

Boreale, Bruni, DeNicola, Loreti:
A pipelining operator (*à la* ORC) is introduced to model the passage of information between sessions; return is used by sessions for passing values to the environment.

SSCC: Stream oriented

Lanese, Martins, Ravara, Vasconcelos:
Relies on explicit streams; primitives for inserting/retrieving data are used both for inter-session communication and for communication with the environment.

CSCC: Message passing oriented

Caires, Vieira:
Three distinct primitives are used for inter-process and inter-session communication and for communicating with the environment.

Outline

A three layered calculus

SOCK is structured along three layers corresponding to different phases of services development and deployment:

- Service behaviour calculus
- Service engine calculus
- Service behaviour calculus

The core component is the behavioural calculus; its operators can be grouped as follows:

- Communication primitives inspired by WSDL operations: One-Way(Input), Request-Response (Input/Output) Notification (Output), Solicit-Response (Output/Input).
- Computational primitives: Assignment.
- Workflow operators: Sequentialization, Parallel Composition, Choice (if then else or input-guarded), Iteration, Compensation and fault handling.

SOCK service behaviour calculus

$$\epsilon ::= s \mid o(\vec{x}) \mid o_r(\vec{x}, \vec{y}, P) \qquad \text{inputs}$$
$$\bar{\epsilon} ::= \bar{s} \mid \overline{o}@k(\vec{x}) \mid \overline{o_r}@k(\vec{x}, \vec{y}) \qquad \text{outputs}$$

$P, Q, \ldots ::=$ processes

$\mathbf{0}$	null process
$\bar{\epsilon}$	output
$x := e$	assignment
$\chi ? P : Q$	if then else
$P; P$	sequence
$P \mid P$	pararallel
$\sum^{+}_{i \in W} \epsilon_i; P_i$	input-guarded non-det. choice
$\chi \rightleftharpoons P$	iteration
$install(u, P)$	install handler
$\{P\}_q$	scope (shortcut for $\{P : \mathcal{H}_0\}_q$)
$throw(f)$	throw
$comp(q)$	compensate

SOCK service behaviour calculus: the semantics

- Action labels are composed by an action name and two parameters, the former parameter expresses a reading condition on the state, the latter parameter expresses a state update. Both parameters are tested within the service engine layer.
- One-Way message exchange rules:

(ONE-WAYOUT)
$$\overline{o}@z(\vec{x}) \xrightarrow{\overline{o}(\vec{v})@l(l/z,\vec{v}/\vec{x}:\_)} \mathbf{0}$$

(ONE-WAYIN)
$$o(\vec{x}) \xrightarrow{o(\vec{v})(\emptyset:\vec{v}/\vec{x})} \mathbf{0}$$

- Assignment rule:

(ASSIGN)
$$\frac{\mathrm{Dom}(\sigma) = \mathrm{Var}(e) \quad [\![e\sigma]\!] = v}{x := e \xrightarrow{\tau(\sigma:v/x)} \mathbf{0}}$$

SOCK service engine calculus

- A service engine executes service behaviour sessions.
  It is composed of:
  - A *declaration* that specifies how the service has to be deployed within the engine
  - An *execution environment* that is the set of running sessions with their (possibly shared) state.
- A *declaration* allows for the specification of:
  - Service behaviour
  - State: It may be *persistent* (shared among the running sessions) or *not persistent* (each session has its own state that expires when it terminates).
  - Correlation set: each session is characterized by the values of some variables of the behaviour specified within the declaration as *correlated variables*
  - Execution modality: A session can be *sequential* (it starts only if there is no other active sessions) or *concurrent* (it is spawned in parallel within the execution environment).

SOCK service engine calculus

- A service engine executes service behaviour sessions.
  It is composed of:
  - A *declaration* that specifies how the service has to be deployed within the engine
  - An *execution environment* that is the set of running sessions with their (possibly shared) state.
- A *declaration* allows for the specification of:
  - Service behaviour
  - State: It may be *persistent* (shared among the running sessions) or *not persistent* (each session has its own state that expires when it terminates).
  - Correlation set: each session is characterized by the values of some variables of the behaviour specified within the declaration as *correlated variables*
  - Execution modality: A session can be *sequential* (it starts only if there is no other active sessions) or *concurrent* (it is spawned in parallel within the execution environment).

SOCK service engine calculus: the syntax

- A service engine is defined in the following way:

$$Y ::= D[H]$$

where D is the declaration and H is the execution environment

- Declaration syntax:

$D ::= !W \mid W^*$	Execution modality (concurrent/sequential)
$W ::= c \triangleright U$	c is the correlation set
$U ::= P_\bullet \mid P_\times$	P is a service behaviour and subscript indicates whether the state is persistent or not

- Execution environment syntax:

$H ::= c \triangleright I$	Execution environment
$I ::= (P, S) \mid I \mid I$	Session consisting of parallel couples of behaviours and states

SOCK service engine calculus: the syntax

- A service engine is defined in the following way:

$$Y ::= D[H]$$

where D is the declaration and H is the execution environment

- Declaration syntax:

$D ::= !W \mid W^*$	Execution modality (concurrent/sequential)
$W ::= c \triangleright U$	c is the correlation set
$U ::= P_\bullet \mid P_\times$	P is a service behaviour and subscript indicates whether the state is persistent or not

- Execution environment syntax:

$H ::= c \triangleright I$	Execution environment
$I ::= (P, S) \mid I \mid I$	Session consisting of parallel couples of behaviours and states

SOCK service engine calculus: the syntax

- A service engine is defined in the following way:

$$Y ::= D[H]$$

where D is the declaration and H is the execution environment

- Declaration syntax:

$D ::= !W \mid W^*$	Execution modality (concurrent/sequential)
$W ::= c \triangleright U$	c is the correlation set
$U ::= P_\bullet \mid P_\times$	P is a service behaviour and subscript indicates whether the state is persistent or not

- Execution environment syntax:

$H ::= c \triangleright I$	Execution environment
$I ::= (P, \mathcal{S}) \mid I \mid I$	Session consisting of parallel couples of behaviours and states

SOCK service engine calculus: persistent or not persistent state

- A *state* is a function storing the value of the variables
- The state is supplied by the engine within the execution environment
- A 'persistent' state is shared among the running sessions
- Semantics for the state: ι is the action name and $\vec{v}/\vec{x}$ and $\mathcal{S}(\vec{x})$ are two parameters exploited within the correlation layer.

(ENGINE-STATE 1)

$$\frac{P \xrightarrow{\iota(\sigma:\vec{v}/\vec{x})} P', \mathcal{S} \vdash \sigma, \iota \neq \tau}{(P, \mathcal{S}) \xrightarrow{\iota(\vec{v}/\vec{x}:\mathcal{S}(\vec{x}))} (P', \mathcal{S} \oplus [\vec{v}/\vec{x}])}$$

SOCK service engine calculus: correlation set

- Semantics rules, where $\vec{v}/\vec{x} \vdash_c \vec{w}$ means that variable $\vec{x}$ is correlated under cset c because its actual value $\vec{w}$ corresponds to the incoming value $\vec{v}$ and $\vec{x}$ belongs to c, or $\vec{x}$ does not belong to c.

$$\text{(CORRELATED)}\quad \frac{l \xrightarrow{\iota(\vec{v}/\vec{x}:\vec{w})} l', \vec{v}/\vec{x} \vdash_c \vec{w}}{l \xrightarrow{\iota,c} l'}$$

- Example:

SOCK service engine calculus: correlation set

- Semantics rules, where $\vec{v}/\vec{x} \vdash_c \vec{w}$ means that variable $\vec{x}$ is correlated under cset c because its actual value $\vec{w}$ corresponds to the incoming value $\vec{v}$ and $\vec{x}$ belongs to c, or $\vec{x}$ does not belong to c.

$$\text{(CORRELATED)}\quad \frac{I \xrightarrow{\iota(\vec{v}/\vec{x}:\vec{w})} I', \vec{v}/\vec{x} \vdash_c \vec{w}}{I \xrightarrow{\iota,c} I'}$$

- Example:

SOCK service engine calculus: correlation set

- Semantics rules, where $\vec{v}/\vec{x} \vdash_c \vec{w}$ means that variable $\vec{x}$ is correlated under cset c because its actual value $\vec{w}$ corresponds to the incoming value $\vec{v}$ and $\vec{x}$ belongs to c, or $\vec{x}$ does not belong to c.

$$\text{(CORRELATED)}\quad \frac{I \xrightarrow{\iota(\vec{v}/\vec{x}:\vec{w})} I', \vec{v}/\vec{x} \vdash_c \vec{w}}{I \xrightarrow{\iota,c} I'}$$

- Example:

$\{x\} \triangleright (a(< x, y >); P, \mathcal{S}[4/x])$

- $\{x\}\triangleright$ defines x as the correlated variable
- $\mathcal{S}[4/x]$ is the actual state of the service where x is initialized with the value 4. Since x is a correlated variable, a message can be received on a only if the first received value is equal to 4

SOCK service engine calculus: execution modality

- The execution modality expresses how the different sessions must be executed
- Sessions can be executed
  - *sequentially*: a session can start only if there are no other active sessions within the execution environment
  - *concurrently*: each session can start independently from the others. It will be spawned in parallel within the execution environment
- Concurrent not persistent rule.

(SPAWN)

$$\frac{(P, \mathcal{S}_\perp) \xrightarrow{\iota, c} (P', \mathcal{S}),\ \nexists \mathcal{S}_i \in \mathsf{extr}(I).(P, \mathcal{S}_i) \xrightarrow{\iota, c} (P', \mathcal{S}'_i), \iota \in In}{c \triangleright P[I] \xrightarrow{\iota} c \triangleright P[I|(P', \mathcal{S})]}$$

A new session is started only if not session exists whose state satisfies correlation set c.

Service system calculus: the syntax

- The service system calculus permits composing located service engines within a system

$$E ::= Y@l \mid E \parallel E$$

- One-Way message exchange rule:

$$\text{(NormalSync)} \quad \frac{Y@l' \xrightarrow{\lambda@l} Y'@l' \;,\; Z@l \xrightarrow{\lambda'} Z'@l \;,\; comp(\lambda, \lambda')}{Y@l' \parallel Z@l \xrightarrow{\tau} Y'@l' \parallel Z'@l}$$

where $comp(\lambda, \lambda')$ verifies that operation names and exchanged values in One-Way λ and Notification $\overline{\lambda}$ are equal.

Outline

COWS: Calculus for Orchestration of Web Services

- Processes (*services*) create instances to serve specific service invocations.
- *Instances* contain concurrent threads (possibly with a *shared state*)
- Services and instances communicate through channels (*endpoints*).
- Endpoint's names can be communicated but received end-point can only be used for sending.
- Communication is regulated by *pattern-matching* that is also used to correlate, *by means of their same contents*, different service interactions logically forming *sessions*.
- The only binder is the *delimitation* operator; it can generate fresh names and regulate the range of application of substitutions generated by communication
- Termination of parallel activities can be forced by a *kill*, but sensitive code can be *protected*.

COWS^{---}: Syntax

s ::=	(services)	(notations)
$u \bullet u'!\bar{e}$	(*invoke*)	e *expressions*
$\mid \sum_{i=0}^{r} p_i \bullet o_i?\bar{w}_i.s_i$	(*receive-guarded choice*)	x *variables*
$\mid s \mid s$	(*parallel composition*)	v *values*
$\mid [d]\, s$	(*delimitation*)	n, p, o *names*
$\mid * s$	(*replication*)	u, d: names|vars
		w: values|vars

- Services are provided and invoked through *communication endpoints*, written as $p \bullet o$ (i.e. 'partner name' *plus* 'operation name')
- Partner names and operation names can be exchanged when communicating (only the 'send capability' is passed over).
- *Only one binding construct*: $[d]\, s$ binds d in the scope s (free/bound names/variables and closed terms defined accordingly)
- $\bar{\cdot}$ denotes tuples of objects, while empty choice will be rendered as **0**.

COWS^{--}: Structural congruence

Delimitation is used to

- generate fresh names (like the restriction of the π-calculus)
- regulate the range of application of substitutions generated by communication (indeed, receive activities do not bind variables)

$$[d]\,\mathbf{0} \equiv \mathbf{0} \qquad [d_1]\,[d_2]\,s \equiv [d_2]\,[d_1]\,s$$

$$s_1 \mid [d]\,s_2 \equiv [d]\,(s_1 \mid s_2) \quad \text{if } d \notin \mathrm{fd}(s_1)$$

. . . *plus* standard laws for $\sum$, $\mid$ and $*$

. . . and the *standard* operational rule

$$\frac{s \equiv s_1 \quad s_1 \xrightarrow{\alpha} s_2 \quad s_2 \equiv s'}{s \xrightarrow{\alpha} s'}$$

COWS^{--}: Structural congruence

Delimitation is used to

- generate fresh names (like the restriction of the π-calculus)
- regulate the range of application of substitutions generated by communication (indeed, receive activities do not bind variables)

$$[d]\,\mathbf{0} \equiv \mathbf{0} \qquad [d_1]\,[d_2]\,s \equiv [d_2]\,[d_1]\,s$$

$$s_1 \mid [d]\,s_2 \equiv [d]\,(s_1 \mid s_2) \quad \text{if } d \notin \mathrm{fd}(s_1)$$

. . . *plus* standard laws for $\sum$, $\mid$ and $*$

. . . and the *standard* operational rule

$$\frac{s \equiv s_1 \quad s_1 \xrightarrow{\alpha} s_2 \quad s_2 \equiv s'}{s \xrightarrow{\alpha} s'}$$

COWS^{--}: Invoke/receive activities & Choice

Invoke activities

- Can proceed only if the expressions in the argument can be evaluated
- *Evaluation function* $[\![\_]\!]$: takes closed expressions and returns values

$$\frac{[\![\bar{e}]\!] = \bar{v}}{p \bullet o!\bar{e} \xrightarrow{(p \cdot o) \lhd \bar{v}} \mathbf{0}}$$

Receive activities & *Choice*

- Offers an alternative choice of endpoints
- It is *not* a binder for names and variables (delimitation is used to delimit their scope)

$$\sum_{i=0}^{r} p_i \bullet o_i?\bar{w}_i.s_i \xrightarrow{(p_i \cdot o_i) \rhd \bar{w}_i} s_i$$

COWS^{--}: Invoke/receive activities & Choice

Invoke activities

- Can proceed only if the expressions in the argument can be evaluated
- *Evaluation function* $[\![\_]\!]$: takes closed expressions and returns values

$$\frac{[\![\bar{e}]\!] = \bar{v}}{p \bullet o!\bar{e} \xrightarrow{(p \cdot o) \lhd \bar{v}} \mathbf{0}}$$

Receive activities & *Choice*

- Offers an alternative choice of endpoints
- It is *not* a binder for names and variables (delimitation is used to delimit their scope)

$$\sum_{i=0}^{r} p_i \bullet o_i?\bar{w}_i.s_i \xrightarrow{(p_i \cdot o_i) \rhd \bar{w}_i} s_i$$

COWS^{--}: Communication

- Take place when two parallel services perform matching receive and invoke activities
- If more then one matching is possible the *receive* that needs fewer substitutions is selected to progress.

$$\frac{s_1 \xrightarrow{(p\cdot o)\rhd \bar{w}} s_1' \quad s_2 \xrightarrow{(p\cdot o)\lhd \bar{v}} s_2' \quad \mathcal{M}(\bar{w},\bar{v}) = \sigma \quad \neg(s_1 \mid s_2 \downarrow^{|\sigma|}_{p\bullet o,\bar{v}})}{s_1 \mid s_2 \xrightarrow{p\cdot o\, \lfloor\sigma\rfloor\, \bar{w}\, \bar{v}} s_1' \mid s_2'}$$

Matching function

$$\mathcal{M}(x,v) = \{x \mapsto v\} \qquad \mathcal{M}(v,v) = \emptyset \qquad \frac{\mathcal{M}(w_1,v_1) = \sigma_1 \quad \mathcal{M}(\bar{w}_2,\bar{v}_2) = \sigma_2}{\mathcal{M}((w_1,\bar{w}_2),(v_1,\bar{v}_2)) = \sigma_1 \uplus \sigma_2}$$

Predicate $s \downarrow^{|\sigma|}_{p\bullet o,\bar{v}}$ checks existence of potential communication conflicts

i.e. the ability of s of performing a receive activity matching $\bar{v}$ over the endpoint $p \bullet o$ which generates a substitution 'smaller' than σ

COWS^{--}: Communication

- Take place when two parallel services perform matching receive and invoke activities
- If more then one matching is possible the *receive* that needs fewer substitutions is selected to progress.

$$\frac{s_1 \xrightarrow{(p\cdot o)\rhd \bar{w}} s_1' \quad s_2 \xrightarrow{(p\cdot o)\lhd \bar{v}} s_2' \quad \mathcal{M}(\bar{w},\bar{v}) = \sigma \quad \neg(s_1 \mid s_2 \downarrow^{|\sigma|}_{p\bullet o,\bar{v}})}{s_1 \mid s_2 \xrightarrow{p\cdot o\, \lfloor\sigma\rfloor\, \bar{w}\, \bar{v}} s_1' \mid s_2'}$$

Matching function

$$\mathcal{M}(x,v) = \{x \mapsto v\} \qquad \mathcal{M}(v,v) = \emptyset \qquad \frac{\mathcal{M}(w_1,v_1) = \sigma_1 \quad \mathcal{M}(\bar{w}_2,\bar{v}_2) = \sigma_2}{\mathcal{M}((w_1,\bar{w}_2),(v_1,\bar{v}_2)) = \sigma_1 \uplus \sigma_2}$$

Predicate $s \downarrow^{|\sigma|}_{p\bullet o,\bar{v}}$ checks existence of potential communication conflicts

i.e. the ability of s of performing a receive activity matching $\bar{v}$ over the endpoint $p \bullet o$ which generates a substitution 'smaller' than σ

COWS^{--}: Parallel composition

Execution of parallel services is interleaved, when no communication is involved:

$$\frac{s_1 \xrightarrow{\alpha} s_1' \qquad \alpha \neq (p \cdot o \lfloor \sigma \rfloor \bar{w}\, \bar{v})}{s_1 \mid s_2 \xrightarrow{\alpha} s_1' \mid s_2}$$

In case of communications, the receive activity with greater priority progresses:

$$\frac{s_1 \xrightarrow{p \cdot o \lfloor \sigma \rfloor \bar{w}\, \bar{v}} s_1' \qquad \neg(s_2 \downarrow^{|\mathcal{M}(\bar{w},\bar{v})|}_{p \bullet o, \bar{v}})}{s_1 \mid s_2 \xrightarrow{p \cdot o \lfloor \sigma \rfloor \bar{w}\, \bar{v}} s_1' \mid s_2}$$

COWS^{--}: Parallel composition

Execution of parallel services is interleaved, when no communication is involved:

$$\frac{s_1 \xrightarrow{\alpha} s_1' \qquad \alpha \neq (p \cdot o \lfloor \sigma \rfloor \bar{w}\, \bar{v})}{s_1 \mid s_2 \xrightarrow{\alpha} s_1' \mid s_2}$$

In case of communications, the receive activity with greater priority progresses:

$$\frac{s_1 \xrightarrow{p \cdot o \lfloor \sigma \rfloor \bar{w}\, \bar{v}} s_1' \qquad \neg\left(s_2 \downarrow_{p \bullet o, \bar{v}}^{|\mathcal{M}(\bar{w}, \bar{v})|}\right)}{s_1 \mid s_2 \xrightarrow{p \cdot o \lfloor \sigma \rfloor \bar{w}\, \bar{v}} s_1' \mid s_2}$$

COWS^{--}: Delimitation

- $[d]\,s$ behaves like s, except when the transition label α contains d
- When the whole scope of a variable x is determined, and a communication involving x within that scope is taking place the delimitation is ignored and the *substitution* for x is performed

$$\frac{s \xrightarrow{\alpha} s' \quad d \notin d(\alpha)}{[d]\,s \xrightarrow{\alpha} [d]\,s'} \qquad \frac{s \xrightarrow{p \cdot o \lfloor \sigma \uplus \{x \mapsto v'\} \rfloor\, \bar{w}\, \bar{v}} s'}{[x]\,s \xrightarrow{p \cdot o \lfloor \sigma \rfloor\, \bar{w}\, \bar{v}} s' \cdot \{x \mapsto v'\}}$$

Substitutions (ranged over by σ):

- functions from variables to values (written as collections of pairs $x \mapsto v$)
- $\sigma_1 \uplus \sigma_2$ denotes the union of σ_1 and σ_2 when they have disjoint domains

$d(\alpha)$ avoids capturing endpoints of actual communications, it denotes the set of names and variables occurring in α

except for $\alpha = p \cdot o \lfloor \sigma \rfloor\, \bar{w}\, \bar{v}$ for which we let $d(p \cdot o \lfloor \sigma \rfloor\, \bar{w}\, \bar{v}) = d(\sigma)$

COWS^{--}: Delimitation

- $[d]\,s$ behaves like s, except when the transition label α contains d
- When the whole scope of a variable x is determined, and a communication involving x within that scope is taking place the delimitation is ignored and the *substitution* for x is performed

$$\frac{s \xrightarrow{\alpha} s' \quad d \notin d(\alpha)}{[d]\,s \xrightarrow{\alpha} [d]\,s'} \qquad \frac{s \xrightarrow{p \cdot o \lfloor \sigma \uplus \{x \mapsto v'\} \rfloor\ \bar{w}\,\bar{v}} s'}{[x]\,s \xrightarrow{p \cdot o \lfloor \sigma \rfloor\ \bar{w}\,\bar{v}} s' \cdot \{x \mapsto v'\}}$$

Substitutions (ranged over by σ):

- functions from variables to values (written as collections of pairs $x \mapsto v$)
- $\sigma_1 \uplus \sigma_2$ denotes the union of σ_1 and σ_2 when they have disjoint domains

$d(\alpha)$ avoids capturing endpoints of actual communications, it denotes the set of names and variables occurring in α

except for $\alpha = p \cdot o \lfloor \sigma \rfloor\ \bar{w}\,\bar{v}$ for which we let $d(p \cdot o \lfloor \sigma \rfloor\ \bar{w}\,\bar{v}) = d(\sigma)$

COWS^{--}: Delimitation

- $[d]\,s$ behaves like s, except when the transition label α contains d
- When the whole scope of a variable x is determined, and a communication involving x within that scope is taking place the delimitation is ignored and the *substitution* for x is performed

$$\frac{s \xrightarrow{\alpha} s' \quad d \notin d(\alpha)}{[d]\,s \xrightarrow{\alpha} [d]\,s'} \qquad \frac{s \xrightarrow{p \cdot o \lfloor \sigma \uplus \{x \mapsto v'\} \rfloor\ \bar{w}\,\bar{v}} s'}{[x]\,s \xrightarrow{p \cdot o \lfloor \sigma \rfloor\ \bar{w}\,\bar{v}} s' \cdot \{x \mapsto v'\}}$$

Substitutions (ranged over by σ):

- functions from variables to values (written as collections of pairs $x \mapsto v$)
- $\sigma_1 \uplus \sigma_2$ denotes the union of σ_1 and σ_2 when they have disjoint domains

$d(\alpha)$ avoids capturing endpoints of actual communications, it denotes the set of names and variables occurring in α

except for $\alpha = p \cdot o \lfloor \sigma \rfloor\ \bar{w}\,\bar{v}$ for which we let $d(p \cdot o \lfloor \sigma \rfloor\ \bar{w}\,\bar{v}) = d(\sigma)$

COWS: Syntax

s ::=	(services)	(notations)
$\mathbf{kill}(k)$	(kill)	k *(killer) labels*
$\mid\ u \bullet u'!\bar{e}$	(invoke)	e *expressions*
$\mid\ \sum_{i=0}^{r} p_i \bullet o_i?\bar{w}_i.s_i$	(receive-guarded choice)	x *variables*
$\mid\ s \mid s$	(parallel composition)	v *values*
$\mid\ \{\!\mid s \mid\!\}$	(protection)	n, p, o *names*
$\mid\ [d]\, s$	(delimitation)	u: names|vars
$\mid\ * s$	(replication)	w: values|vars
		d: labels|names|vars

Only one binding construct: $[d]\, s$ binds d in the scope s
(free/bound names/variables/labels and closed terms defined accordingly)

COWS: Delimitation & Protection

Delimitation is used to:

- generate fresh names
- regulate the range of application of substitutions
- delimit the field of action of *kill* activities

$$s_1 \mid [d]\, s_2 \;\equiv\; [d]\,(s_1 \mid s_2) \quad \text{if } d \notin \mathrm{fd}(s_1) \cup \mathrm{fk}(s_2)$$

Differently from names/variables, the scope of killer labels cannot be extended

Protection

Protect sensitive code from the effect of a forced termination

$$\{\!|\mathbf{0}|\!\} \equiv \mathbf{0} \qquad \{\!|\,\{\!|s|\!\}\,|\!\} \equiv \{\!|s|\!\} \qquad \{\!|[d]\, s|\!\} \equiv [d]\,\{\!|s|\!\}$$

$$\frac{s \xrightarrow{\alpha} s'}{\{\!|s|\!\} \xrightarrow{\alpha} \{\!|s'|\!\}}$$

COWS: Delimitation & Protection

Delimitation is used to:

- generate fresh names
- regulate the range of application of substitutions
- delimit the field of action of *kill* activities

$$s_1 \mid [d]\, s_2 \;\equiv\; [d]\,(s_1 \mid s_2) \quad \text{if } d \notin \mathrm{fd}(s_1) \cup \mathrm{fk}(s_2)$$

Differently from names/variables, the scope of killer labels cannot be extended

Protection

Protect sensitive code from the effect of a forced termination

$$\{\!|\mathbf{0}|\!\} \equiv \mathbf{0} \qquad \{\!|\, \{\!|s|\!\} \,|\!\} \equiv \{\!|s|\!\} \qquad \{\!|[d]\, s|\!\} \equiv [d]\, \{\!|s|\!\}$$

$$\frac{s \xrightarrow{\alpha} s'}{\{\!|s|\!\} \xrightarrow{\alpha} \{\!|s'|\!\}}$$

COWS: Kill activity

Activity **kill**(k) forces termination of all unprotected parallel activities inside an enclosing $[k]$, that stops the killing effect

$$\mathbf{kill}(k) \xrightarrow{\dagger k} \mathbf{0} \qquad \frac{s_1 \xrightarrow{\dagger k} s_1'}{s_1 \mid s_2 \xrightarrow{\dagger k} s_1' \mid \mathit{halt}(s_2)} \qquad \frac{s \xrightarrow{\dagger k} s'}{[k]\, s \xrightarrow{\dagger} [k]\, s'}$$

Kill activities are executed *eagerly*

$$\frac{s \xrightarrow{\alpha} s' \quad d \notin d(\alpha) \quad s \downarrow_d \Rightarrow \alpha = \dagger, \dagger k}{[d]\, s \xrightarrow{\alpha} [d]\, s'}$$

Function halt(s), defined by s.i., strips off the protected activities and returns the service obtained by only retaining the protected activities inside s

Predicate $s \downarrow_d$ checks the ability of s of immediately performing **kill**(d)

COWS: Kill activity

Activity **kill**(k) forces termination of all unprotected parallel activities inside an enclosing $[k]$, that stops the killing effect

$$\mathbf{kill}(k) \xrightarrow{\dagger k} \mathbf{0} \qquad \frac{s_1 \xrightarrow{\dagger k} s_1'}{s_1 \mid s_2 \xrightarrow{\dagger k} s_1' \mid \mathit{halt}(s_2)} \qquad \frac{s \xrightarrow{\dagger k} s'}{[k]\, s \xrightarrow{\dagger} [k]\, s'}$$

Kill activities are executed *eagerly*

$$\frac{s \xrightarrow{\alpha} s' \quad d \notin d(\alpha) \quad s \downarrow_d \Rightarrow \alpha = \dagger, \dagger k}{[d]\, s \xrightarrow{\alpha} [d]\, s'}$$

Function halt(s), defined by s.i., strips off the protected activities and returns the service obtained by only retaining the protected activities inside s

Predicate $s \downarrow_d$ checks the ability of s of immediately performing **kill**(d)

Outline

General Principles: service definitions and invocations

Service definitions

- services expose their protocols (i.e. behaviours)
- services can be deployed dynamically, shut down and updated
- services can handle multiple requests separately

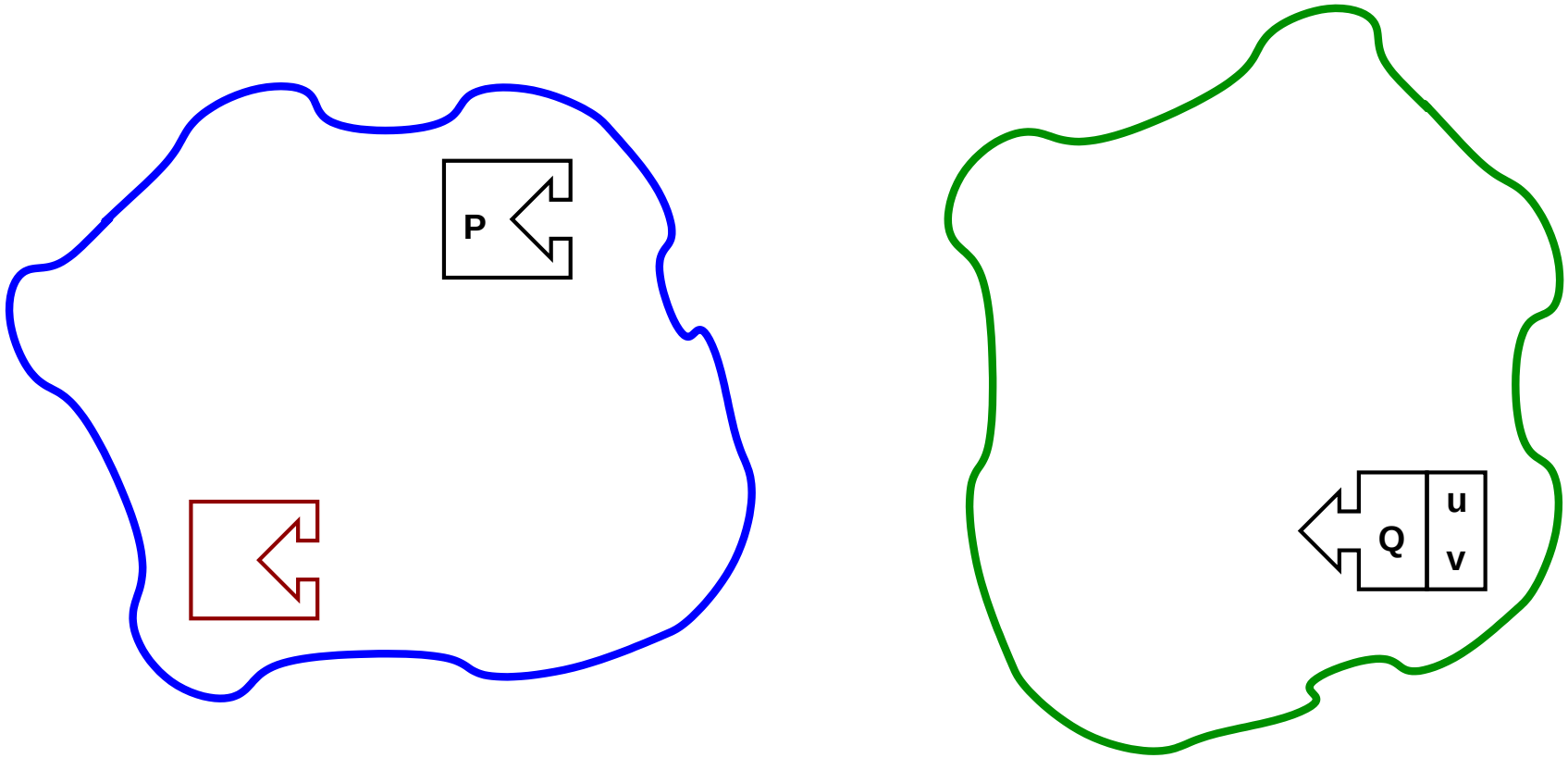
Service invocations

- service calls expose their protocols
- multiple invocations are possible (via data streaming á la Orc)

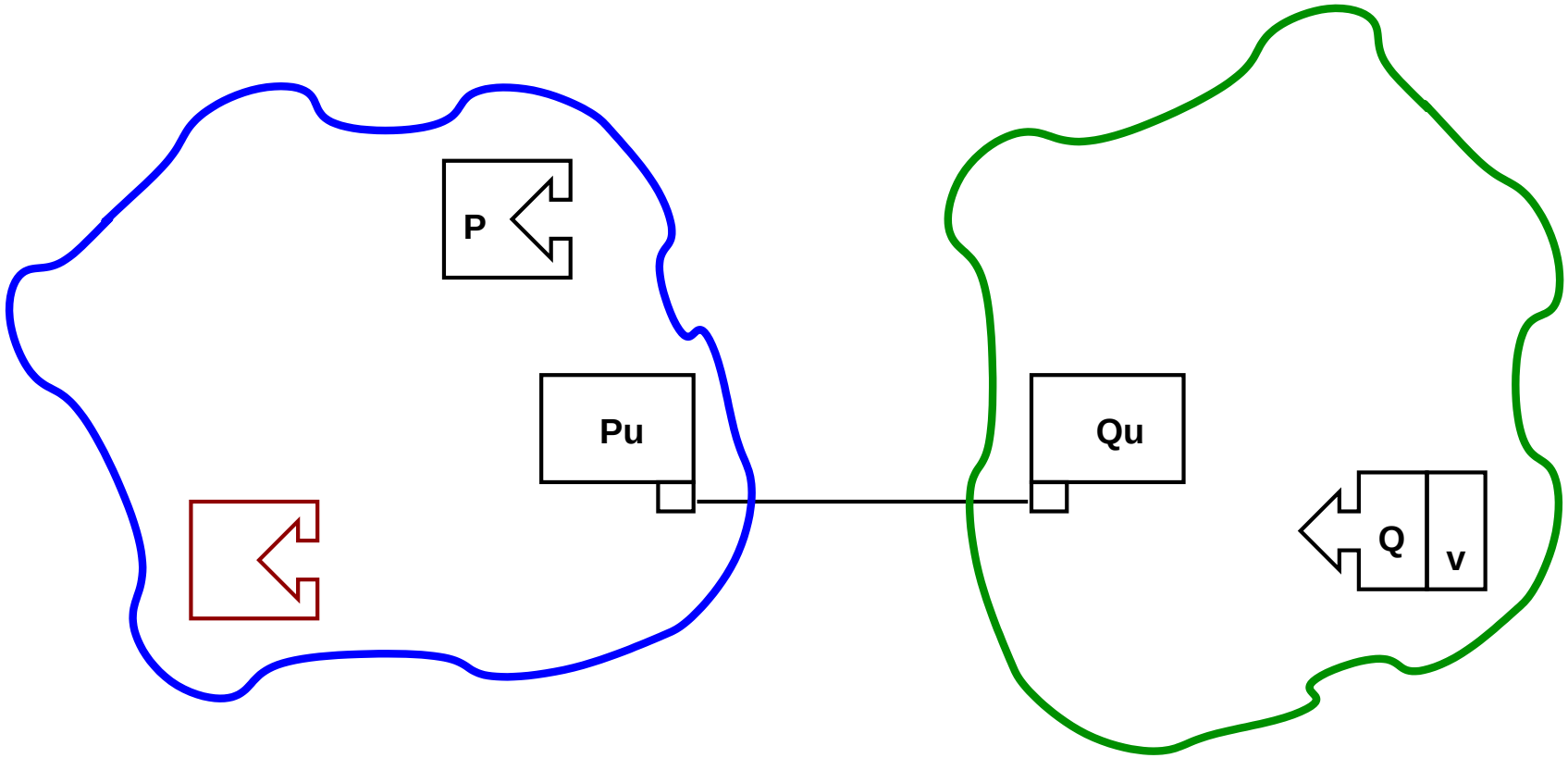
Sessions

- sessions are two-party (service-side + client-side)
- service invocations spawn fresh session parties (locally to each partner)
- communication within sessions is bi-directional
- sessions can be nested
- values can be returned outside sessions
- sessions can be closed by the involved parties

Service Activation, Graphically



Bidirectional Session, Graphically

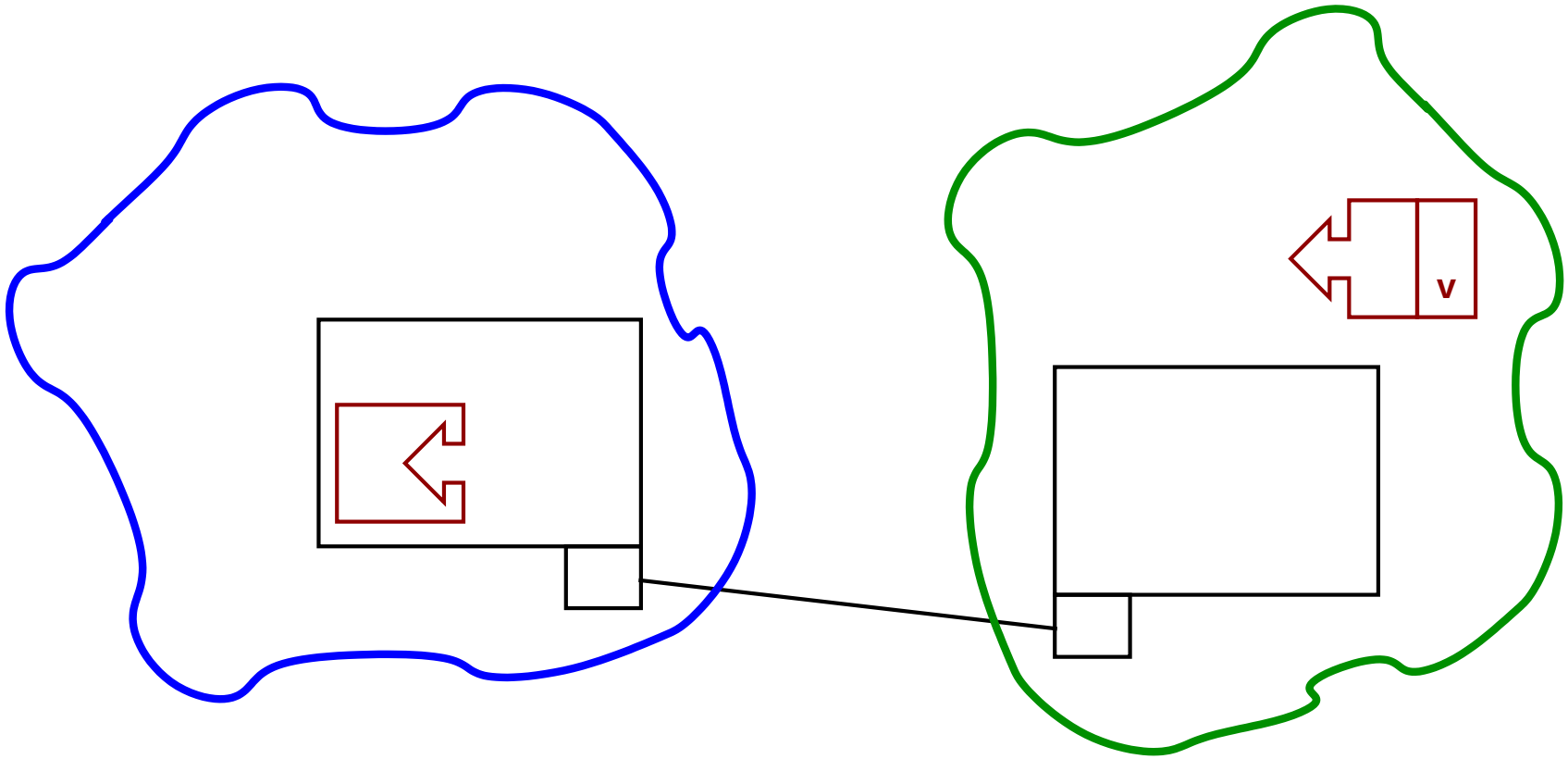


Intra-Session Communication, Graphically

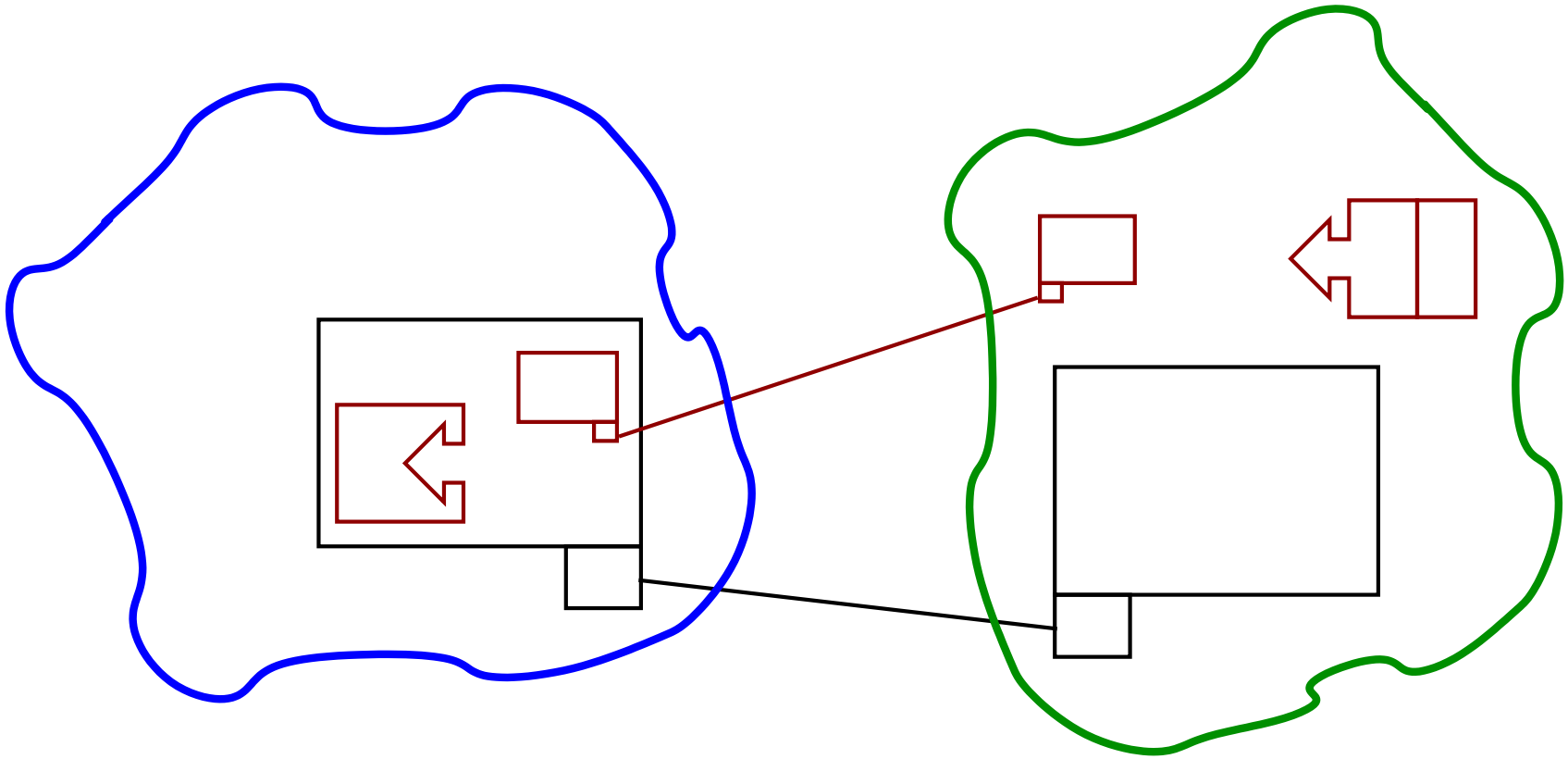
Intra-Session Communication, Graphically

Intra-Session Communication, Graphically

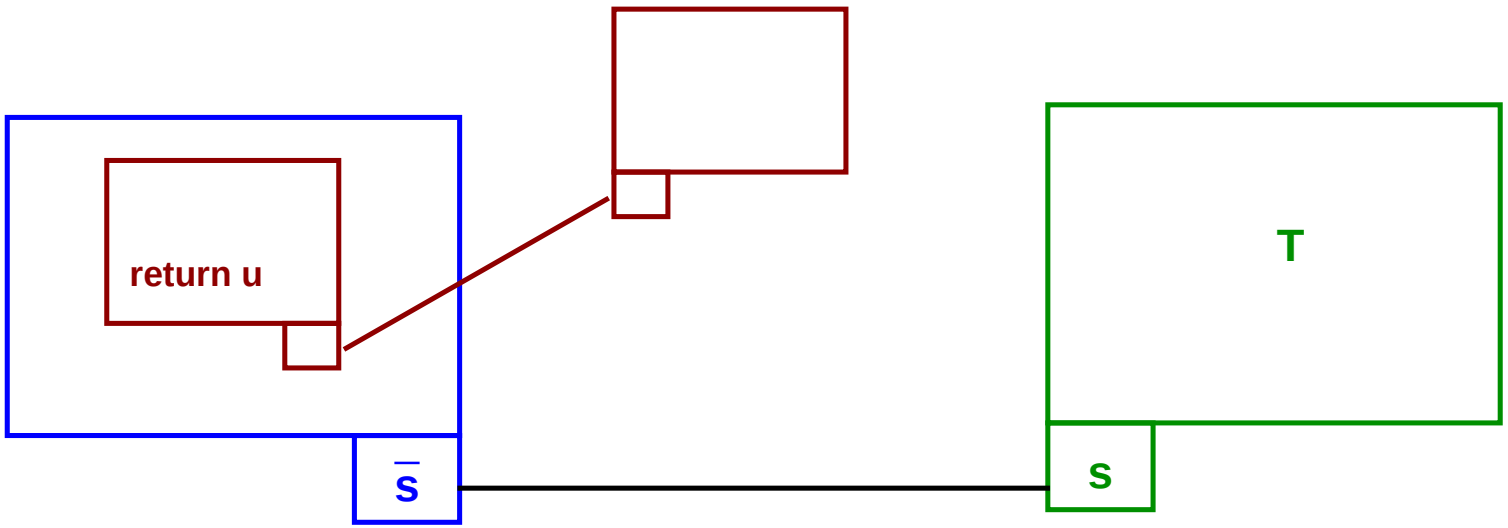
Nested Services and Multi-Sessions, Graphically



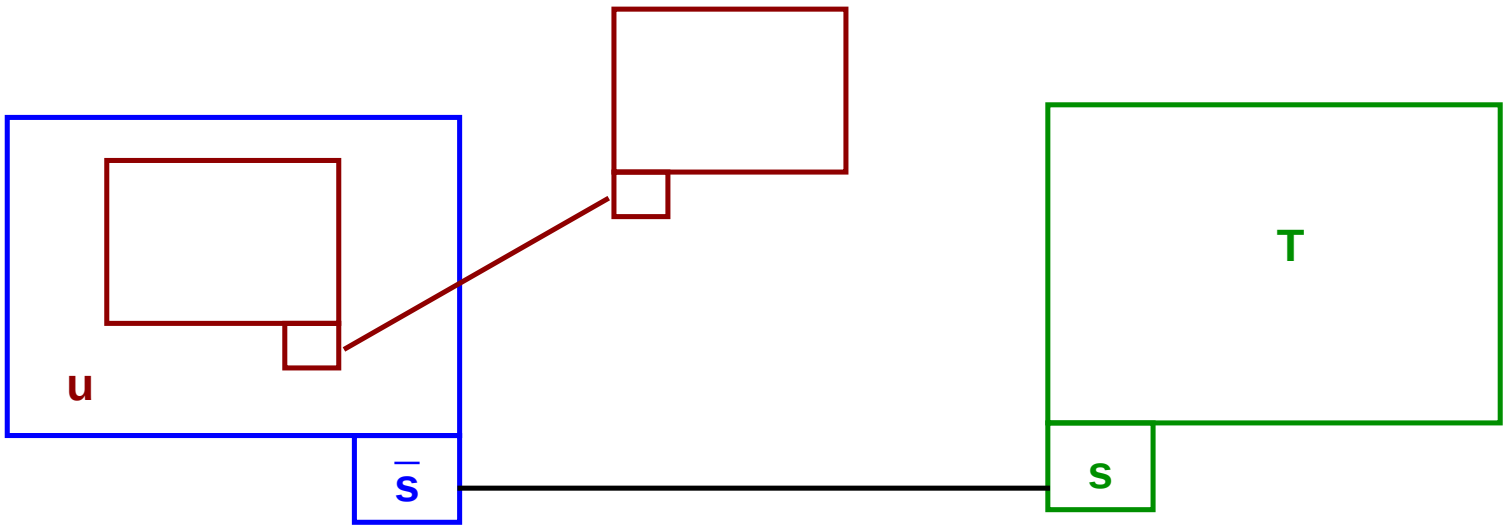
Nested Services and Multi-Sessions, Graphically



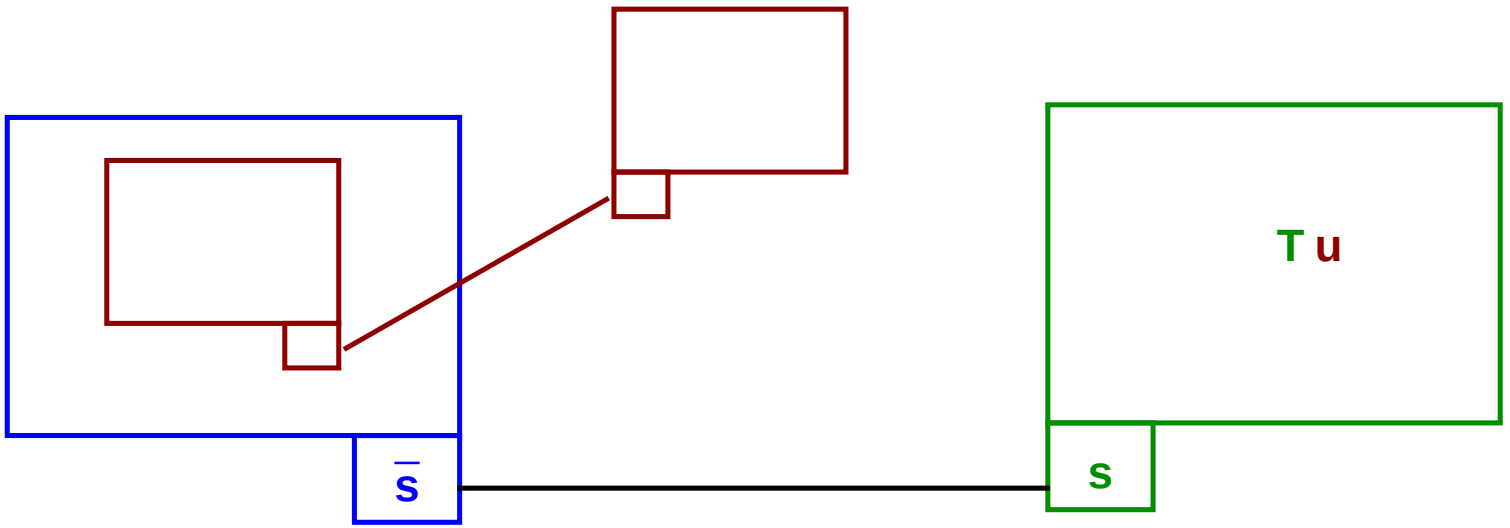
Returning Values, Graphically



Returning Values, Graphically



Returning Values, Graphically



PSCC Syntax (Close-free fragment)

P, Q	$::=$	$\mathbf{0}$	Nil
		$\mid \gamma P$	Concretion (output)
		$\mid \sum_i (\widetilde{F}_i) P_i$	Abstraction (input)
		$\mid\ !P$	Replication
		$\mid s.P$	Service Definition
		$\mid \overline{s}.P$	Service Invocation
		$\mid P > Q$	Pipeline
		$\mid r \triangleright P$	Session
		$\mid P \mid Q$	Parallel Composition
		$\mid (\nu n) P$	Name Restriction
γ	$::=$	$\langle \widetilde{V} \rangle \mid \langle \widetilde{V} \rangle^{\uparrow}$	
V	$::=$	$\ldots$	Values
F	$::=$	$\ldots$	Patterns

PSCC: Abstractions and Concretions

Concretion:

$\langle V \rangle P$ produces a value V and then behaves like P:

$$\langle V \rangle P \xrightarrow{\langle V \rangle} P$$

Abstraction:

$\sum_i (F_i) P_i$ waits for a value matching one of $F_1, \ldots, F_n$ and then activates the corresponding process:

$$\frac{match(F_j, V) = \sigma}{\sum_{i=0}^{k} (F_i) P_i \xrightarrow{(V)} P_j \sigma}$$

Service Definition and Service Invocation

- $s.P$ identifies a definition for service s with body P.
- $\overline{s}.Q$ invokes service s using protocol Q.
- A service invocation causes the activation of a new session; a fresh name r ($r \notin fn(P)$) identifies the two sides of the session.

$$s.P \xrightarrow{s(r)} r \triangleright P \qquad \overline{s}.Q \xrightarrow{\overline{s}(r)} r \triangleright Q$$

$$\frac{P \xrightarrow{s(r)} P' \quad Q \xrightarrow{\overline{s}(r)} Q'}{P|Q \xrightarrow{\tau} (\nu r)(P'|Q')}$$

PSCC: Session

Session:

$r \triangleright P$ is an abstract channel for Client and Server interaction:

$$\frac{P \xrightarrow{\langle V \rangle} P'}{r \triangleright P \xrightarrow{r:\langle V \rangle} r \triangleright P'} \qquad \frac{P \xrightarrow{(V)} P'}{r \triangleright P \xrightarrow{r:(V)} r \triangleright P'}$$

$$\frac{P \xrightarrow{r:(V)} P' \quad Q \xrightarrow{r:\langle V \rangle} Q'}{P|Q \xrightarrow{\tau} P'|Q'}$$

$$\frac{P \xrightarrow{\lambda} P'}{r \triangleright P \xrightarrow{\lambda} r \triangleright P'} \quad \lambda = \tau, s(r), \overline{s}(r)$$

PSCC: Inter-session interactions

Return:

$\langle V \rangle^{\uparrow} P$ can be used to return a value to the enclosing environment:

$$\langle V \rangle^{\uparrow} P \xrightarrow{\uparrow V} P \qquad \frac{P \xrightarrow{\uparrow V} P'}{r \rhd P \xrightarrow{\langle V \rangle} r \rhd P'}$$

Pipeline:

Different activities can be composed by using pipeline $P > Q$. Each value produced by P activates a new instantiation Q' of Q:

$$\frac{P \xrightarrow{\langle V \rangle} P' \quad Q \xrightarrow{(V)} Q'}{P > Q \xrightarrow{\tau} (P' > Q) | Q'} \qquad \frac{P \xrightarrow{\lambda} P' \quad \lambda \neq \langle V \rangle}{P > Q \xrightarrow{\lambda} P' > Q}$$

PSCC Syntax (with close)

P, Q	$::=$	$\mathbf{0}$	Nil
		$\mid \gamma P$	Concretion
		$\mid \sum_i (\widetilde{F}_i) P_i$	Abstraction
		$\mid !P$	Replication
		$\mid$ close	Close
		$\mid s_k.P$	Service Definition
		$\mid \overline{s_k}.Q$	Service Invocation
		$\mid P > Q$	Pipeline
		$\mid r \rhd_k P$	Session
		$\mid \blacklozenge P$	Terminated Session
		$\mid P \mid Q$	Parallel Composition
		$\mid (\nu n) P$	Name Restriction

PSCC: Closing sessions

- The syntax of service definition and service invocation become $\overline{s_k}.Q$ and $s_k.P$, where k is used for identifying the termination handler service to be associated to the other side of the instantiated session.

$$s_{k_1}.P \xrightarrow{s(r)^{k_1}_{k_2}} r \rhd_{k_2} P \qquad \overline{s}_{k_2} Q \xrightarrow{\overline{s}(r)^{k_1}_{k_2}} r \rhd_{k_1} P$$

$$\frac{P \xrightarrow{s(r)^{k_1}_{k_2}} P' \quad Q \xrightarrow{\overline{s}(r)^{k_1}_{k_2}} Q'}{P|Q \xrightarrow{\tau} (\nu r)(P'|Q')}$$

PSCC: Closing sessions

- We associate to each session a service name k which identifies a *termination handler* service $(r \rhd_k P)$
- As soon as P execute close , session r enters a closing state $(\blacklozenge Q)$ and the corresponding termination handler is invoked

$$\text{close} \xrightarrow{\text{close}} \mathbf{0} \qquad \frac{P \xrightarrow{\text{close}} P'}{r \rhd_k P \xrightarrow{\tau} \blacklozenge P' | \overline{k}.\mathbf{0}}$$

- A closing session can only trigger further closing of nested subsessions.

$$r \rhd_k P \xrightarrow{\dagger(k)} P \qquad \frac{P \xrightarrow{\dagger(k)} P'}{\blacklozenge P \xrightarrow{\tau} \blacklozenge P' | \overline{k}.\mathbf{0}}$$

SSCC Syntax

P, Q	$::=$	$\mathbf{0}$	Nil
		$\mid P \mid Q$	Parallel Composition
		$\mid (\nu a)P$	Name Restriction
		$\mid X$	Process Variable
		$\mid \mathrm{rec}\, X.P$	Recursive Process Definition
		$\mid a \Rightarrow P$	Service Definition
		$\mid a \Leftarrow P$	Service Invocation
		$\mid v.P$	Value Sending
		$\mid (x)P$	Value Reception
		$\mid \mathrm{stream}\, P \,\mathrm{as}\, f \,\mathrm{in}\, Q$	Stream
		$\mid \mathrm{feed}\, v.P$	Feed the Process' Stream
		$\mid f(x).P$	Read froom a Stream
u, v	$::=$	a	Service Name
		$\mid \mathrm{unit}$	Unit Value

SSCC: Runtime syntax

We use an extended syntax to give semantics to SSCC.

P, Q	$::=$	$\ldots$	Static Operators
		$\mid r \rhd P$	Server Session
		$\mid r \lhd P$	Client Session
		$\mid (\nu r)P$	Session Restriction
		$\mid$ stream P as $f = \vec{v}$ in Q	Stream with Values

Sessions are not available as programming construct.

Service Definition and Service Invocation

- Service definitions are rendered as $a \Rightarrow P$ where a is the service name and P is the process defining the service behaviour.
- Service invocation is written as $a \Leftarrow P$ where service a is invoked with client protocol P.
- Service invocation and service definition are symmetric.
- A service invocation creates a new session. A fresh name r identifies the two sides of the session. Client and server protocols are instantiated each at the proper side of the session.

$$\frac{r \notin fn(P)}{a \Rightarrow P \xrightarrow{a \Rightarrow (r)} r \rhd P} \qquad \frac{r \notin fn(P)}{a \Leftarrow P \xrightarrow{a \Leftarrow (r)} r \lhd P}$$

$$\frac{P \xrightarrow{a \Rightarrow (r)} P' \quad Q \xrightarrow{a \Leftarrow (r)} Q'}{P|Q \xrightarrow{\tau} (\nu r)(P'|Q')}$$

SSCC: In-Session Communication

Session:

A session $(r \triangleright P)$ is an abstraction for a bidirectional channel used by Client and Server protocols to interact each other:

$$v.P \xrightarrow{\overleftarrow{v}} P \qquad (x)P \xrightarrow{\overrightarrow{v}} P\{v/x\}$$

$$\frac{P \xrightarrow{\overleftrightarrow{v}} P' \;,\; \overleftrightarrow{v} \in \{\overleftarrow{v}, \overrightarrow{v}\} \;,\; \bowtie \in \{\triangleright, \triangleleft\}}{r \bowtie P \xrightarrow{r \bowtie \overleftrightarrow{v}} r \bowtie P'}$$

$$\frac{P \xrightarrow{\lambda} P' \;,\; \bowtie \in \{\triangleright, \triangleleft\} \;,\; \lambda \notin \{\overrightarrow{v}, \overleftarrow{v}\}}{r \bowtie P \xrightarrow{\lambda} r \bowtie P'}$$

$$\frac{P \xrightarrow{r \bowtie \overleftarrow{v}} P' \quad Q \xrightarrow{r \overline{\bowtie} \overrightarrow{v}} Q' \;,\; \bowtie \in \{\triangleright, \triangleleft\}}{P|Q \xrightarrow{r\tau} P'|Q'}$$

SSCC: Extra-Session Communication

The idea

- Processes need to communicate outside sessions (e.g., data from a google client are used by an hotel booking client).
- Models communication between processes on the same "machine" (but we do not want to represent machines explicitly).
- Should be orthogonal w.r.t. the session structure.
- Tradeoff between expressive power and structured communication (for typability, ...)
- We propose stream P as f in Q, modelling a stream named f for communication from process P to process Q.

SSCC: Extra-Session Communication

stream P as f in Q

- P and Q are concurrently executing.
- P can feed data to the nearest enclosing stream using feed $v.P'$ (feeds an unknown context).
- Q can read from stream named f with $f(x).Q'$ (Q can read from multiple sources).
- Feed is non blocking (f acts as a buffer), read is blocking.

$$\text{feed}\, v.P \xrightarrow{\Uparrow v} P \qquad f(x).P \xrightarrow{f \Downarrow v} P\{v/x\}$$

$$\frac{P \xrightarrow{\Uparrow v} P'}{\text{stream}\, P \,\text{as}\, f = \vec{w} \,\text{in}\, Q \xrightarrow{\tau} \text{stream}\, P' \,\text{as}\, f = v :: \vec{w} \,\text{in}\, Q}$$

$$\frac{Q \xrightarrow{f \Downarrow v} Q'}{\text{stream}\, P \,\text{as}\, f = \vec{w} :: v \,\text{in}\, Q \xrightarrow{\tau} \text{stream}\, P \,\text{as}\, f = \vec{w} \,\text{in}\, Q'}$$

CSCC: Key features

Context

- A site, or a conversation context, shared by various partners.

Service instantiation

Service instantiation creates a new (split) conversation context.

- Provider and client code lie in their respective contexts.

Services instances as delegated processes

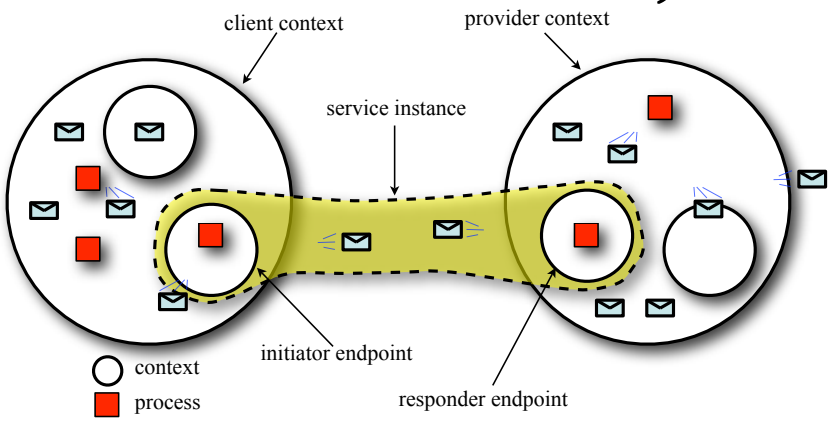
Tasks may be delegated either locally or remotely (loose coupling).

Uniform communication mechanism by message passing

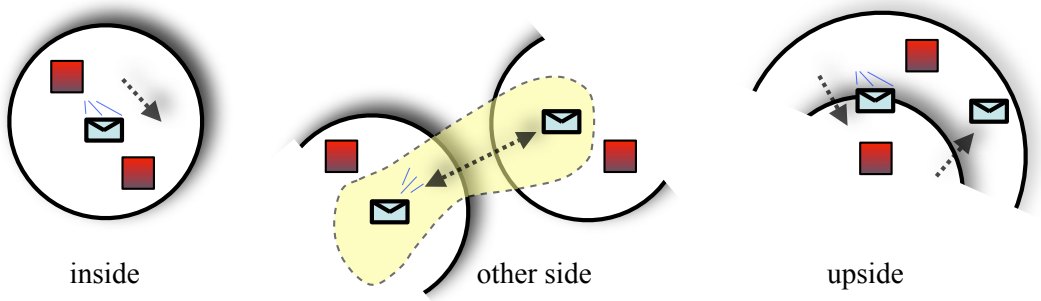
- in the same conversation context (e.g., in a orchestration);
- between nested contexts (e.g., between system and subsystem);
- between two endpoint roles (e.g, in a session).

CSCC: Endpoint Roles and Communication Directions

$$\rho \quad ::= \quad \blacktriangleright \ \text{Responder} \quad \Big\} \ \text{Roles}$$
$$\quad\ \ \mid \quad \blacktriangleleft \ \text{Initiator}$$



$$\alpha \quad ::= \quad \downarrow n \quad \text{Here} \quad \Big\} \ \text{Directions}$$
$$\quad\ \ \mid \quad \uparrow n \quad \text{Up}$$
$$\quad\ \ \mid \quad \leftarrow n \quad \text{There}$$



CSCC: Syntax

$P, Q \quad ::=$

	stop	Inaction
$\mid$	$P \mid Q$	Parallel
$\mid$	(**new** n)P	Restriction
$\mid$	**out** $\alpha(v_1, \ldots, v_n).P$	Out
$\mid$	**in** $\alpha(x_1, \ldots, x_n).P$	In (with choice $\oplus$)
$\mid$	$!P$	Replication

π-Calculus

$\mid$	$n\rho\,[P]$	Context
$\mid$	**here**$(x).P$	Awareness
$\mid$	**instance** $n\rho s \Leftarrow P$	Instantiation
$\mid$	**def** $s \Rightarrow P$	Definition

Services

$\mid$	**try** P **catch** Q	Try-Catch
$\mid$	**throw**$.P$	Throw

Exceptions

CSCC: Semantics (1/4)

π-calculus

$$\mathbf{in}\ \alpha(\overline{x}).P \xrightarrow{(\overline{n})\alpha(\overline{v})} P\{\overline{x}\leftarrow\overline{v}\} \quad (\overline{n} \subseteq \overline{v}) \qquad \mathbf{out}\ \alpha(\overline{v}).P \xrightarrow{\overline{\alpha(\overline{v})}} P$$

$$\frac{P \xrightarrow{\lambda} Q \qquad \lambda \neq \mathtt{throw}}{P \mid R \xrightarrow{\lambda} Q \mid R} \qquad \frac{P \xrightarrow{\lambda} Q \qquad n \notin \lambda}{(\mathbf{new}\ n)P \xrightarrow{\lambda} (\mathbf{new}\ n)Q}$$

$$\frac{P \xrightarrow{\lambda} Q \quad n \in out(\lambda)}{(\mathbf{new}\ n)P \xrightarrow{(n)\lambda} Q} \qquad \frac{P \xrightarrow{(\overline{n})\lambda} P' \quad Q \xrightarrow{(\overline{n})\overline{\lambda}} Q'}{P \mid Q \xrightarrow{\tau} (\mathbf{new}\ \overline{n})(P' \mid Q')}$$

$$\frac{P \mid !P \xrightarrow{\lambda} Q}{!P \xrightarrow{\lambda} Q}$$

CSCC: Semantics (2/4)

Message directions

$$\frac{P \xrightarrow{\uparrow\lambda} Q}{n\rho\,[P] \xrightarrow{\downarrow\lambda} n\rho\,[Q]} \qquad \frac{P \xrightarrow{\downarrow\lambda} Q}{n\rho\,[P] \xrightarrow{n\rho.\downarrow\lambda} n\rho\,[Q]} \qquad \frac{P \xrightarrow{\rightarrow\lambda} Q}{n\rho\,[P] \xrightarrow{n\rho.\rightarrow\lambda} n\rho\,[Q]}$$

Service instantiation

$$\mathbf{def}\ s \Rightarrow P \xrightarrow{(c)\mathtt{def}\ s} c \blacktriangleright [P] \qquad \mathbf{instance}\ n\rho s \Leftarrow P \xrightarrow{\overline{(c)n\rho.\mathtt{def}\ s}} c \blacktriangleleft [P]$$

$$\frac{P \xrightarrow{(c)\mathtt{def}\ s} Q}{n\rho\,[P] \xrightarrow{(c)n\rho.\mathtt{def}\ s} n\rho\,[Q]}$$

CSCC: Semantics (3/4)

Context awareness

$$\mathbf{here}(x).P \xrightarrow{n\rho.\mathtt{here}} P\{x \leftarrow n\} \qquad \frac{P \xrightarrow{n\rho.\mathtt{here}} Q}{n\rho\,[P] \xrightarrow{\tau} n\rho\,[Q]}$$

$$\frac{P \xrightarrow{(\overline{o})\lambda} P' \qquad Q \xrightarrow{(\overline{o})\overline{n\rho\lambda}} Q'}{P \mid Q \xrightarrow{n\rho.\mathtt{here}} (\mathbf{new}\ \overline{o})(P' \mid Q')}$$

Context

$$\frac{P \xrightarrow{\tau} Q}{n\rho\,[P] \xrightarrow{\tau} n\rho\,[Q]} \qquad \frac{P \xrightarrow{m\rho.\lambda} Q \qquad \lambda \neq \mathtt{here}}{n\rho\,[P] \xrightarrow{m\rho.\lambda} n\rho\,[Q]}$$

CSCC: Semantics (4/4)

Exception handling

$$\mathbf{throw}.P \xrightarrow{\texttt{throw}} P$$

$$\frac{P \xrightarrow{\texttt{throw}} R}{\mathbf{try}\ P\ \mathbf{catch}\ Q \xrightarrow{\tau} Q \mid R}$$

$$\frac{P \xrightarrow{\texttt{throw}} R}{P \mid Q \xrightarrow{\texttt{throw}} R}$$

$$\frac{P \xrightarrow{\lambda} Q \qquad \lambda \neq \texttt{throw}}{\mathbf{try}\ P\ \mathbf{catch}\ R \xrightarrow{\lambda} \mathbf{try}\ Q\ \mathbf{catch}\ R}$$

$$\frac{P \xrightarrow{\texttt{throw}} R}{n\rho\,[P] \xrightarrow{\texttt{throw}} R}$$

CSCC: Bisimulation

Definition

A (strong) bisimulation is a symmetric binary relation $\mathcal{R}$ on processes such that, for all processes P and Q, if $P\mathcal{R}Q$, we have:

$$\text{If } P \xrightarrow{\lambda} P' \text{ and } bn(\lambda)\#Q \text{ then there is a process } Q' \text{ such that } Q \xrightarrow{\lambda} Q' \text{ and } P'\mathcal{R}Q'.$$

We denote by $\sim$ (strong bisimilarity) the largest strong bisimulation.

Proposition

Strong bisimilarity is a congruence for all operators.

Equations

- $n \blacktriangleright [P] \mid n \blacktriangleright [Q] \sim n \blacktriangleright [P \mid Q]$.
- $n \blacktriangleleft [\mathbf{out} \uparrow m(\overline{v})] \sim \mathbf{out} \downarrow m(\overline{v})$.

Thank you for your attention!

Questions?