
Notation

Concepts

Active object	Root object of an activity	69
Activity	A process made of a single active object and a set of passive objects	69
Wait-by-necessity	Blocking of execution upon a strict operation on a future: $\alpha[\mathcal{R}[\iota \dots], \sigma_\alpha \dots] \wedge \sigma_\alpha(\iota) = fut(f_i^{\gamma \rightarrow \beta})$	70
Service method	Method started upon activation: m_j in $Active(a, m_j)$	72
Request	Asynchronous remote method call	69
Future	Represents the result of a request before the response is sent back	70
Future value	Value associated to a future $f_i^{\alpha \rightarrow \beta}$ $copy(\iota, \sigma)$ where $\{f_i^{\alpha \rightarrow \beta} \mapsto \iota\} \in F_\alpha$	74
Computed future	A future which has a value associated: $f_i^{\alpha \rightarrow \beta}$ where $f_i^{\alpha \rightarrow \beta} \in dom(F_\beta)$	106
Not updated future	A computed future not yet locally updated	113
Partial future value	Future value containing references to futures	74
Closed term	Term without free variable ($fv(a) = \emptyset$)	65
Source term	Closed term without location: $fv(a) = \emptyset \wedge locs(a) = \emptyset$	65
Reduced object	Object with all fields reduced to a location: $o ::= [l_i = \iota_i; m_j = \varsigma(x_j, y_j)a_j]_{j \in 1..m}^{i \in 1..n}$	66
Potential services	Static approximation of the set of method names appearing in service primitives $\mathcal{M}_{\alpha P}$	103
Interfering requests	Two requests that can be served by the same serve primitive: $[m_1; \iota; f_i^{\alpha \rightarrow \beta}]$ and $[m_2; \iota; f_i^{\gamma \rightarrow \beta}]$ such that there is $M, \{m_1, m_2\} \subseteq M$ and $Serve(M)$ can appear in β	107

$r = [m_j; \iota; f_i^{\alpha \rightarrow \beta}]$	Request: asynchronous remote method call	69
$R_\alpha = \{[m_j; \iota; f_i^{\alpha \rightarrow \beta}]\}$	Pending requests: a queue of requests	88
$R :: r$	Adds a request r at the end of the pending requests	88
R		
$r :: R$	Takes the first request r at the beginning of the pending requests	88
$F :: \{f_i \mapsto \iota\}$	Adds a new future association to the future values	88

General Notation

$\{a_i\}$	List	66
$\{a \mapsto b\}$	Association/finite mapping	66
$\theta ::= \{\{b \leftarrow c\}\}$	Substitution	65
$\xrightarrow{*}$	Transitive closure of any reduction \rightarrow	103
\oplus	Disjoint union	105
$L _M$	Restriction of (RSL) list L to labels belonging to M	108
L_n	n^{th} element of the list L	108
\sqcup	Least upper bound	110
\exists^1	There is at most one	122

Stores

σ	Store: finite map from locations to objects (reduced or generalized reference) $\sigma ::= \{\iota_i \mapsto o_i\}$	66
$dom(\sigma)$	set of locations defined by σ	66
$\sigma :: \sigma'$	Append of disjoint stores	66
$\sigma + \sigma'$	Updates the values defined in σ' by those defined in σ : $(\sigma + \sigma')(\iota) = \begin{cases} \sigma(\iota) & \text{if } \iota \in dom(\sigma) \\ \sigma'(\iota) & \text{otherwise} \end{cases}$	66
$Merge(\iota, \sigma, \sigma')$	Store merge: merges σ and σ' independently except for ι which is taken from σ' : $Merge(\iota, \sigma, \sigma') = \sigma' \theta + \sigma$ where $\theta = \{\{\iota' \leftarrow \iota'' \mid \iota' \in dom(\sigma') \cap dom(\sigma) \setminus \{\iota\}, \iota'' \text{ fresh}\}\}$	90
$copy(\iota, \sigma)$	Deep copy of $\sigma(\iota)$	89
$Copy\&Merge(\sigma, \iota; \sigma', \iota')$	Appends in $\sigma'(\iota')$ a deep copy of $\sigma(\iota)$ $= Merge(\iota', \sigma', copy(\iota, \sigma) \{\{\iota \leftarrow \iota'\}\})$	91

Semantics

\mathcal{R}	Reduction context	67,89
$\mathcal{R}[a]$	Substitution inside a reduction context	67
\rightarrow_S	Sequential reduction	67
\longrightarrow	Parallel reduction	92
\xrightarrow{T}	Parallel reduction where rule T is applied	103
\Longrightarrow	Parallel reduction with future updates, i.e., Parallel reduction preceded by some reply rules	117 284
\xRightarrow{T}	Parallel Reduction with future updates where rule T is applied:	117
	$\xrightarrow{\text{REPLY}^*} \xrightarrow{T}$ if $T \neq \text{REPLY}$ and $\xrightarrow{\text{REPLY}^*}$ if $T = \text{REPLY}$	
$FL(\alpha)$	Future List of α	99
$RSL(\alpha)$	Request Sender List of α :	108
	$(RSL(\alpha))_n = \beta^f$ if $f_n^{\beta \rightarrow \alpha} \in FL(\alpha)$	
\trianglelefteq	RSL comparison: prefix order on sender activities	110
\mathcal{M}_{α_P}	Potential services:	103
	Static approximation of the set of M that can appear in the $Serve(M)$ instructions of α_P :	
	$P \xrightarrow{*} Q \wedge Q = \alpha[\mathcal{R}[Serve(M)], \dots] \parallel \dots$	
	$\Rightarrow \exists M' \in \mathcal{M}_{\alpha_P}, M \subseteq M'$	
$ActiveRefs(\alpha)$	Set of active objects referenced by α :	98
	$\{\beta \mid \exists \iota \in dom(\sigma_\alpha), \sigma_\alpha(\iota) = AO(\beta)\}$	
$FutureRefs(\alpha)$	Set of futures referenced by α :	98
	$\{f_i^{\beta \rightarrow \gamma} \mid \exists \iota \in dom(\sigma_\alpha), \sigma_\alpha(\iota) = fut(f_i^{\beta \rightarrow \gamma})\}$	
FF	Set of Forwarded Futures: $\{(f_i^{\alpha \rightarrow \beta}, \gamma, \delta)\} \in FF$ if $f_i^{\alpha \rightarrow \beta}$ has been transmitted from γ to δ	213

Equivalences

\equiv	Equality modulo renaming (alpha conversion) of locations and futures, and reordering of pending requests	112
\equiv_F	Equivalence modulo future updates also called equivalence modulo replies	113

Properties

$\vdash P \text{ ok}$	Well-formed configuration	99
$RSL(\alpha) \bowtie RSL(\beta)$	RSL compatibility	110
$P \bowtie Q$	Configuration compatibility	110
$P_1 \Downarrow P_2$	Configuration confluence: $\exists R_1, R_2, P_1 \xrightarrow{*} R_1 \wedge P_2 \xrightarrow{*} R_2 \wedge R_1 \equiv_F R_2$	118
$\mathcal{G}(P_0)$	Approximated call graph α can send a request <i>foo</i> to β implies $(\dot{\alpha}, \dot{\beta}, \text{foo}) \in \mathcal{G}(P_0)$	125
Request flow graph	$\alpha \rightarrow_R \beta$ if α has sent a request to β	126
$DON(P)$	Deterministic Object Network	122
$SDON(P)$	Static Deterministic Object Network	125
$TDON(P)$	Tree Deterministic Object Network	126

Syntax of ASP Calculus

Source terms

$a, b \in L ::= x$	variable
$ [l_i = b_i; m_j = \varsigma(x_j, y_j) a_j]_{j \in 1..m}^{i \in 1..n}$	object definition
$ a.l_i$	field access
$ a.l_i := b$	field update
$ a.m_j(b)$	method call
$ clone(a)$	superficial copy
$ Active(a, m_j)$	activates object: deep copy + activity creation m_j is the activity method or \emptyset for FIFO service
$ Serve(M)$	Serves a request among a set of method labels

where M is a set of method labels used to specify which request has to be served.

$$M = m_1, \dots, m_k$$

Intermediate Terms**Terms**

$a, b \in L' ::= x$	variable
$[l_i = b_i; m_j = \varsigma(x_j, y_j) a_j]_{j \in 1..m}^{i \in 1..n}$	object definition
$a.l_i$	field access
$a.l_i := b$	field update
$a.m_j(b)$	method call
$clone(a)$	superficial copy
$Active(a, m_j)$	object activation
$Serve(M)$	service primitive
ι	location
$a \uparrow f, b$	a with continuation b

Configurations

$$P, Q ::= \alpha[a; \sigma; \iota; F; R; f] \parallel \beta[\dots] \parallel \dots$$

Requests

$$R ::= \{[m_j; \iota; f_i^{\alpha \rightarrow \beta}]\}$$

Future Values

$$F ::= \{f_i^{\gamma \rightarrow \alpha} \mapsto \iota\}$$

Store

$$\sigma ::= \{l_i \mapsto o_i\}$$

$o ::= [l_i = \iota_i; m_j = \varsigma(x_j, y_j) a_j]_{j \in 1..m}^{i \in 1..n}$	reduced object
$ AO(\alpha)$	active object reference
$ fut(f_i^{\alpha \rightarrow \beta})$	future reference

Operational Semantics

STOREALLOC:	$\frac{\iota \notin \text{dom}(\sigma)}{(\mathcal{R}[o], \sigma) \rightarrow_S (\mathcal{R}[\iota], \{\iota \mapsto o\} :: \sigma)}$
FIELD:	$\frac{\sigma(\iota) = [l_i = \iota_i; m_j = \varsigma(x_j, y_j)a_j]_{j \in 1..m}^{i \in 1..n} \quad k \in 1..n}{(\mathcal{R}[\iota.l_k], \sigma) \rightarrow_S (\mathcal{R}[\iota_k], \sigma)}$
INVOKE:	$\frac{\sigma(\iota) = [l_i = \iota_i; m_j = \varsigma(x_j, y_j)a_j]_{j \in 1..m}^{i \in 1..n} \quad k \in 1..m}{(\mathcal{R}[\iota.m_k(\iota')], \sigma) \rightarrow_S (\mathcal{R}[a_k \{x_k \leftarrow \iota, y_k \leftarrow \iota'\}], \sigma)}$
UPDATE:	$\frac{\sigma(\iota) = [l_i = \iota_i; m_j = \varsigma(x_j, y_j)a_j]_{j \in 1..m}^{i \in 1..n} \quad k \in 1..n \quad o' = [l_i = \iota_i; l_k = \iota'_k; l_{k'} = \iota_{k'}; m_j = \varsigma(x_j, y_j)a_j]_{j \in 1..m}^{i \in 1..k-1, k' \in k+1..n}}{(\mathcal{R}[\iota.l_k := \iota'], \sigma) \rightarrow_S (\mathcal{R}[\iota], \{\iota \rightarrow o'\} + \sigma)}$
CLONE:	$\frac{\iota' \notin \text{dom}(\sigma)}{(\mathcal{R}[\text{clone}(\iota)], \sigma) \rightarrow_S (\mathcal{R}[\iota'], \{\iota' \mapsto \sigma(\iota)\} :: \sigma)}$

Table 1. Sequential reduction

$\iota \in \text{dom}(\text{copy}(\iota, \sigma))$ $\iota' \in \text{dom}(\text{copy}(\iota, \sigma)) \Rightarrow \text{locs}(\sigma(\iota')) \subseteq \text{dom}(\text{copy}(\iota, \sigma))$ $\iota' \in \text{dom}(\text{copy}(\iota, \sigma)) \Rightarrow \text{copy}(\iota, \sigma)(\iota') = \sigma(\iota')$

Table 2. Deep copy

<p>LOCAL:</p> $\frac{(a, \sigma) \rightarrow_S (a', \sigma') \quad \rightarrow_S \text{ does not clone a future}}{\alpha[a; \sigma; \iota; F; R; f] \parallel P \longrightarrow \alpha[a'; \sigma'; \iota; F; R; f] \parallel P}$
<p>NEWACT:</p> $\frac{\gamma \text{ fresh activity} \quad \iota' \notin \text{dom}(\sigma) \quad \sigma' = \{\iota' \mapsto \text{AO}(\gamma)\} :: \sigma}{\sigma_\gamma = \text{copy}(\iota', \sigma) \quad \text{Service} = (\text{if } m_j = \emptyset \text{ then } \text{FifoService} \text{ else } \iota'.m_j())}{\alpha[\mathcal{R}[\text{Active}(\iota', m_j)]; \sigma; \iota; F; R; f] \parallel P \longrightarrow \alpha[\mathcal{R}[\iota']; \sigma'; \iota; F; R; f] \parallel \gamma[\text{Service}; \sigma_\gamma; \iota''; \emptyset; \emptyset] \parallel P}$
<p>REQUEST:</p> $\frac{\sigma_\alpha(\iota) = \text{AO}(\beta) \quad \iota'' \notin \text{dom}(\sigma_\beta) \quad f_i^{\alpha \rightarrow \beta} \text{ new future} \quad \iota_f \notin \text{dom}(\sigma_\alpha)}{\sigma'_\beta = \text{Copy\&Merge}(\sigma_\alpha, \iota'; \sigma_\beta, \iota'') \quad \sigma'_\alpha = \{\iota_f \mapsto \text{fut}(f_i^{\alpha \rightarrow \beta})\} :: \sigma_\alpha}{\alpha[\mathcal{R}[\iota.m_j(\iota)]; \sigma_\alpha; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \parallel \beta[a_\beta; \sigma_\beta; \iota_\beta; F_\beta; R_\beta; f_\beta] \parallel P \longrightarrow \alpha[\mathcal{R}[\iota_f]; \sigma'_\alpha; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \parallel \beta[a_\beta; \sigma'_\beta; \iota_\beta; F_\beta; R_\beta :: [m_j; \iota''; f_i^{\alpha \rightarrow \beta}]; f_\beta] \parallel P}$
<p>SERVE:</p> $\frac{R = R' :: [m_j; \iota_r; f'] :: R'' \quad m_j \in M \quad \forall m \in M, m \notin R'}{\alpha[\mathcal{R}[\text{Serve}(M)]; \sigma; \iota; F; R; f] \parallel P \longrightarrow \alpha[\iota.m_j(\iota_r) \uparrow f, \mathcal{R}[\Box]; \sigma; \iota; F; R' :: R''; f'] \parallel P}$
<p>ENDSERVICE:</p> $\frac{\iota' \notin \text{dom}(\sigma) \quad F' = F :: \{f \mapsto \iota'\} \quad \sigma' = \text{Copy\&Merge}(\sigma, \iota; \sigma, \iota')}{\alpha[\iota \uparrow (f', a); \sigma; \iota; F; R; f] \parallel P \longrightarrow \alpha[a; \sigma'; \iota; F'; R; f'] \parallel P}$
<p>REPLY:</p> $\frac{\sigma_\alpha(\iota) = \text{fut}(f_i^{\gamma \rightarrow \beta}) \quad F_\beta(f_i^{\gamma \rightarrow \beta}) = \iota_f \quad \sigma'_\alpha = \text{Copy\&Merge}(\sigma_\beta, \iota_f; \sigma_\alpha, \iota)}{\alpha[a_\alpha; \sigma_\alpha; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \parallel \beta[a_\beta; \sigma_\beta; \iota_\beta; F_\beta; R_\beta; f_\beta] \parallel P \longrightarrow \alpha[a_\alpha; \sigma'_\alpha; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \parallel \beta[a_\beta; \sigma_\beta; \iota_\beta; F_\beta; R_\beta; f_\beta] \parallel P}$

Table 3. Parallel reduction (used or modified values are non-gray)

Overview of Properties

The objective of Fig. 2 is to show the dependencies between properties and definitions given in this book. This diagram is very informal and should help the reader to understand the main dependencies between ASP properties and definitions.

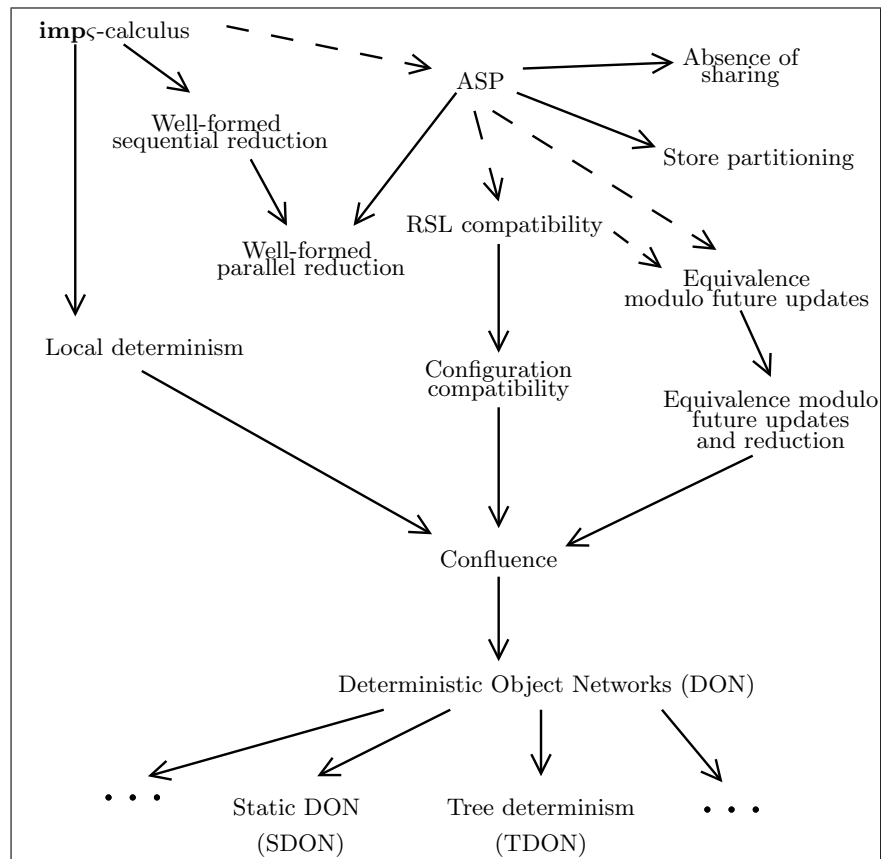


Fig. 2. Diagram of properties

The top left part of Fig. 2 shows properties and definitions related to imperative ς -calculus and which are *local* to an activity.

The *absence of sharing* and *store partitioning* properties are somewhat independent, even if in fact they have important consequences on all the other properties of ASP. These properties are used indirectly for proving all the other ones; for example, without store partitioning, a future value could be altered after the end of the corresponding service which would contradict with the properties of the equivalence modulo future updates.

Most of the properties shown in this book are related to *confluence* and its consequences. The principles of the confluence theorem can be summarized by: *concurrency can only originate from the application of two interfering REQUEST rules on the same destination activity*; for example, the order of updates of futures never has any influence on the reduction of a term. Moreover, an ASP execution is only characterized by the order of the request senders inside each activity.

The bottom part (last line) of the diagram shows the approximation of Deterministic Object Networks (DON) that can be performed. This book focused on two approximations: the static DON (SDON), and the deterministic behavior of programs communicating over a tree (TDON).

Overview of ASP Extensions

man dvips We present here most of the features that have been added to ASP in Part IV. We provide a brief summary, based on the syntax, and most of the reduction rules associated with these features. When several and somehow equivalent reduction rules exist for the same feature, we choose one of them.

Three Confluent Features:

1. Delegation

Delegates to another activity the responsibility to reply to the current request (confluent).

Syntax

$delegate(a)$

Reduction Rules

Parallel DELEGATE:

$$\frac{\sigma_\alpha(\iota) = AO(\beta) \quad \iota'' \notin dom(\sigma_\beta) \quad \sigma'_\beta = Copy\&Merge(\sigma_\alpha, \iota' ; \sigma_\beta, \iota'') \quad f_\emptyset \text{ new future}}{\alpha[\mathcal{R}[delegate(\iota.m_j(\iota'))]; \sigma_\alpha; \iota_\alpha; F_\alpha; R_\alpha; f_i^{\gamma \rightarrow \alpha}] \parallel \beta[a_\beta; \sigma_\beta; \iota_\beta; F_\beta; R_\beta; f_\beta] \parallel P \longrightarrow \alpha[\mathcal{R}[\emptyset]; \sigma_\alpha; \iota_\alpha; F_\alpha; R_\alpha; f_\emptyset] \parallel \beta[a_\beta; \sigma'_\beta; \iota_\beta; F_\beta; R_\beta :: [m_j; \iota''; f_i^{\gamma \rightarrow \alpha}]; f_\beta] \parallel P}$$

Sequential DELEGATE:

$$\frac{\sigma_\alpha(\iota) = [l_i = \iota_i; m_j = \zeta(x_j, y_j) a_j]_{j \in 1..m}^{i \in 1..n} \quad k \in 1..m}{\alpha[\mathcal{R}[delegate(\iota.m_j(\iota'))]; \sigma_\alpha; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \parallel P \longrightarrow \alpha[\mathcal{R}[a_k \{x_k \leftarrow \iota, y_k \leftarrow \iota'\}]; \sigma_\alpha; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \parallel P}$$

Generalized REPLY:

$$\frac{\sigma_\alpha(\iota) = fut(f_i^{\gamma \rightarrow \delta}) \quad F_\beta(f_i^{\gamma \rightarrow \delta}) = \iota_f \quad \sigma'_\alpha = Copy\&Merge(\sigma_\beta, \iota_f; \sigma_\alpha, \iota)}{\alpha[a_\alpha; \sigma_\alpha; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \parallel \beta[a_\beta; \sigma_\beta; \iota_\beta; F_\beta; R_\beta; f_\beta] \parallel P \longrightarrow \alpha[a_\alpha; \sigma'_\alpha; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \parallel \beta[a_\beta; \sigma_\beta; \iota_\beta; F_\beta; R_\beta; f_\beta] \parallel P}$$

2. Explicit Wait

Waits for a future update (confluent).

Syntax

$$waitFor(a)$$

Encoding

$$\begin{aligned} \llbracket [l_i = b_i; m_j = \varsigma(x_j, y_j)a_j]_{j \in 1..m}^{i \in 1..n} \rrbracket &\triangleq [wait = [], l_i = b_i; m_j = \varsigma(x_j, y_j)a_j]_{j \in 1..m}^{i \in 1..n} \\ \llbracket waitFor(a) \rrbracket &\triangleq a.wait \end{aligned}$$

3. Method Update

Changes the code associated to a method (confluent).

Syntax

$$x.foo \Leftarrow b$$

Five Non-confluent Features:

1. Testing Future Reception

Returns “true” if a future is awaited, and “false” if it has already been updated.

Syntax

$$\text{awaited}(a)$$

Reduction Rules

WAITT:

$$\frac{\sigma(\iota) = \text{fut}(f_i^{\alpha \rightarrow \beta})}{(\mathcal{R}[\text{awaited}(\iota)], \sigma) \rightarrow_S (\mathcal{R}[\text{true}], \sigma)}$$

WAITF:

$$\frac{\sigma(\iota) \neq \text{fut}(f_i^{\alpha \rightarrow \beta})}{(\mathcal{R}[\text{awaited}(\iota)], \sigma) \rightarrow_S (\mathcal{R}[\text{false}], \sigma)}$$

2. Non-blocking Service

Serves a request if it is in the request queue, else continues the execution.

Syntax

$$\text{ServeWithoutBlocking}(M)$$

Reduction Rules

SERVEWBSERVE:

$$\frac{R = R' :: [m_j; \iota_r; f'] :: R'' \quad m_j \in M \quad \forall m \in M, m \notin R'}{\alpha[\mathcal{R}[\text{ServeWithoutBlocking}(M)]; \sigma; \iota; F; R; f] \parallel P \longrightarrow \alpha[\iota.m_j(\iota_r) \uparrow f, \mathcal{R}[\Box]; \sigma; \iota; F; R' :: R''; f'] \parallel P}$$

SERVEWBCONTINUE:

$$\frac{\forall m \in M, m \notin R}{\alpha[\mathcal{R}[\text{ServeWithoutBlocking}(M)]; \sigma; \iota; F; R; f] \parallel P \longrightarrow \alpha[\mathcal{R}[\Box]; \sigma; \iota; F; R; f] \parallel P}$$

3. Testing Request Reception

Returns “true” if a corresponding request is in the request queue.

Syntax

$$inQueue(M)$$

Reduction Rules

INQUEUEET:

$$\frac{\exists m \in M, m \in R}{\alpha[\mathcal{R}[inQueue(M)]; \sigma; \iota; F; R; f] \parallel P \longrightarrow \alpha[\mathcal{R}[true]; \sigma; \iota; F; R; f] \parallel P}$$

INQUEUEEF:

$$\frac{\forall m \in M, m \notin R}{\alpha[\mathcal{R}[inQueue(M)]; \sigma; \iota; F; R; f] \parallel P \longrightarrow \alpha[\mathcal{R}[false]; \sigma; \iota; F; R; f] \parallel P}$$

4. Join Pattern Example

The term below encodes a join pattern cell: the cell reacts to the simultaneous presence of two messages, either s and set , or s and get . s is used to store the internal state of the cell.

Encoding a Join Pattern Cell

$$\begin{aligned} Cell \triangleq & Active([s_v = [], set_v = []]; \\ & set = \zeta(this, v) this.set_v := v \\ & s = \zeta(this, v) this.s_v := v \\ & get = \zeta(this) [] \\ & srv = \zeta(this) Repeat(if inQueue(s) \wedge inQueue(set) then \\ & \quad this.setcell() \\ & \quad if inQueue(s) \wedge inQueue(get) then \\ & \quad \quad this.getcell()), \\ & setcell() = \zeta(this)(Serve(set); Serve(s); this.Activity.s(set_v)), \\ & getcell() = \zeta(this)(Serve(get); Serve(s); this.Activity.s(s_v); s_v) \end{aligned}$$

Example of usage

$$Cell.s([]); Cell.set([x = 2]); Cell.get()$$

5. Extended Join Services

$Join((m_{11}, m_{12}, \dots, m_{1n_1}), (m_{21}, \dots, m_{2n_2}), \dots, (m_{k1}, \dots, m_{kn_k}))$

$Join((m_1, m_2), (m_1, m_3)) \triangleq$ *let* $served = false$ *in*
 Repeat
 if $(inQueue(m_1) \wedge inQueue(m_2))$ *then*
 $(Serve(m_1); Serve(m_2); served := true)$
 else if $(inQueue(m_1) \wedge inQueue(m_3))$ *then*
 $(Serve(m_1); Serve(m_3); served := true)$
 Until $(served = true)$

Migration

Simulates the migration: makes the current activity forward the requests to a newly created activity.

Syntax

$$\text{thisActivity.Migrate()}$$

Encoding

$$\begin{aligned} \text{Migrate} &\triangleq \varsigma(\text{this}) \text{let } \text{newao} = \text{Active}(\text{this}, \text{sevice}) \text{ in} \\ &\quad (\text{CreateForwarders}(\text{newao}); \text{FifoService}) \\ \text{CreateForwarders}(\text{newao}) &\triangleq \forall m_j, m_j \Leftarrow \varsigma(x, y) \text{newao}.m_j(y) \end{aligned}$$

Groups

Entity containing several objects that can be accessed as a single one.

Passive Groups

Syntax

$$\text{Group}(a_k^{k \in 1..l})$$

Reduction Rules

$$\mathcal{R} ::= \dots \mid \text{Group}(\iota_k, \mathcal{R}, b_{k'})^{k \in 1..m-1, k' \in m+1..l}$$

Store group:

$$\frac{\iota \notin \text{dom}(\sigma)}{(\text{Group}(\iota_k)^{k \in 1..l}, \sigma) \rightarrow_G (\iota, \{\iota \mapsto \text{Gr}(\iota_k)^{k \in 1..l}\} :: \sigma)}$$

Field access:

$$\frac{\sigma(\iota) = \text{Gr}(\iota_k)^{k \in 1..l}}{(\mathcal{R}[\iota.l_i], \sigma) \rightarrow_G (\text{Group}(\iota_k.l_i)^{k \in 1..l}, \sigma)}$$

Field update:

$$\frac{\sigma(\iota) = \text{Gr}(\iota_k)^{k \in 1..l}}{(\mathcal{R}[\iota.l_i := \iota'], \sigma) \rightarrow_G (\text{Group}(\iota_k.l_i := \iota')^{k \in 1..l}, \sigma)}$$

Invoke method:

$$\frac{\sigma(\iota) = Gr(\iota_k)^{k \in 1..l}}{(\mathcal{R}[\iota_k.m_j(\iota')], \sigma) \rightarrow_G (Group(\iota_k.m_j(\iota'))^{k \in 1..l}, \sigma)}$$

Active Groups

Syntax

$$ActiveGroup(a_1, \dots, a_n, m)$$

Encoding

$$ActiveGroup(a_1, \dots, a_n, m) \triangleq Group(Active(a_1, m), \dots, Active(a_n, m))$$

Components

Primitive Component

A *primitive component* is defined from an activity α , a set of *server interfaces* (SI, a subset of the served methods), and a set of *client interfaces* (CI, references to other activities contained in fields):

$$SI_i \subseteq \bigcup_{M \in \mathcal{M}_{\alpha P_0}} M$$

$$PC ::= C_n < a, srv, \{SI_i\}^{i \in 1..k}, \{CI_j\}^{j \in 1..l} >$$

Composite Component

A composite component is a set of components (either primitive (PC) or composite (CC)) exporting some server interfaces (some SI_i), some client interfaces (some CI_j), and connecting some client and server interfaces (defining a partial binding (CI_i, SI_j)). Such a component is given a name C_n . CC is a composite component and C either a primitive or a composite one:

$$CC ::= C_n \ll C_1, \dots, C_m; \{(C_{i_p}.CI_{j_p}, C_{i'_p}.SI_{j'_p})\}^{p \in 1..k}; \\ \{C_{i_q}.CI_{j_q} \rightarrow CI_q\}^{q \in 1..l}; \{C_{i_r}.SI_{j_r} \rightarrow SI_r\}^{r \in 1..l'} \gg$$

$$C ::= PC | CC$$

where each C_i is the name of one included component C_i ($i \in 1..m$), supposed to be pairwise distinct; each exported SI is only bound once to an included component, and each internal client interface $(C_i.CI_j)$ appears at most one time:

$$\forall p, p' \in 1..k, \forall q, q' \in 1..l, \forall r, r' \in 1..l' \begin{cases} p \neq p' \Rightarrow C_{i_p}.CI_{j_p} \neq C_{i_{p'}}.CI_{j_{p'}} \\ q \neq q' \Rightarrow C_{i_q}.CI_{j_q} \neq C_{i_{q'}}.CI_{j_{q'}} \\ C_{i_p}.CI_{j_p} \neq C_{i_q}.CI_{j_q} \\ r \neq r' \Rightarrow SI_r \neq SI_{r'} \end{cases}$$

Deterministic Primitive Component (DPC)

A DPC is a primitive component defined from an activity α , such that server interfaces SI are disjoint subsets of the served method of the active object of α such that every $M \in \mathcal{M}_{\alpha P_0}$ is included in a single SI_i :

$$\begin{cases} \forall i, k, i \neq k \Rightarrow SI_i \cap SI_k = \emptyset \\ \forall M \in \mathcal{M}_{\alpha P_0}, \forall M_1 \subseteq M, \forall M_2 \subseteq M (M_1 \subseteq SI_i \wedge M_2 \subseteq SI_j) \Rightarrow i = j \end{cases}$$

Deterministic Composite Component (DCC)

A DCC is

- either a DPC,
- or a composite component connecting some DCCs such that the binding between server and client interfaces is one to one. More precisely the following constraints must be added to the ones of Definition 14.2:

$$\left\{ \begin{array}{l} \text{Each } C_i \text{ is a DCC} \\ \forall p, p' \in 1..k, \forall q, q' \in 1..l, \forall r, r' \in 1..l' \end{array} \right. \left\{ \begin{array}{l} p \neq p' \Rightarrow C_{i'_p} \cdot SI_{j'_p} \neq C_{i'_{p'}} \cdot SI_{j'_{p'}} \\ r \neq r' \Rightarrow C_{i_r} \cdot SI_{j_r} \neq C_{i_{r'}} \cdot SI_{j_{r'}} \\ C_{i'_p} \cdot SI_{j'_p} \neq C_{i_r} \cdot SI_{j_r} \\ q \neq q' \Rightarrow CI_q \neq CI_{q'} \end{array} \right.$$

Index

Symbols

π -calculus 26
 $\pi o\beta\lambda$ 51
 $\pi o\beta\lambda$ 251
conc ζ -calculus 54, 250
 ζ -calculus 31
 ζ -calculus XXIX, 249
imp ζ -calculus 63
Øjeblik 49, 250
 π -calculus XXIX, 246
PICT 37
PICT XXXI

A

ABCL 14, 45
active object XXVI, 5, 15, 69, 71, 73
activity 6, 69, 70, 72, 87
Actor Foundry library 25
actors 5, 15, 23, 245
Ada 45
adaptive 266
adaptive implementation 259
AIC 190
alias condition 114, 273
ambient calculus 40, 250
approximated call graph 125
ASP XXIX, 18, 69
asynchronous 28
asynchronous systems 4
atomicity 163
authentication 190

B

balloon types 258

barrier 149, 211
batched futures 261
binary tree 76, 198

C

cactus stack 205
CADP 13
call delegation 48
call-by-copy-restore 7
call-by-move 7
call-by-name 7
call-by-object-reference 7
call-by-reference 7
call-by-sharing 7
call-by-visit 7
capabilities 25
centralized synchronization 46
channel 123
channels in ASP 181, 247
Cilk 204
CML 43
cncurrency 3
cncurrent C 10
communication base 7
communication passing 8
communication timing 9, 225
compatibility 107
compatible configurations XXI, 110
components 44, 169
composite and parallel component 176
composite component 170
computational mobility 192
concurrency 298
concurrent request sending 121, 304

condition variable 46
 confidentiality 190
 configuration 66, 68, 88, 99
 conflicting requests 258
 confluence XXVI, 118, 295
 continuation 88
 copy and merge 91
 copy-restore parameter passing 7
 CSP XXIX, 45
 current future 88, 96
 current request 69
 current term 71, 87

D

DAG 186
 data-driven synchronization 69
 deep copy 89
 delegation 137
 demand driven evaluation 262
 determinism XXVI, 126
 Deterministic Object Networks (DON)
 122
 Directed Acyclic Graphs 186
 disconnected mode 262
 Distribution 4

E

E actor language 25
 eager future update 223
 Eiffel 198
 ELECTRE 13
 Emerald 7
 equivalence modulo future updates
 111, 270, 288
 equivalence modulo replies 111, 270
 Eratosthenes 77, 201
 explicit message acceptance 47
 extended join services 149, 211

F

Fibonacci numbers 80, 206
 FIFO service 72, 130
 form analysis 258
 forwarder 152, 194
 free variables 65
 functional nets 11
 future 11, 36, 70, 73, 88, 94
 future list 99
 future update 70, 73, 213, 262

future values 70, 71, 74, 88

G

garbage collection 241
 generalized reference 88
 generalized reply 139, 216
 global localization 257
 Gordon-Hankin Concur. Calculus 54
 Grid 259
 group atomicity 157
 group proxy 158
 groups 157, 158, 184

H

hard-reactivity 12

I

I-structures 256
 IC2D 210
 implicit message acceptance 47
 initial configuration 97
 integrity 190
 interactivity vs. reactivity 12
 interfering requests 103, 107, 122, 257
 isolation 105

J

Java Messaging System 195
 JMS 195
 join 211
 join pattern 146
 join primitive 11
 join-calculus 11, 42, 250

K

kell-calculus 44

L

lambda expressions 64
 lazy copy 8
 lazy future update 223
 lazy parameter passing 8
 lazy pass-by-value 8
 Lazy Task Creation (LTC) 204
 lazy threads 204
 let 64
 like Current 198
 linearized channels 29
 load-time MOP 189
 local confluence 298

- localization server 194
 - location 63, 65, 66
 - LOTOS 13
- M**
- Meta-Object Protocol (MOP) 189
 - method arguments 63
 - method parameter 63
 - method update 63, 141
 - migration 49, 151, 184, 192
 - MIMD 178
 - mobility 8, 192
 - MultiLisp 249
 - Multilisp 35
 - mutual exclusion 46
 - My Type 198
- N**
- node 191
 - non-blocking services 10, 144
- O**
- object activity 15
 - object RMI 14
 - object topology 258
 - Obliq 49, 250
 - one-sided communications 9
 - out-of-order future updates 178
 - ownership analysis 258
 - ownership types 258
- P**
- parallel activity 45
 - parallel component 176
 - parallel configuration 71
 - parallel reduction 89, 92
 - parallelism 3
 - partial future value 74
 - pass-by-reference 7
 - pass-by-value 7
 - passive object 69
 - path 113
 - pending requests 69, 71, 73, 88, 94
 - pending-queue inspection 10
 - PKI 190
 - PolyTOIL 198
 - potential services 103
 - primitive component 169
 - ProActive XXVII, XXXI, 189
 - process mobility 8
 - Process Networks 30, 185, 248
 - protected 51
 - proxy 73
- Q**
- quasi-parallel activity 45
- R**
- reactive 11, 12
 - reactive (a)synchronous systems 12
 - reconfiguration 181
 - reduced object 66
 - reduction context 67
 - renaming 65, 269
 - rendezvous 73, 95, 225
 - Repeat 89
 - Repeat Until 89
 - reply 96
 - request 69, 73, 88, 93
 - request flow graph 126
 - Request Sender List (RSL) 108
 - restriction 108
 - routing protocol 194
 - runtime MOP 189
- S**
- scatter group 198
 - select primitive 10
 - self 63
 - sequential activity 45
 - sequential degradation 205
 - sequential reduction 67, 329
 - serialized 250
 - served requests 70
 - service method 72, 93
 - service primitive 71, 95, 130
 - sharing 6, 104
 - sieve of Eratosthenes 77, 201
 - Sisal 11
 - soft reactivity 12
 - source term 65
 - SPMD 178
 - static 34
 - static analysis 258
 - Static Deterministic Object Network (SDON) 124, 125
 - store 66, 71, 88
 - store append 66
 - store update 66

store-passing style 23
stores merge 90
strict operation 73
substitution 65
synchronization 11
synchronous 28
synchrony hypothesis 12

T

target method 88
Thal 23
Topology with Cycles 186
Tree DON (TDON) 126
triangle pattern 231
TTL-TTU localization protocol 194

TTL: Time To Live 194
TTS: Time To Send 262
TTU: Time To Update 194
two-sided communications 9

V

virtual node 191

W

wait-by-necessity XXVI, 16, 70, 73, 96
weak migration 192
well-formedness 68, 99, 139

Z

zero-copy 10