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: $\alpha[\mathcal{R}[\iota\ldots],\sigma_\alpha\ldots] \wedge \sigma_\alpha(\iota) = \mathit{fut}(f_i^{\gamma\to\beta})$	70
Service method	Method started upon activation: m_j in $\mathit{Active}(a, m_j)$	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 $f_i^{\alpha\to\beta}$ $\mathit{copy}(\iota,\sigma)$ where $\{f_i^{\alpha\to\beta} \mapsto \iota\} \in F_\alpha$	74
Computed future	A future which has a value associated: $f_i^{\alpha\to\beta}$ where $f_i^{\alpha\to\beta} \in \mathit{dom}(F_\beta)$	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: $fv(a) = \emptyset \wedge \mathit{locs}(a) = \emptyset$	65
Reduced object	Object with all fields reduced to a location: $o ::= [l_i = \iota_i; m_j = \varsigma(x_j, y_j)a_j]^{i\in 1..n}_{j\in 1..m}$	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: $[m_1;\iota;f_i^{\alpha\to\beta}]$ and $[m_2;\iota;f_i^{\gamma\to\beta}]$ such that there is $M, \{m_1, m_2\} \subseteq M$ and $\mathit{Serve}(M)$ can appear in β	107

Cycle of futures	set of future identifiers $\{fut(f_i^{\gamma_i \rightarrow \beta_n})\}$ such that $\beta_0 \ldots \beta_n$ verify: $\begin{cases} \forall i,\ 0 < i \leq n, fut(f_{i-1}^{\gamma_{i-1} \rightarrow \beta_{i-1}}) \in \\ \qquad copy(F_{\beta_i}(fut(f_i^{\gamma_i \rightarrow \beta_i})), \sigma_{\beta_i}) \\ fut(f_n^{\gamma_n \rightarrow \beta_n}) \in copy(F_{\beta_0}(fut(f_0^{\gamma_0 \rightarrow \beta_0})), \sigma_{\beta_0}) \end{cases}$	116

Core Syntax: ASP Source Terms

$[l_i = b_i; m_j = \varsigma(x_j, y_j)a_j]_{j \in 1..m}^{i \in 1..n}$	Object definition	64
$a.l_i$	Field access	64
$a.l_i := b$	Field update	64
$a.m_j(b)$	Method call	64
$clone(a)$	Superficial copy	64
$Active(a, m_j)$	Object activation	71
$Serve(M)$	Request service	71
M	list of method labels	71

Encoding of Classical Primitives

$let\ x = a\ in\ b$	$[; m = \varsigma(z, x)b].m(a)$	64
$a; b$	$[; m = \varsigma(z, z')b].m(a)$	64
$\lambda x.b$	$[arg = [], val = \varsigma(x)b\{\!\{x \leftarrow x.arg\}\!\}]$	64
$(b\ a)$	$(clone(b).arg := a).val()$	64
$Repeat(a)$	$[; repeat = \varsigma(x)a; x.repeat()].repeat()$	89
$FifoService$	$Repeat(Serve(\mathcal{M}))$	89
$Repeat\ a\ Until\ b$	$[; rep = \varsigma(x)a;\ if\ (not(b))\ then\ x.rep()].rep()$	89

ASP Intermediate Terms and Semantic Structures

(a, σ)	Sequential configuration	66
α, β	Activities: $\alpha[a_\alpha; \sigma_\alpha; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha]$	69
	[current term, store, active object, future values, pending requests, current future]	87
ι	Locations	64
$locs(a)$	Set of locations occurring in a	65
P, Q	Parallel configuration	88
$\alpha \in P$	Activity α belongs to the configuration P	98
$a \Uparrow f, b$	a with continuation b	71
	f is the future associated with the continuation	
$AO(\alpha)$	Generalized reference to the activity α	73
$f_i^{\alpha \rightarrow \beta}$	Future identifier	87
$fut(f_i^{\alpha \rightarrow \beta})$	Future reference	88

$r = [m_j; \iota; f_i^{\alpha\rightarrow\beta}]$	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 R	88
$r :: R$	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
$\overset{*}{\rightarrow}$	Transitive closure of any reduction $\rightarrow$	103
$\bigoplus$	Disjoint union	105
$L\big\vert_{M}$	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 $\theta = \{\!\{\iota' \leftarrow \iota'' \mid \iota' \in dom(\sigma') \cap dom(\sigma) \backslash \{\iota\}, \iota'' \text{ fresh}\}\!\}$	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
$\overset{T}{\Longrightarrow}$	Parallel Reduction with future updates where rule T is applied: $\xrightarrow{\text{REPLY}*}\xrightarrow{T}$ if $T \neq$ REPLY and $\xrightarrow{\text{REPLY}*}$ if $T =$ REPLY	117
$FL(\alpha)$	Future List of α	99
$RSL(\alpha)$	Request Sender List of α: $(RSL(\alpha))_n = \beta^f$ if $f_n^{\beta \to \alpha} \in FL(\alpha)$	108
$\trianglelefteq$	RSL comparison: prefix order on sender activities	110
$\mathcal{M}_{\alpha_P}$	Potential services: Static approximation of the set of M that can appear in the $Serve(M)$ instructions of α_P: $P \xrightarrow{*} Q \wedge Q = \alpha[\mathcal{R}[Serve(M)], \ldots] \parallel \ldots$ $\Rightarrow \exists M' \in \mathcal{M}_{\alpha_P},\, M \subseteq M'$	103
$ActiveRefs(\alpha)$	Set of active objects referenced by α: $\{\beta \mid \exists \iota \in dom(\sigma_\alpha),\, \sigma_\alpha(\iota) = AO(\beta)\}$	98
$FutureRefs(\alpha)$	Set of futures referenced by α: $\{f_i^{\beta \to \gamma} \mid \exists \iota \in dom(\sigma_\alpha),\, \sigma_\alpha(\iota) = fut(f_i^{\beta \to \gamma})\}$	98
FF	Set of Forwarded Futures: $\{(f_i^{\alpha \to \beta},\, \gamma,\, \delta)\} \in FF$ if $f_i^{\alpha \to \beta}$ 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

$\vdash P\ \mathrm{OK}$	Well-formed configuration	99
$RSL(\alpha) \bowtie RSL(\beta)$	RSL compatibility	110
$P \bowtie Q$	Configuration compatibility	110
$P_1 \curlyvee P_2$	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 foo to β implies $(\dot{\alpha}, \dot{\beta}, foo) \in \mathcal{G}(P_0)$	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
$\mid [l_i = b_i; m_j = \varsigma(x_j, y_j)a_j]^{i \in 1..n}_{j \in 1..m}$	object definition
$\mid a.l_i$	field access
$\mid a.l_i := b$	field update
$\mid a.m_j(b)$	method call
$\mid clone(a)$	superficial copy
$\mid Active(a, m_j)$	activates object: deep copy + activity creation m_j is the activity method or $\emptyset$ for FIFO service
$\mid Serve(M)$	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.

$$M = m_1, \ldots, m_k$$

Intermediate Terms

Terms

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

Configurations

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

Requests

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

Future Values

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

Store

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

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

Operational Semantics

STOREALLOC:

$$\frac{\iota \notin dom(\sigma)}{(\mathcal{R}[o], \sigma) \rightarrow_S (\mathcal{R}[\iota], \{\iota \mapsto o\} :: \sigma)}$$

FIELD:

$$\frac{\sigma(\iota) = [l_i = \iota_i; m_j = \varsigma(x_j, y_j)a_j]^{i \in 1..n}_{j \in 1..m} \quad k \in 1..n}{(\mathcal{R}[\iota.l_k], \sigma) \rightarrow_S (\mathcal{R}[\iota_k], \sigma)}$$

INVOKE:

$$\frac{\sigma(\iota) = [l_i = \iota_i; m_j = \varsigma(x_j, y_j)a_j]^{i \in 1..n}_{j \in 1..m} \quad k \in 1..m}{(\mathcal{R}[\iota.m_k(\iota')], \sigma) \rightarrow_S (\mathcal{R}[a_k\{\!\{x_k \leftarrow \iota, y_k \leftarrow \iota'\}\!\}], \sigma)}$$

UPDATE:

$$\frac{\begin{array}{c}\sigma(\iota) = [l_i = \iota_i; m_j = \varsigma(x_j, y_j)a_j]^{i \in 1..n}_{j \in 1..m} \quad k \in 1..n \\ o' = [l_i = \iota_i; l_k = \iota'; l_{k'} = \iota_{k'}; m_j = \varsigma(x_j, y_j)a_j]^{i \in 1..k-1,\, k' \in k+1...n}_{j \in 1..m}\end{array}}{(\mathcal{R}[\iota.l_k := \iota'], \sigma) \rightarrow_S (\mathcal{R}[\iota], \{\iota \rightarrow o'\} + \sigma)}$$

CLONE:

$$\frac{\iota' \notin dom(\sigma)}{(\mathcal{R}[clone(\iota)], \sigma) \rightarrow_S (\mathcal{R}[\iota'], \{\iota' \mapsto \sigma(\iota)\} :: \sigma)}$$

Table 1. Sequential reduction

$$\iota \in dom(copy(\iota,\sigma))$$

$$\iota' \in dom(copy(\iota,\sigma)) \Rightarrow locs(\sigma(\iota')) \subseteq dom(copy(\iota,\sigma))$$

$$\iota' \in dom(copy(\iota,\sigma)) \Rightarrow copy(\iota,\sigma)(\iota') = \sigma(\iota')$$

Table 2. Deep copy

LOCAL:

$$\frac{(a,\sigma) \to_S (a',\sigma') \qquad \to_S \text{ does not clone a future}}{\alpha[a;\sigma;\iota;F;R;f] \parallel P \longrightarrow \alpha[a';\sigma';\iota;F;R;f] \parallel P}$$

NEWACT:

$$\frac{\begin{array}{c}\gamma \text{ fresh activity} \quad \iota' \notin dom(\sigma) \quad \sigma' = \{\iota' \mapsto AO(\gamma)\} :: \sigma \\ \sigma_\gamma = copy(\iota'',\sigma) \quad Service = (\text{if } m_j = \emptyset \text{ then } FifoService \text{ else } \iota''.m_j())\end{array}}{\begin{array}{c}\alpha[\mathcal{R}[Active(\iota'',m_j)];\sigma;\iota;F;R;f] \parallel P \longrightarrow \\ \alpha[\mathcal{R}[\iota'];\sigma';\iota;F;R;f] \parallel \gamma[Service;\sigma_\gamma;\iota'';\emptyset;\emptyset;\emptyset] \parallel P\end{array}}$$

REQUEST:

$$\frac{\begin{array}{c}\sigma_\alpha(\iota) = AO(\beta) \quad \iota'' \notin dom(\sigma_\beta) \quad f_i^{\alpha\to\beta} \text{ new future} \quad \iota_f \notin dom(\sigma_\alpha) \\ \sigma'_\beta = Copy\&Merge(\sigma_\alpha,\iota' \ ; \ \sigma_\beta,\iota'') \quad \sigma'_\alpha = \{\iota_f \mapsto fut(f_i^{\alpha\to\beta})\} :: \sigma_\alpha\end{array}}{\begin{array}{c}\alpha[\mathcal{R}[\iota.m_j(\iota')];\sigma_\alpha;\iota_\alpha;F_\alpha;R_\alpha;f_\alpha] \parallel \beta[a_\beta;\sigma_\beta;\iota_\beta;F_\beta;R_\beta;f_\beta] \parallel P \longrightarrow \\ \alpha[\mathcal{R}[\iota_f];\sigma'_\alpha;\iota_\alpha;F_\alpha;R_\alpha;f_\alpha] \parallel \beta[a_\beta;\sigma'_\beta;\iota_\beta;F_\beta;R_\beta :: [m_j;\iota'';f_i^{\alpha\to\beta}];f_\beta] \parallel P\end{array}}$$

SERVE:

$$\frac{R = R' :: [m_j;\iota_r;f'] :: R'' \quad m_j \in M \quad \forall m \in M,\ m \notin R'}{\begin{array}{c}\alpha[\mathcal{R}[Serve(M)];\sigma;\iota;F;R;f] \parallel P \longrightarrow \\ \alpha[\iota.m_j(\iota_r) \Uparrow f, \mathcal{R}[[]];\sigma;\iota;F;R' :: R'';f'] \parallel P\end{array}}$$

ENDSERVICE:

$$\frac{\iota' \notin dom(\sigma) \quad F' = F :: \{f \mapsto \iota'\} \quad \sigma' = Copy\&Merge(\sigma,\iota \ ; \ \sigma,\iota')}{\alpha[\iota \Uparrow (f',a);\sigma;\iota;F;R;f] \parallel P \longrightarrow \alpha[a;\sigma';\iota;F';R;f'] \parallel P}$$

REPLY:

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

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.

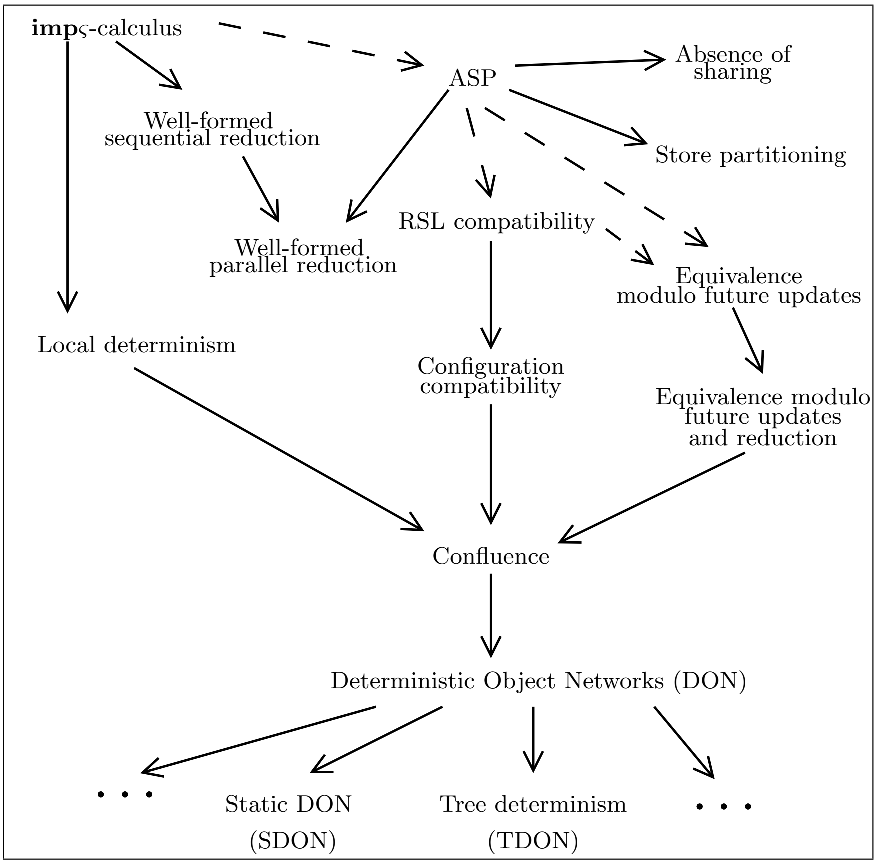

Fig. 2. Diagram of properties

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:

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

Sequential DELEGATE:

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

Generalized REPLY:

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

2. Explicit Wait

Waits for a future update (confluent).

Syntax

$$waitFor(a)$$

Encoding

$$[\![[l_i = b_i; m_j = \varsigma(x_j, y_j)a_j]^{i\in 1..n}_{j\in 1..m}]\!] \triangleq [wait = [], l_i = b_i; m_j = \varsigma(x_j, y_j)a_j]^{i\in 1..n}_{j\in 1..m}$$

$$[\![waitFor(a)]\!] \triangleq a.wait$$

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

$$awaited(a)$$

Reduction Rules

waitT:

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

waitF:

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

2. Non-blocking Service

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

Syntax

$$ServeWithoutBlocking(M)$$

Reduction Rules

ServeWBServe:

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

ServeWBContinue:

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

3. Testing Request Reception

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

Syntax

$$inQueue(M)$$

Reduction Rules

inQueueT:

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

inQueueF:

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

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

$$\begin{array}{l}
Cell \triangleq Active([s_v = [],\ set_v = [];\\
\qquad\qquad set = \varsigma(this, v) this.set_v := v\\
\qquad\qquad s = \varsigma(this, v) this.s_v := v\\
\qquad\qquad get = \varsigma(this)[]\\
\qquad\qquad srv = \varsigma(this) Repeat(if\ inQueue(s) \wedge inQueue(set)\ then\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad this.setcell()\\
\qquad\qquad\qquad\qquad\qquad\quad if\ inQueue(s) \wedge inQueue(get)\ then\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad this.getcell()),\\
\qquad\qquad setcell() = \varsigma(this)(Serve(set); Serve(s); thisActivity.s(set_v)),\\
\qquad\qquad getcell() = \varsigma(this)(Serve(get); Serve(s); thisActivity.s(s_v); s_v)
\end{array}$$

Example of usage

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

5. Extended Join Services

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

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

Migration

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

Syntax

$$thisActivity.Migrate()$$

Encoding

$$\begin{aligned} Migrate \triangleq \varsigma(this)\, let\ newao = Active(this, sevice)\ in \\ (CreateForwarders(newao); FifoService) \end{aligned}$$

$$CreateForwarders(newao) \triangleq \forall m_j,\ m_j \Leftarrow \varsigma(x,y) newao.m_j(y)$$

Groups

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

Passive Groups

Syntax

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

Reduction Rules

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

Store group:

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

Field access:

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

Field update:

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

Invoke method:

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

Active Groups

Syntax

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

Encoding

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

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:

$$\begin{aligned} CC ::= C_n \ll\ & C_1, \ldots, C_m; \{(C_{i_p}.CI_{j_p}, C_{i'_p}.SI_{j'_p})\}^{p \in 1..k}; \\ & \{C_{i_q}.CI_{j_q} \to CI_q\}^{q \in 1..l}; \{C_{i_r}.SI_{j_r} \to SI_r\}^{r \in 1..l'} \gg \\ C ::= PC|CC & \end{aligned}$$

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:

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

Index