School of Informatics, University of Edinburgh

The Laboratory for Foundations of Computer Science

Distributed Programming with Types and OTP Design Principles (full version)

by

Jiansen HE

Supervised by
Professor Philip Wadler
Professor Donald Sannella

Informatics Thesis Proposal

School of Informatics **January 2012**
http://www.informatics.ed.ac.uk/

Distributed Programming with Types and OTP Design Principles (full version)

Jiansen HE

Supervised by
Professor Philip Wadler
Professor Donald Sannella

SCHOOL *of* INFORMATICS
The Laboratory for Foundations of Computer Science

January 2012

Contents

1 Project Description

1.1 Background and Motivation

Concurrent programming is facing new challenges due to the recent advent of multi-core processors, which rely on parallelism, and the pervasive demands of web applications, which rely on distribution. Theoretic works on concurrency can be traced back to Petri net and other approaches invented in 1960s [4]. Since 1980, more attention has been paid to the study of process calculi (a.k.a. process algebras), which provide a family of formal models for describing and reasoning about concurrent systems [2]. The proliferation of Process Calculi marks the advance in concurrency theory. "The Dreams of Final Theories" [2] for concurrency, however, have not been achieved.

In practice, it is difficult to build a reliable distributed system because of its complexity. Firstly, a distributed system shares features and problems with other concurrent systems. For instance, it is non-deterministic so that traditional input-output semantics is inadequate to be used for reasoning about its behaviour. Secondly, scalability of a distributed system is usually a challenge for its developers. For example, developers of a distributed system are unlikely to know the structure of the system in advance. Moreover, the mobility [9] of a distributed system requires the capability of changing its topological structure on the fly. In addition, the latency of communication could be affected by many unpredictable factors. Thirdly, the system should be tolerant of partial failures. For example, when a site fails to respond to messages, the system should address this failure by invoking a recovery mechanism.

Fortunately, some recent works provide a solid basis for the design of distributed programming languages. For example, the ambient calculus [9] models the movement of processes and devices. Bigraphs [26], a more abstract model, is aiming at describing spatial aspects and mobility of ubiquitous computing. Session types [20, 21] guarantees that two parties of a session always communicate in a dual pattern. Lastly, the join-calculus [15] is a remarkable calculus which models features of distributed programming as well as provides a convenient construct, the join patterns, for resources synchronisation. It is also important to note that the syntax of the join-calculus is close to a real programming language and therefore reduces the barrier between understanding the mathematical model and employing this abstract model to guide programming practices.

In addition, good programming principles, such as the OTP design principles[1], have been developed and verified by the long-term implementation practice. OTP is a platform for developing concurrent, distributed, fault tolerant, and non-stop Erlang applications [3]. In the past 20 years, the Erlang language was used to build large reliable applications like RabbitMQ, Twitterfall, and many telecommunications systems.

Nevertheless, Erlang is an untyped functional programming language whereas a typed language detects certain forms of data misuse earlier. Therefore, it would be pleasant to combine advantages of static typing and OTP principles. At the time of this writing, this idea has been partly realised in the akka framework [23]. In spite of some attempts, the akka framework inherits some limitations of using untyped actors from the Erlang programming language. This situation left us research potentials in this line.

1.2 Project Aim

The aim of this project is to *build a library that supports OTP design principles in a typed setting.* The project intends to explore factors that encourage programmers to employ good programming principles. This project will demonstrate how types and appropriate models help the construction of reliable distributed systems.

[1] OTP is stand for Open Telecom Platform. It provides a set of libraries for developing Erlang applications. For the above reason, it is also known as Erlang/OTP design principles

1.3 Project Objectives

1. To identify a number of medium-sized applications implemented in Erlang using OTP principles. Those examples will be served as references to evaluate frameworks that support OTP principles. Candidate examples shall cover a wide range of aspects in distributed programming, including but not limited to (a) transmitting messages and potentially computation closures, (b) modular composition of distributed components, (c) default and extensible failure recovery mechanism, (d) eventually consistency model for shared states, and (e) dynamic topology configuration and hot code swapping.

2. To re-implement and compare identified applications within existing frameworks. The primitives purpose of this work is to be aware of prominent features and limitations of existing frameworks. As a research which contains certain level of overlaps with other existing and evolving frameworks, this research should distinguish itself from others by devised solutions to some problems which are difficulty to be solved by alternatives. Moreover, the evaluation component formalised in this work is likely to be novel for comparing frameworks aiming at supporting distributed computation. The same evaluation component will be used for evaluating our later developed library as well.

3. To implement a library that supports OTP design principles in a typed setting. The library could be built either on top of existing frameworks or with lower level constructs. It important to note that the library might be based on an abstraction that is more general than the actor model. Although OTP design principles were proposed for the Erlang language, which is based on the Actor model, we believe that it should be possible to rephrase those good programming principles in other models. To demonstrate the wide applications of OTP design principles, the library interface should provide a certain level of generality so that it could be implemented by and used within different models.

4. To demonstrate how the proposed library meets the demands of distributed programming. At this stage, the proposed library will be used to (a) demonstrate how to write small programs aiming at different requirements; (b) demonstrate its scalability of programming in the large by reimplementing applications used in objective 1 and 2.

5. To summarise some challenges in distributed programming and how to get around those difficulties with the proposed library and other programming frameworks. The summary will reveal how language libraries, which is used for practical purposes, are related to their theoretical guidance underneath. It may also plot research blanks for future study.

2 Literature Review

Process calculi (or process algebras) is a family of algebraic approaches to study concurrent systems [4]. A process calculus need to address following questions: 1. What are primitive components in a concurrent system? 2. What kind of computations and process behaviours could be modelled in this calculus? 3. What does equivalence between two processes mean in this calculus? In the past 30 years, a number of process calculi have been proposed and studied. Each calculus answers the first two questions from slightly different perspectives. For the third question, it is worth to note that a particular calculus may have more than one interpretation for process equivalences on different levels.

This chapter will present selective results in concurrent theories and their implementations. §2.1 will use the π-calculus as an example to sketch research topics in the area of process calculi, which provides formal basis for the design of concurrent languages. §2.2 and §2.3 will focus on the programmability of the join-calculus and the ambient-calculus. Both sections will introduce how to the subject calculus addresses problems in distributed programming and how ideas in the subject calculus are applied in real programming languages. Algebraic properties of the subject calculus, however, will not be investigated in depth in this short review. Then, §2.4 will introduce OTP design principles which make distributed Erlang programs more reliable and maintainable. §2.5 will present indirect and direct encodings for the λ-calculus in process calculi. Lastly, §2.6 will give a short survey on a general framework for implementing concurrent programming languages.

2.1 The π-calculus

2.1.1 Syntax and Notations

The π-calculus [27, 33] is a name passing calculus, in the sense that computations are represented by passing *names* through *channels*. The π-calculus uses lower-case letter to denote a name, which could be either a channel or a variable.

Syntax of the π-calculus given in Table 1 also denotes part of its semantics. The prefix of a process denotes its capability, which could be (i) receiving names from a channel; (ii) sending names through a channel; or (iii) performing an internal action that cannot be observed from outside. Informally, a process could : (1) evolve to another process after performing an action; (2) evolve as two processes who are running at the same time; (3) evolve as one of the two processes; (4) evolve to another process when two names become equal; (5) a process with a private name; (6) infinite parallel composition of the same process; or (7) an inert process that does nothing. Finally, two processes are structurally congruent, $\equiv$, if they are equal up to structure rules listed in Table 2

In the rest of this section, we distinguish reduction relation($\longrightarrow$) from labelled transition relations($\xrightarrow{\alpha}$). Specifically, the assertion $P \longrightarrow P'$ states that process P can evolve to process P' as a result of an action *within* P. On the other hand, $P \xrightarrow{\alpha} Q$ indicates that P can evolve to Q after performing action α. Reduction relations are formally defined in Table 3 and transition relations are formally defined in Table 4.

α	:: =		action	P	:: =		process
	|	$u(\widetilde{x})$	input		|	$\alpha.P$	prefix
	|	$\overline{u}\langle\widetilde{y}\rangle$	output		|	$P \mid P$	composition
	|	τ	silent/internal		|	$P + P$	summation
					|	$[x = y].P$	match
					|	$(\nu x).P$	restriction
					|	$!P$	replication
					|	0	inert process

Table 1: Syntax of the π-calculus

SC-MAT	$[x = x]\alpha.P$	$\equiv$	$\alpha.P$
SC-SUM-ASSOC	$P_1 + (P_2 + P_3)$	$\equiv$	$(P_1 + P_2) + P_3$
SC-SUM-COMM	$P_1 + P_2$	$\equiv$	$P_2 + P_1$
SC-SUM-INACT	$P + 0$	$\equiv$	P
SC-COMP-ASSOC	$P_1 \mid (P_2 \mid P_3)$	$\equiv$	$(P_1 \mid P_2) \mid P_3$
SC-COMP-COMM	$P_1 \mid P_2$	$\equiv$	$P_1 \mid P_2$
SC-COMP-INACT	$P \mid 0$	$\equiv$	P
SC-RES	$\nu z\ \nu w\ P$	$\equiv$	$\nu w\ \nu z\ P$
SC-RES-INACT	$\nu z\ 0$	$\equiv$	0
SC-RES-COMP	$\nu z\ (P_1 \mid P_2)$	$\equiv$	$P_1 \mid \nu z\ P_2, if\ z \not\in fn(P_1)$
SC-REP	$!P$	$\equiv$	$P \mid !P$

Table 2: The axioms of structural congruence

$$\text{R-INTER}\ \frac{}{(\overline{x}\langle y\rangle.P_1 + M_1) \mid (x(z).P_2 + M_2) \longrightarrow P_1 \mid P_2\{y/z\}}$$

$$\text{R-PAR}\ \frac{P_1 \longrightarrow P_1'}{P_1 \mid P_2 \longrightarrow P_1' \mid P_2} \qquad \text{R-RES}\ \frac{P \longrightarrow P'}{\nu z P \longrightarrow \nu z P'}$$

$$\text{R-STRUCT}\ \frac{P_1 \equiv P_2 \quad P_2 \longrightarrow P_2' \quad P_2' \equiv P_1'}{P_1 \longrightarrow P_1'} \qquad \text{R-TAU}\ \frac{}{\tau P + M \longrightarrow P}$$

Table 3: Reduction rules in the π-calculus

$$\textsc{Out}\ \frac{}{\overline{x}y.P \xrightarrow{\overline{x}y} P} \qquad \textsc{Inp}\ \frac{}{x(z).P \xrightarrow{xy} P\{y/z\}}$$

$$\textsc{Tau}\ \frac{}{\tau.P \xrightarrow{\tau} P} \qquad \textsc{Mat}\ \frac{\pi.P \xrightarrow{\alpha} P'}{[x=x]\pi.P \xrightarrow{\alpha} P'}$$

$$\textsc{Sum-L}\ \frac{P \xrightarrow{\alpha} P'}{P+Q \xrightarrow{\alpha} P'}$$

$$\textsc{Par-L}\ \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}\ \mathsf{bn}(\alpha) \cap \mathsf{fn}(Q) = \emptyset$$

$$\textsc{Comm-L}\ \frac{P \xrightarrow{\overline{x}y} P' \quad Q \xrightarrow{xy} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

$$\textsc{Close-L}\ \frac{P \xrightarrow{\overline{x}(z)} P' \quad Q \xrightarrow{xz} Q'}{P \mid Q \xrightarrow{\tau} \nu z\,(P' \mid Q')}\ z \notin \mathsf{fn}(Q)$$

$$\textsc{Res}\ \frac{P \xrightarrow{\alpha} P'}{\nu z\,P \xrightarrow{\alpha} \nu z\,P'}\ z \notin \mathsf{n}(\alpha) \qquad \textsc{Open}\ \frac{P \xrightarrow{\overline{x}z} P'}{\nu z\,P \xrightarrow{\overline{x}(z)} P'}\ z \neq x$$

$$\textsc{Rep-act}\ \frac{P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P' \mid !P}$$

$$\textsc{Rep-comm}\ \frac{P \xrightarrow{\overline{x}y} P' \quad P \xrightarrow{xy} P''}{!P \xrightarrow{\tau} (P' \mid P'') \mid !P}$$

$$\textsc{Rep-close}\ \frac{P \xrightarrow{\overline{x}(z)} P' \quad P \xrightarrow{xz} P''}{!P \xrightarrow{\tau} (\nu z\,(P' \mid P'')) \mid !P}\ z \notin \mathsf{fn}(P)$$

Table 4: Transition rules in the π-calculus

2.1.2 Equivalence Relations

Equivalence relations are important concepts in an algebraic system. Unfortunately, the variety of equivalence relations in the π-calculus and other classical process calculi are usually obstacles to learning a calculus. This section will give a gentle introduction to the most important equivalence relation in the π-calculus, the strong barbed bisimulation, followed by short definitions cited from [33] in a designed order. By reading this section, readers will obtain a basic understanding for all equivalences in the Figure 1 and most other common equivalences in literature.

Barbed relations study the behaviour of a process via examining its potential interactions with the environment. Formally,

Definition 2.1. *For each name or co-name μ, the observability predicate $\downarrow_\mu$ is defined by:*
(1) $P \downarrow_\mu$ if P can perform an input action via channel μ
(2) $P \downarrow_{\overline{\mu}}$ if P can perform an output action via channel μ

Strong barbed relations can be defined as follows:

Definition 2.2. *A relation S is a* **strong barbed bisimulation** *if whenever $(P,Q) \in S$,*
(1) $P \downarrow_\mu$ implies $Q \downarrow_\mu$
(2) $P \xrightarrow{\tau} P'$ implies $Q \xrightarrow{\tau} Q'$ for some Q' with $(P', Q') \in S$
(3) $Q \downarrow_\mu$ implies $P \downarrow_\mu$
(4) $Q \xrightarrow{\tau} Q'$ implies $P \xrightarrow{\tau} P'$ for some P' with $(P', Q') \in S$

Definition 2.3. *P and Q are* **strong barbed bisimilar** *if $(P,Q) \in S$ for some barbed bisimulation S.*

Definition 2.4. *P and Q are* **strong barbed congruence** *if $C[P]$ and $C[Q]$ are strong barbed bisimilar for any context C[2].*

Definition 2.5. *P and Q are* **strong barbed equivalent** *if $P \mid R$ and $Q \mid R$ are strong barbed bisimilar for any process R.*

Bisimulation, **bisimilarity**, **congruence**, **and equivalent**, as revealed in the barbed relations, are four related concepts in process calculi. For this reason, in the rest of this subsection, where various of equivalent relations will be introduced, only one of the four concepts in each group will be defined explicitly. The meaning of the rest three relations in each group should be apparent.

Definition 2.6. *A relation S is a* **reduction bisimulation** *if whenever $(P,Q) \in S$,*
(1) $P \xrightarrow{\tau} P'$ implies $Q \xrightarrow{\tau} Q'$ for some Q' with $(P', Q') \in S$
(2) $Q \xrightarrow{\tau} Q'$ implies $P \xrightarrow{\tau} P'$ for some P' with $(P', Q') \in S$

In terms of evaluation strategy, the π-calculus could adopt either the early instantiation scheme or the late instantiation scheme. In the early instantiation scheme, variables are instantiated as soon as an input message is received, more precisely, $\frac{}{a(x).P \xrightarrow{\overline{a}v} P\{v/x\}}$. On the contrary, in the late instantiation scheme, bound variables of input actions are instantiated only when they are involved in an internal communication.

Definition 2.7. *A binary relation S on processes P and Q is an* **early simulation**, *$\sim_e$, if PSQ implies that*
1. If $P \xrightarrow{\alpha} P'$ and α is a free action[3]*, then for some Q', $Q \xrightarrow{\alpha} Q'$ and $P'SQ'$*
2. If $P \xrightarrow{x(y)} P'$ and $y \notin n(P,Q)$, then for all w, there is Q' such that $Q \xrightarrow{x(y)} Q'$ and $P\{w/y\}SQ\{w/y\}$
3. If $P \xrightarrow{\bar{x}(y)} P'$ and $y \notin n(P,Q)$, then for some Q', $Q \xrightarrow{\bar{x}(y)} Q'$ and $P'SQ'$

[2] A context, $C[\cdot]$ is a term written using the same syntax of the π-calculus and an additional constant $\cdot$. Substituting the $\cdot$ by a process P result in a π-calculus term $C[P]$

[3] an internal action or a free output.

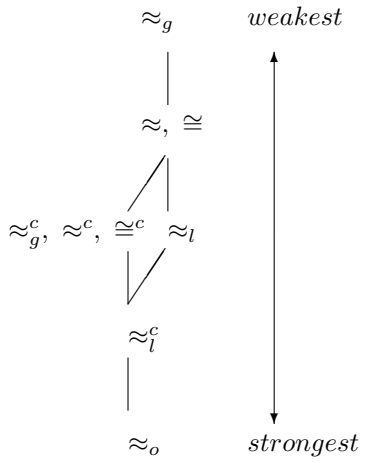
Definition 2.8. *A binary relation S on agents is a* **late simulation**, $\sim_l$, *if PSQ implies that*
1. If $P \xrightarrow{\alpha} P'$ *and* α *is a free action, then for some Q',* $Q \xrightarrow{\alpha} Q'$ *and P'SQ'*
2. If $P \xrightarrow{x(y)} P'$ *and* $y \notin n(P,Q)$*, then for some Q',* $Q \xrightarrow{x(y)} Q'$ *and for all w,* $P\{w/y\}SQ\{w/y\}$
3. If $P \xrightarrow{\bar{x}(y)} P'$ *and* $y \notin n(P,Q)$*, then for some Q',* $Q \xrightarrow{\bar{x}(y)} Q'$ *and P'SQ'*

Definition 2.9. *P and Q are* **full bisimilar**, $P \approx^c Q$ *, if* $P\sigma \approx Q\sigma$ *for every substitution* σ.

Definition 2.10. *A binary relation R over processes is an* **open bisimulation**, $\approx_o$, *if for every pair of elements* $(p,q) \in R$ *and for every name substitution* σ *and every action* α*, whenever* $p\sigma \xrightarrow{\alpha} p'$ *then there exists some q' such that* $q\sigma \xrightarrow{\alpha} q'$ *and* $(p', q') \in R$.

Definition 2.11. *A relation is* **ground bisimulation**, $\approx_g$, *iff whenever* $P \approx_g Q$*, there is* $z \notin fn(P, Q)$ *such that if* $P \xrightarrow{\alpha} Q$*, where* α *is an action, then then* $Q \stackrel{\alpha}{\Rightarrow} \approx_g P$.

To conclude this subsection, Figure 1, cited from [33], presents the hierarchy of equivalences in the π-calculus.



$\approx_g$:	*ground bisimilarity*	$\approx$:	*bisimilarity*	$\cong$:	*barbed equivalence*
$\approx_g^c$:	*ground congruence*	$\approx^c$:	*full bisimilarity*	$\cong^c$:	*barbed congruence*
$\approx_l$:	*late bisimilarity*	$\approx_l^c$:	*late congruence*	$\approx_o$:	*open bisimilarity*

Figure 1: Hierarchy of equivalences in the π-calculus

2.2 The Join-Calculus and the JoCaml Programming Language

Join-calculus [17] is a name passing process calculus that is designed for the distributed programming. There are two versions of the join-calculus, namely the core join-calculus and the distributed join-calculus. The core join-calculus could be considered as a variant of the π-calculus. It is as expressive as the asynchronous π-calculus in the sense that translations between those two calculi are well formulated. A remarkable construct in the join-calculus is join patterns, which provides a convenient way to express process synchronisations. This feature also makes the join-calculus closer to a real programming language. The distributed join-calculus extends the core calculus with location, process migration, and failure recovery. This proposal uses the short phrase "join-calculus" to refer to the distributed join-calculus which includes all components in the core join-calculus. The syntax (Table 5), the scoping rules (Table 6), and the reduction rules (Table 7) of the join-calculus are well summarized in [16].

P	$::=$		processes	D	$::=$		definition
	$\mid$	$x\langle\widetilde{v}\rangle$	asynchronous message		$\mid$	$J \triangleright P$	local rule
	$\mid$	**def** D **in** P	local definition		$\mid$	$\top$	inert definition
	$\mid$	$P \mid P$	parallel composition		$\mid$	$D \wedge D$	co-definition
	$\mid$	$\mathbf{0}$	inert process		$\mid$	$a\ [D : P]$	**sub-location**
	$\mid$	$go\langle a, \kappa\rangle$	**migration**		$\mid$	$\Omega a\ [D : P]$	**dead sub-location**
	$\mid$	$halt\langle\rangle$	**termination**	J	$::=$		join-pattern
	$\mid$	$fail\langle a, \kappa\rangle$	**failure detection**		$\mid$	$x\langle\widetilde{v}\rangle$	message pattern
					$\mid$	$J \mid J$	synchronous join-pattern

Constructs whose explanation is in **bold** font are only used in the distributed join-calculus. Other constructs are used in both distributed and local join-calculus.

Table 5: Syntax of the distributed join-calculus

$J:$	$dv[x\langle\widetilde{v}\rangle]$	$\stackrel{def}{=}$	$\{x\}$	$rv[x\langle\widetilde{v}\rangle]$	$\stackrel{def}{=}$	$\{u \in \widetilde{v}\}$	
	$dv[J \mid J']$	$\stackrel{def}{=}$	$dv[J] \cup dv[J']$	$rv[J \mid J']$	$\stackrel{def}{=}$	$rv[J] \uplus rv[J']$	
$D:$	$dv[J \triangleright P]$	$\stackrel{def}{=}$	$dv[J]$	$rv[J \triangleright P]$	$\stackrel{def}{=}$	$dv[J] \cup (fv[P] - rv[J])$	
	$dv[\top]$	$\stackrel{def}{=}$	$\emptyset$	$fv[\top]$	$\stackrel{def}{=}$	$\emptyset$	
	$dv[D \wedge D']$	$\stackrel{def}{=}$	$dv[D] \cup dv[D']$	$fv[D \wedge D']$	$\stackrel{def}{=}$	$fv[D] \cup fv[D']$	
	$dv[a\ [D : P]]$	$\stackrel{def}{=}$	$\{a\} \uplus dv[D]$	$fv[a\ [D : P]]$	$\stackrel{def}{=}$	$\{a\} \cup fv[D] \cup fv[P]$	[h]
$P:$	$fv[x\langle\widetilde{v}\rangle]$	$\stackrel{def}{=}$	$\{x\} \cup \{u \in \widetilde{v}\}$	$fv[go\langle a, \kappa\rangle]$	$\stackrel{def}{=}$	$\{a, \kappa\}$	
	$fv[\mathbf{0}]$	$\stackrel{def}{=}$	$\emptyset$	$fv[halt\langle\rangle]$	$\stackrel{def}{=}$	$\emptyset$	
	$fv[P \mid P']$	$\stackrel{def}{=}$	$fv[P] \cup fv[P']$	$fv[fail\langle a, \kappa\rangle]$	$\stackrel{def}{=}$	$\{a, \kappa\}$	
	$fv[$**def** D **in** $P]$	$\stackrel{def}{=}$	$(fv[P] \cup fv[D]) - dv[D]$				

Well-formed conditions for D: A location name can be defined only once; a channel name can only appear in the join-patterns at one location.

Table 6: Scopes of the distributed join-calculus

str-join	$\vdash P_1 \mid P_2$	$\rightleftharpoons$	$\vdash P_1,\ P_2$	
str-null	$\vdash \mathbf{0}$	$\rightleftharpoons$	$\vdash$	
str-and	$D_1 \wedge D_2 \vdash$	$\rightleftharpoons$	$D_1,\ D_2 \vdash$	
str-nodef	$\top$	$\rightleftharpoons$	$\vdash$	
str-def	$\vdash$ **def** D **in** P	$\rightleftharpoons$	$D\sigma_{dv} \vdash P\sigma_{dv}$	(range(σ_{dv}) fresh)
str-loc	$\varepsilon a\ [D : P] \vdash_\varphi$	$\rightleftharpoons$	$\vdash_\varphi \parallel \{D\} \vdash_{\varphi\varepsilon a} \{P\}$	(a frozen)
red	$J \triangleright P \vdash_\varphi J\sigma_{rv}$	$\longrightarrow$	$J \triangleright P \vdash_\varphi P\sigma_{rv}$	(φ alive)
comm	$\vdash_\varphi x\langle\widetilde{v}\rangle \parallel J \triangleright P \vdash$	$\longrightarrow$	$\vdash_\varphi \parallel J \triangleright P \vdash x\langle\widetilde{v}\rangle$	($x \in dv[J]$, φ alive)
move	$a[D : P \mid go\langle b, \kappa\rangle] \vdash_\varphi \parallel \vdash_{\psi\varepsilon b}$	$\longrightarrow$	$\vdash_\varphi \parallel a\ [D : P \mid \kappa\langle\rangle] \vdash_{\psi\varepsilon b}$	(φ alive)
halt	$a[D : P \mid halt\langle\rangle] \vdash_\varphi$	$\longrightarrow$	$\Omega a[D : P] \vdash_\varphi$	(φ alive)
detect	$\vdash_\varphi fail\langle a, \kappa\rangle \parallel \vdash_{\psi\varepsilon a}$	$\longrightarrow$	$\vdash_\varphi \kappa\langle\rangle \parallel \vdash_{\psi\varepsilon a}$	($\psi\varepsilon a$ dead, φ alive)

Side conditions: in **str-def**, σ_{dv} instantiates the channel variables $dv[D]$ to distinct, fresh names; in **red**, σ_{rv} substitutes the transmitted names for the received variables $rv[J]$; φ is dead if it contains Ω, and alive otherwise; "a frozen" means that a has no sublocations; εa denotes either a or Ωa

Table 7: The distributed reflexive chemical machine

P	$::=$		processes	D	$::=$		definition
	$\mid$	$x\langle\widetilde{v}\rangle$	asynchronous message		$\mid$	$J \triangleright P$	local rule
	$\mid$	**def** D **in** P	local definition		$\mid$	$\top$	inert definition
	$\mid$	$P \mid P$	parallel composition		$\mid$	$D \wedge D$	co-definition
	$\mid$	$\mathbf{0}$	inert process	J	$::=$		join-pattern
	$\mid$	$\mathrm{x}(\widetilde{V}); P$	**sequential composition**		$\mid$	$x\langle\widetilde{v}\rangle$	message pattern
	$\mid$	**let** $\widetilde{u} = \widetilde{V}$ **in** P	**named values**		$\mid$	$J \mid J$	synchronous join-pattern
	$\mid$	**reply** $\widetilde{V}$ **to** x	**implicit continuation**	V	$::=$		values
					$\mid$	x	value name
					$\mid$	$\mathrm{x}(\widetilde{V})$	**synchronous call**

$\mathrm{x}(\widetilde{v})$	$=$	$x\langle\widetilde{v}, \kappa_x\rangle$	(in join-patterns)
reply $\widetilde{V}$ **to** x	$=$	$\kappa_x\langle\widetilde{V}\rangle$	(in process)
$x\langle\widetilde{V}\rangle$	$=$	**let** $\widetilde{v} = \widetilde{V}$ **in** $x\langle\widetilde{v}\rangle$	
let $\widetilde{u} = \widetilde{V}$ **in** P	$=$	**let** $u_1 = V_1$ **in let** $u_2 = \cdots$ **in** P	
let $\widetilde{u} = \mathrm{x}(\widetilde{V})$ **in** P	$=$	**def** $\kappa\langle\widetilde{u}\rangle \triangleright P$ **in** $x\langle\widetilde{V}, \kappa\rangle$	
let $u = v$ **in** P	$=$	$P\{v/u\}$	
$\mathrm{x}(\widetilde{V}); P$	$=$	**def** $\kappa\langle\rangle \triangleright P$ **in** $x\langle\widetilde{V}, \kappa\rangle$	

Table 8: The core join-calculus with synchronous channel, sequencing, and let-binding

2.2.1 The Local Reflexive Chemical Machine (RCHAM)

The denotational semantics of the join-calculus is usually described in the domain of a reflexive chemical machine (RCHAM). A local RCHAM consists of two parts: a multiset of definitions D and a multiset of active processes P. Definitions specify possible reductions of processes, while active processes can introduce new names and reaction rules.

The six chemical rules for the local RCHAM are **str-join**, **str-null**, **str-and**, **str-nodef**, **str-def**, and **red** in Table 7. As their names suggest, the first 5 are structure rules whereas the last one is reduction rule. Structure rules correspond to reversible syntactical rearrangements. The reduction rule, **red**, on the other hand, represents an irreversible computation.

Finally, for the ease of writing programs, the local join-calculus could be extended with synchronous channel, sequencing, and let-bindings as in Table 8. The distributed join-calculus could be extended similarly.

2.2.2 Distributed Solutions

Distributed system in the join-calculus is constructed in three steps: first, definitions and processes are partitioned into several local solutions; then, each local solution is attached with a unique location name; finally, location names are organized in a location tree.

A distributed reflexive chemical machine (DRCHAM) is simply a multiset of RCHAMs. It is important to note that message pending to a remotely defined channel will be forwarded to the RCHAM where the channel is defined before applying any **red** rule. The above process is a distinction between the join-calculus and other distributed models. The side effect of this evaluation strategy is that both channel and location names must be pairwise distinct in the whole system. As a consequence, a sophisticate name scheme is required for a language that implements the join-calculus.

To support process migration, a new contract, $go\ \langle b, \kappa\rangle$, is introduced, together with the **move** rule. There are two effects of applying the move rule. Firstly, site a moves from one place (φa) to another ($\psi\varepsilon\mathrm{a}$). Secondly, the continuation $\kappa\langle\rangle$ may trigger another computation at the new location.

2.2.3 The Failure Model

A failed location in the join-calculus cannot respond to messages. Reactions inside a failed location or its sub-locations are prevented by the side-condition of reduction rules. Nevertheless, messages and locations are allowed to move into a failed location, but will be frozen in that dead location (str-loc).

To model failure and failure recovery, two primitives $halt\langle\rangle$ and $fail\langle\cdot,\cdot\rangle$ are introduced to the calculus. Specifically speaking, $halt\langle\rangle$ terminates the location where it is triggered (rule halt), whereas $fail\langle a, \kappa\rangle$ triggers the continuation $\kappa\langle\rangle$ when location a fails (rule detect).

2.2.4 The JoCaml Programming Language

The JoCaml programming language is an extension of OCaml. JoCaml supports the join-calculus with similar syntax and more syntactic sugars. When using JoCaml to build distributed applications, users should be aware of following three limitations in the current release (version 3.12) [14]:

1. Functions and closures transmission are not supported. In the join-calculus, distributed calculation is modelled as sending messages to a remotely defined channel. As specified in the **comm** rule, messages sent to a remotely defined channel will be forwarded to the place where the channel is defined. In some cases, however, programmers may want to define an computation at one place but execute the computation elsewhere. The standard JoCaml, unfortunately, does not support code mobility.

2. Distributed channels are untyped. In JoCaml, distributed port names are retrieve by enquiring its registered name (a string) from name service. Since JoCaml encourages modular development, codes supposed to be run at difference places are usually wrote in separated modules and complied independently. The annotated type of a distributed channel, however, is not checked by the name service. Invoking a remote channel whose type is erroneously annotated may cause a run-time error.

3. When mutable value is required over the web, a new copy, rather than a reference to the value, is sent. This may cause problems when a mutable value is referenced at many places across the network.

2.3 The Ambient Calculus and the Obliq Programming Language

2.3.1 The Ambient Calculus

The ambient calculus provides a formal basis for describing mobility in concurrent systems. Here mobility refers to both *mobile computing* (computation carried out in mobile devices) and *mobile computation* (code moves between the network sites) [9]. In reality, there is an additional security requirement for mobility, that is, the authorization for an agent to enter or exit certain administrative domain (e.g. a firewall). The ambient calculus solves the above problems with a fundamental concept: ambient. The three key attributes of a ambient are:

- a name for access control (enter, exit, and open the ambient).
- a collection of local processes/agents that control the ambient.
- a collection of sub-ambients.

An atomic computation in the ambient calculus is a one-step movement of an ambient. Although the pure ambient calculus with mobility is Turing-complete [9], communication primitives are necessary to comfort the encoding of other communication based calculi such as the π-calculus. The full calculus is given through Table 9 to 11, cited from [9]. It is important to note that communication in the ambient calculus are local. In other words, value (name or capability) communication only happens between two processes inside the same ambient.

Mobility and Communication Primitives

$P,Q ::=$	processes
$(\nu n)P$	restriction
$\mathbf{0}$	inactivity
$P \mid Q$	composition
$!P$	replication
$M[P]$	ambient
$M.P$	capability action
$(x).P$	input action
$\langle M \rangle$	async output action
$M ::=$	capabilities
x	variable
n	name
$in\ M$	can enter into M
$out\ M$	can exit out of M
$open\ M$	can open M
ε	null
$M.M'$	path

Free names (revisions and additions)

$$fn(M[P]) \triangleq fn(M) \cup fn(P) \qquad fn(x) \triangleq \emptyset$$
$$fn((x).P) \triangleq fn(P) \qquad fn(n) \triangleq \{n\}$$
$$fn(\langle M \rangle) \triangleq fn(M) \qquad fn(\varepsilon) \triangleq \emptyset$$
$$fn(M.M') \triangleq fn(M) \cup fn(M')$$

Free variables

$$fv((\nu n)P) \triangleq fv(P) \qquad fv(x) \triangleq \{x\}$$
$$fv(\mathbf{0}) \triangleq \emptyset \qquad fv(n) \triangleq \emptyset$$
$$fv(P \mid Q) \triangleq fv(P) \cup fv(Q) \qquad fv(in\ M) \triangleq fv(M)$$
$$fv(!P) \triangleq fv(P) \qquad fv(out\ M) \triangleq fv(M)$$
$$fv(M[P]) \triangleq fv(M) \cup fv(P) \qquad fv(open\ M) \triangleq fv(M)$$
$$fv(M.P) \triangleq fv(M) \cup fv(P) \qquad fv(\varepsilon) \triangleq \emptyset$$
$$fv((x).P) \triangleq fv(P) - \{x\} \qquad fv(M.M') \triangleq fv(M) \cup fv(M')$$
$$fv(\langle M \rangle) \triangleq fv(M)$$

Table 9: Syntax and scope in the ambient-calculus

$P \equiv P$	(Struct Refl)
$P \equiv Q \Rightarrow Q \equiv P$	(Struct Symm)
$P \equiv Q, Q \equiv R \Rightarrow P \equiv R$	(Struct Trans)
$P \equiv Q \Rightarrow (\nu n)P \equiv (\nu n)Q$	(Struct Res)
$P \equiv Q \Rightarrow P \mid R \equiv Q \mid R$	(Struct Par)
$P \equiv Q \Rightarrow !P \equiv !Q$	(Struct Repl)
$P \equiv Q \Rightarrow n[P] \equiv n[Q]$	(Struct Amb)
$P \equiv Q \Rightarrow M.P \equiv M.Q$	(Struct Action)
$P \mid Q \equiv Q \mid P$	(Struct Par Comm)
$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$	(Struct Par Assoc)
$!P \equiv P \mid !P$	(Struct Repl Par)
$(\nu n)(\nu m)P \equiv (\nu m)(\nu n)P$	(Struct Res Res)
$(\nu n)(P \mid Q) \equiv P \mid (\nu n)Q$ if $n \notin fn(P)$	(Struct Res Par)
$(\nu n)(m[P]) \equiv m[(\nu n)P]$ if $n \neq m$	(Struct Res Amb)
$P \mid \mathbf{0} \equiv P$	(Struct Zero Par)
$(\nu n)\mathbf{0} \equiv \mathbf{0}$	(Struct Zero Res)
$!\mathbf{0} \equiv \mathbf{0}$	(Struct Zero Repl)
$P \equiv Q \Rightarrow M[P] \equiv M[Q]$	(Struct Amb)
$P \equiv Q \Rightarrow (x).P \equiv (x).Q$	(Struct Input)
$\varepsilon.P \equiv P$	(Struct ε)
$(M.M').P \equiv M.M'.P$	(Struct .)

Table 10: Structure congurence in the ambient-calculus

$n[in\ m.\ P \mid Q] \mid m[R] \longrightarrow m[n[P \mid Q] \mid R]$	(Red In)
$m[n[out\ m.\ P \mid Q] \mid R] \longrightarrow n[P \mid Q] \mid m[R]$	(Red Out)
$open\ n.\ P \mid n[Q] \longrightarrow P \mid Q$	(Red Open)
$P \longrightarrow Q \Rightarrow (\nu n)P \longrightarrow (\nu n)Q$	(Red Res)
$P \longrightarrow Q \Rightarrow n[P] \longrightarrow n[Q]$	(Red Amb)
$P \longrightarrow Q \Rightarrow P \mid R \longrightarrow Q \mid R$	(Red Par)
$P' \equiv P, P \longrightarrow Q, Q \equiv Q' \Rightarrow P' \longrightarrow Q'$	(Red $\equiv$)
$(x).P \mid \langle M \rangle \longrightarrow P\{x \leftarrow M\}$	(Red Comm)

Table 11: Reduction in the ambient-calculus

2.3.2 The Obliq Programming Language

At the time of writing this report, there is no real language that implements the ambient-calculus[4]. Instead, this section will introduce the Obliq language, which has certain notions of ambient and influenced the design of the ambient calculus.

Obliq[8] is one of the earliest programming languages which support distributed programming. The language was designed before the pervasive of web applications. It only supports simple object model which is a collection of fields. Each field of an Obliq object could be a value (a constant or a procedure), a method, or an alias to an object. An object could either be constructed directly by specifying its fields, or be cloned from other objects.

The four operations which could be performed on objects are:

- selection: a value field of a object could be selected and transmitted over the web. If the selected value is a constant, the value will be transmitted. By contrast, if the selected value is a method, values of its arguments will be transmitted to the remote site where the method is defined, the computation is performed remotely, and the result or an exception is returned to the site of the selection.
- updating: when an updating operation is performed on an remote object, the selected filed is updated to a value that might be sent across the web. If the selected filed is a method, a transmission of method closure is required.
- cloning: cloning an object will yield a new object which contains all fields of argument objects or raise an error if field names of argument objects conflict.
- aliasing: After executing an aliasing method, a.x := **alias** y **of** b **end**, further operations on x of a will be redirected to y of b.

It is important to note that Obliq, as some other languages in the pre-web era, does not distinguish local values from distributed values. By contrast, Waldo etc. [35] pointed out that distinct views must be adopted for local and distributed objects, due to differences in latency, memory access, partial failure, and concurrency.

2.4 The Erlang Language and The OTP Design principles

Erlang [3] is an untyped functional programming language used for constructing concurrent programs. It is widely used in scalable real-time systems. The reliability of Erlang systems is largely attributed to the five OTP design principles, namely, supervision trees, behaviours, applications, releases, and release handling [1]. The following subsections will give a brief introduction to the three principles closely related to distributed computing.

2.4.1 Supervision Trees

Supervision tree is probably the most important concept in the OTP design principle. A supervision tree consists of workers and supervisors. Workers are processes which carry out actual computations while supervisors are processes which inspect a group of workers or sub-supervisors. Since both workers and supervisors are processes and they are organised in a tree structure, the term *child* is used to refer to any supervised process in literature. An example of the supervision tree is presented in Figure 2 [5], where supervisors are represented by squares and workers are represented by circles.

In principle, a supervisor is accountable for starting, stopping and monitoring its child processes according to a list of *child specifications*. A child specification contains 6 pieces of information [1]: i) a internal name for the supervisor to identify the child. ii) the function call to start the child

[4] Cruz and Aguirre [10] proposed a virtual machine for the ambient calculus

[5] This example is cited from OTP Design Principles/Overview. Restart strategies of each supervisor, however, are removed from the figure since they are not related to the central ides discussed here.

process. iii) whether the child process should be restarted after the termination of its siblings or itself. iv) how to terminate the child process. v) whether the child process is a worker or a supervisor. vi) a singleton list which specifies the name of the callback module.

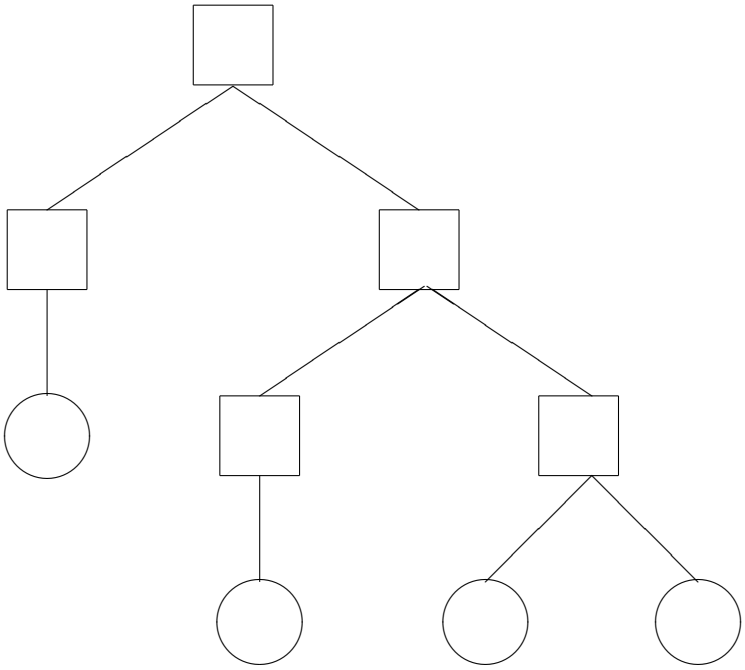


Figure 2: a supervision tree

2.4.2 Behaviours

Behaviours in Erlang, like interface or traits in the objected oriented programming, abstracts common structures and patterns of process implementations. With the help of behaviours, Erlang code can be divided into a generic part, a behaviour module, and a specific part, a callback module. Most processes, including the supervisor in §2.4.1 , could be implemented by realising a set of predefined callback functions for one or more behaviours. Although ad-hoc code and programming structures may be more efficient, using consistent general interfaces make code more maintainable and reliable. Standard Erlang/OTP behaviours include: i) *gen_server* for constructing the server of a clientserver paradigm. ii) *gen_fsm* for constructing finite state machines. iii) *gen_event* for implementing event handling functionality. iv) *supervisor* for implementing a supervisor in a supervision tree. Last but not least, users could define their own behaviours in Erlang.

2.4.3 Applications

The OTP platform is made of a group of components called applications. To define an application, users need to annotate the implementation module with "-behaviour(application)" statement and implement the start/2 and the stop/1 functions. Applications without any processes are called library applications. In an Erlang runtime system, all operations on applications are managed by the *application controller* process [6].

This project is concerns in distributed applications which may be used in a distributed system with several Erlang nodes. An Erlang distributed application will be restarted at another node when its current node goes down. A distributed application is controlled by both the application

[6] registered as application_controller

controller and the distributed application controller [7], both of which are part of the *kernel* application. Two configuration parameters must be set before loading and launching a distributed application. First, possible nodes where the distributed application may run must be explicitly pointed. Then, all nodes configured in the last step will be sent a copy of the same configuration which include three parameters: the time for other nodes to start, nodes that must be started in given time, and nodes that can be started in given time.

2.5 Encoding Functions in Process Calculi

Process Calculi are used to describe the structure and behaviour of concurrent processes. As a basis for the design of programming languages, a calculus should be able to encode canonical calculations with functions. Encoding the λ-calculus, the canonical form of functional programming, to a processes calculus could be done either indirectly or directly.

2.5.1 Indirect Encoding

In [25], Milner gave translation rules from both the lazy λ-calculus and the call-by-value λ-calculus to his π-calculus. Based on this work, in an higher-order π-calculus, functions could be encoded into processes and then passed around the network as a value. Later, Sangiorgi pointed out that the higher-order approach is unnecessary since the plain π-calculus could simulate this higher-order feature by passing a name that points to the encoding process[32].

The main drawback of the two indirect encoding approaches is its inefficiency in translation and reduction. Indirect encoding will yield a mass of intermediate variables. Moreover, without a direct representation for functions, complex substitutions of variables for functions are unavoidable during reductions.

2.5.2 The Blue Calculus: Encoding Functions in a Direct Style

Boudol's blue calculus, π^*, is a direct extension of both the λ-calculus and the π-calculus. In fact, Boudol defined two calculi in [6]: a name-passing λ-calculus (λ^*) and an extended π-calculus without summation and matching (π^*). The λ^*-calculus has λ-style syntax and π-style reduction relation (Table 12). It is used as an intermediate language when translating λ-terms to π^*-terms. The π^*-calculus contains primitives from both the λ^*-calculus and the π-calculus so that the translation from both languages is straightforward.

$L ::= x \mid \lambda x L \mid (Lx) \mid (\mathsf{def}\ x = L\ \mathsf{in}\ L)$	*syntax*
$x \neq z \Rightarrow (\mathsf{def}\ x = N\ \mathsf{in}\ L)z \equiv (\mathsf{def}\ x = N\ \mathsf{in}\ Lz)$	*structural congruence*
$(\lambda x L)z \rightarrow [z/x]L$	*reduction*
$(\mathsf{def}\ \cdots x = L \cdots\ \mathsf{in}\ xy_1 \cdots y_n) \rightarrow (\mathsf{def}\ \cdots x = L \cdots\ \mathsf{in}\ Ly_1 \cdots y_n)$	
$L \rightarrow L' \Rightarrow (Lz) \rightarrow (L'z)$	*context rules*
$L \rightarrow L' \Rightarrow (\mathsf{def}\ x = N\ \mathsf{in}\ L) \rightarrow (\mathsf{def}\ x = N\ \mathsf{in}\ L')$	
$L \rightarrow L'\ \&\ N \equiv L \Rightarrow N \rightarrow L'$	

Table 12: The λ^*-calculus

The λ^*-calculus differs from the λ-calculus in two aspects: (i) The argument (N) in an application (MN) must be a variable. (ii) A convenient notation for name declaration, $def\ x\ =\ N\ in\ M$, is allowed. It is important to note that the λ^*-calculus only contains the call-by-name evaluation. This simplifies subsequent studies on relationship between the λ-calculus and other calculi. Translations from λ to λ^* is similar to Launchbury's encoding in [24]:

[7] registered as dist_ac

$$
\begin{array}{rcl}
x^* & = & x \\
(\lambda x M)^* & = & \lambda x M^* \\
(MN)^* & = & (def\ v = N^*\ in\ (M^* v)) \quad (v \text{ fresh})
\end{array}
$$

As mentioned earlier, both the λ^*-calculus and the π-calculus (without summation and matching) could be translated to the π^*-calculus (see Table 14). In addition, a CPS [8] transform from the π^*-calculus to the π-calculus is given in [6] as well. Lastly, Silvano Dal-zilio [11] proposed a implicit polymorphic type sytem for the π^*-calculus as an improvement of the original simple type system in [6].

syntax:

$$
\begin{array}{rcll}
P & ::= & A \mid D \mid (P \mid P) \mid (\nu x)P & \text{processes} \\
A & ::= & u \mid (\lambda u)P \mid (Pu) & \text{agents} \\
D & ::= & \langle u\ =\ P \rangle \mid \langle u\ \Leftarrow P \rangle & \text{declarations}
\end{array}
$$

structural equivalence:

$$
\begin{array}{rcll}
(P \mid Q) & \equiv & (Q \mid P) & \text{commutativity} \\
((P \mid Q) \mid R) & \equiv & (P \mid (Q \mid R)) & \text{associativity} \\
((\nu u)P \mid Q) & \equiv & (\nu v)(P \mid Q)\ \ (u\ is\ not\ free\ in\ Q) & \text{scope migration} \\
(P \mid Q)u & \equiv & (Pu \mid Qu) & \text{distributivity} \\
((\nu u)P)v & \equiv & (\nu u)(Pv)\ \ (u \neq v) & \\
Du & \equiv & D & \\
\langle u\ =\ P \rangle & \equiv & \langle u \Leftarrow (P \mid \langle u\ =\ P \rangle) \rangle & \text{duplication}
\end{array}
$$

reduction:

$$
\begin{array}{rcll}
((\nu u)P)v & \rightarrow & [v/u]P & \beta \\
(u \mid \langle u \Leftarrow P \rangle) & \rightarrow & P & \text{resource fetching}
\end{array}
$$

Table 13: The π^*-Calculus

$$
\begin{array}{rcl@{\qquad}rcl}
[x]u & = & \overline{x}u & [\overline{u}v_1 \cdots v_k] & = & uv_1 \cdots v_k \\
[\lambda x L]u & = & u(x,v)[L]v & [u(v_1, \cdots, v_k)P] & = & \langle u \Leftarrow (\lambda v_1 \cdots v_k)[P] \rangle \\
[Lx]u & = & (\nu x)([L]v \mid \overline{v}xu) & [!u(v_1, \cdots, v_k)P] & = & \langle u\ =\ (\lambda v_1 \cdots v_k)[P] \rangle \\
[def\ x\ =\ N\ in\ L]u & = & (vx)([L]u \mid\ !x(v)[N]v) & [P \mid Q] & = & ([P] \mid [Q]) \\
 & & & [(\nu u)P] & = & (\nu u)[P]
\end{array}
$$

Table 14: Translation from λ^*-calculus and π-calculus to π^*-calculus

2.6 Implementation Strategies

This section will give a short survey on Peter Van Roy's general framework for the implementation of concurrent programming languages [29, 30]. [29] abstracts a four-layer-structure which smoothly applies to four languages designed for different purposes. The four-layer-structure and case studies in [29] are summarized in Table 15. The rest of this section will introduce the main results in [30], which is focused on implementation details.

2.6.1 A General Purpose Virtual Machine

The basic abstract machine in [30] consists of three elements: statement $\langle s \rangle$, environment E, and assignment store σ. Explanations for concepts related to this VM are given as follows:

[8] continuation passing style

Layer	Erlang	E	Mozart	Oz
strict functional language	Y	Y	Y	Y
deterministic concurrency	N	Y	unknown	Y
asynchronous message passing	Y	Y	Y	Y
global mutable state	Y	N	Y	Y

Table 15: a shared structure for 4 programming languages

- an *assignment store* σ contains a set of variables. Variables could be bound, unbound, or partially bound.
- an *environment* E stores mappings from variable identifiers to entities in σ.
- a *semantic statement* is a pair of ($\langle s \rangle$, E).
- an *execution state* is a pair of (ST, σ), where ST is a stack of semantic statements.
- a *computation* is a sequence of transitions between execution states.

2.6.2 Extending the Basic VM

The basic VM described in the last section is capable to model most of computation paradigms directly or indirectly. More pleasantly, several restrictions and extensions could be added to this basic VM on demand.

- MST(multiset of semantic stacks). Concurrent programming would be supported via simply replacing the ST by MST. Using the multiset structure for storing semantic stacks gives the flexibility of coordinating identical threads.
- bound or unbound variables. In the book, unbound variables refer to variables which have not been assigned a value. In sequential computing, unbound variables should not be allowed since the program will be halting forever. In concurrent computing, however, unbound variables are acceptable since it might be bound by another thread later.
- single-assignment variable or multiple-assignment variable. Although using single-assignment variable should be encouraged to avoid side-effects, multiple-assignment variable may ease the implementation of imperative languages. Alternatively, both kinds of variable could appear in the same language, such as the Scala language, with different notations.
- variables lifetime. There are three moments in a variables lifetime. i) creation, ii) specification, and iii) evaluation. Adjacent moments may or may not be happen at the same time. Different combinations yields Table 16 cited from [30].
- more kinds of stores. If the state of a shared store could be regarded as the status of a program, evaluation rules on store variables implies features of the programming paradigm. The first three examples in the next subsection demonstrate this idea.
- new statement syntax. When the encoding of certain feature is expensive within the current kernel syntax, adding a new form of statement is one of the easiest way to increase the expressiveness of the language. The gained expressiveness, however, may not be free. For example, the number evaluation rules will be increased dramatically and reasoning about programs will be more complex. Moreover, properties enjoyed by the old model may no longer be hold in the new one.

	sequential with values	sequential with values and dataflow variables	concurrent with values and dataflow variables
eager execution	strict functional programming (e.g., Scheme, ML) (1)&(2)&(3)	declarative model (e.g., Prolog) (1),(2)&(3)	data-driven concurrent model (1),(2)&(3)
lazy execution	lazy functional programming (e.g., Haskell) (1)&(2),(3)	lazy FP with dataflow variables (1),(2),(3)	demand-driven concurrent model (1),(2),(3)

(1): Declare a variable in the store
(2): Specify the function to calculate the variable's value
(3): Evaluate the function and bind the variable

(1)&(2)&(3): Declaring, specifying, and evaluating all coincide
(1)&(2),(3): Declaring and specifying coincide; evaluating is done later
(1),(2)&(3): Declaring is done first; specifying and evaluating are done later and coincide
(1),(2),(3): Declaring, specifying, and evaluating are done separately

Table 16: Popular computation models in languages

2.6.3 Example Paradigms

Following are 5 examples chose from [30]. Each example demonstrates how a popular modern programming feature could be implemented with techniques described in §2.6.2.

- Laziness could be realised by adding a need-predict store.
- Message passing concurrency could be realised by adding a mutable store, together with two primitives NewPort and Send.
- Explicit state could be realised by introducing a mutable cell store. A cell records a mapping from a name value to a store variable. Although a name value might be mapped to different store variables at different time, changes to name-variable mapping will not affect the internal state of a program.
- Relational programming could be realised by adding choice and failure statements to kernel.
- Constraint programming could be realised by adding a constraint store and a batch of primitive operations.

The book [30] provides at least two prominent contributions. Firstly, it provides a framework which unifies most of popular programming paradigms. Secondly, combining different versions of statements, environment and assignment store yields different programming models. To the author of this research proposal, those three components are three dimensions of the "language space". Researchers in the area of programming languages could use this coordinate system is two ways. On the one hand, it provides a convenient formalism to compare different programming languages. On the other hand, regardless of their usages, new programming paradigms could be defined by filling "blank coordinates" of the space. In fact, some models in the book does not correspond to any well known languages.

3 Some Research Problems

3.1 Name Server that Combines Static and Dynamic Typing

In distributed programming which supports modular compilation, distributed components are implemented according to agreed communication interfaces. In untyped settings, developers strictly follow the system specification to ensure that one part will be compatible with others. In typed settings, developers have the advantage of checking the consistency between implementation and the predefined communication interface at compilation time, before the implementation is plugged into the running system. Unfortunately, most of distributed systems are not static but require dynamic updating individual components, and therefore change communication interfaces now and then.

To detect incompatible interfaces at the earliest opportunity, we could implement a novel name server that maps a pair of name and type to a channel expecting values of that type. The novel name server will provide two benefits. Firstly, ill-typed message sending to a registered channel will be rejected before it is sent. Secondly, components that relies on a service could be informed when the new communication interface is no longer backward compatible.

Although building a name server described as above will provide promising features, it is not clear yet how many changes are required to embed the name server into standard Scala or the akka framework.

3.2 Hot Code Swapping

Hot code swapping plays an important role in systems running on OTP platform. To support peaceful hot code swapping, the Erlang/OTP run-time keeps at most two versions of a module simultaneously.

In an experiment, I implemented a ProcessWrapper class that maintains two pointers to two instances of my GenServer class. Messages sending to the current version of a GenServer implementation is delegated via an instance of ProcessWrapper. Comparing to the Erlang platform, this design has two limitations. For one thing, for a GenServer potentially to be swapped, it must be specified in advance. For another, one needs to swap the entire GenServer rather than the specific server implementation within a GenServer. Those two problems demand further investigations.

The akka library [23] also provides a form of hot swap on the message loop of an actor. In akka, different versions of message loops are stored at a stack in corresponding ActorContext. Users could send the actor some special messages, defined in the current message loop, to trigger the context.become/unbecome method. The downside of this method is three-fold. Firstly, developers need to specify not only that the code will be potentially hot swapped but also how to trigger the swap. Secondly, passing new message loop, which is a function, over network is still a problem. Lastly, different from the behaviour of Erlang programs, only one message loop is "active" at a time.

3.3 Actor Parametrised on the Type of Expecting Messages

In regular supervision meetings, Professor Wadler and I discussed the possibility of using actors that are parametrised on the type of message it expects to receive. This proposal sounds like a feasible solution to enhance type-safety of communication between actors. However, we do not find such a version in our working platforms, neither in pure Scala nor in akka libraries. Further investigation reveals following problems of implementing the proposed Actor in Scala and akka as a compatible alternative:

1. The *sender* field. To simplify the work of replying to synchronous request, both akka actor and scala actor has a *sender* field that records the sender of the latest received message. Including a *sender* field in our proposed Actor class raises two challenges. For one thing, the type of the *sender* filed is only known at run-time. Therefore, the type parameter of an actor should not be removed at compiler time so that it could be transmitted with the

message. For another, the sender may also be able to deal with messages whose type is not known by the receiver. Understanding full type information of messages expected by the sender is unnecessary for the receiver.

2. Linked actors. Following the Erlang design, Actors may link together so that the exit signal of a dying actor will be broadcasted to other actors. In this case, only exit signals are transmitted between actors. Same as problem 1, knowing type parameters of all linked actors is unnecessary but increase the burden of transmitting type information.

3. Layers in a supervision tree. The akka framework provides APIs to enquiry the parent actor and child actors of an actor. In this situation, it may not be a good idea to leak type information between layers. Similar issues will be discussed in §3.4.

3.4 Wadler's Type Pollution Problem

Most of modern systems are built in layers, where each layer provides services to the layer immediately above it, and implements those services using services in the layer immediately below it. If a layer is implemented using an actor, then the actor will receive both messages form the layer above and the layer below via its only channel. This means that the higher layer could send a message that is not expected from it, so as the lower layer.

One solution to the type pollution problem is using sub-typing. Let a layer has type Actor[T], denoting that it is an actor that expects message of type T. With sub-typing, one could publish the layer as Actor[A] to the layer above while publish the same layer as Actor[B] to the layer below. The system could check if both A and B are subtypes of T.

Another solution is using separate channels for requests from the higher level and responses from the lower level. However, the plain actor model has to be abandoned in this solution. Currently, we identified two alternatives for the plain actor model: join calculus [15] and session types [20, 21]. In join calculus, one could define different channels to be used for different messages. In session types, a channel is only used for communication between two parties at any time; therefore, after initialising a session requested by the layer above, the middle layer requests a new session for communications with the layer below. The join calculus has been implemented both as language extensions [5] and libraries [19, 28, 31]. The session type, on the other hand, has only been implemented as prototype languages [21, 22]. It could be a challenge to implement session types as a library.

3.5 Eventual Consistency for Shared States

Users of a distributed system would expect receiving consistent value when using shared states. Unfortunately, known as the CAP theorem [18], strong consistency, availability, and partition tolerance could not be obtained at the same time in distributed systems. In practice, developers usually weaken strong consistency to eventually consistency to improve availability and partition tolerance. Eventually consistency is an active topic both in theoretical study[7, 34] and in practise[12]. For our project, we would like to apply the eventually consistency model to build global consistent name server.

4 Project Progress and Plans

4.1 Progress Achieved since the First Panel Review

With suggestions from panel members, I have been working on the more focused aim stated in §1.2. Research objectives are also tailored to more specific tasks accordingly. Meanwhile, I studied more related works on concurrent programming and type theories.

Before the new year, three medium-sized applications has been identified. Both the ATM example and the main application of the elevator controller example have been ported into Scala using akka libraries. The later example contains an interface for properties checking using QuickCheck. I am working on porting QuickCheck properties specified in the Erlang version to my Scala implementation. The porting of a larger example, Riak, will be started soon.

For the work of implementing a prototype Scala library that supports OTP principles, I found a way of mimicking hot code swap via object delegation. However, this strategy requires at least one non-swappable wrapper process, which is not needed in the Erlang/OTP platform. I also identified some problems of implementing actors which have a type parameter that specifies the type of expecting messages. To overcome the bottleneck of solving those problems, I postponed this work for learning lessons from the design of existing frameworks.

In the last report, I mentioned Walder's type pollution problem when using the Actor model to construct layered systems. One solution is using subtyping on the type parameter of the reference to a typed actor. However, this solution is awkward and does not cope with the dynamic configuration of distributed systems. To improve this situation, I recently proposed using union types to plot a type in a type hierarchy on the fly. At the meantime, Professor Walder pointed out that another solution could be using session types, where each channel is only used for the communication between two parties.

4.2 Plan for the Second Year

In the rest of the second year, I will port the Riak example and summarise problems that I encountered when translating the three examples into Scala using akka. More translation exercises will be done if proper examples are found.

Based on the summary, I shall started the the design and implementation of library that supports OTP principles. This work will require iterative and incremental development on requirement analysis, interface design, functionality implementation, and system evaluation. In addtion, a flexible plan should be given in advance to guide and monitor the pace of progress.

4.3 Plan for the Third Year

The beginning of the third year will continue the work of library implementation. After that, the library will be published with a technical report that covers: (i) functionalities provided by the library, (ii) alternative choices of library design and reasons for adopting the chosen design, (iii) benchmark results of selected canonical distributed applications.

In the meantime of receiving feedbacks from programming communities, I will re-implement the selected applications with the new library. Testing results for the new implementation will be published in an updated version of library documentation.

Finally, as promised in the research objectives, a more comprehensive summary of common challenges in distributed programming and their solutions will be given. The summary will reveal how language libraries, which is used for practical purposes, are related to their theoretical guidance underneath. It may also plot research blanks for future study.

4.4 Potential Risks

As a novel and ambitious research, some tasks may be tougher than my expectation. Also, results of some tasks may differ from my intuition. Risk analyses for planed objectives are given below.d

Firstly, sample applications used in the first and second objectives, which are the basis of research problems specification and results evaluation, must be carefully selected. It would be better to select open-source applications implemented in high-quality code. For a candidate application, we need to analysis the proportion of implementation for OTP features to the implementation for other problems out of our main research interests. Fortunately, we have identified three appropriate applications.

Secondly, some research problems rooted in the center of distributed programming, and therefore is inevitably also investigated by other researchers. Although it seems feasible to re-implement identified applications in other frameworks and then identify problems of using those frameworks, our chosen comparing frameworks are vigorous and evolving quickly. For example, significant changes have been made to the actor model in the new akka library (version 2.0). The evolving of comparing frameworks contributes to plausible solutions to some research problems as well as pressures of providing devised simpler solutions for tough problems. In addition, since there is no uniform model for distributed programming, it would be hard to evaluate to what extend our library interface is a general framework for distributed programming.

Lastly, apart from above general difficulties, I also encountered implementation difficulties listed in §3. At the time of this writing, it is not clear to me how those problems could be solved in a neat way. I anticipate that new problems will be added to the list when I write more programs in type safe-less environment.

Despite the identified and potential troubles which may distract this research from a perfect result, none of above risks should be the principle hindrance of achieving the main objectives listed in §1.3. With proper plans, wide sources of suggestions, and the commitment to research objectives, the proposed research should be suitable for a PhD project.

A Scala Join (version 0.3) User Manual

The Scala Join Library (version 0.3) improves the scala joins library [19] implemented by Haller and Cutsem. Main advantages of this library are: (i) providing uniform *and* operator, (ii) supporting pattern matching on messages, (iii) supporting theoretically unlimited numbers of join patterns in a single join definition (iv) using simpler structure for the **Join** class, and most importantly, (v) supporting communications on distributed join channels.

A.1 Using the Library

A.1.1 Sending messages via channels

An elementary operation in the join calculus is sending a message via a channel. A channel could be either asynchronous or synchronous. At the caller's side, sending a message via an asynchronous channel has no obvious effects in the sense that the program will always proceed. By contrast, when a message is sent via a synchronous channel, the current thread will be suspended until a result is returned.

To send a message m via channel c, users simply apply the message to the channel by calling $c(m)$. For the returned value of a synchronous channel, users may want to assign it to a variable so that it could be used later. For example,

```
val v = c(m) // c is a synchronous channel
```

A.1.2 Grouping join patterns

A join definition defines channels and join patterns. Users define a join definition by initializing the **Join** class or its subclass. It is often the case that a join definition should be globally *static*. If this is the case, it is a good programming practice in Scala to declare the join definition as a *singleton object* with the following idiom:

```
object join_definition_name extends Join{
  //channel declarations
  //join patterns declaration
}
```

A channel is a singleton object inside a join definition. It extends either the **AsyName**[**ARG**] class or the **SynName**[**ARG**, **R**] class, where **ARG** and **R** are generic type parameters. Here, **ARG** indicates the type of channel parameter whereas **R** indicates the type of return value of a synchronous channel. The current library only supports unary channels, which only take one parameter. Fortunately, this is sufficient for constructing nullary channels and channels that take more than one parameter. A nullary channel could be encoded as a channel whose argument is always **Unit** or any other constants. For a channel that takes more than one parameter, users could pack all arguments in a tuple.

Once a channel is defined, we can use it to define a join pattern in the following form

```
case <pattern> => <action>
```

The <*pattern*> at the left hand side of ⇒ is a set of channels and their formal arguments, connected by the infix operator *and*. The <*action*> at the right hand side of ⇒ is a sequence of Scala statements. Formal arguments declared in the <*pattern*> must be pairwise distinct and might be used in the <*action*> part. In addition, each join definition accepts one and only one group of join patterns as the argument of its *join* method. Lastly, like most implementations for the join calculus, the library does not permit multiple occurrences of the same channel in a single join pattern. On the other side, using the same channel in an arbitrary number of different patterns is allowed.

We conclude this section by presenting a sample code that defines and uses join patterns.

Listing 1: Example code for defining join patterns (join_test.scala)

```
import join._
import scala.concurrent.ops._ // spawn
object join_test extends App{// for scala 2.9.0 or later
  object myFirstJoin extends Join{
    object echo extends AsyName[String]
    object sq extends SynName[Int, Int]
    object put extends AsyName[Int]
    object get extends SynName[Unit, Int]

    join{
      case echo("Hello") => println("Hi")
      case echo(str) => println(str)
      case sq(x) => sq reply x*x
      case put(x) and get(_) => get reply x
   }
  }

  spawn {
    val sq3 = myFirstJoin.sq(3)
    println("square(3) = "+sq3)
  }
  spawn { println("get: "+myFirstJoin.get()) }
  spawn { myFirstJoin.echo("Hello"), myFirstJoin.echo("Hello World") }
  spawn { myFirstJoin.put(8) }
}
```

One possible result of running the above code is:

```
>scalac join_test.scala
>scala join_test
square(3) = 9
Hi
Hello World
get: 8
```

A.1.3 Distributed computation

With the distributed join library, it is easy to construct distributed systems on the top of a local system. This section explains additional constructors in the distributed join library by looking into the code of a simple client-server system, which calculates the square of an integer on request. The server side code is given at Listing 2 and the client side code is given at Listing 3.

Listing 2: Server.scala

```
import join._
object Server extends App{
  val port = 9000
  object join extends DisJoin(port, 'JoinServer){
    object sq extends SynName[Int, Int]
    join{ case sq(x) => println("x:"+x); sq reply x*x }
    registerChannel("square", sq)
  }
  join.start()
}
```

Listing 3: Client.scala

```
object Client{
  def main(args: Array[String]) {
    val server = DisJoin.connect("myServer", 9000, 'JoinServer)
    //val c = new DisSynName[Int, String]("square", server)
    //java.lang.Error: Distributed channel initial error:
    //        Channel square does not have type Int => java.lang.String ...
    val c = new DisSynName[Int, Int]("square", server)//pass
    val x = args(0).toInt
    val sqr = c(x)
    println("sqr("+x+") = "+sqr)
    exit()
}}
```

```
> scala ServerTest
Server 'JoinServer Started...
                                                    >scala Client 5
x:5
                                                    sqr(5) = 25
                                                    >scala Client 7
x:7
                                                    sqr(7) = 49
```

In Server.scala, we constructed a distributed join definition by extending class **DisJoin**(**Int**, **Symbol**). The integer is the port where the join definition will listen and the symbol is used to identify the join definition. The way to declare channels and join patterns in **DisJoin** is the same as the way in **Join**. In addition, channels which might be used at remote site are registered with a memorizable string. At last, different from initializing a local join definition, a distributed join definition has to be explicitly started.

In Client.scala, we connect to the server by calling DisJoin.connect. The first and second arguments are the hostname and port number where the remote join definition is located. The last argument is the name of the distributed join definition. The hostname is a String which is used for the local name server to resolve the IP address of a remote site. The port number and the name of join definition should be exactly the same as the specification of the distributed join definition.

Once the distributed join definition, server, is successfully connected, distributed channels could be initialized as instances of DisAsyName[ARG](channel_name, server) or DisSynName[ARG, R](channel_name, server). Using an unregistered channel name or declaring a distributed channel whose type is inconsistent with its referring local channel will raise a run-time exception during the channel initialization. In later parts of the program, the client is free to use distributed channels to communicate with the remote server. The way to invoke distributed channels and local channels are the same.

A.2 Implementation Details

A.2.1 Case Statement, Extractor Objects and Pattern Matching in Scala

In Scala, a partial function is a function with an additional method: $isDefinedAt$, which will return $true$ if the argument is in the domain of this partial function, or $false$ otherwise. The easiest way to define a partial function is using the case statement. For example,

```
scala> val myPF : PartialFunction[Int,String] = {
     |  case 1 => "myPF apply 1"
     | }
myPF: PartialFunction[Int,String] = <function1>

scala> myPF.isDefinedAt(1)
res1: Boolean = true

scala> myPF.isDefinedAt(2)
res2: Boolean = false

scala> myPF(1)
res3: String = myPF apply 1

scala> myPF(2)
scala.MatchError: 2
       at $anonfun$1.apply(<console>:5)
       ...
```

In addition to basic values and case classes, the value used between *case* and ⇒ could also be an instance of an *extractor object*: object that contains an *unapply* method[13]. For example,

```
scala> object Even {
     |   def unapply(z: Int): Option[Int] = if (z%2 == 0) Some(z/2) else None
     | }
defined module Even

scala>  42 match { case Even(n) => Console.println(n) } // prints 21
21

scala>  41 match { case Even(n) => Console.println(n) } // prints 21
scala.MatchError: 41
       ...
```

In the above example, when a value, say x, attempts to match against a pattern, $Even(n)$, the method $Even.unapply(x)$ is invoked. If $Even.unapply(x)$ returns $Some(v)$, then the formal argument n will be assigned with the value v and statements at the right hand side of ⇒ will be executed. By contrast, if $Even.unapply(x)$ returns $None$, then the current case statement is considered not matching the input value, and the pattern examination will move towards the next case statement. If the last case statement still does not match the input value, then the whole partial function is not defined for the input. Applying a value outside the domain of a partial function will rise a *MatchError*.

A.2.2 Implementing local channels

Both asynchronous channel and synchronous channel are subclasses of trait *NameBase*. The reason why we introduced this implementation free trait is that, although using generic types to restrict the type of messages pending on a specific channel is important for type safety, a uniform view for asynchronous and synchronous channels simplifies the implementation at many places. For example, the three methods listed in Listing 4 are common between those two kinds of channels and are important for the implementation of Join and DisJoin class.

Listing 4: The NameBase trait

```
trait NameBase{ // Super Class of AsyName and SynName
  def argTypeEqual(t:Any):Boolean
  def pendArg(arg:Any):Unit
  def popArg():Unit
}
```

Listing 5: Code defines local asynchronous channel

```
class AsyName[Arg](implicit owner: Join, argT:ClassManifest[Arg]) extends NameBase{
  var argQ = new Queue[Arg] //queue of arguments pending on this name

  override def pendArg(arg:Any):Unit = {
    argQ += arg.asInstanceOf[Arg]
  }

  def apply(a:Arg) :Unit = synchronized {
    if(argQ.contains(a)){ argQ += a }
    else{
      owner.trymatch(this, a) // see if the new message will trigger any pattern
    }
  //other code
}
```

Asynchronous channel is implemented as Listing 5. The implicit argument *owner* is the join definition where the channel is defined. The other implicit argument, **argT**, is the descriptor for the run time type of **Arg**. Although **argT** is a duplicate information for **Arg**, it is important for distributed channels, whose erased type parameter might be declared differently between different sites. We postpone this problem until §A.2.4.

As shown in the above code, an asynchronous channel contains an argument queue whose element must have generic type **Arg**. Sending a message via a channel is achieved by calling its *apply* method, so that $c(m)$ could be written instead of c.apply(m) for short in Scala. Based on the linear assumption that no channel should appear more than once in a join pattern, reduction is possible only when a new message value is pending on a channel. Therefore, if the new message has the same value as another pended message, it should be attached to the end of the message queue; Otherwise, the join definition will be notified to perform a pattern checking and fire a possible pattern, if there is one.

As listing 6 shows, in addition to firing a pattern or pending a message to the message queue, an invocation on synchronous channel also needs to return a result value to the message sender. Since many senders may be waiting for a return value at the same time, for each reply invocation, the library need to work out which message the result is replied for. To this end, messages with the same value is tagged with different integers. The library uses msgTags to store the message that matches current fireable pattern. When a reply method is called, the channel inserts a integer-message pair and its corresponding reply value to the result queue and notifies all fetch threads that are waiting for a reply. With the help of the synchronized method, only one thread could attempt to fetch the reply value at a time.

Listing 6: Code defines local synchronous channel

```
class SynName[Arg, R](implicit owner: Join, argT:ClassManifest[Arg], resT:ClassManifest[R])
 extends NameBase{
  var argQ = new Queue[(Int,Arg)] // argument queue
  var msgTags = new Stack[(Int,Arg)] // matched messages
  var resultQ = new Queue[((Int,Arg), R)] // results

  private object TagMsg{
    val map = new scala.collection.mutable.HashMap[Arg, Int]
    def newtag(msg:Arg):(Int,Arg) = {
      map.get(msg) match {
        case None =>
          map.update(msg,0)
          (0,msg)
        case Some(t) =>
          map.update(msg, t+1)
          (t+1, msg)
   }}}

  def pushMsgTag(arg:Any) = synchronized {
    msgTags.push(arg.asInstanceOf[(Int,Arg)])
  }

  def popMsgTag:(Int,Arg) = synchronized {
    if(msgTags.isEmpty) { wait(); popMsgTag }
    else{ msgTags.pop }
  }

  def apply(a:Arg) :R = {
    val m = TagMsg.newtag(a)
    argQ.find(msg => msg._2 == m._2) match{
      case None => owner.trymatch(this, m)
      case Some(_) => argQ += m
    }
    fetch(m)
  }

  def reply(r:R):Unit = spawn {synchronized {
    resultQ.enqueue((msgTags.pop, r))
    notifyAll()
  }}

  private def fetch(a:(Int,Arg)):R = synchronized {
    if (resultQ.isEmpty || resultQ.front._1 != a){
      wait(); fetch(a)
    }else{ resultQ.dequeue()._2 }
  }
  //other code
}
```

A.2.3 Implementing the join pattern using extractor objects

The unapply method for local synchronous channel

In this library, join patterns are represented as a partial function. To support join patterns and pattern matching on message values, the library provides the unapply method for local channels. The unapply method for synchronous channel is given in Listing 8. The unapply method for asynchronous channel is almost the same as the synchronous version, except that it does not need

to deal with message tags.

Listing 7 gives the core of the unapply method of synchronous channel. The five parameters sent to the unapply method are:

(i) *nameset*: channels that could trigger the first fireable pattern.

(ii) *pattern*: the join pattern itself.

(iii) $fixedMsg$: a map from channels to corresponding message values. If the current channel is a key of the map, the unapply method returns its mapped value.

(iv) *dp*: an integer indicates the depth of pattern matching. The *dp* is useful for optimizations and debugging.

(v) $bandedName$: a banded channel name. If the current channel is the same as the $bandedName$, the unapply method returns None.

When a channel is asked to select a message that could trigger a *pattern*, it first check rule (iii) and (v). If neither rule applies, the channel returns the first message that matches the pattern and adds this channel to the *nameset*, if such a message exists. We consider a message of the current channel triggers a join pattern if the join pattern cannot be fired without the presence of messages on the current channel and will be fired when that message is bound to the current channel.

Listing 7: Core of the unapply method of local synchronous channel

```
case (nameset: Set[NameBase], pattern:PartialFunction[Any, Any],
          fixedMsg:HashMap[NameBase, Any], dp:Int, banedName:NameBase) => {
       //other code
       if (this == banedName) {return None}
       if(fixedMsg.contains(this)){
         Some(fixedMsg(this).asInstanceOf[(Int,Arg)]._2)
       }else{

         def matched(m:(Int,Arg)):Boolean = {
         //    pattern cannot be fired without the presence of message on current channel
         //and pattern can be fired when m is bound to the current channel
             (!(pattern.isDefinedAt(nameset, pattern, fixedMsg+((this, m)), dp+1, this))
          && (pattern.isDefinedAt((nameset, pattern, fixedMsg+((this, m)), dp+1, banedName))))
         }

       var returnV:Option[Arg] = None

       argQ.span(m => !matched(m)) match {
          case (_, MutableList()) => { // no message pending on this channel may trigger the pattern
            returnV = None
          }
          case (ums, ms) => {
            val arg = ms.head // the message could trigger a pattern
            nameset.add(this)
            if(dp == 1) {pushMsgTag(arg)}
            returnV = Some(arg)
          }
       }
       returnV
     }
}
```

The above code implements the core algorithm and could be improved for better efficiency. Firstly, if the value of a message has been proved not to trigger the join pattern, the *matched*

method invoked by the span iteration does not need to run complex test for that value. To this end, a HashSet $checkedMsg$ could be introduced to record checked message values. The set should be cleared after the span iteration. Secondly, when a message is selected, popping it to the head of the message queue will save the later work of removing that message from the queue. Lastly, each channel that triggers a pattern only needs to be added to the *nameset* once. Although inserting an element to a hashset is relatively cheap, the cost could be further reduced to the cost of comparing two integers. The full implementation for the unapply methods of synchronous channel is given at Listing 8. The first case statement is used for improving the efficiency of code involving singleton pattern, where a pattern only contains one channel. More explanation for this decision will be given in §A.2.3.

Listing 8: The unapply method of local synchronous channel

```
  def unapply(attr:Any) : Option[Arg]= attr match {
    case (ch:NameBase, arg:Any) => {// For singleton patterns
      if(ch == this){ Some(arg.asInstanceOf[(Int,Arg)]._2))
      }else{ None }
    }
    case (nameset: Set[NameBase], pattern:PartialFunction[Any, Any],
          fixedMsg:HashMap[NameBase, Any], dp:Int, banedName:NameBase) => {
      if (this == banedName) {return None}
      if(fixedMsg.contains(this)){ Some(fixedMsg(this).asInstanceOf[(Int,Arg)]._2))
      }else{
        var checkedMsg = new HashSet[Arg]

        def matched(m:(Int,Arg)):Boolean = {
          if (checkedMsg(m._2)) {false} // the message has been checked
          else {
            checkedMsg += m._2
            (!(pattern.isDefinedAt(nameset, pattern, fixedMsg+((this, m)), dp+1, this))
            && (pattern.isDefinedAt((nameset, pattern, fixedMsg+((this, m)), dp+1, banedName))))
          }
        }

        var returnV:Option[Arg] = None
        argQ.span(m => !matched(m)) match {
           case (_, MutableList()) => { returnV = None }
           case (ums, ms) => {
             val arg = ms.head // the message could trigger a pattern
             argQ = (((ums.+=:( arg )) ++ ms.tail).toQueue) // pop this message to the head of message queue
             if(dp == 1) {nameset.add(this); pushMsgTag(arg)}
             returnV = Some(arg._2)
           }
        }
        checkedMsg.clear
        returnV
      }
    }
}
```

The *Join* class and the *and* object As said in the earlier section, join patterns are represented as a partial function in this library. An instance of the Join class is responsible for storing the join definition and attempting to fire a pattern on request. If the requested channel message association could fire a pattern, all channels involved in that pattern will be asked to remove the matched message; otherwise, the channel will be notified to pend the message to its message queue.

Although this library encourages using join patterns as a convenience constructor to synchro-

nizing resources, actor model is popular at the time of implementing this library and not all channels need to be synchronized with others. For this reason, this library gives singleton patterns the privilege on pattern examination. Readers may wonder to what extent the efficiency will be affected by the above decision. To answer this question, consider a typical join definition where p patterns are defined and there are m channels on each pattern. At the time of a new message's arrival, there are n messages pending on each channel on average. On average, this library needs $O(p)$ time to check all patterns as if they are singleton patterns before spending $O(pmn)$ time checking all patterns as join patterns. Therefore, the additional checking will not significantly increase the cost of checking join patterns but will benefit programs that use singleton patterns.

Listing 9: Code defines the Join class

```
class Join {
  private var hasDefined = false
  implicit val joinsOwner = this
  private var joinPat: PartialFunction[Any, Any] = _

  def join(joinPat: PartialFunction[Any, Any]) {
   if(!hasDefined){
     this.joinPat = joinPat
     hasDefined = true
   }else{
     throw new Exception("Join definition has been set for"+this)
   }
  }

  def trymatch(ch:NameBase, arg:Any) = synchronized {
    var names: Set[NameBase] = new HashSet
    try{
      if(ch.isInstanceOf[SynName[Any, Any]]) {ch.asInstanceOf[SynName[Any,Any]].pushMsgTag(arg)}
      if(joinPat.isDefinedAt((ch, arg))){// optimization for singleton pattern
        joinPat((ch,arg))
      }else{
        if(ch.isInstanceOf[SynName[Any, Any]]){
          joinPat((names, this.joinPat, (new HashMap[NameBase, Any]+((ch, arg))), 1, new SynName))
          ch.asInstanceOf[SynName[Any,Any]].pushMsgTag(arg)
        }else{
          joinPat((names, this.joinPat, (new HashMap[NameBase, Any]+((ch, arg))), 1, new AsyName))
        }
        names.foreach(n => {
          if(n != ch) n.popArg
        })
      }
    }catch{
      case e:MatchError => {// no pattern is matched
        if(ch.isInstanceOf[SynName[Any, Any]]) {ch.asInstanceOf[SynName[Any,Any]].popMsgTag}
        ch.pendArg(arg)
      }
    }
  }
}
```

The last thing is to define an *and* constructor which combines two or more channels in a join pattern. Indeed, this is surprisingly simple to some extent. Thanks to the syntactic sugar provided by Scala, the infix *and* operator in this library is defined as a binary operator that passes the same argument to both operands.

Listing 10: Code defines the and object

```
object and{
  def unapply(attr:Any) = {
      Some(attr,attr)
  }
}
```

A.2.4 Implementing distributed join calculus

The **DisJoin** class extends both the **Join** class, which supports join definitions, and the **Actor** trait, which enables distributed communication. In addition, the distributed join definition manages a name server which maps strings to its channels. Compared to a local join definition, a distributed join definition has two additional tasks: checking if distributed channels used at a remote sites are annotated with correct types and listening messages sending to distributed channels. The code of the **DisJoin** class is presented in Listing 11.

Listing 11: Code defines the DisJoin Class

```
class DisJoin(port:Int, name: Symbol) extends Join with Actor{
  var channelMap = new HashMap[String, NameBase] //work as name server

  def registerChannel(name:String, ch:NameBase){
    assert(!channelMap.contains(name), name+" has been registered.")
    channelMap += ((name, ch))
  }

  def act(){
    RemoteActor.classLoader = getClass().getClassLoader()
    alive(port)
    register(name, self)

    loop(
      react{
        case JoinMessage(name, arg:Any) => {
          if (channelMap.contains(name)) {
            channelMap(name) match {
              case n : SynName[Any, Any] => sender ! n(arg)
              case n : AsyName[Any] => n(arg)
         }}}

        case SynNameCheck(name, argT, resT) => {
          if (channelMap.contains(name)) {
            sender ! (channelMap(name).argTypeEqual((argT,resT)))
          }else{
            sender ! NameNotFound
          }
        }
        case AsyNameCheck(name, argT) => {
          if (channelMap.contains(name)) {
            sender ! (channelMap(name).argTypeEqual(argT))
          }else{
            sender ! NameNotFound
       }}}
    )
}}
```

In this library, a distributed channel is indeed a stub of a remote local channel. When a distributed channel is initialized, its signature is checked at the place where its referring local channel is defined. Later, when a message is sent through this distributed channel, the message and the channel name is forwarded to the remote join definition where the referring local channel is defined. Consistent with the semantic of distributed join calculus, reduction, if any, is performed at the location where the join pattern is defined. If the channel is a distributed synchronous channel, a reply value will be sent back to the remote caller. Listing 12 illustrates how distributed synchronous channel is implemented. Distributed asynchronous channel is implemented in a similar way.

Listing 12: Code defines distributed synchronous channel

```
class DisSynName[Arg:Manifest, R:Manifest](n:String, owner:scala.actors.AbstractActor){
  val argT = implicitly[ClassManifest[Arg]]//type of arguments
  val resT = implicitly[ClassManifest[R]]//type of return value

  initial()// type checking etc.

  def apply(arg:Arg) :R = synchronized {
    (owner !? JoinMessage(n, arg)).asInstanceOf[R]
  }

  //check type etc.
  def initial() = synchronized {
    (owner !? SynNameCheck(n, argT, resT)) match {
      case true => Unit
      case false => throw new Error("Distributed channel initial error:"+
                                    "Channel " + n + " does not have type "+
                                    argT+ " => "+resT+".")
      case NameNotFound => throw new Error("name "+n+" is not found at "+owner)
}}}
```

Lastly, the library also provides a function that simplifies the work of connection to a distributed join definition.

```
object DisJoin {
  def connect(addr:String, port:Int, name:Symbol):AbstractActor = {
    val peer = Node(addr, port)//location of the server
    RemoteActor.select(peer, name)
  }
}
```

A.3 Limitations and Future Improvements

A.3.1 Assumption on linear pattern

As with most of implementations that support join patterns, this library assumes that channels in each join pattern are pairwise distinct. Nevertheless, the current prototype implementation does not check the linear assumption for better simplicity.

Under the current implementation, a non-linear pattern

- will never be triggered if the channel involves a non-linear channel that takes two or more different messages. For example, the pattern $\{case\ c(1)\ and\ c(2) \Rightarrow println(3)\}$ will never fire.
- will work as a linear pattern if the all occurrences of a non-linear channel could take the same message. In this case, one or more variable names could be used to indicate the same message value. For example, $\{case\ c(m)\ and\ c(n) \Rightarrow println(m+n)\}$ will print 4 when c(2) is called.

A.3.2 Limited number of patterns in a single join definition

Due to the limitation of the current Scala compiler (version 2.9.1), the library also has an upper limit for the number of patterns and the number of channels in each pattern. Although the pattern of this limitation is not clear, the writer observed that a "sorted" join-definition may support more patterns.

For example,

```
case A(1) and B(1) and C(1) => println(1)
case A(2) and B(2) and D(2) => println(2)
```

is a "better" join-definition than

```
case A(1) and B(1) and C(1) => println(1)
case D(2) and B(2) and A(2) => println(2)
```

The compiler error: "java.lang.OutOfMemoryError: Java heap space" usually indicates the above limitation.

A.3.3 Unnatural usages of synchronous channels

Users of the current library may define a join-definition as follows

```
// bad join definition
object myjoin extends Join{
 object A extends AsyName[Int]
 object S extends SynName[Int, Int]
 join {
   case A(1) => S reply 2
   case S(n) => println("Hello World")
 }
}
```

In addition to the linear assumption, the current library further assumes that:

(i) the action part only reply values to synchronous channels that appeared in the left hand side of $\Rightarrow$. For this reason, the first pattern in the above example is invalid.

(ii) all synchronous channels in a pattern, if any, will receive one and only one value when the pattern fires. For this reason, the second pattern in the above example is invalid.

For assumption (i), the writer assumes that a program only needs to send a reply value to a synchronous channel on request. For assumption (ii), we think that all invocations on synchronous channels are expecting a reply value. Unlike some other libraries such as Cω. this library permits multiple synchronous channels in a single pattern.

Violating any of the above assumptions may be accepted by the system but usually causes deadlock or unexpected behaviour at run time.

A.3.4 Straightforward implementation for synchronous channels

Readers may have noticed that the implementation for synchronous channels are implemented according to its straightforward meaning rather than its formal definition in the join-calculus, which translates synchronous channels to asynchronous channels.

Admittedly, the translation in the join-calculus is a clever strategy to mimic the straightforward meaning of synchronous channels with less constructs. As the Scala programming language provides low-level concurrency constructs, we think that a direct implementation for the synchronous channel is easier than and could be consistent with the indirect translation.

A.3.5 Type of the join pattern and the unapply methods

As an implementation in a static typed language, users would expect a clear type for the join pattern and the *unapply* methods in **AsyName**, **SynName**, and the *and* object. If the join pattern has type $\mathbf{T} \Rightarrow \mathbf{Unit}$, then the *unapply* methods in **AsyName** and **SynName** should have type $\mathbf{T} \Rightarrow \mathbf{Option}[\mathbf{Arg}]$, and the *unapply* method in the *and* object should have type $\mathbf{T} \Rightarrow \mathbf{Option}[(\mathbf{T}, \mathbf{T})]$. The unusual implementation that passes partial function to the unapply methods indicates that **T** is a recursive type. Furthermore, due to the optimization for singleton patterns, **T** is also a Either type.

For earnest readers, **T** is $\forall \mathbf{T}.\mathbf{Either}[(\mathbf{NameBase}, \mathbf{Any}), (\mathbf{HashSet}[\mathbf{NameBase}], \mathbf{PartialFunction}[\mathbf{T}, \mathbf{Unit}], \mathbf{HashMap}[\mathbf{NameBase}, \mathbf{Any}], \mathbf{Int}, \mathbf{NameBase})]$. Defining such a complex data type in a separate place may not be more helpful for readers than typing all parameters in each case statement. Moreover, general users do not need to understand this complex type to use this library. For above reasons, we simply replace **T** with the **Any** Type and manually verify type correctness of our implementation.

References

[1] Ericsson AB. Otp design principles user's guide, 2011.

[2] S. Abramsky. What are the fundamental structures of concurrency? we still don't know! In *Algebraic process calculi: the first 25 years and beyond*, BRICS Notes Series NS-05-03, pages 1–5, June 2005.

[3] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[4] J. C. M. Baeten. A brief history of process algebra. Technical report, Theor. Comput. Sci, 2004.

[5] Nick Benton, Luca Cardelli, and Cédric Fournet. Modern concurrency abstractions for c#. *ACM Trans. Program. Lang. Syst.*, 26:769–804, September 2004.

[6] Grard Boudol. The π-calculus in direct style, 1997.

[7] Sebastian Burckhardt, Manuel Fahndrich, Daan Leijen, and Mooly Sagiv. Eventually consistent transactions. In *the 22n European Symposium on Programming (ESOP)*. Springer, 2012.

[8] Luca Cardelli. Obliq: A language with distributed scope. *Computing Systems*, 8:27–59, 1995.

[9] Luca Cardelli and Andrew D. Gordon. Mobile ambients. *Theor. Comput. Sci.*, 240(1):177–213, 2000.

[10] Luis Rodrigo Gallardo Cruz and Oscar Olmedo Aguirre. A virtual machine for the ambient calculus. In *Electrical and Electronics Engineering, 2005 2nd International Conference on*, pages 56–59, 2005.

[11] Silvano Dal-Zilio. Implicit polymorphic type system for the blue calculus. Technical report, 1997.

[12] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41:205–220, October 2007.

[13] Burak Emir, Martin Odersky, and John Williams. Matching objects with patterns. In *ECOOP 2007 Object-Oriented Programming, volume 4609 of LNCS*, pages 273–298. Springer, 2007.

[14] Cédric Fournet, Fabrice Le Fessant, Luc Maranget, and Alan Schmitt. Jocaml: A language for concurrent distributed and mobile programming. In *In Advanced Functional Programming*, pages 129–158. Springer Verlag, 2003.

[15] Cédric Fournet and Georges Gonthier. The join calculus: A language for distributed mobile programming. In *In Proceedings of the Applied Semantics Summer School (APPSEM), Caminha*, pages 268–332. Springer-Verlag, 2000.

[16] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *Proceedings of the 7th International Conference on Concurrency Theory*, CONCUR '96, pages 406–421, London, UK, 1996. Springer-Verlag.

[17] Cédric Fournet, Georges Gonthier, and Inria Rocquencourt. The reflexive cham and the join-calculus. In *In Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385. ACM Press, 1995.

[18] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33:51–59, June 2002.

[19] Philipp Haller and Tom Van Cutsem. Implementing joins using extensible pattern matching. In *Proceedings of the 10th international conference on Coordination models and languages*, COORDINATION'08, pages 135–152, Berlin, Heidelberg, 2008. Springer-Verlag.

[20] Kohei Honda. Types for dyadic interaction, 1993.

[21] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *In ESOP98, volume 1381 of LNCS*, pages 122–138. Springer-Verlag.

[22] Raymond Hu, Olivier Pernet, Dimitrios Kouzapas, Nobuko Yoshida, and Kohei Honda. Type-safe eventful sessions in java.

[23] Typesafe Inc. Akka documentation: Release 2.0-m3, June 2012.

[24] John Launchbury. A natural semantics for lazy evaluation. pages 144–154. ACM Press, 1993.

[25] Robin Milner. Functions as processes. In *Proceedings of the 17th International Colloquium on Automata, Languages and Programming*, ICALP '90, pages 167–180, London, UK, 1990. Springer-Verlag.

[26] Robin Milner. *The Space and Motion of Communicating Agents*. Cambridge University Press, New York, NY, USA, 2009.

[27] Robin Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Journal of Information and Computation*, 100:1–77, 1992.

[28] Tomas Petricek. Reactive programming with events. Master's thesis, Charles University in Prague, 2010.

[29] Peter Van Roy. Convergence in language design: a case of lightning striking four times in the same place. flops. In *in the Same Place, 8th International Symposium on Functional and Logic Programming (FLOPS 2006), April 2006, Springer LNCS*, pages 2–12. Springer, 2006.

[30] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, Cambridge, MA, USA, 2004.

[31] Claudio Russo and Nick Benton. The joins concurrency library. In *In: Proc. PADL*, pages 260–274, 2007.

[32] Davide Sangiorgi. From pi-calculus to higher-order pi-calculus - and back. In *Proceedings of the International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, TAPSOFT '93, pages 151–166, London, UK, 1993. Springer-Verlag.

[33] Davide Sangiorgi and David Walker. *The π-Calculus: A Theory of Mobile Processes*. Cambridge University Press, New York, NY, USA, 2001.

[34] W Vogels. Eventually consistent [distributed system]. *Communications of the ACM*, 52(1):40–4, 2009.

[35] Jim Waldo, Geoff Wyant, Ann Wollrath, and Samuel C. Kendall. A note on distributed computing. In *Selected Presentations and Invited Papers Second International Workshop on Mobile Object Systems - Towards the Programmable Internet*, pages 49–64, London, UK, 1997. Springer-Verlag.