Functions, Concurrency, Distribution and Mobility

Zeliha Dilsun Kırlı

Laboratory for Foundations of Computer Science, The University of Edinburgh
e-mail: zdk@dcs.ed.ac.uk.
tel:+44 131 650 48 89

Abstract

There are efforts within the research area of programming languages to identify and exploit those features of functional languages which could offer a solution to the challenges posed by the complex character of computation in modern networks. These efforts include implementations of languages which are primarily functional but also support concurrency, distribution and mobility. In this paper, we focus on such languages. We present a survey on different approaches to language design and provide a framework to make a systematic comparison of them. The survey briefly introduces frequently cited formal models of distributed and mobile computation. It follows by a closer look at the programming languages Concurrent ML, Facile, PLAN, Erlang and the Join-Calculus Language.

1 Introduction

The desire to exploit the recent developments in telecommunication technology has led to a new vision for computation in modern networks. Computers are now seen as parts of a global computing platform sharing local and remote resources rather than as mainly self-contained computing devices using local resources and occasionally communicating with each other. Therefore, the basic assumptions about distributed computing and programming languages are being revised to allow for better use of the global infrastructure. A consequence of this has been the emergence of the *mobile computation* paradigm, also known as the *mobile agents* paradigm, along with technologies supporting it.

The key characteristic of the mobile computation paradigm is to give programmers control over the mobility of code or active computations across the network. This marks the latest approach to the concept of *mobility* which has been investigated and exploited in different forms over decades. In the late 1970s, mobility was used in the form of process migration, transferring a process between two computers to enable load distribution or fault resilience. In the 1980s, portable computers became available and researchers focused on issues arising from the physical mobility of computers as well as process migration. Providing mobility as a programming language feature, as is done in more recent languages, allows users to write mobile agents which move across the network and carry out tasks on their behalf [1]. An agent can be defined as a whole computational component together with its state, the code it needs, and some resources required to carry out its task. Telescript [2], Aglets [3], Agent TCL [4], Concordia [5], Mole [6], TACOMA [7], Sumatra [8], Ara [9], MOA [10] and Voyager [11] are some of the most recent and advanced enabling technologies for mobile agents.

It is a natural question to ask if or how mobile computation can be smoothly integrated with a previously existing paradigm. By smoothly we mean without having to make significant compromises from the strengths of the paradigm. In this paper, we explore the question for the case of functional programming by surveying some approaches to programming language design issues relevant to mobile computation.

Functional programs are concise, secure and elegant [12]. These qualities arise from the strong points of functional programming languages such as higher-order features, robust type systems and automatic memory management. Despite these strengths, in some cases a different programming paradigm may seem to suit the programming task in hand or a programmer may wish to transcend the features of a purely functional language to make more direct use of computational resources. To address these demands, the integration of functional, imperative, concurrent and distributed paradigms have been explored through implementations of languages such as Concurrent ML [13], Poly/ML [14], Facile [15, 16], Concurrent Haskell [17], Erlang [18], the Join-calculus language [19] and PLAN [20]. Understanding the fundamental ideas of these languages can provide insight into the question of how well the benefits of functional programming languages can carry over to programming languages for mobile computation. In the pursuit of our subject, there are also gains to be made by looking at the general developments in the foundational studies of the field.

In Section 2 we introduce the programming language design issues which set the focus and the scope of our survey. In Section 3 we give a brief overview of the prominent formal models for concurrent, distributed and mobile computation. We follow by examining the programming languages Concurrent ML, Facile, PLAN, Erlang and the Join-calculus language individually, aiming to identify their approaches to the issues introduced in Section 2.

2 Language Design Issues

This section presents some programming language design issues which are essential for mobile computation. The choices made in these design issues greatly influence the way in which a language supports mobile computation. Note that this section covers those issues which appear to be the most crucial from the language design perspective. Other significant issues for mobile computation such as interoperability and efficiency are not discussed in detail.

2.1 Concurrency

In general, concurrency is a desirable programming feature for mainly two different purposes; to facilitate the development of interactive systems and to speed up applications by dividing programs into parts that can be run in parallel. Computation in a network is inherently a concurrent activity. The different nodes of a network can be used to carry out different tasks in parallel.

The specification and creation of *processes* in a concurrent programming language can be either static, where the set of processes is fixed or dynamic where a language feature is provided for creating new processes on the fly. The means of providing *communication* and *synchronisation* are inextricably associated with the discussion of any concurrent activity. Shared-memory languages use a mutable shared state to implement process communication and provide mechanisms to prevent processes from interfering with each other. On the other hand, distributed-memory languages use message passing primitives. A naming convention is adopted to specify both ends of the communication. Process names, ports or channels may be used to designate the communication partner. The degree of synchronisation also varies from language to language; the message passing may be of asynchronous, synchronous or request-reply type.

2.2 Distribution

The physical *distribution* of processes among different nodes of a network may be desirable for several reasons; improving efficiency by making use of multiple processors, increasing reliability and fault tolerance, accessing resources which reside on another node and accommodating the distributed nature of applications such as booking systems and conference systems.

The issues concerning concurrency are valid for distributed systems with the additional complexity of taking into consideration the different physical sites of computation. For example, synchronous message-passing becomes a global consensus problem in a distributed setting. The distribution of processes among different physical locations makes locality an important notion to be addressed carefully by language designers. Localities induced by computations dispersed across several locations are also essential to many failure models used in distributed programming.

2.3 Mobility

The interaction between the computational units moving across the network and the computational environments they are relocated to characterises different forms of mobility. A classification regarding code and execution state management can be made as follows. *Weak mobility* is the ability to transfer code across different computational components where no migration of execution state is involved. Code may be accompanied by some initialisation data. The more difficult case to achieve is *strong mobility* which is the ability to transfer both code and execution state [21].

Data space management is another issue which needs to be resolved by the language design and implementation. Resource relocation and binding reconfiguration are the major problems. When a computational unit moves to a new computational environment, the set of bindings to resources accessible by it must be rearranged. The way of doing this depends on the nature of resources involved and the type of binding to the resources. A survey of the approaches regarding data space management can be found in [21]. A similar survey has been conducted in [22].

2.4 Safety and Security

A major motivation for mobile computation is to make better use of the global computation infrastructure by facilitating the sharing of its resources among mobile computational entities. In order to bring about the desired advantages of mobile computation, safety and security must be taken into consideration with particular emphasis. In general, safety aims at prevention of mistakes and unintended behaviour of programs. Safety is a precondition for security. However, security is concerned with a wider range of issues such as secrecy, integrity and prevention of malicious attacks.

Type systems are well known tools used by several programming languages to enforce type safety. By type safety we mean that programs cannot corrupt the runtime system so that further execution of the program is not faithful to the language semantics [23]. Languages may choose to use static typing, dynamic typing or a combination of these two.

In the context of mobile computation, providing type safety alone may not be sufficient and programming languages may be compelled to employ more heavyweight mechanisms such as those which rely on cryptographic techniques for trust management.

3 Foundational Models

In parallel to the developments in language technology, several foundational models of distributed computation have started to appear. These are generally in the form of process calculi. It is possible to view these calculi as miniature programming languages which are simple enough to highlight points of interest. They either offer improvements over the existing calculi for traditional distributed computation so that the mobile computation paradigm can be accounted for or suggest profoundly different perspectives. In this section, we give an overview of the most frequently cited calculi in the foundational study of mobile computation.

3.1 Calculi for Concurrency

Earlier formalisms of concurrency such as Petri Nets [24], CSP [25] and CCS [26] started by considering static connectivity. CCS and CSP provide an abstract model of computation where the basic resources are communication channels and the basic computation is carried out by input, output actions on these channels. Processes are constructed as combinations of actions by using combinators such as sequencing, composition and choice. The π-calculus [27] followed these formalisms by offering a richer model which includes features such as dynamic process and channel creation, transmission of names and a static scoping discipline. This rich model relies on the basic notion of naming and communication of these names between processes. The π-calculus is often described as a calculus for mobility where the mobility refers to the mobility of the names.

The π-calculus is a candidate for being the canonical calculus for concurrent computation with its expressive power and relatively tractable semantic theory. Since its first presentation several variants of the π-calculus have been proposed such as [28, 29] and their behavioural properties and plausible type systems have been investigated [30]. The main concerns for the calculi of this section, however, have been concepts such as causality, conflict and concurrency between actions. The issues related to physical distribution of processes were left unaddressed.

3.2 Location Calculi

In order to address the issue of physical distribution, many authors choose to select a variant of the π-calculus and to add new features to it. In this way, they exploit the basic powerful notions of the π-calculus and benefit from the theoretical development around it. Another advantage of this approach is that it gives the opportunity to relate the results obtained by different studies in the common framework of the π-calculus.

3.2.1 π_l, π_1, π_{1l}

π_l-calculus was presented by Amadio and Prasad [31] to be able to account for the notions of locality and failure for the programming language Facile. A π_l program consists of a number of processes running on one or more locations where the number of locations can dynamically change due to failure or generation of new nodes. The calculus focuses on making clear the dependence of channels and processes on the nodes where they reside. They take the view that the distribution of processes can be perceived by the absence of certain communication capabilities due to failures.

In a later work [32] Amadio introduces the π_1-calculus and studies its typing system. The main idea is to enforce the unicity of the receptor by typing. The advantage of this approach can be seen in the modelling of the interaction of objects when an object calls the method of another uniquely determined object.

The π_{l1}-calculus further enriches the π_1-calculus by adding explicit locations, failures, mobility of processes and failure detectors.

3.2.2 Join-Calculus

The join-calculus by Fournet and Gonthier [33, 34] can be considered as a variant of the asynchronous π-calculus which focuses on better locality and static scoping principles. It is viable for a realistic distributed implementation as it avoids global consensus for communication. In the join-calculus syntactic restrictions ensure that all channels have a unique location. It is assumed that the underlying network knows the location of every channel.

The conceptual model [33] was obtained by an extension of the generic model of the chemical abstract machine [35]. In a later work the join-calculus was extended with explicit locations and primitives for mobility [34]. The resulting Distributed join-calculus allows the expression of mobile agents moving between different physical sites. A location resides on a physical site and contains a group of processes. It can also be moved to another site taking all its sublocations with it. Join locations can be organised into a tree structure. This feature of the join-calculus offers a simpler model of failure. Section 8 introduces the high-level programming language which is built on the join-calculus.

3.2.3 Dπ, Distributed-π

Dπ, a distributed variant of the π-calculus was presented by Hennessy and Riely [36]. It is different from the calculi presented by Amadio [32] in two major respects. It ignores location failures and restricts communication to be local. Dπ supports global migration of passive code [37]. A series of works on sophisticated type systems for resource access control, safe and secure execution of mobile agents in open systems [36, 38, 39] build upon the model of Dπ.

Distributed π-calculus of Sewell [40] combines the location and migration primitives from the Distributed join-calculus with asynchronous communication in the π-calculus style. It is given a type system in which the input and output capabilities of channels may be either global, local or absent. The issues of global and local subtyping and capability inference for this type system are studied in [40]. Distributed π-calculus supports global migration of active code [37].

A more recent work by Amadio, Boudol and Lhoussaine [41] presents the Receptive Distributed π-calculus which can be seen as a simplification of Dπ

and an extension of the π_1-calculus. The authors show that a simple static analysis ensures the receptiveness of channel names. Together with a simple type system this analysis guarantees a local deadlock-freedom property.

3.2.4 Ambient Calculus

The Ambient Calculus of Cardelli and Gordon [42, 43] is a process calculus which is different in spirit to the π-calculus model of computation. It focuses on process mobility rather than process communication. It attempts to capture the notions of locality, mobility and ability to cross barriers between different administrative domains. The key abstraction is the idea of an ambient.

An ambient is a named location which may contain processes and subambients, and it can move as a unit inside or outside other ambients. Processes within an ambient may cause their enclosing ambient to move. Ambient calculus supports local and asynchronous communication. It can also be considered as supporting local migration of active code [37].

The typing aspects relevant to the control of exchange of values during the communication and control of movement of ambients through other ambients are investigated in [44, 45]. The authors of the ambient calculus have recently been working on a modal logic which can express spatial as well as temporal properties of ambients [46].

3.2.5 Seal (σ) Calculus

The σ-calculus of Vitek and Castagna [47] shares goals with the Ambient Calculus but has design differences. It aims to provide a model for secure distributed applications over large scale open networks such as the Internet. It extends the π-calculus with location mobility and resource access control. Security issues have been emphasised in the development of the σ-calculus to allow context independent proofs of security.

Seals are named, hierarchically-structured locations. A seal can contain a hierarchy of subseals. Communication occurs synchronously over the channels and is restricted to be either local or neighbourly. Seals may be moved over channels.

The mobility of a seal is under the control of its environment. There are mechanisms to control the propagation of names of channels to control external access to local resources. The notion of portal is proposed as the key mechanism to control inter-seal reactions. Unlike many other calculi, resource access issues are addressed by mechanisms different from type systems. The authors state that types appear too rigid a mechanism to model the dynamic resource allocation of real systems. They argue in favour of using types to constrain other behaviours of seals to facilitate proofs of security properties.

3.3 Other Foundational Models

The literature includes many models which contribute to the foundational work on modern and more traditional distributed systems. To list a few, the Actor Model presented in [48], the coordination model Linda [49, 50], A Calculus with Code Mobility [22] and the Spi calculus [51] are other influential works in the field.

4 Concurrent ML (CML)

4.1 Development

Concurrent ML (CML) [13] is a programming language, developed by John Reppy, which integrates high-level abstraction mechanisms with concurrency primitives. It has its roots in PML [52], an ML-like language used in the Pegasus system at AT&T Bell Laboratories. The concurrency primitives of PML were re-implemented on top of Standard ML of New Jersey [53] at Cornell University and this evolved into CML as presented in Reppy's dissertation.

Reppy's s introduction to CML starts with his observation on the breakthrough in sequential programming brought about by the successful exploitation of procedural abstraction and data abstraction. At the time CML was designed, the practice of concurrent programming had mainly been based on low-level, systems programming languages which provided abstractions of hardware whereas Reppy aimed at facilitating programmer-defined abstractions for concurrent programming. His work on CML evolved around the question "What is the right notion of abstraction for concurrent programming"?

4.2 Higher-Order Concurrency

The basic design philosophy of CML is to provide abstraction mechanisms which allow the programmer to organise complex programs in a structured way while also facilitating the expression of concurrency, synchronisation, and choice.

CML supports dynamic process and channel creation. Reppy claims that shared-memory communication is ill-suited for an ML based language because it relies on mutable state and leads to an imperative programming style. He argues in favour of distributed memory (message-passing) communication, synchronous message passing in particular.

The key underlying idea of CML is to separate the operation of synchronisation from the mechanism for describing synchronisation and communication protocols. CML introduces a new abstract type of values called *event*. Events represent potential communication and synchronisation actions. These abstracted actions are performed only when they are synchronised on.

By the introduction of the *event* datatype synchronous operations are elevated to being first-class values. One can draw an analogy between synchronous operations and functions; an event being analogous to a function abstraction and synchronisation being analogous to function application. CML also provides combinators for construction of more complex events from simpler ones.

Property	**Function Values**	**Event Values**
Type constructor	$\rightarrow$	event
Introduction	λ-abstraction	recvEvt, sendEvt, ...
Elimination	application	sync
Combinators	$\circ$, map, ...	choose, wrap, ...

Reppy calls this style of concurrent programming, with the ability to express a wide range of concurrency paradigms by using events and a small set of primitives and combinators, *higher-order concurrent programming*.

An overview of common CML operations is given in Figure 1. The operation guard f creates an event out of the pre-synchronisation action f. When the value

```
type    thread_id
type    'a chan
type    'a event

val   spawn:(unit → unit) → thread_id

val   channel: unit → 'a chan
val   recv: 'a chan → 'a
val   send: ('a chan * 'a) → unit
val   recvEvt: 'a chan → 'a event
val   sendEvt: ('a chan * 'a) → unit event

val   guard: (unit → 'a event) → 'a event
val   wrap : ('a event * ('a → 'b)) → 'b event
val   choose: 'a event list → 'a event

val   sync: 'a event → 'a
val   select: 'a event list → 'a
```

Figure 1: Overview of CML operations

is synchronised on, the guard function f is evaluated and its result is used in the synchronisation. The operation wrap(ev,f) wraps the event value ev with the post-synchronisation function f. When this event is synchronised on, the function f is applied to the synchronisation result of event ev. It should be relatively obvious what the rest of the operations do.

4.3 Generalised selective communication

According to Reppy, support for generalised selective communication is essential for a concurrent programming language. He points out the limitation imposed by the notion of selective communication of CSP [25] in which only input guards are allowed. He attempts to generalise this notion so that both input and output operations are allowed as a guard.

The interplay between abstraction and generalised selective communication has been a key issue influencing the design of CML. The examples in [13] and [54] may convince the reader that the new abstraction mechanism *event* is indeed indispensable for a realistic integration of abstraction with selective communication, hiding the details of the communication protocols while allowing the expression and implementation of selective communication.

4.4 Mobility

Since CML is not intended to be a language for distributed programming, the kind of mobility support we can talk of is a rather restricted form which arises from the fact that channels can be sent along channels. One can view CML channels and the send and recv operations as providing an implementation of a typed version of the π-calculus. The key difference is that in the π-calculus one can send free channel names along channels whereas in CML a channel has to be created in some scope before being sent.

The combination of local scoping inherited from ML let bindings and the mobility of channels leading to scope extrusion allows programmers to express controlled communication of processes in a relatively flexible setting.

4.5 Safety and Security

CML adopts static typing to enforce type safety in the style of Standard ML [55]. The polymorphic type system of ML has been adapted to support polymorphism of channels. The typing scheme of Standard ML of New Jersey which uses weak types is inherited in the implementation of CML. However, the work on the static semantics uses the type system presented in the Definition of Standard ML. In his dissertation Reppy proves a type safety property for a subset of CML which includes the essential concurrency extensions such as channels and events.

4.6 Formal Foundations

Reppy has also presented a dynamic semantics for a small language λ_{cv} that models concurrency features of CML. The type safety result mentioned above has been obtained with respect to the dynamic semantics of λ_{cv}.

The key design decisions of CML were not made to produce a clean algebra but to produce a language to be used in large-scale computer programming. However, it turned out that the algebra of events was observed to have some pleasant properties. An investigation of categorical structure as well as denotational semantics has been developed by Alan Jeffrey [56].

CML has also been studied within the field of static analysis [57, 58, 59, 60].

4.7 Applications and Extensions

The design of CML was motivated by the need to support the programming of interactive applications rather than facilitating programming in a loosely coupled distributed environment or exploiting parallelism in the hardware. Its practicality has been demonstrated by the implementation of a multi-threaded X window system toolkit called eXene and interactive applications on top of eXene.

One can build higher-level linguistic support for distributed programming on top of CML. Distributed ML developed by Krumvieda [61] extends CML with new abstractions to model distributed communication.

5 Facile

5.1 Development

Facile is a higher-order, mostly functional programming language which attempts to encompass functional, imperative, concurrent, and distributed programming paradigm in a single programming language.

The original work on Facile began at the State University of New York in Stony Brook. It focused on the formal foundations of the functional, concurrent language integration [62] and on abstract implementation models. This work was influenced by the work on process calculi such as CCS and CHOCS [26, 63]. Starting in 1991 a group at the European Computer-Industry Research Centre

(ECRC) refined and implemented the language and the Facile Antigua Release was made freely available in 1994. Facile was implemented as an extension of the Standard ML of New Jersey compiler [53]. It was developed further by Knabe to support the mobile computation paradigm [16].

A major principle in the design of Facile is the symmetric integration of different programming paradigms so that every paradigm can use any other paradigm as a subcomponent for its expression. For example, a function may be implemented as a system of communicating processes and the internals of a process may be implemented using functions.

The designers of Facile emphasise the importance of simplicity and coherence of concepts and language constructs. The numbers of concepts and constructs must be relatively few, easy to understand and their meaning must not be too sensitive to their context. Most of the language constructs, except a few which involve behaviours, can be expressed in the spirit of the λ-calculus using function application and values.

Facile adopts the principle of uniform treatment of values from Standard ML. All values are treated equally. For example, scripts, channels, guards, nodes and libraries are all first class values. This principle enables Facile to inherit many of the benefits of Standard ML as well as facilitating the programming of applications which require dynamic connectivity.

5.2 Concurrency

Facile's model of computation depends on several concurrently executing processes. Processes can be created dynamically and they execute by evaluating expressions. The behaviour of a process is syntactically described by a behaviour expression. Essentially behaviour expressions are not expressions because they do not denote values. The simplest behaviour expression is **terminate** which denotes a dead process. The other basic form of behaviour expression is **activate** exp where exp evaluates to a process *script*. A script can be thought of as the code executed by a process. The language provides constructs, script and **activate** for converting a behaviour expression into a script and vice versa. Behaviour expressions also include parallel composition of behaviour expressions and the non-deterministic choice.

Processes communicate over synchronous channels. Processes have no identity except for the channels they can use in communication. Any value which can be defined in the language can also be communicated over channels. Process scripts and channels are first-class values. They can be passed as arguments, returned as values of expressions and can be communicated over channels.

We choose to give an overview of Facile by means of an example. The basic operations are similar to those of CML except for the characteristic event synchronisation mechanism described in Section 4. Figure 2 and Figure 3 below show different implementations of the same function illustrating the fact that Facile can support different programming approaches.

5.3 Distribution

To address the locality of processes the notion of *node* has been introduced. A Facile system can be viewed as a collection of nodes each of which host a number of processes. A node corresponds to a virtual processor with an address space.

```
proc fib_server(a,b) = let fun fib(i) = if (i = 0) or (i = 1) then 1
                                        else fib(i-1) + fib(i-2)
                       in b ! (fib(a?))
                       end;
                       terminate
```

Figure 2: Processes use functions

```
proc fib_server(a,b) =
    let fun fib(i) = if (i = 0) or (i = 1) then 1
                     else
                         let   val (in1,out1) = (channel(int), channel(int));
                               val (in2,out2) = (channel(int), channel(int));
                         in
                               spawn(out1 ! (fib(in1 ?)); terminate);
                               spawn(out2 ! (fib(in2 ?)); terminate);
                               (in1 ! (i-1));
                               (in2 ! (i-2));
                               ((out1 ?) + (out2 ?))
                         end
    in b ! (fib(a?))
    end;
    terminate
```

Figure 3: Functions use processes

Nodes can be created dynamically and may reside on different computers in a network. The language also provides the constructs r_spawn and r_channel to create processes and channels at specific nodes.

Since the implementation of the choice operator of CCS leads to problems in a real distributed setting, Facile adopts a different version of the choice operator which is discussed in detail in [62]. Facile also provides some general constructs to implement delay and time-out mechanisms to get around the problems posed by blocked communications. To define the semantics of these constructs the designers of the language follow the developments in timed process algebra.

5.4 Mobility

The fact that functions are first class values means that we can create functions at run time, apply them to arguments, pass them to other functions as arguments and receive them as results. We can also transmit them over communication channels. All these properties imply that mobile agents have a natural representation as functions in Facile.

However, there are other requirements for Facile to be generally accepted as a mobile computation language for the global computing platform. One such requirement is the ability to deal with heterogeneity of network nodes. Different nodes may be of different architectures and therefore support different value representations. Knabe demonstrates different approaches for dealing with this issue and implements a language which can be classified as a weakly mobile language [16]. He also develops a way to combine strong typing, remote resource access and independent compilation which are desirable properties for a language for mobile computation.

5.5 Safety and Security

Facile adopts static typing in the style of ML. The original definition of Facile presents a monomorphic type system for Facile. However, the implementation adopts the weak polymorphic type system of Standard ML of New Jersey [53]. The authors have also presented a polymorphic sound type and effect system for a subset of Facile along with a type inference algorithm [64].

5.6 Formal foundations

A clean and well understood semantics has been the main motivation from the very early days of Facile. This has lead to a number of works for the formal foundations of Facile such as [62, 65, 66]. It continues to be of interest to researchers of process calculi.

5.7 Applications

The focus of Facile is to provide a well-founded platform to support the development of end-user applications rather than system software. An effort has been made to enable Facile implementations in a network of heterogeneous platforms. Interfaces with foreign environments have been developed.

Facile has been used in the implementation of the user interface of Calumet, a cooperative work application which supports teleconferencing [67]. Einrichten is another medium scale application which experiments with Facile. It was developed by ECRC as a collaborative interior design system [68]. Finally, to study the feasibility of mobile agent based computation in Facile, a technology called Mobile Service Agents has been developed [69].

6 Packet Language for Active Networks (PLAN)

6.1 Development

PLAN (Packet Language for Active Networks) [20] is a domain specific, simple functional language for programs which form the packets of an active network. It is based on a subset of ML with some primitives to express remote evaluation. It is being developed as a part of the SwitchWare project [70] at the University of Pennsylvania which is one of the prominent projects within the area of active network research [71].

The concept of active networking has been motivated by the desire to bring programmability to networks. Active networks are *active* in the sense that switches perform customised computations on the packets flowing through them. This can be contrasted with the approach adopted in traditional networks where the nodes transport data from one end system to another passively, computation being limited to header processing and signalling in packet-switching and connection-oriented networks respectively. The principal advantages of active networking are to be seen in the enabling of adaptive protocols, implementation of application-specific functions at strategic points within the network and the deployment of new services at a faster pace [71].

The SwitchWare project has been exploring how to make the network programmable by allowing switches to be dynamically extended with new services and by allowing packets themselves to be programs. The idea of packets as programs is being explored through the design and implementation of PLAN. The SwitchWare architecture is based on three layers. The top layer consists of active packets which are mobile entities containing both code and data which replace the header and payload of conventional packets. The middle layer consists of extensions which may be dynamically loaded or which can be part of the basic functionality of a switch. They communicate with other switches by using active packets. The lowest layer is static and provides a secure foundation for the layers above itself.

The active packet layer is intended for high-level control while the complex functionality resides in services provided by the middle layer. Thus, PLAN was designed to support lightweight programmability for packets while also providing a scripting language for general services which may employ heavyweight computations. It is a small language with a resource-limited semantics. The expressiveness is limited to allow active nodes to evaluate PLAN programs without the need for authentication. All programs are guaranteed to terminate by limiting the resources consumed at each node. Packets and their descendents can visit only a fixed number of nodes. The designers of PLAN also put emphasis on the language being sufficiently well defined so that advances in type theory, programming language semantics and formal methods can be exploited to address issues related to safety and security. The most recent implementation of PLAN has been carried out in Objective CAML [72].

6.2 Concurrency and Distribution

As a language designed specifically for active networks, PLAN supports concurrent and distributed execution of programs carried in active packets. It is a purely functional language and its packets do not communicate with each other. This ensures non-interference among concurrently executing programs. Service layer extensions may be written in other general purpose languages and so can introduce possibilities for communication.

A PLAN application consists of a series of PLAN packets which comprise a task. A host application, which is a program that runs on an end-node in the network and which may be written in another language, constructs a PLAN packet and injects it into the active network through a port connected to the local PLAN interpreter.

Packets

A PLAN packet encapsulates a *chunk (code hunk)* and the fields *evaluation destination, resource bound, routing function name*, *source* and *handler*.

<table>
<tr><td colspan="8">Packet</td></tr>
<tr><td colspan="3">chunk</td><td rowspan="2">evalDest</td><td rowspan="2">RB</td><td rowspan="2">routFun</td><td rowspan="2">source</td><td rowspan="2">handler</td></tr>
<tr><td>code</td><td>entry point</td><td>bindings</td></tr>
</table>

A chunk is composed of three components: PLAN code, a function name to serve as an entry point and values to serve as bindings for the function's arguments.

The code consists of a series of definitions which bind names to functions, values and exceptions where the names of the services available at the node of definition form the initial bindings in the namespace. The arguments are evaluated locally in a call-by-value fashion and the actual evaluation of the function call is delayed until the packet arrives at its destination. It is invoked in an environment where all top-level bindings are available. This is the point where PLAN departs from the discipline of static scoping which it adopts elsewhere.

Chunks are first-class data values which can be passed as arguments, returned from functions, can be stored in data structures and manipulated as bags of bits. One can force the execution of a chunk by the core service eval.

The roles of the remaining fields of a packet are as follows. The routing function serves to define how the packet will be transported from the current node to the evaluation destination. The resource bound sets the limit on the number of hops the packet or any of its descendents can make. This restricts the global network resource usage of a PLAN application. The source field names the packet's oldest ancestor and the handler field provides the name of a service routine on the source which will handle certain error communications.

Network Primitives

PLAN programs create new packets through calls to the network primitives OnRemote and OnNeighbour. The call OnRemote(C,evalDest, Rb, routFun) creates a new packet which will evaluate the chunk C on node evalDest. As we have noted above, the bindings of the chunk are determined locally while the function application is evaluated remotely. The arguments Rb and routFun correspond respectively to the resource bound and the routing function name fields of the created packet. Until it reaches its evaluation destination, Rb is decremented by one at each hop and the packet is terminated if the resource bound is exhausted. The network primitive OnNeighbor is similar to OnRemote the difference being that the created packet must execute on a neigbour of the current node.

Services

PLAN programs can call core services which are present on all active nodes in the same way as they call locally defined functions. Core services are guaranteed to terminate. The services thisHost, getHostByName, getNeighbors, getRB, defaultRoute, print are examples of core services presented in the Specification of PLAN [73] to be provided as standard library functions. In addition to these, there are a number of service packages which extend the functionality of PLAN programs. For example, the service package resident enables PLAN programs to leave state on the nodes they visit to facilitate exploring the network topology. Note that service packages such as these may create the possibility for unsafe operations and therefore PLAN may have to impose certain safety and security requirements to permit their employment. This will be discussed in more detail below.

6.3 Mobility

In Section 6.2 we have introduced the execution model for PLAN programs in an active network. Mobility of PLAN packets and the remote evaluation of chunks encapsulated in these packets is at the heart of the execution of PLAN programs. Given its domain specific approach, it is not straightforward to compare PLAN with general purpose languages to classify the kind of mobility it supports. Nevertheless, we consider PLAN to be strongly mobile due to the fact that PLAN programs are able to initiate their own evaluation at a remote site as well as taking with them a collection of resources which they may need at that remote site.

6.4 Safety and Security

Expressiveness and Resource Limitations

PLAN has a limited set of simple constructs for flow of control. It supports statement sequencing, conditional execution, iteration over lists by folding and exceptions in the style of ML. Recursive functions and unbounded iteration are not allowed to ensure the termination of programs. Besides these limitations on its expressiveness, we have also noted its resource limited semantics. All these restrictions are intended to enforce safety and security in a simple way. Indeed, pure PLAN programs, which use core services only, can run with no need for authentication.

Type System

PLAN is strongly typed which implies that well typed programs cannot go wrong. It requires that programs are statically typeable but it also allows dynamic type checking. A discussion about the relative merits of static and dynamic type checking for PLAN can be found in [73].

Heavyweight Mechanisms

The safety and security of pure PLAN programs can be ensured by the mechanisms presented above. However, PLAN programs can also call service routines which are written in general purpose languages. This constitutes a potential threat to the safety and security of the system.

To make service calls safe, the pure part of PLAN has been complemented with a system of trust management [74]. According to this system, each node administrator creates a policy which restricts the use of unsafe services to selected users through a process of authorisation. Packets are then required to authenticate themselves before accessing the priviliged services. The technique employed by PLAN, which is based on expanding or contracting a packet's service environment based on its level of privilege, is called *namespace-based security*.

6.5 Formal Foundations

PLAN has recently been provided with a specification which aims to define a mathematically precise operational semantics [73]. It is intended to set a standard for implementations and to support proofs of key properties of PLAN which all conformant implementations must obey.

The designers of PLAN have aimed to keep the language simple. It employs the λ-calculus as its core and adopts many features of the programming language ML which is known for its well-defined foundations. Hence, it is possible to exploit much of the existing work in programming language theory. However, the complexity of service extensions poses challenges for safety and security requirements. The formulation and verification of these remains as further investigated.

7 Erlang

7.1 Development

Erlang [18] grew out of the experiments conducted by the Ericsson Computer Science Laboratories to facilitate development of telecommunication systems with less effort and fewer errors. Telecommunication systems pose a range of requirements which are difficult to deal with by conventional programming languages such as the requirement for introducing software upgrades without interrupting the execution of the system and fault-tolerance.

Erlang was built upon ideas from logic and functional programming communities and programming languages such as Chill and Ada which have industrial applications in programming control systems. In 1999, an open-source Erlang was released. It contains the implementation of Erlang as well as a large part of Ericsson's middleware for building distributed high-availability systems [75].

7.2 Concurrency

Erlang supports a process-based model of concurrency which has asynchronous message-passing. Processes can be created dynamically by the primitive spawn. The expression spawn(Module, Function, ArgList) creates a process which will evaluate the function in a specifed module and returns the identifier of the newly created process. Note that Erlang uses a call-by-value semantics in function application. A process identifier is a first-class object and can be manipulated like any other object. Process identifiers in Erlang are necessary for specifying the communication partner.

Message passing is the only form of communication in Erlang. A message can be any Erlang value and can be sent to another process whose process identifier is known. The expression pid ! message causes the value denoted by message to be sent the process with the identifier pid. The primitive receive is used to receive messages.

```
receive
    Message1 [when Guard1] → Actions1;
    Message2 [when Guard2] → Actions2;
    ...
end
```

Each process has a mailbox and all messages sent to the process are stored in the mailbox in the same order as they arrived. The first message in the mailbox is matched against the patterns in the receive construct, if a match is found the message is removed from the mailbox and the corresponding actions are evaluated. Otherwise, the other messages in the mailbox are tried in the same fashion. The unmatched messages remain in the mailbox in the same order as they were stored to be matched upon the next receive. Optionally, the receive primitive can be augmented with a timeout.

7.3 Distribution

The notion of *node* abstracts an Erlang virtual machine. A distributed Erlang system is a network of Erlang nodes where a node corresponds to a processor. The primitive r_spawn is used to create a process on a specific remote node: r_spawn(Node, Mod, Func, ArgList). Almost all operations allowed on process identifiers are also allowed on remote process identifiers.

7.4 Mobility

Erlang does not support mobility in any of the senses described in Section 2. However, there have been some attempts to extend Erlang in the direction of mobility support [76, 77].

In Erlang, there is no direct way for a process to move. If a process wants to move it may spawn a new process on a remote node and then terminate on the local node. The state is carried over through the arguments to the spawned process. The code to be executed on the remote node, which is a function in a module, must exist in the remote system's default search path. Furthermore, the modules referenced in the loaded module are also resolved on the remote system.

SafeErlang [77] extends Erlang with a remote loading mechanism so that the code no longer needs to be available on the local system but can be loaded from an arbitrary place. Therefore SafeErlang can be considered to support weak mobility. In SafeErlang, module identifiers (Mid) are introduced to serve as packaging of code moved around the distributed system. They store information on the origin and the representation of the code.

7.5 Safety and Security

Erlang adopts dynamic typing. There have been attempts to develop soft type systems so that some advantages of static typing can be enjoyed by Erlang programs. Various publications on type systems for Erlang can be reached through Erlang's web site [75].

The authors of SafeErlang [77] and SSErl [78] identify the features of Erlang which create major security holes for Erlang systems in the case of computation in an open system. These are mainly due to the lack of control over resources of the system and the lack of hierarchy among the nodes. SafeErlang attempts to deal with these issues by introducing the concepts of *capability* and *subnode*. SSErl is a system with similar goals to SafeErlang. The author of SSErl states that SafeErlang has a stronger focus on code mobility whereas his concern is a simpler and more comprehensive implementation of capabilities and subnodes.

A capability is a globally unique, unforgeable name for some resource such as a module, node, pid or port and a list of operations permitted on that resource by holders of the capability such as exit, link, register, restrict and send. A capability is created and verified upon use on the node which manages the referenced resource.

In order to allow a hierarchy within a system, so that custom context of services become available, SafeErlang enables the creation of subnodes with a primitive. The expression subnode(Name, Options) returns a reference to a new node containing a full capability. It can be used to manipulate the node and spawn processes on it. The Options can be used to restrict the system resources the node is allowed to use.

7.6 Code Replacement

The target application domain for Erlang was telecommunications. Code replacement during operation is a common requirement in large soft real-time control systems which typically use long-lived computations. This section is about the language feature of Erlang which facilitates code updates without interrupting the execution of the system.

In Erlang, the use of an explicitly qualified name (M:f) causes the function f in a module M to be dynamically linked into the run-time code. Every time a function call is made using this notation, the system will evaluate the function using the latest version of the code. For example if the user wishes to replace the code for module M without interrupting its execution, recompiling and loading the new code for module M will cause the new code for function f to take effect upon its next application.

7.7 Formal Foundations

It is reported on the Web pages of the Ericsson Computer Science Laboratory that a specification of the Erlang programming language is being developed. It is stated that the intention is not to write a formal definition like the Definition of Standard ML but a specification close in style to that of Java and Common Lisp.

There is a research group working on the verification of Erlang programs. The goal of the project is stated as producing a method and a prototype tool-set for verification. One of the publications related to this project is [79]. In this paper a logic and a proof system is introduced for proving properties of open distributed systems. A core fragment of the Erlang programming language is used as a realistic setting to demonstrate the results. Erlang is considered as a first-order actor language with datatypes, buffered asynchronous communication and dynamic process spawning.

8 The Join-Calculus Language

8.1 Development

The join-calculus language [19] is an experimental high-level language based on the process calculus with the same name presented by Fournet and Gonthier in [33] and its distributed and mobile extension presented in [34]. Two implementations of the join-calculus have been carried out in the Objective CAML [72] environment. The first one, Join-Calculus language system, contains a compiler and an interpreter written in Objective CAML. The language is the join-calculus extended with basic types where most of the Objective-CAML libraries appear as primitives. An interface to Objective-CAML modules has also been provided. The second one, JoCaml System, is based on the standard Objective CAML distribution extended with a join-calculus library. The language is Objective CAML extended with join-calculus definition and locations. Although we focus on language features in general which are not implementation specific, this section should be considered to refer to the Join-Calculus language system when there is any scope for confusion.

The original work on the join-calculus was motivated by the observation of the gap between theory and practice of concurrent computation in distributed systems. According to the authors, the existing calculi provided elegant theories however they overlooked implementation issues. On the other hand, the large set of constructs found in programming languages for building concurrent distributed applications was an obstacle to their theoretical investigation. The design of the join-calculus was an attempt to provide a simple formal model of concurrent, distributed computation which can also be used a foundation for a practical programming language suitable for computation in modern networks.

The higher-level programming language has been derived directly from the join-calculus by providing some syntactic convenience.

8.2 Concurrency

Join-calculus programs are composed of processes and expressions. Processes are executed asynchronously for their side effects and they return no value. Expressions are evaluated synchronously to yield a result. Processes are created dynamically by the spawn construct and expressions are introduced by the do construct. Processes communicate by sending messages over channels.

Channels are divided into two categories: synchronous and asynchronous. Synchronous channels return results while the asynchronous ones do not. A characteristic feature of the join-calculus is that channels and the processes which listen on these channels are defined by a single construct. This permits implementing functions by channels. Users can create new channels with a **let** construct. Note that this is not the same kind of **let** binding familiar from functional programming. For example, assuming that echo was previously defined, **let** echo_twice(x) = echo(x) | echo(x) defines a new channel echo_twice. Sending a value on echo_twice would cause the process (echo(x) | echo(x)) to be fired where x is bound to the value sent. Activating the definition by the statement spawn echo_twice("b") would cause "bb" to be put on the screen.

Synchronous names can be used to support a functional programming style as in the following example. The synchronous channel fib can be considered as implementing the Fibonacci function.

```
let fib(n) = if n <= 1 then reply 1
             else reply fib(n-1) + fib(n-2)
```

As can be seen in the above examples, the channels of the join-calculus have a different character than those of many process calculi such as the π-calculus. The join-calculus channels are not bi-directional, they can only send messages. This is due to the fact that the definition of the receptor process and the definition of the channel receiving the input for the process coincide. Note that bi-directional processes can be encoded in the join-calculus [19].

8.3 Join-patterns

Join-patterns extend the key mechanism of channel definitions presented above. It is possible to define several channels simultaneously by using join-patterns.

To trigger the following print_string process, messages must be sent both on fruit and cake: **let** fruit(f) | cake(c) = print_string(f ↑" "↑ c)

One can observe from the example above how join-patterns can be used to express non-determinism. Sending several messages by the following statement could cause possible different different combinations to be output.

```
spawn fruit("apple") | fruit("pear") | cake("pie") | cake("crumble")
```

By using composite join-patterns different synchronisation patterns can be specified. For more examples the user is referred to [19]. Briefly, join-patterns are a very powerful mechanism to encode data structures, control structures for concurrency such as locks and control structures for general purpose programming such as loops.

8.4 Distribution

The join-calculus uses asynchronous message-passing which is the basic operation provided in most distributed systems. This fact facilitates its distributed implementation.

The execution of join-calculus programs can be distributed over numerous heterogeneous machines. At any time, a process or an expression runs on a given machine. Resources such as definitions and processes are never silently relocated by the system. The next section will address the facility for migration of computational units under the language control.

The scope for defined names and values is independent of the localisation. This implies that when a channel name is known it can be used without knowing whether it is local or remote. Locality is transparent in this sense. Programs can be written without being concerned about where their parts will actually be executed.

8.5 Mobility

The previous section on distribution has drawn a picture in which the localisation of processes and expressions is highly static. For all the reasons which motivate the mobile computation paradigm, it is desirable to have control over the adjustment of localities. The join-calculus language supports mobility in the strong sense described in Section 2.

A unit of locality called *location* is introduced. A location contains a collection of definitions and processes. Every location is given a name and location names are first class values. Locations can move from one place to another while processes and definitions are statically attached to their locations. As a result of the migration the location becomes a sublocation of its destination. Distributed computations can be organised as trees of nested locations. Migrations are triggered by a process inside a location by means of a go statement.

Several examples on mobility can be found in [19]. The following example given in Figure 4 is included to give an idea about how mobile agents can

```
loc here end                                  (* client side *)
do ns.register("here",here)

loc mobile                                    (* server side *)
  init
    let here = ns.lookup("here")
    in go(here);
    let sqr = ns.lookup("square")
    in
      let sum(s,n) = reply (if n=0   then s else sum(s+sqr(n), n-1))
      in let result = sum(0,5)
         in print_string("q:sum(5) = "string_of_int(result));
  end
```

Figure 4: Mobile Agent

be programmed in the join-calculus language. Suppose that the purpose is to improve the efficiency of the function sum which uses a function called sqr. A reasonable way of achieving this would be to move to the place of the sqr function and call it locally. On the side of the sqr function a new location called "here" can be created and registered on the name-server. On the client side another location called mobile can created so that it gets the name "here" and migrates there.

8.6 Safety and Security

The join-calculus has a static type system which follows closely that of Standard ML. A new type generalisation criterion which extends the polymorphism of ML to join-definitions has been established in [80]. The authors prove the correctness of their type system with respect to a chemical semantics and also relate typed extensions of the core join-calculus to functional languages.

Furthermore, Abadi, Fournet and Gonthier investigate the secure implementation of channel abstractions on a variant of the join-calculus in [81].

8.7 Formal Foundations

As we have already noted, the join-calculus language is an experimental language. Although it has features such as a modules system and garbage collection which encourage the development of practical applications, the sound theoretical basis it provides attracts more attention. It is a direct descendant of the process calculus with the same name which was introduced in Section 3. The join-calculus is being referred to by many calculi and programming language designers as a departure point for their work or for comparison. There are numerous works and results with theoretical aspects. Translations from the join-calculus to other calculi and vice versa, notions for behavioural equivalences and investigation of proof techniques are examples of such works [33, 34].

A very recent work by Fournet and Schmitt [82] focuses on the implementation of ambients from the Ambient Calculus [42] in the JoCaml system. This work includes a translation scheme from the Ambient Calculus to the join-calculus while providing a comparison between the two calculi.

Language	CML	Facile	PLAN	Erlang	Join-Calculus
Process Creation	*dynamic*	*dynamic*	*dynamic*	*dynamic*	*dynamic*
Communication	*sync.*	*sync.*	*No*	*async.*	*async.*
Distribution	*No*	$\surd$	$\surd$	$\surd$	$\surd$
Mobility	*channel*	*weak*	*strong*	*weak*	*strong*
Static Typing	$\surd$	$\surd$	$\surd$	*soft*	$\surd$
Dynamic Typing	*No*	*No*	$\surd$	$\surd$	*No*

Figure 5: Summary

9 Conclusions

We have paid particular attention for our survey to include representatives of different existing approaches to language design. When we were faced with a choice between two languages, we were biased for the one which has relatively better understood formal foundations. Therefore, most of the languages we have considered have close links with the functional language ML which has a complete formal definition [55].

Figure 5 summarises the approaches taken by different language designs with respect to the issues which we have considered in detail throughout this paper. Our concise reading of this picture is as follows. All of the languages we have considered support dynamic creation of concurrent computational units indicating that this is a requirement to accommodate the dynamics of modern networks. The means of communication remain to be an issue of choice depending on the targeted application domain of the language. Increasing demand for control over the mobility of computations can be observed by inspecting the chronological order of the developments around these languages. We should also note that the difficulty of achieving strong mobility without compromising expressiveness, safety or security is a commonly agreed fact. Type systems are seen as useful tools for providing safety. The guarantees offered by static typing seem to be attractive to all, however some languages find flexibility equally important and seek to reconcile the advantages of the two.

Acknowledgements

I am grateful to Stephen Gilmore for his guidance in the process of writing this paper. I also would like to thank Jane Hillston whose comments helped me improve various points. My research is funded by a studentship from the University of Edinburgh.

References

[1] D. Milojičić, F. Douglis, and R. Wheeler, editors. *Mobility.* ACM Press, 1999.

[2] J. E. White. Telescript Technology:Mobile Agents. In D. Milojičić, F. Douglis, and R. Wheeler, editors, *Mobility*, chapter 14, pages 460–493. ACM Press, 1999. Also appeared in J. Bradshaw editor, *Software Agents* AAAI Press/ MIT Press, 1996.

[3] D. Lange and M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison Wesley Longman, Reading, MA, September 1998.

[4] R. Gray, D. Kotz, G. Cybenko, and D. Rus. D'agents: Security in a multiple-language, mobile agent system. In G. Vigna, editor, *Mobile Agent Security*. Springer Verlag, 1998.

[5] A. Castillo, M. Kawaguchi, N. Paciorek, and D. Wong. Concordia as enabling technology for cooperative information gathering. In *Proceedings of the Japanese Society for Artificial Intelligence Conference*, pages 280–283, June 1998.

[6] J. Bauman, F. Hohl, K. Rothermel, and M. Straßer. Mole-concepts of a mobile agent system. *World Wide Web*, 1(3):123–137, September 1998.

[7] F. B. Schneider. Towards fault-tolerant and secure agentry. In *Proceedings of the Eleventh International Workshop on Distributed Algorithms*, September 1997.

[8] M. Ranganathan, A. Acharya, S.D. Sharma, and J. Saltz. Network-aware mobile programs. In *Proceedings of the USENIX 1997 Annual Technical Conference*, pages 91–103, January 1997.

[9] H. Peine. Security concepts and implementation for the Ara mobile agent system. In *Proceedings of the Seventh IEEE Workshop on Enabling Technologies:Infrastructure for Collaborative Enterprises*, pages 236–242, June 1998.

[10] D. S. Milojičić, D. Chauhan, and W. laForge. Mobile objects and agents (moa), design, implementation and lessons learned. *IEE Distributed System Engineering*, 5:1–14, 1998.

[11] G. Glass. Objectspace Voyager core package technical overview. White Paper. available from `http://www.objectspace.com`.

[12] S. Gilmore. Programming in Standard ML'97: A tutorial introduction. Technical Report ECS-LFCS-97-364, Laboratory for Foundations of Computer Science, 1997.

[13] J. Reppy. *Higher-order Concurrency*. PhD thesis, Department of Computer Science, Cornell University, 1992.

[14] D. Matthews. *ML with Concurrency: Design, Analysis, Implementation, and Application*, chapter 3, pages 31–58. Springer, 1997. Also available as a University of Edinburgh Technical Report ECS-LFCS-91-174.

[15] B. Thomsen, L. Leth, S. Prasad, T. M. Kuo, A. Kramer, and F.C. Knabe. Facile Antigua Release programming guide. Technical Report ECRC-93-20, Europian Computer Industry Research Centre, Munich, Germany, Dec. 1993.

[16] F.C. Knabe. *Language Support for Mobile Agents*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, Dec. 1995.

[17] S. L. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages (POPL '96)*, pages 295–308, Florida, January 1996.

[18] J. Armstrong, M. Williams, C. Wikstrom, and R.Virding. *Concurrent Programming in Erlang*. Prentice Hall, 1993.

[19] C. Fournet and L. Maranget. *The Join-Calculus Language release 1.04*. Institut National de Recherce en Informatique et Automatique, 1997. Available from `http://pauillac.inria.fr/join/`.

[20] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. PLAN: A Packet Language for Active Networks. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming Languages*, pages 86–93. ACM, 1998.

[21] A. Fugetta, G. P. Picco, and G. Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24, 1998.

[22] T. Sekiguchi and A. Yonezawa. A calculus with code mobility. In H. Bowman and J. Derrick, editors, *Formal Methods for Open Object-Oriented Distributed Systems (FMOODS)*, volume 2. Chapman and Hall, 1997.

[23] A. Appel. A critique of Standard ML. *Journal of Functional Programming*, 3(4):391–430, 1993.

[24] C. A. Petri. Fundamentals of theory of asynchronous information flow. In *Proceedings of IFIP Congress '62*, pages 386–390. North Holland, 1962.

[25] C. A. R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, 1985.

[26] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[27] R. Milner, J.Parrow, and D. Walker. A calculus of mobile processes I and II. *Information and Computation*, 100:1–72, 1992.

[28] K. Honda and M. Tokoro. An object calculus for asynchronous communication. In P. America, editor, *Proceedings of ECOOP '91*, pages 133–147, 1991.

[29] D. Sangirogi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis, University of Edinburgh, 1992.

[30] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. In *23rd ACM Symposium Principles of Programming Languages*, pages 358–371, 1996. Full version to appear in *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1999.

[31] R.M. Amadio and S. Prasad. Localities and failures. In *Proceedings of 14th FST and TCS Conferences*, number 880 in LNCS, pages 205–216. Springer-Verlag, 1994.

[32] R. Amadio. An asynchronous model of locality, failure, and process mobility. In *Proceedings of COORDINATION '97*, number 1282 in LNCS. Springer Verlag, 1997.

[33] C. Fournet and G. Gonthier. The reflexive CHAM and the join-calculus. In *Proceedings 23rd ACM Symposium on Principles of Programming Languages (POPL '96)*, pages 372–385, Florida, Jan 1996. ACM.

[34] C. Fournet, G. Gonthier, L. Maranget, and D. Remy. A calculus of mobile agents. In *Proceedings of CONCUR '96*, number 1119 in LNCS, 1996.

[35] G. Berry and G.Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1):217–248, 1992.

[36] M. Hennesy and J. Riely. Resource access control in systems of mobile agents. In *Proceedings of 3rd International Workshop on High-Level Concurrent Languages (HCLC '98)*, volume 3, 1998.

[37] M. Hennessy. A survey of location calculi. Proceedings of the Fifth International Workshop on Expressiveness in Concurrency, 1998. Invited Talk.

[38] M. Hennesy and J. Riely. Type-safe execution of mobile agents in anonymous networks. In *Secure Internet Programming: Proceedings of 4th Workshop on Mobile Object Systems*, volume 1603 of *LNCS*, pages 95–117, Brussels, 1998. Springer-Verlag.

[39] M. Hennesy and J. Riely. Trust and partial typing in open systems of mobile agents. In *Proceedings of 26rd ACM Symposium on Principles of Programming Languages (POPL '99)*, pages 93–104, San Antonio, Texas, January 1999.

[40] P. Sewell. Global/local subtyping and capability inference for a distributed π-calculus. In S. Skyum K. G. Larsen and G. Winskel, editors, *Proceedings of ICALP '98*, volume 1443 of *LNCS*. Springer Verlag, 1998.

[41] R. Amadio, G. Boudol, and C. Lhoussaine. The receptive distributed π-calculus. To appear in the Springer LNCS as Proc. of FST-TCS 99,Madras, 1999. Presented at the 5th Mobile Object Systems Workshop, Lisbon.

[42] L. Cardelli and A. Gordon. Mobile ambients. In M. Nivat, editor, *Foundations of Software Science and Computational Structures*, number 1378 in LNCE, pages 140–155. Springer-Verlag, 1998.

[43] Luca Cardelli. Abstractions for mobile computation, 1998. Available from `http://www.luca.demon.co.uk`.

[44] L. Cardelli and A.D. Gordon. Types for mobile ambients. In *Proceedings of 26rd ACM Symposium on Principles of Programming Languages (POPL '99)*, pages 79–92. ACM Press, 1999.

[45] L. Cardelli, A.D. Gordon, and G. Ghelli. Mobility types for mobile ambients. In *Proceedings of ICALP'99*, volume 1644 of *LNCS*, pages 230–239. Springer Verlag, 1999.

[46] L. Cardelli and A. Gordon. Anytime, anywhere: modal logics for mobile ambients. To appear, 1999.

[47] J. Vitek and C. Castagna. Seal: A framework for secure mobile computations. To appear in LNCS series on Internet Programming Languages, 1999. Available from `http://www.cs.purdue.edu/homes/jv/publist.html`.

[48] G. Agha. *Actors — A model of concurrent computation in distributed systems.* The MIT Press, 1986.

[49] R. De Nicola, G. Ferrari, and R. Pugliese. Coordinating mobile agents via blackboards and access rights. In *Proceedings of COORDINATION'97.* Springer Verlag, 1997.

[50] R. De Nicola, G. Ferrari, R. Pugliese, and B. Venneri. Types for access control. To appear in Theoretical Computer Science, Submitted in 1998.

[51] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols. In *Proceedings of the Fourth ACM Conference on Computer and Communications Security*, Zurich, 1997. ACM Press.

[52] J. H. Reppy. Synchronous operations as first-class values. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 250–259, 1988.

[53] A. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[54] J. Reppy. *ML with Concurrency: Design, Analysis, Implementation, and Application*, chapter 2, pages 31–58. Springer-Verlag New York, Inc., 1997.

[55] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML: Revised 1997.* The MIT Press, 1997.

[56] A. Jeffrey. A fully-abstract semantics for a concurrent functional language with monadic types. In *Proceedings of LICS '95*, pages 255–264, 1995.

[57] H. R. Nielson and F. Nielson. From CML to process algebras. *Theoretical Computer Science*, 155, 1996.

[58] H. R. Nielson and F.Nielson. Static and dynamic processor allocation for higher-order concurrent languages. In *Proceedings of TAPSOFT '95*, number 915 in LNCS. FASE.

[59] H. R. Nielson and F. Nielson. Higher-order concurrent programs with finite communication topology. In *Proceedings of 21rd ACM Symposium on Principles of Programming Languages (POPL '94).* ACM Press.

[60] K.L. Gasser, F.Nielson, and H.R.Nielson. Systematic realisation of control flow analysis for CML. In *Proceedings of ICFP '97.* ACM Press, 1997.

[61] C. D. Krumvieda. *Distributed ML: Abstractions for Efficient and Fault-Tolerant Programming.* PhD thesis, Department of Computer Science, Cornell University, 1993.

[62] A. Giacalone, P. Mishra, and S. Prasad. Facile: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2):121–160, April 1989.

[63] B. Thomsen. *Calculi for Higher Order Communicating Systems.* PhD thesis, Imperial College, London University, 1989.

[64] B. Thomsen. Polymorphic sorts and types for concurrent functional programs. In J. Glauert, editor, *Proceedings of the 6th International Workshop on the Implementation of Functional Languages*, Norwich,UK, 1994.

[65] L. Leth and B. Thomsen. Some facile chemistry. *Formal Aspects of Computing*, 7(E):67–110, 1995.

[66] R. M. Amadio. Translating core facile. Technical Report ECRC-94-3, European Computer-Industry Research Centre, 1994.

[67] J. P. Talpin, P. Marchal, and K. Ahlers. Calumet — a reference manual. Technical Report ECRC-94-30, European Computer-Industry Research Centre, 1994.

[68] K. H. Ahlers, D.E. Breen, C. Crampton, E. Rose, M. Tucheryan, R. Whitaker, and D. Geer. Distributed augmented reality for collaborative design applications. Technical Report ECRC-95-03, European Computer-Industry Research Centre, 1995.

[69] B. Thomsen, F. Knabe, L. Leth, and P. Y. Chevalier. Mobile agents set to work. *Communications International*, July 1995.

[70] D. S. Alexander, W. A. Arbaugh, M. W. Hicks, P. Kakkar, A. D. Keromytis, J. T. Moore, C. A. Gunter, S. M. Nettles, and J. M. Smith. The SwitchWare active network architecture. *IEEE Network Special Issue on Active and Controllable Networks*, 12(3):37–45, 1999.

[71] David L.Tennhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, January 1997.

[72] X. Leroy. Objective CAML, 1997. `http://pauillac.inria.fr/ocaml/`.

[73] P. Kakkar, M. Hicks, J. T. Moore, and C. A. Gunter. Specifying the networking programming language PLAN. In *Proceedings of Third International Workshop on Higher Order Operational Techniques in Semantics*, September 1999. To appear.

[74] M. Hicks and A. D. Keromytis. A secure PLAN. In Stefan Covaci, editor, *Proceedings of the First International Workshop on Active Networks*, volume 1653 of *LNCS*, pages 307–314. Springer-Verlag, June 1999. Extended version at `http://www.cis.upenn.edu/~switchware/papers/secureplan.ps`.

[75] Home page of Erlang. `http://www.erlang.org/`.

[76] J. Arthursson, J. Engblom, I. Johnson, R. Mirza, G. Naeser, M. Olsson, R. Ottenhag, D. Sahlin, M. Schmid, B. Spolander, and E. Zofonoon. A platform for mobile agents. In *Proceedings of the Second International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology*, London, April 1997.

[77] G. Naeser. SafeErlang. Master's thesis, Computing Science Department, Uppsala University, Sweden, June 9 1997.

[78] L. Brown. SSErl: Prototype of Safer Erlang. Presented in Erlang Users Conference in 1997. Available from `http://www.adfa.edu.au/~lpb/papers/ssp97/`.

[79] M. Dam and L. Freudlund. On the verification of open distributed systems. In *Proceedings of the 1998 ACM Symposium on Applied Computing*.

[80] C. Fournet, C. Laneve, L. Maranget, and D. Remy. Implicit typing à la ML for the join calculus. In *Proceedings of CONCUR '97*.

[81] M. Abadi, C. Fournet, and G. Gonthier. Secure implementation of channel abstractions. In *Proceeding of the LICS' 98.*

[82] C. Fournet and A. Schmitt. An implementation of ambients in JoCaml. Position paper for the 5th Mobile Object Systems Workshop, Lisbon, 1999.