MIKADO Global Computing Project IST-2001-32222

Mobile Calculi Based on Domains

Models of distribution and mobility: state of the art

MIKADO Deliverable D1.1.1

Title :	Models of distribution and mobility: state of the art
Editor :	I. Castellani (INRIA)
Authors :	G. Boudol, I. Castellani (INRIA), F. Germain, M. Lacoste (France Telecom R&D)
Contributors :	G. Boudol, I. Castellani, J.B. Stefani (INRIA), F. Germain, M. Lacoste (France Telecom R&D), M. Boreale, R. Pugliese (U. of Florence), E. Giovannetti (U. of Turin), A. Ravara (U. of Lisbon), M. Hennessy, V. Sassone (U. of Sussex)
Classification :	Deliverable D1.1.1, Public, Version 1.0
Reference :	RR/WP1/1
Date :	August 2002

Abstract

This document presents a comparative analysis of various models and languages for mobile code, incorporating some notion of *site* or more generally of *domain*. These models and languages will be referred to as *distributed mobile calculi*. Our study will focus on three aspects of these calculi: distribution, mobility and security. Moreover, it will emphasize the notion of domain as a primitive programming concept and illustrate its semantics in the various calculi.

Contents

Introduction 3

– PART I: ANALYSIS CRITERIA 5

1 Distribution 5

2 Mobility 9

3 Security 11

– PART II: LANGUAGES AND MODELS 17

1 The π_{1l}-calculus 17

2 Dπ - a distributed version of the π-calculus 20

3 A Lexically Scoped Distributed π-calculus (*lsd*π) and the Distributed Typed Concurrent Objects (DiTyCO) 24

4 Pict and NomadicPict 29

5 The SPI-calculus 32

6 The DJoin-calculus 34

7 Mobile Ambients 38

8 Safe Mobile Ambients 44

9 Boxed Ambients 47

10 The Seal-calculus 51

11 Mobile Resources 56

12 The M-calculus 59

13 KLAIM 63

14 Obliq 68

15 Agent Tcl 71

Introduction

The aim of this document is to give an overview of models and languages for mobility, which incorporate some notion of *site* or more generally of *domain*. Such partitioning of processes into sites or domains is necessary to give a precise account of migration: it will provide the reference frame in which migration takes place. Our stand here, and more generally in the MIKADO project, is that domains should be called to play a more important role in calculi for mobility, and become a programming concept in its own right.

Our analysis will by no means be exhaustive. Some early proposals will be left out, mainly for lack of available documentation. This concerns in particular "commercial products" such as ODYSSEY, AGLETS, OBJECTSPACE VOYAGER..., which consist of JAVA libraries supporting, to some extent, the possibility of programming agent migration. For these languages the accessible documentation amounts to little more than straight publicity advertisements. As concerns migration, it is perhaps worth pointing out immediately some limitations of the JAVA language: at present it is only possible in JAVA to "serialise", i.e. to put into transmissible form, restricted types of objects. In particular it is not possible to capture in this way the state of a thread at a given stage of execution. Hence JAVA and more generally languages where "everything is an object", do not allow the concept of migrating agent to be implemented in its full generality, as it was proposed, for instance, in the TELESCRIPT manifesto. One of the objectives of the MIKADO project is to go beyond these limits, hence the fact that our analysis will not touch on these extensions of JAVA should not penalize our study.

We shall mostly concentrate here on models and languages which are presented as process calculi or strongly based on them, and particularly on Milner, Parrow and Walker's π-calculus [MPW92]. These include distributed versions of the π-calculus like the $D\pi$-calculus [HR98] and the Nomadic π-calculus [SWP98], the distributed version of the JOIN calculus [FG96], as well as new calculi expressly designed with distribution and security concerns in mind, such as the AMBIENT calculus of Cardelli and Gordon [CG98] and the SEAL calculus of Castagna and Vitek [VC99b] and the Mobile Resources calculus of Godskesen, Hildebrandt and Sassone [GHS02a]. On a slightly different strand, combining process algebras with other programming paradigms - respectively object-oriented and coordination-oriented - stand the calculus DI-TYCO [VLS98], the distributed version of the language TYCO [VB98, LSV99] by Lopes, Vasconcelos and Bastos, and the calculus KLAIM [NFP98] of De Nicola, Ferrari and Pugliese, which is based on the coordination language LINDA. Finally the M-CALCULUS of Schmitt and Stefani [SS02] and its sub-calculus, the K-CALCULUS [FL02], studied by Germain and Lacoste, although still at an early stage of development, represent a concrete attempt at conceiving a language around the primitive notion of domain.

The word "mobility" will be used here to indicate the actual *movement* of processes from one place to another, what is commonly called *migration*. We shall therefore not consider the more implicit form of mobility exemplified by the π-calculus and its higher-order variants, CHOCS [Tho93] and HOPI [San92, San93], which consists in the *transmisnsion* of processes or links to processes as arguments of communication.

Our study will concentrate on three aspects of a distributed mobile calculus: *distribution*, *mobility* and *security*. It will emphasize the existence of "groups of entities" (resources or processes) which share some behavioural characteristics or some common interface with the environment, while remaining relatively independent from other groups. The notion of site or *locality* provides a typical example of such a group. These groups, however, may play more general rôles. They will generally be referred to as *domains*. For instance, a domain may be a communication unit (an address space) or a migration unit (a group of processes moving as a block). It can also act as a unit of failure (the failure of an entity entailing that of the whole group), of security (a group of processes sharing the same security policy), or as an "administrative" domain.

Although domains may have different meanings, according to the calculus, some common features may be identified:

- A domain is the host of a number of processes and resources. Sometimes, the domain itself is assimilated to a particular process which plays the rôle of a controller for all the entities in its scope.
- Domains can be *observed* or *controlled*. The control of domains may be motivated, for instance, by failure management or security concerns. The observation of domains may be required to implement migration or distant communication.
- A domain can be *named*, and as such designated as the target of a migration or a communication (if communication across domains is allowed).
- The structure of domains may be *flat* or *hierarchical*: in the latter case domains are usually organised as a tree or a forest. Structured domains allow for a finer control of mobility and failure, and for a more flexible handling of security issues when modelling networks partitioned into administrative or "trust" domains.

As said previously, domains are the reference frame for migration. Migration is intended as the movement of processes or some other entity having executable content (such as locations themselves), as opposed to simple data transfer. One can distinguish between *subjective* and *objective* migration. In the first case the migration instruction comes from the migrating entity itself (possibly from a process enclosed in it, if this entity is for instance a location), while in the second it is outside its control.

A further classification can be made into *strong* and *weak* migration. In the first case a process can be interrupted during its execution and moved together with its current environment to another location where execution is pursued. In this case, where "active" code can be moved, one often speaks of *migrating agent*. In the case of weak mobility, instead, only "passive" code can be moved.

The rest of this document is organised as follows. Part 1 explains the criteria for the analysis, namely distribution, mobility and security. It is divided into three sections, one for each criterion. Part 2 is devoted to the description of the various calculi: it takes the form of a series of "fiches", each describing in some detail a particular calculus. This study is partly based on [Cas01], [Zil01] and [Sew00].

PART I: ANALYSIS CRITERIA

1 Distribution

Our analysis of distribution will be developed around the notion of domain. More precisely, distribution will be *defined as* the existence of domains. A domain will group a number of processes (agents, resources) which can be identified as a "behavioural unit" for some criterion of interest.

At present, each calculus or language provides a unique semantics for its domain concept, either in explicit form (in most cases), or in semi-implicit form. This semantics differs in the various calculi, depending on their particular "orientation": whether towards an explicit localisation on a network, or towards failure, or towards security management, etc.

The attributes we shall examine in this section for comparing the various notions of domains will be essentially their topological structure, their semantics, the bearing they have on communication and their naming policy. The issue of failure will be addressed in a more synthetic way.

1.1 The concept of domain

All distributed mobile calculi offer some expression of the concept of domain, but none of them goes as far as to completely "reify" this concept. Among the various calculi considered here, only the AMBIENT calculus proposes an explicit domain primitive, called *ambient*.

The basic expression of a domain, provided by all calculi, is its *name*. More specific identifications for domains may be present in some calculi: the $\pi_{1\ell}$-calculus, for instance, has "locality processes", which control a given locality (as opposed to "localised processes" which reside in a locality) and "configurations", the DJOIN calculus has "localised solutions", "distributed solutions" and "localised definitions", the SEAL calculus has "seals" and "process bodies", and so on.

Having a name allows a domain to be treated as a *resource*. A domain can for instance be designated as the target of a distant communication, or as the destination of a process or domain migration.

All calculi agree in viewing domains as *hosts* for processes and resources (or other domains, if nesting of domains is allowed). In some cases the domain itself is regarded as a particular process, which plays the rôle of a controller for all entities in its group.

The semantics of domains differs according to the primary vocation of the calculus: this vocation is for instance "controlled migration" and security in the AMBIENT and SEAL calculi, failure management in the DJOIN and $\pi_{1\ell}$ calculi, and "controlled communication" in the calculus Dπ. Each calculus gives a unique semantics for its notion of domain, which captures all its orientations. For instance, in the DJOIN calculus, domains are described both as migration units and as failure units.

1.1.1 Domain topology

In most of the considered calculi (for instance in the DJOIN, AMBIENT and SEAL calculi) the space of domains is hierarchically structured as a *tree* or a *forest* of trees. In the $\pi_{1\ell}$ calculus, there is an indirect hierarchy in that the space of domains (called "localities") is the set of leaves in a tree of configurations. However, the set of domains in itself is flat in this calculus.

A tree-like structure of domains allows for migration to be expressed as a local *reconfiguration* in a tree. This form of migration, which amounts to moving a sub-tree from one node to the other will be called *topological migration*. In a tree-like space of domains, each node represents a domain, with children nodes representing sub-domains.

Some of the studied calculi, like the $\pi_{1\ell}$ calculus, assume the existence of an *ether* in which the domains live. This ether is the support for messages in transit from one domain to another. The concept of ether may be helpful in formalising the notions of asynchrony and distant communication.

1.1.2 Domain semantics

This section gives a synthetic description of the semantics of domains in the various calculi, pointing at their different orientations and at their implications in terms of infrastructure requirements.

- Orientation towards *communication*: entities belonging to the same domain are unified by their ability to communicate with each other. In other words, communication is restricted to being *local* or *intra-domain*. The domain itself may be distributed over different address spaces on the same machine, or over different machines. Several calculi (AMBIENT, Dπ, Nomadic π-calculus) use domains in this way, as *communication units*, forbidding direct communication from one domain to the other.

- Orientation towards *mobility*: entities belonging to the same domain are unified by their behaviour under the effect of a migration instruction. The whole domain moves as a block. If the structure of domains is hierarchical, the migration of a domain entails the migration of all its sub-domains. This is the case, for instance, in the calculi DJOIN, AMBIENT and SEAL. This orientation will be further examined in Section 2.

- Orientation towards *failures*: entities belonging to the same domain are unified by their reaction to failures. The failure of one of the entities inhibits all the others. The failure of a domain is usually expressed by its inability to emit/receive messages, to receive visiting processes, or to adopt incoming domains as sub-domains. This hypothesis facilitates the writing of protocols for failure detection and control. This orientation is chosen in the calculi $\pi_{1\ell}$ and DJOIN. It will be shortly reconsidered in Section 1.1.5.

- Orientation towards *security*: entities belonging to the same domain are unified by their access protocol or their security policy. All the studied calculi include a “minimal” security notion via the handling of *name scoping* (for both channels and domains). Going further along the way, some calculi introduce explicit notions of *boundary* and *entrance authorisation* for a domain (AMBIENT), or *access authorisation* for a resource in a domain (SEAL).

1.1.3 Observation and control of domains

The need for observing or controlling domains is often related to failure management ($\pi_{1\ell}$, DJOIN). In this case the observation/control may apply to a *state*, when the notion of state is made explicit in the calculus (in $\pi_{1\ell}$, for instance, a domain - a locality - is closely associated with a process acting as a state controller). The control of domains may be also motivated by security concerns, as in the AMBIENT and SEAL calculi. In the AMBIENT calculus, for instance, the control of domains is ensured by primitives which only allow for “stepwise migration”: a domain can move inside an adjacent one or outside the immediately including one, and a domain can dissolve an immediate sub-domain allowing all its processes to step at the upper level. In a recent variation of the AMBIENT calculus, the SAFE AMBIENTS by Levi and Sangiorgi [LS00], this control is pushed even further by requiring the simultaneous consensus of both the source and the target of each migration. In the AMBIENT calculus, another form of control is provided by the mechanism of *capabilities* (of operating on a given domain with a given primitive), which may guard a domain and make it accessible only once that capability has been used. A similar phenomenon is found in the DJOIN calculus, where the activation of a domain may be guarded by a *join pattern*, which is consumed by effect of a distributed synchronisation. Finally, in the DJOIN calculus, it is possible to activate or inhibit a domain dynamically by using the *definition* construct.

1.1.4 Domains and communication

The term *local communication* is synonymous here of intra-domain communication, while *distant communication* means inter-domain communication.

Local communication

We start by considering the question of *local naming*, that is the naming of a resource (typically represented by a receptor process) by a process belonging to the same domain. The name of the resource must be syntactically known by the process that uses it (i.e. the process must be in the scope of this name). Such name may have been acquired through a communication. Within a domain, a name may denote a unique receptor process ($\pi_{1\ell}$), or a set of receptor processes (DJOIN, SEAL).

- **Local communication policy**

  Local communication is communication which takes place between processes in the same domain. In some calculi this is the only possible form of communication, and processes have to convene in a common domain in order to interact. Typically, a process wishing to send a message will have to move to the domain of the receiver. Clearly this choice only makes sense in calculi with a *migration* primitive, where distant communication can be simulated.

  In all calculi except the SEAL and Mobile Resources calculi, local communication is *asynchronous*, polyadic and unicast (two processes cannot receive simultaneously the same message on the same name). In the SEAL and Mobile Resources calculi local communication is *synchronous* and unicast. Local communication may be directed, that is, based on names ($\pi_{1\ell}$, DJOIN, SEAL), or anonymous (AMBIENT).

  In all the studied calculi, except the Mobile Resources calculus which does not allow name passing in its present form, processes can transmit both channel names and domain names. The AMBIENT calculus allows for the communication of more precise entities, namely *capabilities* (a capability can be viewed as a permission for a specific use of a name). In all the considered calculi reception is *static*, meaning that the reception of a name does not turn its receptor into an "owner" of that name. A process receiving a channel name through communication acquires the right of emitting on that name or forwarding it to other processes. Similarly, a process that receives a domain name acquires the right to designate that domain as the target of a migration (DJOIN, AMBIENT), or as an object of observation/control ($\pi_{1\ell}$).

- **Semantics of local reception**

  The ability to receive on a given name may be *unique* in a domain, in the sense that the domain contains at most one receptor process on each name (the "unique receptor property"), as in $\pi_{1\ell}$, or it may be *multiple*, as in the DJOIN and Dπ calculi. Moreover in the DJOIN calculus, all receptors on a given name are necessarily in the same domain, i.e. domains determine a partition over receptor names. This is important from an implementation point of view, since it allows the decision between conflicting synchronisations to be made locally, and therefore simplifies the underlying infrastructure.

  In most of the studied calculi (apart from the Nomadic π-calculus and the SEAL calculus), reception is static, in the sense that it is not possible to create new receptors for a received name. Finally, in all calculi a received name becomes "known", that is, it can be used for future referencing.

Distant communication

We consider here the issue of *distant naming*, which is the naming of a resource (or domain) by a process belonging to a different domain. Like for local naming, the name needs to be syntactically known by the process that intends to use it. All the studied calculi include a notion of *distributed scope* for names, extending to the entire system. This allows entities to be known and referenced from distant domains.

Distant naming may be *global*, in the sense that every entity may be named from any domain ($\pi_{1\ell}$, DJoin) or *restricted*, in which case only neighbouring domains or sub-domains (Mobile Resources) in the tree-structure may be named (AMBIENTS, SEAL). Finally, the localisation may be *absolute* ($\pi_{1\ell}$, DJOIN, AMBIENT) or *relative* to the domain from which the request is issued (SEAL, Mobile Resources).

- **Distant communication policy**

  Distant communication is communication which takes place between processes belonging to different domains. As for naming, distant communication may be *global* (or *transparent*) in the sense that distant destinations are reachable as if they were local (DJOIN, $\pi_{1\ell}$), restricted to sub-domains (Mobile Resources) or to the immediate neighbourhood (SEAL), or even completely forbidden (AMBIENT, *D*π-calculus).

  The localisation of the target may be implicit or explicit in the message[1]. The message delivery may be *implicit* (a message is emitted towards some name, and automatically received on that name), or *explicit* (a message emitted towards a name must first explicitly exit the source domain, and then enter the target domain to reach its receiver). This is the case for instance in the $\pi_{1\ell}$ calculus, where the two phases are clearly distinguished. Similarly, the DJOIN captures two different phases in distant communication, an atomic exit-entry operation from one domain to the other, followed by the reception of the message.

- **Logical routing**

  We call *logical routing* the routing within a hierarchical structure of domains. This captures the notion of routing within a network when domains are used as communication units.

  The routing of messages is *explicit* in the AMBIENT calculus, in Dπ, NOMADIC PICT, Mobile Resources and in SEAL (where it is limited to the immediate neighbourhood). In the other calculi under examination, distant communication is transparent and the path towards the target process is implicit[2].

In most calculi both local and distant communication (when allowed) are *asynchronous* (in the sense that the sending of a message is non-blocking[3]). In general, distributed calculi have tended to adopt asynchronous rather than synchronous communication disciplines, as the former seem more primitive, are easier to implement and are closer to the asynchronous message delivery mechanisms of actual networks. In the SEAL calculus, both local and distant communication are *synchronous*. This choice is due to the fact that the primary concern of the SEAL calculus is security, and therefore a more rigid communication discipline is preferred (the implementation issue is less crucial here since only short-range communication is allowed). Similarly, communication is synchronous in the Dπ-calculus, where only local communication is allowed.

1.1.5 Domains and failures

In some of the studied calculi ($\pi_{1\ell}$, DJOIN) the notion of *partial failure* is put forward as one of the main concerns. These calculi represent in a primitive way (as a domain) a group of processes unified by the same reaction to failure, in the sense that a failure manifests itself simultaneously in all of them as the *blocking*, temporary or permanent, of some of their capacities of emitting/receiving messages or of emitting/receiving migrating entities. In both cases, a domain fails atomically, together with all its sub-domains. Such failure is always detectable from other domains ($\pi_{1\ell}$, DJOIN), and may also be detectable from inside the failed domain (DJOIN).

[1]In the case of control messages targeted at some domain, the localisation of the target must be explicit, since the name of the domain also represents, in a way, the channel on which the message is sent. This presupposes that domain names are unique at a global level (whence the necessity of a corresponding infrastructure mechanism).

[2]Implicit routing needs a distributed infrastructure mechanism to ensure the transparency.

[3]This is not to be confused with the use of “asynchronous” and “synchronous” in Milner’s calculi CCS and SCCS (Synchronous CCS) [Mil83]. In SCCS synchrony means that parallel components proceed together in lock-step.

As regards the *representation* of failure, the status of a failed domain may appear as a special state ($\pi_{1\ell}$), or be recorded in a label (DJOIN). In DJOIN, a stopped locality takes the status of a *definition* (an endemic form of locality). In all cases, a stopped locality is supposed to have the ability to respond to a failure test.

2 Mobility

The analysis of mobility in distributed mobile calculi will be organised around three notions:

- **Domains**. We concentrate here on the mobility aspect of domains. Domains are therefore viewed simply as reference points for mobility: a domain is was one quits and what one enters when performing a migration. When dealing with mobility in actual systems, this notion is sometimes left implicit, or reduced to that of address (an Internet address for instance).
- **Migrating entities**. We do not consider client/server models (RPC or RMI) based on simple migration of data. Migrating entities will have in general an *executable content*, which can take various forms (domains, objects, closures).
- **Context**. The context is what constitutes the *environment* of the mobile entity at the moment of migration. This notion remains implicit in process calculi, while the management of contexts during migration is an important part of the implementation of languages for mobility.

These three notions are not disjoint: in some models the migrating entities are the domains themselves, while in others the migrating entity includes – or more exactly brings along – a portion of the context. In general, a migrating entity is included in a domain, while its context may extend over several domains. In a classical semantic model, the context consists of the *environment*, which associates which each identifier a "denotable value" (which may be in particular a memory address) and of the *memory* (or *store*), which associates a value with each memory address. Each element of the context is named, by means of an identifier or an address, and the context of a given entity may be loosely defined as the portion of the environment and of the memory that the entity knows, through the free occurrence of the corresponding names. One could also have a more generous view of what constitutes the context of an entity, including for instance physical devices or program libraries for which the entity possesses a link, but this will generally not be the case in the models and languages we consider.

As concerns migration, a first distinction can be made between *strong migration* and *weak migration*: in the first form of migration, an *execution unit* may be interrupted, sent with its current state to another domain, and resumed at its arrival, whereas in the second case the migration entity is not an execution unit but only a piece of *executable code*. This distinction will be further refined in the course of our analysis.

One can also distinguish migration by actual **movement**, whose semantics is intuitively clear, and migration by **replication**, where a *clone* of the migrating entity is created in the target domain, while an image or *proxy* of the entity remains in the source domain. In fact this distinction applies mainly to the context of the migrating entity. However there exist some examples of code migration based on replication: the best known is the mechanism of *code on demand* à la JAVA, which should be regarded as a "limit case" of mobility[4]. This mechanism may be found also in OBLIQ, and, in a different form, in AGENT TCL, where it is possible to clone agents at a distance.

Following Cardelli and Gordon, a further distinction can be made between *objective migration* and *subjective migration*: in the first case the migration instruction is outside the control of the migrating entity, while in the latter the instruction emanates from the migrating entity itself.

This distinction effects a clear separation between models allowing domain migration, where the distinction can be usefully applied, and models for code migration, where mobility seems to be always expressed

[4]We recall that one of the announced objectives of the MIKADO project is to go beyond JAVA as concerns mobility. We shall only study here proposals that clearly address this objective.

in objective form. We shall use this distinction mainly to qualify a “style” of migration. In a similar spirit, we shall analyse the effect of migration on the *state of execution* of the migrating entity: migration may be *blocking* for the migrating entity – this is typically the case for migration of threads or continuations – or *asynchronous*, as in the case of agent or domain migration: here the migrating entity can receive and satisfy requests of communication while undergoing a migration.

Concerning the context, a finer analysis may be made. Indeed, the context consists necessarily of a *public part* – at the very least a name server, but in general also a set of libraries required for the execution of the migrating code – and possibly of a *private part*, whose scope is restricted. It appears that, in all cases, the behaviour of the migrating entity with respect to the public part of its context is that of *dynamic binding*: the value of a public name is the one it has in the current site of execution, and there is no migration of public values. On the other hand, all the studied models rest on *static binding* as concerns private names. We shall not, however, insist here on these distinctions between public and private name bindings, (nor between static and dynamic binding) which will not play a rôle in our classification.

It should be noted that “private” here does not mean “exclusive”: some elements of the context, whose scope is known, may be shared by different migrating entities, assuming, of course, that the model allows for parallel execution of these entities, even if only in simulated form (for instance a system of threads with a scheduling policy).

One can distinguish various methods for handling the links of a migrating entity with its context, according to the “behaviour” of the context with respect to migration:

- **fixed context**. Here the elements of the context stay in the source domain (the domain where the migration starts from). There are two sub-cases to consider as regards the relation of the migrating entity with its context:
  - the relation is *broken*. It may happen that the link is re-established in the target domain, by dynamic linking with a resource having the same name as the “disconnected” one.
  - the relation is *maintained* at a distance. This may be done either *implicitly* (a request from the migrating entity towards its original context reaches it “spontaneously”), or explicitly, by means of an intermediary or *proxy* located in the target domain[5].
- **mobile context**. Again we may distinguish two sub-cases, according to the type of mobility of the context elements:
  - mobility by actual **movement**. If the context element is shared between the migrating entity and other execution units, we find ourselves in a similar situation as in the previous case (the link may be broken or maintained in some way).
  - mobility by **replication**. The migrating entity brings along a copy of (a part of) the context. Here we could still distinguish the case where the context and its copy must remains identical, whatever their evolution might be (“coherence”), or are allowed to evolve differently. It should be noted that in an asynchronous universe, the idea of perfect coherence is not very plausible: to guarantee such a property one should rather recourse to a proxy, which would bring us back to one of the previous cases.

Clearly this classification applies to each element of the context rather than globally to the whole context. It will be used in the points 3 et 4 of the “analysis grid” that we present next. Our criteria for the analysis of mobility in the various models and languages will be the following:

1. **nature of migrating entities**. These may be domains, objects, or code in various forms, like for instance threads or closures [6]. The main point here is to identify the “style” of the model under

[5]this case can in fact be reduced to the previous one if the resource wich is referenced is itself a proxy. One difference being that the latter is installed at the time of migration, to allow for distant referencing.

[6]As we said earlier, we are only interested here in migrating entities having an executable content.

consideration. The exact nature of the migrating entity will be further specified in the following steps.

2. **way of expressing migration**. We analyse here the way in which the migration is performed (which can be completely implicit, as in the case of "code on demand"), and its caracteristics with respect to the execution of the migrating entity. This is the stage where the distinctions between objective and subjective migration, and between blocking and asynchronous migration, will come into play.

3. **mobility in the course of execution**. We distinguish here the models where the migration of an executable entity takes place before its execution, from those where a migration instruction may continue to affect the migrating entity in the course of its execution. Typical paradigms that fall in the first category are those of *code on demand* (COD) and *remote evaluation* (REV). In the second category one can mention the *agent* model and the migration of threads or domains.

4. **migration of associated state**. This occurs when the migration of an entity, for instance of a thread, provokes that of all or part of the *state* of the memory that the entity has access to when the migration starts. If such a state migration takes place, in conjunction with the actual migration of the entity, one speaks of *migrating agent*.

5. **handling of distant references**. When a part of the execution context of the migrating entity cannot or must not be moved or copied, one has to provide an underlying mechanism to allow the migrating entity to keep a link with this part of the context through the network (i.e. across domains). This is sometimes referred to as "global computation" (Cardelli).

There are several possible combinations among the different options one can choose concerning the above points. Not all of them have been experimented yet. A trend that has been largely explored is that of mobility in object-oriented languages, particularly under the form of developments based on JAVA.

In object-oriented languages, the notion of state and context are rather clear, given that each object encapsulates its state (there is therefore no possibility of sharing – if one does not move threads), and the context can be seen as consisting of the objects it knows, i.e. whose methods it can invoke. Most extensions of JAVA with mobility implement a form of mobility where the migrating entities are objects.

The migration may be *objective* or *subjective* (self-inflicted). The migrating object brings along its state and maintains distant references towards the other objects, possibly by means of *proxies* (communication being effected by method invocation). In order to preserve some degree of strong mobility, i.e. of mobility in the course of computation, the migrating object executes a special method upon its arrival in the target domain, which can be a parameter of the migration instruction. It is to be noted however that, since the JAVA implementation does not support the *serialisation* – and hence the mobility – of threads, the strong mobility character remains rather limited.

This scenario is found for instance, with some variants, in the TELESCRIPT/ODYSSEY proposals (TELESCRIPT is what launched the idea of *mobile agent*, an agent moving from place to place to gather informations, and execute operations on behalf of a user; this idea is only partially implemented in ODYSSEY), AGLETS, OBJECTSPACE VOYAGER. In this document we shall not include a specific description of these extensions of JAVA, because the available documentation is not sufficiently explicit to allow a detailed analysis of them (it would be necessary to experiment with the software itself to get a more precise idea of the semantics).

3 Security

In this section, *security* will be seen as the ability for a system to bring protection guarantees to its resources, both in terms of *secrecy* (protection of assets against unauthorized disclosure of information, and possibly unauthorized use of resources) and *integrity* (protection of resources against unauthorized modification).

The third traditional aspect of computer security which covers *availability* issues ("the property of being accessible and useable upon demand by an authorized entity" [fS88]) will not be addressed here, as protection from denial of service attacks may be seen as more relevant to dependability in a broad sense, including consideration of failures, and in any case, beyond the scope of this section on security. Before examining how security is guaranteed in the studied models and languages, we recall two fundamental distinctions which must be kept in mind when dealing with security:

- At the access control-level, *objects* must clearly be distinguished from *subjects*. While the former are passive entities like resources or information, the latter are active system components like processes or users, which are going to perform requests upon objects, which then may or may not be authorized. This fundamental perspective for access control was first described in [Lam74]. Subjects are endowed with special attributes like an identity or privileges, as captured by the term *principal*.

- It is also important to establish a clear separation between a *security policy* and the *security features* or *security mechanisms* which are going to implement that policy. The policy stems from user-level security requirements and can be defined by a system administrator independantly of the security mechanisms. It can be formally specified using a *security model*. Some of the studied calculi like the SEAL-calculus or the SPI-calculus can be used to express security models. The security mechanisms allow to enforce a particular security policy, within the limits set by the expressiveness of the security model. Some of these mechanisms can be present explicitly at the language level (PICT, OBLIQ, JAVA) when the language offers explicit security-oriented primitives: in that case, some properties can be statically checked, which opens the way for formal proofs. Other security features are found at the run-time level only.

The implementations of the studied models and languages have in common to be mobile code systems. Several security threats can occur in such systems, thus requiring protection mechanisms. To simplify, and help identify the weak spots in such architectures, a mobile code system can be seen as made of three elements [VST97]: first, a *computational environment* (CE) including a computer, operating system, and run-time hosting a number of resources. This CE is akin to the notion of *context* defined in the section on mobility. Second, *execution units* (EU) able to migrate from CE to CE, which are called *migrating entities* in the section on mobility. Third, the network which connects the various CEs.

This architecture can be subject to several types of security attacks: replay (attack where a previously sent message is captured and retransmitted for illegitimate purposes), masquerading (attack where an entity pretends to be a different one, e.g., spoofing using forged IP addresses), unauthorized modification and/or disclosure of information, the last two types of attacks being direct violation of integrity and secrecy. Those attacks can be directed towards three main parts of the architecture described previously:

1. *The communications between two CEs* must be secured to avoid unauthorized disclosure (secrecy) or modification (integrity) of information by a malicious user. Communication channels can be made secure with dedicated encryption protocols like SSL/TLS, and by using authentication mechanisms like Kerberos. Those techniques will be studied in sections 3.1 and 3.3. In a mobile code system, this situation corresponds to the case when a remote EU migrates into a local CE: the incoming EU must be identified and granted access rights.

2. *The communications between an EU and a CE* must also be protected, which includes protecting the CE resources from unauthorized use by the EU, but also protecting the EU against a malicious CE which could disclose information without the EU being aware of it, or deny the EU access to services. To simplify, we will not consider such a situation and assume the CE to be trustworthy.

3. *The communications between two EUs* can be protected using general communications security techniques (section 3.3).

The notion of *protection/security domain* is essential in this model. Generally speaking, a security domain corresponds to a CE scope, including all its EUs. Item (1) belongs to *inter-domain security* (between two CEs), while items (2) and (3) are part of *intra-domain security* (within CE limits). This remark allows to make the link between this section, more focused on security mechanisms, and the other sections of the document which are built around the general notion of domain.

In what follows, we present the various criteria chosen to analyze the models and languages for mobility from a security viewpoint. The next sections classify the protection primitives by type of security features: identification and authentication (section 3.1), access control (section 3.2), communications security (section 3.3), and auditing (section 3.4).

3.1 Identification and Authentication

Underlying most authentication mechanisms is a form of *identification*, i.e., each entity must have one or several *identities* that must be difficult to forge. Identities are verified during the *authentication* process. Authentication can be *weak*, e.g., a password is transmitted, or *strong* when no secret is revealed to the outside world ; this can be typically achieved with public-key cryptosystems like in AGENT TCL. The authentication protocol can also be *two-party* where a client wants to be authenticated by a server or *third party* where an authentication server performs the process, e.g., in Kerberos the two parties which try to authenticate one another share a *session key* provided by the authentication server, key that can also be used to encrypt messages.

In the studied models and languages, authentication when considered at all, can be implemented using a separate server, as in a secure version of OBLIQ implemented with Secure Network Objects [ABvDW96]. However, in a system like AGENT TCL, authentication is two-party: the CE plays the role of an authentication server, when an agent server authenticates an agent, or during mutual authentication of two agent servers.

The use of security-oriented calculi like the SPI-calculus allows to formalize authentication protocols used in real systems. Other tools include the BAN-logic of Abadi et al. [BAN90] or theories about trust like [ABLW92].

3.2 Access Control

The resources to be protected are varied, typically a channel name (SPI-calculus, SEAL-calculus) or an object (JAVA, OBLIQ). Access control checks may be performed at various levels of granularity: channel (SPI-calculus), object (OBLIQ), class (JAVA), method (JAVA), or group of classes (JAVA).

3.2.1 Access Control Policy

The motivation behind defining an access control policy, and more generally a security policy is to establish a clear separation between a policy and the mechanisms that must be used to implement it. Few models and languages explicitly represent access control policies, apart from JAVA where a policy is captured by a special `java.security.Policy` class which associates a number of code properties with given permissions.

3.2.2 Authorization Model

A commonly used access control model is the *access matrix* $M = (M_{ij})$ where M_{ij} is the access right, e.g., read, write, etc., of subject i on object j. For realistic systems, storing the whole content of the access matrix appears impossible. Thus, it is more convenient to use two shorter representation of the matrix:

- *Access Control Lists (ACLs)* define for each object, the list of subjects that have the right to access it. ACLs are a column by column representation of the access matrix.

- *Capability Lists (CLs)* define for each subject the list of objects it is allowed to access. CLs are a line by line representation of the access matrix.

Table 1 summarizes the pros and cons of these two representations.

Operation	*ACL*	*CL*
Authentication	Easy, using an external authentication server.	Easy, since the capability is held by the subject. Thus, authentication can occur within the subject.
Access Rights Lookup	Easy, using a simple list traversal.	Difficult, since all capabilities must be kept track of.
Access Rights Revocation	Easy, by a simple list modification.	Difficult, because the capabilities must be invalidated, and invalidated capabilities must be kept track of.
Access Rights Extension	Easy, by a simple list modification.	Easy, but the propagation of capabilities can weaken the overall system security.

Table 1: Some Approaches for Access Control.

Some approaches use both capabilities and ACLs to control access to resources: using the concept of *session*, ACLs are only used during the creation of a new session. During that session, the system provides capabilities to subjects wishing to access a number of ressources, in order for access control to be more efficient. Those capabilities are destroyed at the end of the session. In general such systems using multiple access control mechanisms require adapters, necessary to allow the different authorization systems to interoperate. This may cause increased security weaknesses.

Other models are possible such as the use of *security clearances* attached to a principal in multi-level security systems. In those systems, objects are given a *security level*. Access is only allowed when the security clearance of a principal matches at least the security level of the object it is trying to access. Those label-based systems are in general difficult to administer.

For all these models, we call *privileges* the security attributes of a subject, and *control attributes* the security attributes of an object.

In the studied models and languages, the various approaches are fairly well represented: capabilities (AMBIENTS, SEAL, JAVA, OBLIQ), ACLs (OBLIQ, AGENT TCL), security levels and clearances (AGENT TCL). In several of these calculi (SPI, SEAL, AMBIENTS, Mobile Resources, OBLIQ), lexical scoping allows to control the use of channel names, seen as capabilities to communicate or to access resources. In some others (SEAL, Mobile Resources), restrictions on the use of channel names allow to control migration. In the various "fiches", the present section tries to describe how privileges and control attributes are put to use together to realize access control.

3.2.3 Privilege Delegation

When a client invokes a method on a server object *o*, to complete the operation begun by the client, *o* can perform invocations on other objects. From an access control viewpoint, the problem of delegation can be formulated as whether an intermediary object in the invocation chain will use its own privileges or the ones that have been delegated by client objects. A related security threat is masquerading which can be countered by authentication and access control techniques.

In the models studied, delegation of access rights is not taken into account. This problem is probably still too open to be represented at the level of the programming model. Only a few systems like ORB (Object Request Brokers) implementing CORBA specifications provide some elements of an answer to the problem.

3.2.4 Security Domains

To reduce the size of the access control matrix, the programmer can bring together objects which share the same control attributes in a *security/protection domain*. Another commonly found type of security-related domain is a *user group* which brings together a set of subjects with the same privileges.

Several of the studied calculi capture the notion of security domain using *lexical scoping* (AMBIENTS, SPI, SEAL, Mobile Resources, KALI SCHEME, OBLIQ): the restriction operator narrows the names known by a given process, thus allowing to guarantee secrecy and integrity properties, since those names can only be communicated through the barrier delimited by scoping using a rule of scope extrusion. Such a rule could capture the crossing of a boundary between two protection domains (SPI, AMBIENTS, SEAL). Restrictions on the use of scope extrusion can be imposed (SEAL). In the studied models and languages, security domain management remains implicit (JAVA): in general, a security policy is attached to the protection domain, but the policy does not appear explicitly in the language itself, apart from the JAVA case where a special class allows to manipulate security policies. Some languages (AGENT TCL) also incorporate the notion of *trust domain*.

3.2.5 Security Policy Administration

Security domain management can include:

- *The management of the domain itself:* to create, destroy a domain object, or position it in a hierarchy of domains.
- *The managements of the domain members:*, e.g., to move an object between domains.
- *The management of the policy associated with the domain object:* the policy can cover access control, delegation, audit, non-repudiation, or default message protection.

In the studied models and languages, only JAVA and AGENT TCL take into account the last item. The definition of a global security policy for a domain by federation of the policies of the subdomains is left out. The first two items are left implicit at the level of models and languages. Only in specific implementations like a CORBA ORB are they explicitly managed.

3.3 Communications Security

The establishment and use of secure channels mainly relies on cryptographic techniques. Message secrecy can be guaranteed using cryptosystems which are either asymmetric (public-key cryptography) like RSA, or symmetric (secret-key encryption algorithms) like DES or IDEA. Message data integrity can be guaranteed with digital signature algorithms that use message digest functions like MD5[7]. Advanced security systems also provide certificate [8] management facilities used in non-repudiation services[9].

[7] A *message digest* is generally a oneway function which produces a fixed-length representation of the message, like a hash, which can then be signed using the sender's private key.

[8] "A *certification authority (CA)* guarantees the link between the user and cryptographic key by signing a document that contains user name, key, name of the CA, expiry date, etc" [Gol99]. Certificates can be used to acknowledge a fact, e.g., a message was sent or received, or a right, e.g., a user is entitled to some specified access rights.

[9] *Non-repudiation* is a way "to create evidence that data has been sent or received, so that the sender (or receiver) cannot later deny this fact" [Gol99].

Most of the studied calculi guarantee message secrecy and integrity using lexical scoping. Only some specific security-oriented calculi (SPI, SJOIN) go beyond, and include directly at syntax-level security-oriented primitives like encryption or hash constructs, or propose security-driven extensions (KALI SCHEME, SEAL-calculus). The implementations of some languages with security features like JAVA or AGENT TCL include both symmetric and asymmetric encryption mechanisms and digital signatures algorithms. Among the studied languages, only JAVA seems to include certificate management facilities.

3.4 Auditing

Audit trails can help a system administrator trace a security violation once it occurred. The trails are made of logs of typed events. An *audit policy* filters the security-related events that should be logged to avoid a log size overflow. Auditing is not taken into account in the models and languages studied in this document. This security feature should be reserved for systems with important security requirements. For instance, a CORBA ORB allows an administrator to specify an audit policy, which covers events such as principal authentication, privilege update, or failed service requests.

PART II: LANGUAGES AND MODELS

1 The π_{1l}-calculus

1.1 References

[Ama97] R. Amadio, An Asynchronous Model of Locality, Failure, and Process Mobility, Coordination 97, Lect. Notes in Comp. Sci. 1282, 1997. Extended version appeared as INRIA Research Report 3109.

[Ama00] R. Amadio, On Modelling Mobility, Theoretical Computer Science 240 (2000) 147-176.

1.2 Overview

The π_{1l}-calculus is an asynchronous π-calculus([10]), extended with a notion of *locality* and a related model of failures. The π_{1l}-calculus was the first π-calculus based model to take into account failures of nodes in a network, together with code mobility. A distinctive feature of this calculus, with respect to the standard π-calculus, is that it ensures statically the unicity of receivers. It is also a "local" π-calculus, that is, the names transmitted as argument of messages cannot be used to create receivers. This restriction, that the JOIN also satisfies, has important consequences on the algebraic theory of the calculus, which is much more satisfactory than in the standard case.

1.3 Distribution

1.3.1 Basic entities

There are three kinds of basic entities in the π_{1l}-calculus: channels, which are simply names, processes, written using the standard constructs of the π-calculus, and localities that contain running processes. Associated with each locality there is a "location process" providing information on the status of the corresponding site, running or stopped.

Making domains apparent

Domains in the π_{1l}-calculus manifest themselves in two ways: first, the model features an explicit notion of site, written $\{P\}a$, representing the process P running at the locality named a. Second, for each locality name a, there is an associated process $Loc(a)$ which has two states $Loc_R(a)$ when the corresponding site is running, and $Loc_S(a)$, when it is halted.

Names and references

There are two kinds of names, channel names and location names. Both are values that can be communicated through a channel. A location name denotes a unique site, and in particular there may be only one location process $Loc(a)$ associated with a given name. Similarly, for each channel name there is at most one corresponding receiver. The scope of the names is distributed, in the sense that names may be used in any site. As in the π-calculus, there is a primitive, denoted ν, for generating (globally) fresh names.

[10]see the Section on PICT and NOMADICPICT

1.3.2 Concept of domain

Topology

The network of sites in the π_{1l}-calculus is “flat”: there is no nesting of localities, and therefore the distributed structure described in this model has only two levels, one for the processes, and the other for the localities. At the same level as localities, that is at the network level, one also finds messages going to their destination.

Semantics

A site is a unit of failure: when a locality is stopped, that is when its location process is in the state $Loc_S(a)$, it can no longer emit messages or accept migrating processes. On the other hand, it can still receive messages, but this has no observable consequence. A domain in the π_{1l}-calculus is not a communication unit, nor a migration unit.

Observability/controllability of domains

The status of a domain can be observed using the construct

$$\mathsf{ping}(a, b_1, b_2)$$

This returns a message on the name b_1 if the locality a is running, and a message on b_2 if it is stopped. In the language one also has the possibility to stop a locality, with a message

$$\mathsf{stop}(a)$$

but once a locality is stopped, there is no means to restart it. This is the only way to control the behaviour of a domain. As we said, a stopped locality cannot emit messages nor accept incoming processes.

1.3.3 Local communication

The formal definition of the calculus only describes a form of global communication, but of course local communication in a given site between a message and a receiver process is possible, as in the π-calculus. It corresponds to the following sequences of transitions:

$$\begin{array}{rcl}\{\overline{a}c \mid a(b).P \mid Q\}a \mid Loc_R(a) & \rightarrow & \overline{a}c \mid \{a(b).P \mid Q\}a \mid Loc_R(a) \\ & \rightarrow & \{[c/b]P \mid Q\}a \mid Loc_R(a)\end{array}$$

(The fact that the locality is alive is only needed at the first step.)

1.3.4 Remote communication

In the pi_{1l}-calculus, communication is, by default, remote. A message sent to the channel a enters at the network level (thus exiting the site where it has been emitted, if this site is running), and then reaches without any explicit routing its corresponding receiver (if any). This receiver is unique. In the version presented in [1], this receiver cannot move; this restriction is removed in [2].

1.4 Mobility

1.4.1 Migrating entities

The migrating entities are the processes of the calculus (a domain cannot migrate since sites cannot be nested). The messages, that are either messages in the usual sense, that is values sent on a channel, or control messages, i.e. $\mathsf{stop}(a)$ or $\mathsf{ping}(a, b_1, b_2)$ can also be regarded as migrating entities. In [1] there is a restriction on migrating processes: these cannot contain a receiver on a public channel. In [2], this restriction no longer holds, and various models of mobility, corresponding to various restrictions on the general model are discussed. The need for a mechanism to track moving receivers is mentioned, but not addressed. In the presence of failures, this is not an easy task.

1.4.2 Expressing migration

Migration is expressed by the statement

$$\mathsf{spawn}(a, P)$$

by which the process P is moved to the locality a, provided that this locality is running. This is thus an "objective" kind of mobility. The process P in $\mathsf{spawn}(a, P)$ is not allowed to run or to receive messages. Then the migration instruction is blocking with respect to the migration entity, that is, migration is synchronous.

1.4.3 Migration during execution

Since the migrating process P in $\mathsf{spawn}(a, P)$ is "passive", migration does not occur during the execution of the moving entity. In the π_{1l}-calculus, migration is a form of "remote evaluation".

1.4.4 Migrating an associated state

As in most models based on the π-calculus, in π_{1l} there is no notion of a state. When a statement $\mathsf{spawn}(a, P)$ is executed, only the process P is moved to the locality a. However, one should notice that this statement may be nested within recursive definitions, and in this case what migrates is not only P but also the processes that have been substituted for the recursion variables in P. The "state" of these recursive processes, that is their parameters, is then also moved.

1.4.5 Remote references

A migrating process keeps all the references it may have through the network, without any explicit routing or forwarding mechanism, since communication is global in π_{1l}. However, some references may become inaccessible, due to the failure of some sites.

1.5 Security

Not addressed.

2 Dπ - a distributed version of the π-calculus

2.1 References

[HR98] M. Hennessy and J. Riely. Resource Access Control in Systems of Mobile Agents. In Proceedings HLCL 98, Electronic Notes in Computer Science, volume 16, 1998.

2.2 Overview

The calculus Dπ [HR98] was designed to be a minor extension of the π-calculus with which nelementary semantic notions of distribution could be studied. A syntactic category of *locations* or *sites* is introduced and all processes now exist, and execute, at a specific named location. There is a new mechanism for processes to *move* from one site to another.

Thus a typical system takes the form

$$l[\![P]\!] \mid (\mathsf{new}\, a : \mathrm{T})(l[\![Q]\!] \mid k[\![R]\!]) \tag{1}$$

Here there are three *located processes,*

- P located at l
- Q also located at l
- R located at k.

As with the π-calculus names may be privately shared between groups of processes; in Dπ it does not matter where these are located. So in (1) above the name a may be shared privately between Q and R.

The formal syntax (annotated with types) is given in Figure 1 and is two-tiered. *Processes* look very much like π-calculus terms, with the added migration construct

$$\mathsf{goto}\, l.P$$

while *systems* may be viewed as a collection of such processes, located at specific locations and running in parallel. Thus locations form a **flat, unstructured** space. However they are treated as names in the π-calculus and so may be generated dynamically, and shared privately between groups of processes.

The language has a reduction semantics, given by a reduction relation between systems

$$M \longrightarrow N,$$

informed by that of the π-calculus. The reduction rules are given modulo a structural equivalence; this is governed by identities much as you would find in the π-calculus, such as

$$(\mathsf{new}\, n)(M \mid N) = M \mid (\mathsf{new}\, n)N \quad \text{if } \mathsf{n}(n) \notin \mathsf{fn}(M),$$

for managing *name extrusion*, and extra house-keeping ones such as

$$k[\![P \mid Q]\!] = k[\![P]\!] \mid k[\![Q]\!] \tag{2}$$

which enable systems to be re-arranged so that co-located processes may communicate.

There are two main axioms driving the reduction semantics:

$$k[\![\mathsf{goto}\, l.P]\!] \longrightarrow l[\![P]\!]$$
$$k[\![c!\langle V\rangle\, Q]\!] \mid k[\![c?(X)\, P]\!] \longrightarrow k[\![Q]\!] \mid k[\![P\{\!|^{V}\!/x|\!\}]\!]$$

$P, Q ::=$	*Processes*
$u!\langle v\rangle\, P$	Output
$u?(X : \mathrm{A})\, P$	Input
$\mathsf{goto}\, u.P$	Migration
$\mathsf{if}\ u = v\ \mathsf{then}\ P\ \mathsf{else}\ Q$	Matching
$(\mathsf{new}\, n : \mathrm{T})\, P$	Name creation
$P \mid Q$	Composition
$* P$	Replication
stop	Termination
$M, N ::=$	*Systems*
$l[\![P]\!]$	Located Process
$M \mid N$	Composition
$(\mathsf{new}\, n : \mathrm{T})\, M$	Name Creation

Figure 1: Syntax of Dπ

The first controls the migration of processes and the second communication. The latter dictates that the only communication allowed is *local*; to communicate with a non-local process it is necessary to first migrate to its site.

The values exchanged in communication may be predefined base values, such as integers or booleans, or names. Their types determine whether they are to be interpreted as location names or channel names. In the latter case they are presumed to be *local channels*. Non-local channels are transmitted using structured values; for example the structured value $a@k$ indicates the name of a (non-local) channel named a, located at the site k. Patterns can be used to deconstruct structured values into their components. Here is an example of their use, a simple server located at s:

$$s[\![*\mathsf{quest}?(x, y@z)\mathsf{goto}\, z.y!\langle \mathsf{isprime}(x)\rangle\ .\mathsf{stop}]\!]$$

At the port quest, it accepts some integer, bound to x, and a return address, deconstructed using the pattern $y@z$; it assumes z is bound to some location, and y to some channel local to that location. The server checks if the value is prime, and sends a process to the received location and reports the answer on the local channel y.

A typical client for the server, located at site c, is given by:

$$c[\![(\mathsf{new}\, r)\, \mathsf{goto}\, s.\mathsf{quest}!\langle v, r@c\rangle\, \mathsf{stop} \mid r?(z)\ldots]\!]$$

The client generates a new local channel r and sends a process to the server site s to deliver the request $\langle v, r@c\rangle$; here v is the integer to be analysed, and $r@c$ is the return address. It then awaits the reply on the local channel r.

Much of the published research on Dπ has concentrated on developing type systems for the language, particularly type systems for controlling access to resources, interpreted as local channels, and ensuring the integrity of sites.

2.3 Distribution

2.3.1 Basic entities

The basic entities in Dπ are *anonymous* agents or processes, represented syntactically by terms of the form

$$k[\![P]\!]$$

Here P is the code of the agent, and k is its current location. We say P is a process or agent *located at* k. Unlike in Pict or distributed-π [PT01], the agent itself has no name and can not be referred to either directly or indirectly.

Making domains apparent

The domains of a system are simply the set of (free) locations occurring in the syntactic description of the system. Since systems are identified up to a structural congruence containing the law (2) these domains (or locations) have only indirect representation in the syntax.

Names and references

Names, both of channels and locations, are handled in much the same way as in the π-calculus. They may be generated dynamically, and be shared privately between sets of agents using the standard mechanism of *scope extrusion*.

2.3.2 Concept of domain

A domain is considered to be a location where computation takes place. To facilitate the hosting of computations a given location provides primitive services, in the form of local channels. These services are elementary, in the sense that the provision of a channel c at a location k does not guarantee that when a process currently located at k sends a message to c, a response is guaranteed (in [ABL99], this is guaranteed by a static analysis). This form of service needs to be programmed. The typing system simply ensures that if a process calls on a channel, or resource named c at location k, then a channel with this name has been declared at k. It also enables the selective distribution of capabilities on local resources.

Topology

The topology of domains is a discrete flat structure. It can grow dynamically, but whenever a new location is generated it is simply added to this flat structure.

Semantics

As already stated locations encompass or bound computations, and facilitate computation by providing local resources. However they are static, in the sense that once they are generated they can never disappear, although of course they can become inconsequential by virtue of never being visited by agents.

Observability/controllability of domains

The observability of a location (and even individual resources there) can be controlled by types. When a new location is generated at a given declaration type, say $\mathbf{L}$, its name can be distributed selectively at supertypes of $\mathbf{L}$, thereby allowing the recipient limited access to resources at l. However simply knowing the name l allows any agent to migrate there.

Apart from this control of resources at a location, there is no mechanism from influencing the computations there.

2.3.3 Local communication

The *only* form of communication is local, that is the exchange of data on local channels. The type of the channels determines if data refers to local information, global information such as location names, or non-local information such as the names of channels at non-local sites.

2.3.4 Remote communication

This has to be programmed. To send the value v from location h to a channel a located at k it is necessary to spawn a process at h which immediately moves to h to effect the delivery:

$$h[\![\mathsf{goto}\, k.a!\langle v\rangle\, \mathsf{stop}]\!] \tag{3}$$

2.4 Mobility

2.4.1 Migrating entities

Locations are static entities. It is the agents, anonymous threads, which migrate from location to location:

$$h[\![P \mid \mathsf{goto}\, k.Q]\!]$$

Here the agent Q is sent from h to k while concurrently P continues to execute at h. Note that this may be viewed as a form of distributed *spawn* operation, or an asynchronous *remote procedure call*.

2.4.2 Expressing migration

The only form of migration is at the level of threads, as already explained. It takes the form

$$\mathsf{goto}\, k.Q$$

where Q is a process. This is an objective, blocking form of migration.

2.4.3 Migration during execution

There is no execution during migration. Indeed in $\mathsf{goto}\, k.Q$, the process only starts executing after reaching k. In a system such as (3) above , since goto is *blocking*, the thread Q can not execute while it is at h; only when it reaches k may it start.

2.4.4 Migrating an associated state

This must be programmed. See [ABL99] for an example of how, using the primitive goto a state, such as a memory, can be made to migrate dynamically between sites.

2.4.5 Remote references

These have direct expression in the syntax, with structured identifiers such as $a@k$, and the type system ensures that when such a value is received then k is really a location containing a resource called a.

2.5 Security

Security issues are not directly addressed in Dπ, but this calculus can form a basic framework in which at least certain security oriented concepts can be formalised and investigated. For example in [HR98, RH99] the type system is extended so as to allow for untrusted, or indeed malicious, agents to be described. Two simple forms of dynamic typechecking are described, in which agents are interrogated on the basis of their origin. A proof is then given that this protects *good* locations from the behaviour of malicious agents.

3 A Lexically Scoped Distributed π-calculus (*lsd*π) and the Distributed Typed Concurrent Objects (DiTyCO)

3.1 References

[VLS98] Vasco T. Vasconcelos, Luís Lopes and Fernando Silva. Distribution and Mobility with Lexical Scoping in Process Calculi. In Proceedings HLCL 98, ENTCS 16(3), 1998.

[HR98] M. Hennessy and J. Riely. Resource Access Control in Systems of Mobile Agents. In Proceedings HLCL 98, Electronic Notes in Computer Science, volume 16, 1998.

[LFSV00] Luís Lopes, Álvaro Figueira, Fernando Silva and Vasco T. Vasconcelos. A Concurrent Programming Environment with Support for Distributed Computations and Code Mobility. In Proceedings CLUSTER'00, IEEE Press, 2000.

[RMVL02] António Ravara, Ana G. Matos, Vasco T. Vasconcelos and Luís Lopes. A Lexically Scoped Distributed π-Calculus. DI/FCUL TR 02–4, Department of Computer Science, University of Lisbon, 2002.

3.2 Overview

DITYCO is a proposal for a simple model of distribution for mobile processes that adds a distribution layer to an underlying name-passing (π-calculus like) calculus: conventional processes compute within sites; inter-site computation is achieved by message sending and object migration, both obeying a lexical scope [VLS98]. Programming in DITYCO is network-aware: channels can be local or remote; the distinction is explicit in the syntax.

*lsd*π [RMVL02], the theoretical foundation of DITYCO, is an asynchronous and distributed π-calculus with local communication on named channels and process migration, along the lines of DPI [HR98]. Unlike DPI, where all channels are global and the programmer must explicitly locate channels in migrating processes, *lsd*π follows the DITYCO model of distribution for mobile calculi, which provides both for remote communication and for process migration, making explicit migration primitives superfluous. Included in DITYCO, but not present in the current version of *lsd*π, is the ability to download process definitions from remote servers.

When embodying π-calculus with a distributed setting, the notion of "resources of a site" identifies naturally with the collection of channels of that site. Formally, this information may be explicit in the name of the channel, e.g. using a suffix '@*s*'. Cohabitation of both simple and compound forms of channel names may lead to confusion as to the physical location of a channel. For instance, in DPI, we have the following one-step reduction.

$$s[(\nu x)\,(\text{go } r.x?()P \mid x?()Q)] \to (\nu x@s)\,(s[x?()Q] \parallel r[x?()P])$$

On the right-hand side term, site r cannot internally identify the physical location of channel x. Moreover, DPI presents no notion of location of a variable (a notion important for programmers), which extends naturally to a distributed setting. *lsd*π consolidates these issues by imposing a lexical scope discipline to the networks. By inspecting the name of the channel and its site of occurrence, the physical location of a channel can be known at all times.

The implementation of DITYCO has three layers: network, nodes and sites, although only two—network and sites—are logical [LFSV00]. Sites form the basic sequential units of computation. Nodes have a one-to-one correspondence with physical IP nodes and may have an arbitrary number of sites computing either concurrently or in parallel. This intermediate level does not exist in the formal model. It makes the architecture more flexible by allowing multiple sites at a given IP node. Finally, the network is composed of multiple DiTyCO nodes connected in a static IP topology. Message passing and code mobility occurs at the level of sites, and at this level the communication topology is dynamic.

3.3 Distribution

3.3.1 Basic entities

The basic entities of *lsdπ* are: (1) a structured set of names; (2) processes; and (3) networks. Networks are collections of sites in parallel, sites being named areas where processes run sharing a local name space. Names refer both to sites and to channels (local or remote).

Names and references

There are two kinds of names: channels (denoted by a)—used as links for synchronising processes via communication—and locations (denoted by s)— used to give identity to the sites. Each free channel belongs to a specific site, fixed throughout the computation. The calculus adheres to the *lexical scoping in a distributed context* of Obliq, meaning that, in order to determine where a certain free channel belongs to, one just have to inspect the code for the process where the channel occurs. The rule is simple: located channels $a@s$ belong to the site where they are (*explicitly*) located, s; simple channels a are (*implicitly*) located at the current site, the site where the process they occur at is located. In the present version of the calculus, locations (site names) can not be passed around.

Names are built on top of two (countable and disjoint) sets: *simple channels* a,b,x,y, and *sites* s,r,t. Compound channels—pairs channel-site, like $a@s$—form *located channels*, designating a channel a at site s. Let $u,v,\ldots$ stand for both simple and located channels, and let $\tilde{v}$ stand for a sequence of channels. Take x as a variable ranging over simple channels and $\tilde{x}$ as a sequence of pairwise distinct variables. Furthermore, let n,m stand for both sites and channels, and let g,h stand for both sites and located channels.

$$\begin{array}{rrcccccc}
\textit{Channels,} & u,v & ::= & a & \mid & a@s & & \\
\textit{Globals,} & g,h & ::= & & & a@s & \mid & s \\
\textit{Names,} & n,m & ::= & a & \mid & a@s & \mid & s
\end{array}$$

Making domains apparent

A site is the construction of the calculus that materialises the notion of domain. Its constitutive entities are a name—its identity—and a process—its contents. In a network, a site may appear divided into several components, i.e., there may be several occurrences of sites with the same identifier. The following grammars inductively define the sets of processes and networks.

$$\begin{array}{rrl}
\textit{Processes,} & P,Q & ::= \ \mathbf{0} \ \mid \ (P \mid Q) \ \mid \ (\nu n)P \ \mid \ u!\langle\tilde{v}\rangle \ \mid \ u?(\tilde{u})P \\
\textit{Networks,} & N,M & ::= \ \mathbf{0} \ \mid \ (N \parallel M) \ \mid \ (\nu g)N \ \mid \ s[P]
\end{array}$$

Receptors, of the form $u?(\tilde{u})P$, and *messages*, of the form $u!\langle\tilde{v}\rangle$, are the basic processes in the calculus. A receptor is an input-guarded process. A message has a name u for target and carries a sequence of channels $\tilde{v}$. The remaining constructors are fairly standard in name-passing process calculi: process $(P \mid Q)$ denotes the *parallel composition* of processes; process $(\nu n)P$ denotes the *restriction of the scope* of the name n to the process P (often seen as the creation of a new name or site, visible only within P; moreover, if n is a site, no channel explicitly located at that site is visible outside P); and *inaction* $\mathbf{0}$, denotes the terminated process.

The basic units of *Networks* are processes running at a given site, $s[P]$, where free simple channels in P are *implicitly located* at s (while located channels are considered to be *explicitly located*). The parallel composition of networks, $(N \parallel M)$, performs a simple merge of networks, and $(\nu g)N$ represents the restriction of the scope of a global to a network. The empty network *inaction* is denoted by $\mathbf{0}$.

3.3.2 Concept of domain

Topology

A network is a flat space of sites: sites cannot contain other sites, and each site is directly accessible from any other site. The novelty of *lsdπ* is the communication topology that differs substantially from that of DPI.

The authors envisage a "natural" definition for the free names of a process or of a network, according to the classical definitions for the λ-calculus, and meeting the intuitions of any π-calculist. A channel is local to a site if it occurs as a simple channel in that site, or if it occurs explicitly located at that site anywhere in the network. Amongst the binders of the calculus, two cases deserve a special mention: $(\nu s)\,N$ makes all free channels local to s invisible outside N; and $(\nu\, a@s)\,N$ creates a new free site s. The free names of a network $s[P]$ are s and the free names of P, where the simple channels are made explicitly located at s.

Lexical scoping together with compound names introduce subtleties in the definition of free names. Consider the network $r[(\nu a)\, a@s!\langle\rangle]$. It is not clear whether channel a belongs to r or to s, as it is not obvious whether $a@s$ should be free or bound. Now consider the network $r[b?(x)x?()(\nu a)\, a@s!\langle\rangle]$, which after interacting with another network like $r[b!\langle c@t\rangle]$, yields $t[c?()(\nu a)\, a@s!\langle\rangle]$. Channel a belongs to whatever site x belongs to, possibly neither r nor s (t in this case). *lsdπ* addresses this problem by imposing *syntactic restrictions* on processes otherwise defined by a context-free grammar (rejecting, among others, the above processes), and makes sure that these restrictions carry through, all the way from alpha-congruence to reduction.

Semantics

The reduction semantics of *lsdπ* is based on the notions of substitution, alpha-congruence, structural congruence, and finally reduction. Inside a site processes compute by communication, just as in the π-calculus, turning sites into units of local computation. Networks evolve due to migration of processes between sites, as well as via local computations within sites.

Values are simple channels a and located channels $a@s$; parameters are simple channels only. This means that one cannot rely on the usual substitution of the π-calculus, where one substitutes channels by channels. To make sure that the concept is right, *lsdπ* uses a general notion of substitution: a total function on names (sites, channels, located channels), defined along the lines of the substitution for the λ-calculus. If not treated carefully, alpha congruence could lead syntactically correct processes into incorrect ones. The function referred above is then used to define *name replacement* (used in alpha-congruence), *name instantiation* (the substitution that arises from communication, as in $a!\langle\tilde{v}\rangle \mid a?(\tilde{x})P$), and *name translation* (the substitution that happens during migration, as in $a@s!\langle\tilde{v}\rangle$ or $a@s?(\tilde{x})P$).

The structural congruence rules for networks are inspired on those proposed by Hennessy and Riely for DPI, while those for processes are adapted from the standard rules of the π-calculus. Some of these rules deserve a special mention:

1. $(s[P] \parallel s[Q]) \equiv s[P \mid Q]$

   Processes within a site may be split/aggregated, allowing for the isolation of processes prepared for migration.

2. $((\nu a)\,P) \mid Q \equiv (\nu a)\,(P \mid Q)$, if $(\{a, a@\_\} \cap \mathsf{fn}(Q) = \emptyset$

   $(\nu\, a@s)\, s[P] \equiv s[(\nu a)\, P]$, if $a@\_ \notin \mathsf{fn}(P)$

   $(\nu a)\,(\nu s)\,P \equiv (\nu s)\,(\nu a)\,P$, if $a@s \notin \mathsf{fn}(P)$

   These rules define the scope a binder may take: the scope can expand and contract in such a way that no name is captured or released, and no syntactic conflict appears, acording to the definition of free names.

3. $s[a@s?(\tilde{x})P] \equiv s[a?(\tilde{x})P]$

   $s[a@s!\langle\tilde{v}\rangle] \equiv s[a!\langle\tilde{v}\rangle]$

   These rules embody the notion that simple channels always belong to the site where the process is running. This clarification is needed only at communication time, when the channel is actualy used.

The rules for reduction are based on the following axioms:

1. $a?(\tilde{x})P \mid a!\langle\tilde{v}\rangle \to P\{\tilde{v}/\tilde{x}\}$, standard in the π-calculus.
2. $r[a@s!\langle\tilde{v}\rangle] \to s[(a@s!\langle\tilde{v}\rangle)\Upsilon]$, $\Upsilon = \sigma(a@s!\langle\tilde{v}\rangle, r, s)$, if $r \neq s$

   $r[a@s?(\tilde{x})P] \to s[(a@s?(\tilde{x})P)\Upsilon]$, $\Upsilon = \sigma(a@s?(\tilde{x})P, r, s)$, if $r \neq s$

   These rules were proposed by Vasconcelos *et al.* for DiTyCO. The substitution is defined in order to keep channels “local by default”: when migrating process P from site r to site s, the free channels of P must be renamed accordingly: thus simple channels become explicitly located at r; channels located at s become simple (dropping ‘$@s$’); all the others remain as they were.

The type system is a simplified form of that of DPI, adapted to deal with the lexical scope of channels. Types for channels are the usual types in the simply typed π-calculus, $Ch(\gamma_1,\ldots,\gamma_n)$, describing a channel capable of carrying a series of channels of types $\gamma_1,\ldots,\gamma_n$. The types of sites only capture the types of the (free) channels at the site: if a_1 to a_n of types γ_1 to γ_n contain the free channels of site s, then site s has the type $\{a_1{:}\gamma_1,\ldots,a_n{:}\gamma_n\}$. The type system is simple and intuitive—a straightforward extension of that for simply typed π-calculus; it assumes types for sites only (types for channels are taken from that of the site the channel belongs to), and enjoys subject-reduction.

Observability/controllability of domains

Not addressed.

3.3.3 Local communication

Only this form is allowed, as described above. Communication is restricted to processes located at local channels, allows names to be exchanged, and is disciplined by a simple type system à la π-calculus.

3.3.4 Remote communication

There is no form of remote communication: a process prefixed at a located channel ($a@s$) must first migrate to its site (s) in order to engage in a communication. A type system guarantees that any migrating process respects the arity of the channels used in the destination site.

3.4 Mobility

3.4.1 Remote references

Remote references are denoted by explicitly located channels $a@s$. Moreover, programmers may refer to a channel a belonging to a site s by its local name a or by its remote name $a@s$, reflecting two distinct views of a channel: the local view and the network (global) view.

3.4.2 Migrating entities

Processes prefixed at a located channel (remote messages $a@s!\langle\tilde{v}\rangle$ and remote receptors $a@s?(\tilde{x})P$) are the migrating entities of the calculus. Although there is no form of migration for arbitrary processes, it can be encoded in the calculus at the expenses of an extra message exchange (and two migrations): $[\![\text{go } s.P]\!] \stackrel{\text{def}}{=} (\nu\, a@s)\,(a@s!\langle\rangle \mid a@s?()[\![P]\!])$.

3.4.3 Expressing migration

Migration is subjective (i.e. self-inflicted) and implicit in the lexical scope of channels. Processes located at a remote channel migrate towards the location where the channel belongs to, according to the reduction rules for process migration.

3.4.4 Migration during execution

*lsd*π supports weak mobility only: inert code is migrated prior to its execution, as it is clear from the rules for process migration.

3.4.5 Migrating an associated state

Free variables in a migrating process are translated, in order to preserve the original lexical binding, reflecting its new location. Simple names, which were local, become remote, so they must be explicitly located at the source site, thus presenting their compound form; names located at the target site become local, so they drop their site annotation, thus becoming simple; all other names remain unchanged.

3.5 Security

Not addressed.

4 Pict and NomadicPict

4.1 References

[Tur96] D. Turner, The Polymorphic Pi-calculus: Theory and Implementation, PhD Thesis, University of Edinburgh (1995).

[Pie97] B. Pierce, Programming in the Pi-Calculus: A Tutorial Introduction to Pict (Pict Version 4.0), Technical Report, Indiana University, March 1997.

[SWP98] P. Sewell, P. Wojciechowski, B. Pierce, Location-independent communication for mobile agents: a two-level architecture, Tech. Rep. 462, Computer Lab, University of Cambridge (1998).

The reference web page for PICT is:

http://www.cis.upenn.edu/ bcpierce/papers/pict/Html/Pict.html

and the one for NOMADICPICT is:

http://www.cl.cam.ac.uk/users/pes20/nomadicpict.html

4.2 Overview

Here we survey both PICT and NOMADICPICT. The former is a programming language based on the π-calculus, which does not support distributed programming nor mobile code, while the second, which integrates the former, introduces a few primitives for programming migrating agents in a distributed environment.

The π-calculus is a process calculus where communication uses named channels, which is suited for the description of concurrent systems whose communication topology is dynamically evolving. In a sense the π-calculus is a model for mobile processes, since in it one has the ability to pass the name of a channel as a communicated value, and such a name also serves to access processes (those which are waiting for an input on that channel name).

PICT is a high-level programming language, using a functional style, which is built on a set of primitive constructs called COREPICT. This is an asynchronous π-calculus (without summation), statically typed, with product and record types. The more abstract constructs are introduced as syntactic sugar, as they may be translated in the core calculus. NOMADICPICT extends this with a model for migrating agents, which is very close to the one originally proposed with TELESCRIPT.

4.3 Distribution

4.3.1 Basic entities

There are four kinds of basic entities in NOMADICPICT: channels, which are simply names, processes, written using the standard constructs of the π-calculus, agents, that are named processes $a[P]$, and named sites, where agents are run. The latter are only known by their names in the language.

Making domains apparent

There is no construct for domains in PICT, as this language does not take spatial distribution into account. In NOMADICPICT, sites are domains, but there is no specific construct for them in the language: they can only be named, for migration purposes. An agent $a[P]$ can also be regarded as a domain, though with a different semantics. The linguistic construct for introducing agents is

$$\mathsf{agent}\ a = P\ \mathsf{in}\ Q$$

which builds a new agent named a with body P, which is run in parallel with Q, in the same site. This construct binds the name a, and therefore builts a fresh name for each newly created agent.

Names and references

The named channels may be regarded as references to processes. In PICT and NOMADICPICT the naming space is global: there may be several emitters and/or receivers on a given name, and the corresponding channel is unique. Its name may be used in any site, and its scope is lexical, and is determined by the restriction operator ν of the π-calculus. Using this operator requires a mechanism for generating globally fresh names.

4.3.2 Concept of domain

Topology

In NOMADICPICT there is a two-level hierarchy of domains, each with a different semantics: agents contain processes, and are executed in sites.

Semantics

A site in NOMADICPICT is a communication unit: the agents that are running in the same site may communicate with each other. In an agent, the processes can communicate, by means of named channels as in the π-calculus. An agent is a unit of migration.

Observability/controllability of domains

There is no means in NOMADICPICT to control a domain. Agents can be observed in a sense, since another agent can detect if an agent named a is located in the same site, and in this case it can send a message to a, or more accurately to a channel of a.

4.3.3 Local communication

Local communication occurs in agents as in the asynchronous π-calculus, that is through named channels, with a blocking input construct (replicated or not). The values of the core language, and in particular the names (of channels, agents and sites), can be communicated. In a site, the agents can communicate using a specific construct

$$\mathsf{iflocal}\ \langle a \rangle c!v \rightarrow P\ \mathsf{else}\ Q$$

The semantics is that if an agent b is executing this instruction, and if the agent a is in the same site as b, then the message $c!v$ is sent to a (that is, it is added to the processes running inside a), and the continuation P is executed in b. If there is no agent a co-located with b, the continuation Q is executed in b.

4.3.4 Remote communication

There is no remote communication in NOMADICPICT: each communication must occur within a site. However, a protocol for transparent remote communication is implemented in the language, using migration, and this provides a higher level mode of communication.

4.4 Mobility

4.4.1 Migrating entities

The migrating entity in NOMADICPICT is the agent, that is a named process $a[P]$, which can move from site to site. Also, a message can be sent from an agent to another.

4.4.2 Expressing migration

Migration occurs when a statement

$$\mathsf{migrate\ to}\ s \rightarrow P$$

is executed. This is a subjective construct: the agent enclosing this statement is moved to the site s, and then the process P is added to this agent. This is similar to migration in JOIN or AMBIENTS, although the topology of domains is different (and there is no way to “dissolve” an agent like an ambient with the open construct).

4.4.3 Migration during execution

The migration statement can be used as any other construct of the language, and therefore the movement of an agent can occur at any time during the execution of this agent. Moreover, migration is asynchronous with respect to the processes executing in the migrating agent, and therefore, to synchronise the behaviour of an agent with its movements, one has to use the continuation P of the expression $\mathsf{migrate\ to}\ s \rightarrow P$ (though one cannot control the incoming messages).

4.4.4 Migrating an associated state

There is no explicit notion of state in PICT. However, as in the π-calculus, one can encode reference cells for instance by means of channels. Since the messages and the receivers that an agent contains move as part of the agent during migration, one may consider that the state of the agent is moving with the agent itself. However, part of the state of the agent may be located in another agent, and this part is lost.

4.4.5 Remote references

The migrating agent still knows after having migrated the channel names that are free in the agent. However, these names cannot be considered as remote references, since they can only be used for communication inside an agent, or, at the site where the agent is, by means of the “iflocal” construct. The “else” clause of this construct provides a kind of exception, raised if the communication partner is not there, at the same site.

4.5 Security

Not addressed.

5 The SPI-calculus

5.1 References

[AG99] M. Abadi, A.D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1-70, Academic Press, 1999.

[Bor01] M. Boreale. Symbolic trace analysis of cryptographic protocols. *ICALP'01*, LNCS 2076, pp.667-681, Springer-Verlag, 2001.

[BNP99] M. Boreale, R. De Nicola, R. Pugliese. Proof Techniques for Cryptographic Processes. *LICS'99*, Proceedings, IEEE Computer Society Press, pp.157-166, 1999. Full version to appear in *SIAM Journal on Computing*.

5.2 Overview

The spi-calculus extends the π-calculus with a set of cryptographic primitives (shared- and public-key encryption, hashing, ...): this allows for the description of security protocols as systems of concurrent processes that can exchange encrypted data. A distinguishing feature of the spi-calculus is its reliance on the powerful scoping constructs of the π-calculus to get a clean formalization, at a linguistic level, of such concepts as 'nonce', and 'newly generated key'. The calculus is mainly intended for analysis and verification of security protocols, only marginally as a 'core' programming language.

5.3 Distribution

5.3.1 Basic entities

Processes are the basic entities of the calculus. Distribution is absent, in the sense that all processes belong to a single site.

Making domains apparent Since distribution is absent, this aspect is not addressed in the calculus.

Names and references There are essentially two datatypes: *names* and *ciphertexts*. Depending on its usage, a name can represent either a basic data value, an encryption key or a channel. A name can also serve as a reference to a more complex object, possibly coded up as a process. Exactely like in π-calculus, new names can be invented at run time using the restriction operator, which is lexically scoped. Ciphertexts result from encrypting any piece of data with a key.

5.3.2 Concept of domain

Not addressed.

5.3.3 Local communications

Synchronous message passing, à la π-calculus.

5.3.4 Remote communications

Absent.

5.4 Mobility

5.4.1 Migrating entities

Names and ciphertexts are the only entities that can explicitly be passed around.

5.4.2 Expressing migration

Exactly like in π-calculus, names representing references can freely be passed around, thus causing the set of resources/objects available to a process to change over time. This is the only form of mobility that can be expressesed directly in the calculus. More complex forms of mobility, like process/agent passing, can be simulated by reference-passing.

5.4.3 Migration during execution

Not addressed.

5.4.4 Migration during execution

Not addressed.

5.4.5 Remote references

Not addressed.

5.5 Security

Its built-in cryptographic primitives make the spi-calculus ideal for modelling communication of encrypted data (including coding of agents) over public, possibly insecure, channels. More complex security services can be coded up as protocols.

Type and sorting systems can be adopted to control the way processes use channels, keys and other resources. However, when programming or reasoning on spi-calculus processes, the point of view is taken that an external spi-calculus-calculus process (e.g., an attacker) may not, in general, obey the given typing/sorting discipline.

6 The DJoin-calculus

References

[FG96] C. Fournet, G. Gonthier, The reflexive CHAM, and the join calculus, POPL'96 (1996) 372-385.

[FGL$^+$96] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, D. Rémy, A calculus of mobile agents, CONCUR'96, LNCS 1119 (1996) 406-421.

[Lév97] J.-J. Lévy, Some results in the Join-calculus, TACS'97, LNCS 1281 (1997) 233-249.

6.1 Overview

The JOIN calculus with localities (DJOIN) is a model for mobile programming that includes explicit features for co-localisation of receivers and partial failure, especially conceived to support a distributed implementation. This model is equipped with a "chemical semantics", based on the reflexive abstract machine RCHAM.

6.2 Distribution

6.2.1 Basic entities

The basic entities of the DJOIN-calculus are messages, *definitions* (which play the role of passive rules), *def-processes* (processes containing a definition), *patterns* of joint messages and *solutions* (multisets of def-processes, multisets of active rules).

Making domains apparent

Domains are given by localities in the DJOIN. A locality manifests itself as a locality definition (loc-definition), and through the concepts of *localised solution* and *distributed solution* (parallel composition of localised solutions).

Names and references

The DJOIN-calculus manipulates channel names and locality names, governed by a rule of syntactic scoping. A locality name denotes a unique location. Names have a distributed scope, in the sense that they may be referenced from a remote locality. This implies the existence of an infrastructure mechanism to handle the global resolution of names. Different processes may receive on the same channel name, but in this case they must be co-localised (cf. semantics of local reception).

6.2.2 Concept of domain

Topology

Localities constitute a tree-structure. Moreover the set of localities determines a partition over receptors. The concept of ether, in which the localised solutions float, is not made explicit. Because of the global unicity of locality names, any process at a given locality can reference any other locality directly by its name. The hierarchical structure of localities is relevant for migration and failures, but not for communication.

Semantics

A DJOIN-locality, different from the root locality, is both a migration and a failure unit. A locality may migrate in an atomic way, and its migration entails the migration of all its sublocalities. Similarly, a locality can fail, provoking the simultaneous failure of all its processes and of all its sublocalities. A failed locality cannot exhibit any reaction, nor receive any message or migrating locality. The handling of partial failures is a crucial feature of the semantics of localities in the DJOIN-calculus.

A locality is also an execution unit. Communication may of course take place inside a locality but localities are not communication units, since communication across localities is possible.

Observability/controllability of domains

A locality may be marked as "live" or "dead", and may be stopped (in a subjective way, i.e. by effect of an instruction contained in the locality itself). The state of a locality may be observed (in a objective way, i.e. from the outside the locality).

Furthermore, since the creation of localities may be nested within other constructions of the language, this creation may occur dynamically. This gives some form of control over domains.

6.2.3 Local communication

Local naming

The local naming of a channel is direct, via its name.

Local communication policy

Local communication is asynchronous (in the sense that message emission is non-blocking), but it allows the synchronised reception of several messages. It is also unicast, and doesn't allow more than one receiver on a given channel name during every execution. The transmitted arguments are channel names or locality names.

Semantics of reception

Several processes may receive on a given name provided they reside in the same locality. Local reception on a name is static (there is no "capture" of receptors through communication). Moreover the join-patterns are replicated. A received name becomes known, ans thus may be used as the target of a message.

6.2.4 Remote communication

Since the names defined in a join-pattern are globally unique, there is no difference between local and remote communication, apart from the fact that the latter can be prevented by a failure of one of the concerned localities.

Remote naming

The referencing of a distant channel is direct (through its name). A name is defined in a unique locality, therefore it is possible to use distant referencing in a way which is transparent wrt localisation. Similarly, referencing of a locality may be done directly through the name of the locality.

Distant communication policy

Distant communication is asynchronous, and takes place in two atomic steps: the exit of the message from the current domain followed by its entering the remote domain (first step), and the actual consumption of the message in the target domain, if this domain is able to find partners that match the message by an appropriate join-pattern (second step). Both steps are possible only if the concerned domains are live (in fact a message can still enter a failed locality).

Logical Routing

Due to the transparency assumption, routing in the tree of localities is “spontaneous” and therefore not made explicit.

6.3 Domains and failures

The failures that are taken into account are the permanent failures, which only apply to localities different from the root. The failure of a locality implies the failure of all its sublocalities. A failed locality cannot emit any message, nor be the source of any migration towards another locality. On the other hand, a message or a locality may migrate towards a failed locality. The failure of a locality may be detected from any other live locality. The “activation” of a locality definition remains possible in principle even if this locality has failed.

6.4 Mobility

6.4.1 Migrating entities

In the DJOIN-calculus only localities can migrate. A migrating locality becomes an immediate sub-locality of the locality designed as the target of the migration. This has the effect of modifying the topology of the locality tree, but does not have any influence on communication, except through failures: the migration of a locality towards another one has the effect of making the moving locality dependent on the target locality as concerns failures. Conversely, a locality may escape the failure of one of its ancestor localities by moving to another locality.

There is no possibility for an individual process (or for a definition) within a locality to exit this locality, unless the process is a simple message targeted at a name defined elsewhere. One of the DJOIN-calculus characteristics is in fact that a definition is associated with a unique locality and statically determined.

6.4.2 Expressing migration

Migration is expressed by the instruction $\mathtt{go}(a,\kappa)$ where a is the locality of the destination and κ is a continuation. The effect of this instruction is to turn the surrounding locality (the one that contains the instruction) into a sub-locality of a, and to emit the signal κ when the operation is terminated. It is therefore a subjective form of migration.

This instruction is asynchronous wrt the migrating entity. In particular, if some message is targeted to the locality in which the $\mathtt{go}(a,\kappa)$ is executed, the moment in which this message will be handled (and the handling itself) is a priori independent from the moment in which the migration happens. If the contents of the locality is a sequential program, its execution can be synchronised with the migration, using the continuation κ.

6.4.3 Migration during execution

The migration instruction may be used as any other instruction of the language, hence the migration of a locality may take place while processes inside it are still executing. It is possible to control to some extent the behaviour of the migrating entity (by blocking the processes until the migration is finished), but not the flow of incoming messages.

6.4.4 Migrating an associated state

There is no real notion of state associated with a locality. One could say that the context of the migrating entity remains fixed – taking the context to be the set of definitions for which the locality possesses a reference, but which are not in the locality itself.

6.4.5 Remote references

A migrating locality maintains its links with the context, which are in fact just names, and are sufficient to establish remote communications. This preservation of links is again completely transparent.

6.5 Security

There exists a version of the JOIN-calculus, the SJOIN, that integrates under an abstract form primitives for encrypting and decrypting messages, analogous to those of the SPI-calculus (cf the corresponding fiche). The study of this version of the calculus is deferred to a later time.

7 Mobile Ambients

[CG98] L. Cardelli and A. D. Gordon. Mobile ambients. In M. Nivat, editor, *Proceedings of Foundations of Software Science and Computation Structures (FoSSaCS)*, volume 1378 of *LNCS*, pages 140–155. Springer-Verlag, Berlin, 1998.

[CG99] L. Cardelli and A. D. Gordon. Types for mobile ambients. In *26th Annual Symposium on Principles of Programming Languages (POPL) (San Antonio, TX)*, pages 79–92. ACM, 1999.

[Cas01] I. Castellani. Process algebras with localities. In J. Bergstra, A. Ponse, and S. Smolka, editors, *Handbook of Process Algebra*, pages 945–1045. North-Holland, Amsterdam, 2001.

[GC99] A. D. Gordon and L. Cardelli. Equational properties of mobile ambients. In W. Thomas, editor, *Proceedings FoSSaCS '99, (Amsterdam, The Netherlands, April 1999)*, volume 1578 of *LNCS*, pages 212–226. Springer-Verlag, Berlin, 1999.

7.1 Overview

Cardelli and Gordon's calculus of *Mobile Ambients* (MA) [CG98] was the first process calculus containing an explicit notion of mobile domain. Its extreme simplicity and elegance, along with its expressive power, is presumably the reason of its immediate success and its following widespread acceptance as the new basic theoretical paradigm for mobile computing.

An *ambient* is a named area delimited by a boundary, where multi-threaded computation takes place. In other words, an ambient is a named container that may host any number of π-calculus-like processes running in parallel and locally communicating with each other; the communication primitives are $(x).P$, which performs an input that binds the variable x in P, and $\langle M\rangle$, which outputs the expression M.

Processes, as is standard in process calculi, are intended to model, through the usual basic constructs of sequential and parallel composition, sequential threads of computations running concurrently: sequential composition allows to build a new process $M.P$ out of a sequence of instructions M and a possibly multi-threaded process P; parallel composition allows to build the new process $P|Q$ consisting of the processes P and Q running in parallel. As usual, the null process is the basis for the inductive process construction.

Ambients are themselves processes; as a matter of fact, from a formal point of view, the syntactic category *ambient* does not exist: there are only (ambient) names $m,n,\ldots$, and the construct $n[P]$ which represents a process consisting of an ambient named n containing the (usually multithreaded) process P.

An ambient is thus a very general construct, intended to model a computing location, or an administrative domain, a software agent, a computing resource, etc. Unlike in D-π, different occurrences of a name n in expressions of the form $n[\ldots]$ represent distinct ambients with the same name: $n[P] \mid n[Q]$ is not equivalent to $n[P \mid Q]$.

As is apparent from the syntax given below, an ambient-process may contain other ambient-processes along with "naked" processes: ambients, which are by definition non-overlapping, can therefore be nested at any level.

Mobility is given by the instructions $a?l\langle\tilde{v}\rangle n$ and $a!l\langle\tilde{v}\rangle n$, which cause the enclosing ambient to respectively enter or exit an ambient named n, and by the instruction $\mathsf{open}\, n$, which opens an ambient named n. The complete syntax is:

$$\begin{array}{rcl} M,N & ::= & n \mid x \mid a?l\langle\tilde{v}\rangle M \mid a!l\langle\tilde{v}\rangle M \mid \mathsf{open}\, M \mid \varepsilon \mid M.M' \\ P,Q & ::= & 0 \mid M.P \mid P \,\|\, Q \mid (\nu n)P \mid\, !P \mid M[P] \mid (x).P \mid \langle M\rangle \end{array}$$

The operational semantics is given by the reduction rules supplemented by the definition of a structural congruence $\equiv$, which allows redexes to be formed by trivial syntactic restructuring of terms. The reduction rules for mobility and communication are:

$$\begin{array}{lcl}
n[a?l\langle\tilde{v}\rangle m.P \mid Q] \mid m[R] & \rightarrow & m[n[P \mid Q] \mid R] \\
m[n[a!l\langle\tilde{v}\rangle m.P \mid Q] \mid R] & \rightarrow & n[P \mid Q] \mid m[R] \\
\mathsf{open}\, m.P \mid m[Q] & \rightarrow & P \mid Q \\
(x).P \mid \langle M\rangle & \rightarrow & P\{x \leftarrow M\}
\end{array}$$

The contextual reduction rules are:

$$\begin{array}{rcl}
P \rightarrow Q & \Rightarrow & n[P] \rightarrow n[Q] \\
P \rightarrow Q & \Rightarrow & (\nu n)P \rightarrow (\nu n)Q \\
P \rightarrow Q & \Rightarrow & P \mid R \rightarrow Q \mid R \\
P' \equiv P,\ P \rightarrow Q,\ Q \equiv Q' & \Rightarrow & P' \rightarrow Q'
\end{array}$$

The structural congruence is defined as the minimal reflexive, transitive and symmetric relation which is a congruence and moreover:

1. satisfies $!P \equiv P \mid !P$;
2. makes the operator $\mid$ commutative, associative, with 0 as zero element;
3. allows to stretch the scopes of restrictions, to permute restrictions and to cancel a restriction followed only by 0;
4. satisfies $\varepsilon.P \equiv P$ and $(M.M').P \equiv M.M'.P$.

7.2 Distribution

7.2.1 Basic entities

As recalled in the overview, the MA calculus has an explicit simple notion of domain as a particular kind of named container-process, isolated from its environment by a frontier that cannot be crossed by communication.

Making domains apparent

Formally, an ambient is an occurrence of the construct $M[P]$. A simple type system such as the one in [CG99] ensures that meaningless terms such as $(a?l\langle\tilde{v}\rangle m)[P]$ or $n.P$ cannot arise, thus remedying the "syntactic anomaly" in the calculus definition. An occurrence under a prefix or an input represents an ambient still to be activated, a sort of creation instruction still to be executed.

As remarked in the overview, different occurrences of the construct $n[P]$ with the same n represent distinct ambients with the same name; any action involving the name of sibling homonymous ambients selects nondeterministically one of them. Thus MA may be used to model service providers, where the resource's or even the server's identity is immaterial, as long as they provide the same service.

Names and references

Mobility primitives move (whole) ambients, while communication either passes names, i.e., (possibly ambiguous) references to ambients, or it passes capabilities, which also act through names ($a?l\langle\tilde{v}\rangle n$, $a!l\langle\tilde{v}\rangle n$, etc.).

7.2.2 Concept of domain

Topology

The topology of the domain space results to be a forest of trees, and it is thus well suited to model hierarchical authorization domains arising from the treatment of security.

Semantics

An ambient is a mobility, communication, and security unit, in the sense that:

1. An ambient migrates as a whole, driven by some enclosed process, and taking with them all the enclosed processes and ambients.
2. Direct communication through input/output primitives is only allowed within an ambient; communication between distinct ambients may be performed via ambient movement and opening. The ambient is therefore an essential notion for communication also in the sense that it serves to model messages travelling from one location to another.
3. Finally, the fact that mobility and opening primitives only concern whole ambients and no special purpose security primitive is provided, automatically makes the ambient the elementary security cell.

Observability/controllability of domains

Each ambient's external behaviour, which amounts to its mobility, is collectively controlled by all the processes running within its boundary; there is no notion of a privileged control process. Such control, coming from inside the ambient, is a form of subjective control. The dissolution of the ambient's boundary, however, which is the most disruptive and potentially dangerous operation, can only be performed by a sibling process external to the ambient: which is a strong form of objective control. Finally, a prefixed ambient, i.e., a process of the form $M.N[P]$, or an ambient guarded by an input operation, like $(x).M[P]$, can be considered as controlled by the prefix or input operations, since it comes into existence (and therefore becomes observable) only when (and where) the prefix has been completely consumed or the input performed.

7.2.3 Local communication

Local communication is anonymous and asynchronous; it could be viewed as a form of π-calculus communication on the anonymous channel belonging to each ambient (and not visible outside that ambient). Asynchronicity is, as usual, simply obtained by the syntactic constraint that an output cannot prefix a process but may only occur as the last operation. Input is blocking, of course, its sequential thread. Ambient names, capabilities (i.e., mobility instructions) and sequences of capabilities may be passed as arguments; this is essential for handling security, as will be explained at the point 7.4.

7.2.4 Remote communication

Extra-domain communication is forbidden in the MA calculus: communication and interaction between different domains can only be obtained through ambient migration and opening; the notion of ambient therefore also models messages, packets, and any kind of migrating processes.

It must be stressed that the existence of the objective open primitive is the only means to ensure the possibility of remote communication and interaction in a calculus where input/output is purely local: the objective capacity to dissolve ambients and the local character of communication are therefore strictly connected.

7.2.5 Domains and failures

The calculus, being primarily motivated as a model for global computing on the network, does not possess a notion of domain failure, for reasons very clearly expressed by the authors in the original paper [CG98]: “on the Internet, node failure is almost irrelevant compared to inability to reach nodes.. . . ” Again quoting from the same paper, “failure is only represented, in a weak but realistic sense, as becoming forever unreachable”.

For example, a process of the form $(\nu n)n[P]$, where the name n does not occur in P, is observationally equivalent to the null process [GC99], since the name n, being restricted, is invisible in any context, and direct communication cannot cross an ambient’s boundary. Observe that, owing to scope extrusion, the ambient n may go around exiting and entering other ambients, but it cannot emit nor absorb any ambient, if P does not contain the name n. Thus it is just an unobservable process moving around.

7.3 Mobility

7.3.1 Migrating entities

Migration in MA clearly is by actual movement, though of course proxy creation may be encoded.

“Naked” processes cannot directly migrate through a go primitive, like in some other calculi; only whole ambients may move, driven by some enclosed process, and take with them all enclosed processes. For this, one has the first two *mobility constructs*: $\mathsf{in}\ n.P$ (to enter a sibling ambient named n), and $\mathsf{out}\ n.P$ (to exit a parent ambient named n).

This kind of mobility has been called *subjective* by Cardelli and Gordon [CG98], since the migration command originates from within the migrating ambient itself; in contrast, an *objective* move would be one by which an external process orders an ambient to move, e.g., a construct $\mathsf{move}\ m\ \mathsf{in}\ n$ whose reduction rule would be:

$$\mathsf{move}\, m\, \mathsf{in}\, n.R \mid m[P] \mid n[Q] \;\rightarrow\; R \mid n[m[P] \mid Q]$$

This kind of primitives cannot be encoded – in a context-free way – by MA primitives; they can however be emulated – in a context-dependent way – for specific ambients.

The calculus also offers a third mobility construct, $\mathsf{open}\ n.P$, to open an ambient named n, i.e., to dissolve its boundary thus directly exposing the contained processes; also in this case the ambient n must be a sibling of the process performing the action.

While the in and out operations are dual of each other, the $\mathsf{open}\, m$ instruction has a more definitive character and does not possess an inverse operation proper. It provides a form of objective migration, since it clearly has the effect of moving into the parent of m all the processes (either naked or consisting of ambients) contained in m.

In conclusion, the migrating entities of the Ambient Calculus are primarily the domains (i.e., the ambients); however, owing to the open primitive, (naked) processes too have a limited form of (objective) migration, from one (dissolved) ambient into its parent.

7.3.2 Expressing migration

As remarked in the previous point, the primitives in and out implement a form of subjective ambient migration, while the open implements a limited form of objective process migration.

The tree topology allows for a close control of movements: as we saw, the atomic actions only allow to cross one boundary at a time: this *proximity mobility*, as has been called [Cas01], is well-suited to model navigation through a hierarchy of *administrative domains*, which was the original inspiration for the calculus.

The in and out operations require the names respectively of the destination and the source ambient. On the contrary, the identity of the migrating ambient is implicit: it is the one containing the process where the migrating instruction occurs.

7.3.3 Migration during execution

Since an ambient is a named multithreaded process, and migrating instructions occurring in one thread (i.e., one sequential process) are not blocking for the other threads, we have *strong mobility*: a (heavy-weight) process may move while it is running.

On the other hand, since a migrating instruction in a thread blocks its continuation, a sequence of moving instructions may navigate an ambient-process through the forest of domains.

For example, the ambient-process $k[\,a!l\langle\tilde{v}\rangle m.\,a?l\langle\tilde{v}\rangle h.\,a?l\langle\tilde{v}\rangle n \parallel P\,]$ migrates from the source m to the destination n in three steps, while the other internal process P runs concurrently (of course, if P contains migrating instructions too, it may interfere with the previous sequence and drive k into another destination).

7.3.4 Migrating an associated state

The computational model of MA is stateless, i.e., the underlying programming paradigm is analogous to the one of functional programming, where however function call is completely replaced by communication between processes.

The calculus, being an abstract general formalism in the same category of λ-calculus or π-calculus, for this point of the classification cannot therefore be ascribed to a particular category, since this would depend on how the state itself is encoded.

For example, in the functional style the state of a computation is modelled by the instantiation values of its variables. In this sense, a process always trivially takes its own state with it (but see the following point).

On the other hand, if by state one means the set of siblings whose names are known to the migrating ambient, of course this is not moved.

Finally, observing that ambients are "closed" entities, one might consider the migrating ambient as consisting of threads that move along with their context or state.

7.3.5 Remote references

The ambient names known by a migrating entity are kept during migration, but they are always interpreted locally: so they will denote, after migration, different ambients – if any – from the ones denoted before migration. It may therefore be concluded that there is no notion of remote reference.

7.4 Security

The primitives for entering, exiting, or opening an ambient require the ambient's name, which is therefore a sort of password that must be known for being able to perform the action. Such names may be communicated from one process to another; communication arguments may also consist of mobility instructions or sequences of mobility instructions. Since there is no primitive allowing to extract the name m from the expressions $\mathsf{in}\,m$, $a!l\langle\tilde{v}\rangle m$, $\mathsf{open}\,m$, or to extract a subexpression M from an expression M', a process can give away, instead of an ambient's full name, which grants unrestricted power over it, only a particular capability (or sequence of capabilities) for that ambient. For example, if the redex $\langle a!l\langle\tilde{v}\rangle m\rangle \mid (x)P$ is reduced, the process P acquires the capability of driving its enclosing ambient out of m, but not the one of making it re-enter m.

On the other hand, the identification of the source for the $a?l\langle\tilde{v}\rangle n$ (i.e., the common enclosing ambient), and of the destination for the $a!l\langle\tilde{v}\rangle n$, which is the ambient enclosing n, are implicit.

In this way an ambient, if it has the capability of going out of its surrounding ambient n, is able to do it in whatever environment the ambient n ends up: which may be considered to be safe, since ambients are "closed" entities that may only be opened by the host ambient. Following this view, an ambient named Odysseus may come out of the horse into Troy but, if some Trojan does not open Odysseus' ambient to let out the real destroying process, the Greek hero may only take a harmless walk in the city. See however different choices for such primitives in the other ambient calculi described in this report.

Finally, the construct $\nu n.P$ for name scoping limits a name's visibility to a portion of the network, which – exactly like in the π-calculus – may change dynamically by effect of communication (the well-known phenomenon of scope extrusion).

8 Safe Mobile Ambients

[CG98] L. Cardelli and A. D. Gordon. Mobile ambients. In M. Nivat, editor, *Proceedings of Foundations of Software Science and Computation Structures (FoSSaCS)*, volume 1378 of *LNCS*, pages 140–155. Springer-Verlag, Berlin, 1998.

[GYY] X. Guan, Y. Yang, and J. You. Making ambients more robust. In *Proc. Int'l. Conf. on Software: Theory and Practice*, pages 377–384, Beijing, China, 2000.

[LS00] F. Levi and D. Sangiorgi. Controlling interference in ambients. In *Proceedings POPL 2000*, pages 352–364. ACM Press, 2000.

[Zim00] P. Zimmer. On the expressiveness of pure mobile ambients. In L. Aceto and B. Victor, editors, *Preliminary Proceedings of EXPRESS '00*, volume NS-00-2 of *BRICS Notes Series*, pages 81–104, 2000.

8.1 Overview

The calculus of *Safe Mobile Ambients* (SA) [LS00] is obtained from *Mobile Ambients* (MA) [CG98] by adding to the three mobility actions three corresponding *coactions*: $\overline{\mathsf{in}}\, m$, $\overline{\mathsf{out}}\, m$, $\overline{\mathsf{open}}\, m$, with the consequent modification of the three related reduction rules; for an action *op m* to take place (with $op = \mathsf{in}, \mathsf{out}, \mathsf{open}$), it is necessary that the ambient *m* gives its consent, through the simultaneous execution of the corresponding coaction, as follows:

$$\begin{array}{rcl} n[a?l\langle\tilde{v}\rangle m.P_1 \mid P_2] \mid m[\overline{\mathsf{in}}\, m.Q_1 \mid Q_2] & \rightarrow & m[n[P_1 \mid P_2] \mid Q_1 \mid Q_2] \\ m[n[a!l\langle\tilde{v}\rangle m.P_1 \mid P_2] \mid \overline{\mathsf{out}}\, m.Q_1 \mid Q_2] & \rightarrow & n[P_1 \mid P_2] \mid m[Q_1 \mid Q_2] \\ \mathsf{open}\, n.P \mid n[\overline{\mathsf{open}}\, n.Q_1 \mid Q_2] & \rightarrow & P \mid Q_1 \mid Q_2 \end{array}$$

Movement therefore always results from a handshaking between the migrating ambient and the exited or entered ambient. The open primitive, requiring now the agreement of the ambient to be opened, loses its character of purely objective move.

All the other constructs and rules are unchanged w.r.t. MA, with the exception, irrelevant here, that replication (the construct $!P$) is replaced by recursion (with the construct $\mathsf{rec}\, X.P$), and consequently the equivalence rule $!P \equiv P \mid !P$ is replaced by the obvious unfolding reduction rule.

In the following, we will only cover the points affected by the new definition, and we will limit ourselves to indicating the differences w.r.t. MA. It is intended that the omitted points are as in MA.

8.2 Distribution

8.2.2 Concept of domain

Semantics

Like in MA, the ambient is a mobility, communication, and security unit. A particular stress is in a broad sense on security, since the primary motivation for the introduction of coactions is that it enables to develop a better algebraic theory for a bisimulation-based behavioural equivalence (see also the point 8.4).

Observability/controllability of domains

An ambient's behaviour results both from the subjective control exerted by the enclosed processes, as in MA, and from the consent given by the ambient where the coaction is consumed. In particular, in dissolving an ambient the open and $\overline{\mathsf{open}}$ primitives play a completely symmetric role: the coaction might be viewed as a self-open action, whose coaction is the open.

Observe that MA-like behaviour for an ambient n may be obtained in the obvious way [LS00] by placing inside n the processes $!\overline{\mathsf{in}}\,n$, $!\overline{\mathsf{out}}\,n$, $!\overline{\mathsf{open}}\,n$ (where $!P$ is intended as a shorthand for $\mathsf{rec}\,X.(P\|X)$).

8.2.3 Local communication

Local communication is exactly the same as in MA.

The pure MA calculus without communication primitives had been proved in the original paper [CG98] to be Turing-complete, thus supporting the view that (actual or virtual) space distribution and mobility may play a fundamental role in a theory of computation appropriate for global computing.

Actually, being Turing Machines essentially sequential devices, the result was of mainly theoretical interest, since TM encodings of concurrent mobile systems will be necessarily awkward.

An encoding of SA communication primitives into SA without communication has been recently found [Zim00], similar to the encoding of objective moves for specific ambients in MA [CG98]. This again seems to stress the basic character of domain mobility w.r.t. communication, which may be viewed as a movement of information.

8.2.4 Remote communication

Since, like in MA, remote communication is obtained by migration and opening, the stronger constraints on mobility clearly affect remote communication.

8.3 Mobility

8.3.2 Expressing migration

As observed in the overview, the three mobility primitives in, out and $\overline{\mathsf{open}}$ are forms of subjective migration, which require an objective authorization by handshaking respectively with the primitives $\overline{\mathsf{in}}$, $\overline{\mathsf{out}}$ and open.

8.4 Security

The introduction of the coactions is explicitly motivated by the aim of eliminating the *grave interferences*, i.e., situations where “the activity of a process is damaged or corrupted because of the activities of other processes” [LS00].

Such situations may arise, in the original MA, when an expression intended to model a particular real-world process or system is put in a context that models a possible reality different from the one first envisaged. This is due to the fact that in MA mutually exclusive reductions are possible which represent semantically contrasting behaviours, as in the following examples taken from [LS00]:

$$\mathsf{open}\,n \mid n[a?l\langle\tilde{v}\rangle m.P] \mid m[Q] \quad\to\quad \begin{cases} a?l\langle\tilde{v}\rangle m.P \mid m[Q] \\ \mathsf{open}\,n \mid m[n[P] \mid Q] \end{cases}$$

$$h[n[a?l\langle\tilde{v}\rangle m.P \mid a!l\langle\tilde{v}\rangle h.R] \mid m[Q]\,] \quad\to\quad \begin{cases} h[m[n[P \mid a!l\langle\tilde{v}\rangle h.R] \mid Q]\,] \\ n[a?l\langle\tilde{v}\rangle m.P \mid R] \mid h[m[Q]\,] \end{cases}$$

In the first example, the ambient m may either be dissolved, or migrate into n (where it cannot be opened any more); in the second, n may exit its parent h or else enter its sibling m. In either example the two different reducts are certainly non-equivalent w.r.t. any reasonable notion of equivalence, and it’s hard to imagine a real-world situation modelled by a nondeterministic choice between two so contrasting behaviours.

However, because of the kind of nondeterminism inherent in MA, it is difficult to avoid that these situations may arise when running a “MA program”, or better when plugging a MA module into a (maybe unforeseen) context. The above examples are therefore more likely to represent unintended behaviours coming along with those intentionally specified: i.e., programming errors.

The result is the difficulty of designing MA systems provably correct (w.r.t. the specifications) in all contexts; which of course is also the difficulty of designing protocols and algorithms for security that be guaranteed free from loopholes.

With coactions, each ambient may decide whether, at a given instant, it may accept a sibling ambient coming in, or whether it may let out a contained ambient. Observe however, in the reduction rules, that the argument of the coaction is the same as the one in the action: the name of the authorizing, not of the authorized ambient. The handshaking for in and out is therefore not symmetric: an ambient cannot directly authorize one ambient of a given name to come in or to go out.

On the other hand, the above remarked symmetry of the two opening constructs changes the potentially dangerous character of the original open into a much more manageable operation.

As a matter of fact, in [LS00] the introduction of coactions mainly serves the purpose of allowing to define a sophisticated type system that statically controls the mobility and eliminates the possibility of grave interferences, thus enhancing the intrinsic safeness of the new design tool.

A variation of the calculus – called *RObust AMbients* (ROAM) – where the argument of the coaction must be the name of the authorized ambient is studied in [GYY]. Its reduction rules for mobility are:

$$\begin{array}{rcl} n[a?l\langle\tilde{v}\rangle m.P \mid Q] \mid m[\overline{\mathsf{in}}\, n.R \mid S] & \rightarrow & m[n[P \mid Q] \mid R \mid S] \\ m[n[a!l\langle\tilde{v}\rangle m.P \mid Q] \mid \overline{\mathsf{out}}\, n.R \mid S] & \rightarrow & n[P \mid Q] \mid m[R \mid S] \\ \mathsf{open}\, n.P \mid n[\overline{\mathsf{open}}\,.Q \mid R] & \rightarrow & P \mid Q \mid R \end{array}$$

An ambient may enter or exit another ambient only if the latter gives the former an individual nominal permit valid for one trip; for migration to be allowed, all concerned ambients must therefore know (or at least possess capabilities on) each other's name. Observe however that because of the homonymy phenomenon, inherent in all ambient calculi, names do not always univocally identify ambients (names are not fingerprints). The more restrictive discipline allows the definition of a flexible type system, where evolving behaviours are characterized at a finer grain by evolving types.

9 Boxed Ambients

[BCC01] M. Bugliesi, G. Castagna, and S. Crafa. Boxed ambients. In *TACS 2001, 4th. International Symposium on Theoretical Aspects of Computer Science*, volume 2215 of LNCS, pages 38–63, Sendai, Japan, 2001. Springer-Verlag, Berlin.

[Car99] L. Cardelli. Abstractions for mobile computations. In J. Vitek and C. Jensen (eds), *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, volume 1603 of LNCS, pages 51–94, 1999. Springer-Verlag, Berlin.

[CG98] L. Cardelli and A. D. Gordon. Mobile ambients. In M. Nivat, editor, *Proceedings of Foundations of Software Science and Computation Structures (FoSSaCS)*, volume 1378 of LNCS, pages 140–155. Springer-Verlag, Berlin, 1998.

[CGN01] G. Castagna, G. Ghelli, and F. Zappa Nardelli. Typing Mobility in the Seal Calculus. In *CONCUR 2001*, volume 2154 of LNCS, pages 82–101. Springer-Verlag, Berlin, 2001.

[GYY] X. Guan, Y. Yang, and J. You. Making ambients more robust. In *Proc. Int'l. Conf. on Software: Theory and Practice*, pages 377–384, Beijing, China, 2000.

[LS00] F. Levi and D. Sangiorgi. Controlling interference in ambients. In *Proceedings POPL 2000*, pages 352–364. ACM Press, 2000.

[MS02] M. Merro and V. Sassone. Typing and subtyping mobility in boxed ambients. In *Proceedings CONCUR 02*, LNCS, 2002.

9.1 Overview

Boxed Ambients (BA) [BCC01] are a variation of *Mobile Ambients* (MA) [CG98] that follows a radical approach for controlling the destructive character of the open primitive while ensuring the possibility of remote communication: it merely drops the open altogether, while providing new inter-domain primitives for communication across an ambient boundary, inspired by the *Seal* calculus [CGN01] (with only intra-ambient communication and without the open, in the absence of any form of "naked process" mobility, ambients would be completely isolated from each other, which is of course absurd)

In particular, besides local communication as in MA, there is the possibility of atomically forwarding an input request – or dropping an output – into the enclosing ambient or into a named enclosed ambient, through the constructs $(x)^{\eta}P$ and $a!\langle\tilde{v}\rangle M^{\eta}P$, where $\eta = M, \uparrow$: M is an ambient name or a variable standing for an ambient name, the symbol $\uparrow$ denotes communication with the parent.

An output dropped across a boundary can be collected by a local input in the other ambient; symmetrically, an input request across a boundary can collect a local output in the other ambient, as stated by the following reduction rules:

$$\begin{array}{lrcl}
(\textit{local commun.}) & (x)P \mid \langle M\rangle Q & \to & P\{x:=M\} \mid Q \\
(\textit{input from child}) & (x)^{n}P \mid n[\,\langle M\rangle Q \mid R\,] & \to & P\{x:=M\} \mid n[\,Q \mid R\,] \\
(\textit{input from parent}) & \langle M\rangle P \mid n[\,(x)^{\uparrow}Q \mid R\,] & \to & P \mid n[\,Q\{x:=M\} \mid R\,] \\
(\textit{output to child}) & \langle M\rangle^{n}P \mid n[\,(x)Q \mid R\,] & \to & P \mid n[\,Q\{x:=M\} \mid R\,] \\
(\textit{output to parent}) & (x)P \mid n[\,\langle M\rangle^{\uparrow}Q \mid R\,] & \to & P\{x:=M\} \mid n[\,Q \mid R\,]
\end{array}$$

Observe that each of two untagged communication primitives inherited from MA occurs three times in the above rules, i.e., both in the intra-domain and in the inter-domain communication. Such instructions are therefore not, as in MA, purely local input/output primitives, rather they are forms of undirected input or output, in contrast with the directed communication requests represented by the tagged versions.

This implies a certain supplementary amount of nondeterminism: an undirected input may synchronize either with a local output or with an output from (a process in) a child or parent ambient (i.e., across one boundary); the analogous holds for undirected output.

The six communication primitives are clearly redundant: for example, an input by an ambient from a parent's output may be performed either as an undirected input from a directed output, or as a directed input from an undirected output. Of course, the two different implementations are not equivalent, since the respective synchronizations take place in different ambients and are therefore subject to different possible "interferences".

As a matter of fact, in [BCC01] the natural alternative is also discussed in which communication primitives inherited from MA keep their purely local character, and communication across a boundary must be directed from both sides. The reduction rules for the tagged primitives then become:

$$\begin{array}{rcl} (x)^n P \mid n[\,a!\langle\tilde{v}\rangle M^{\uparrow} Q \mid R\,] & \rightarrow & P\{x := M\} \mid n[\,Q \mid R\,] \\ a!\langle\tilde{v}\rangle M^n P \mid n[\,(x)^{\uparrow} Q \mid R\,] & \rightarrow & P \mid n[\,Q\{x := M\} \mid R\,] \end{array}$$

This solution "provides ambients with full control of the exchanges they may have with their children" [BCC01], but in turn it makes difficult the modelling of other kinds of communication protocols easily implemented by the original BA primitives (such as the possibility for an ambient to broadcast a message to any entering agent, like mobile phone companies do whenever a cell-phone crosses a national border).

9.2 Distribution

9.2.2 Concept of domain

Semantics

The absence of the open operation implies that ambients, once activated, live forever (since there is no way of destroying them); in any realistic situation a garbage collection is therefore needed to dispose of ambients no longer involved in any potential computation.

9.2.3 Local communication

Communication may be local, or it may be across one boundary: from child to parent – which is implicit, and therefore has not to be named – or from parent to (named) children. Direct communication between sibling ambients is not possible: it must be mediated by communication with the parent.

Differently from Mobile Ambients and *Safe Ambients* (SA) [LS00], communication is synchronous, which is simply expressed in the syntax by the fact that the output primitive is now a prefix $a!\langle\tilde{v}\rangle MP$ exactly like input, also blocking its continuation. Asynchronous communication, which has convincingly been argued to be the main form of communication in mobile and distributed computing [Car99], may of course be considered a special case of synchronous communication: namely, one between two processes where synchronous output has a null continuation.

In [BCC01], however, various other possible versions of the calculus w.r.t. the issue of synchronous vs. asynchronous communication are discussed, especially in relation with typing. In particular, a calculus is considered where the reduction rules for directed output are replaced by asynchronous versions, and a new reduction rule for undirected output is introduced:

$$\begin{array}{lrcl} (\textit{asynch. undirected output}) & a!\langle\tilde{v}\rangle MP & \rightarrow & a!\langle\tilde{v}\rangle M \parallel P \\ (\textit{asynch. output to child}) & a!\langle\tilde{v}\rangle M^n P \mid n[\,Q\,] & \rightarrow & P \mid n[\,a!\langle\tilde{v}\rangle M \mid Q\,] \\ (\textit{asynch. output to parent}) & n[\,a!\langle\tilde{v}\rangle M^{\uparrow} P \mid Q\,] & \rightarrow & a!\langle\tilde{v}\rangle M \mid n[\,P \mid Q\,] \end{array}$$

If the rules for input are kept unchanged, their overlapping with the asynchronous version of undirected output gives rise to a form of nondeterminism in communication, which may be performed either in one step atomically, or in two steps. For example:

$$a!\langle\tilde{v}\rangle MP \mid n[\,(x)^{\uparrow}Q \mid R\,] \quad \to \quad \begin{cases} P \mid n[\,Q\{x := M\} \mid R\,] \\ a!\langle\tilde{v}\rangle M \mid P \mid n[\,(x)^{\uparrow}Q \mid R\,] \ \to \ \cdots \end{cases}$$

This phenomenon may be avoided by adopting an asynchronous version of the input rules too, consisting of an input instruction that only accepts asynchronous (i.e., non-prefix) output:

$$\begin{array}{rcl} (x)P \mid a!\langle\tilde{v}\rangle M & \to & P\{x := M\} \\ (x)^{n}P \mid n[a!\langle\tilde{v}\rangle M \mid Q] & \to & P\{x := M\} \mid n[Q] \\ a!\langle\tilde{v}\rangle M \mid n[\,(x)^{\uparrow}Q \mid R\,] & \to & n[\,Q\{x := M\} \mid R\,] \end{array}$$

Every communication is then carried out following a two-step protocol, always with the intermediation of the form $a!\langle\tilde{v}\rangle M$ which may be interpreted as an ambient's memory cell.

9.2.4 Remote communication

As shown at the previous points, in the synchronous version communication across a boundary is performed atomically. General remote communication can only be carried out, like in MA and SA, through a sequence of intermediate steps; unlike in those calculi, it may be realized by communication proper, crossing one boundary at a time, without the need of ambient movement (though, of course, communication obtained via mobility is still possible).

9.3 Mobility

9.3.1 Migrating entities

The absence of the open primitive implies that ambients really are the only migrating entities: there is no objective nor subjective process mobility.

In the asynchronous version, however, a very particular and limited kind of (objective) process mobility is the one of the output expressions of the form $a!\langle\tilde{v}\rangle M$. That shows once more how, at a sufficiently abstract level, the distinction between communication and migration tends to blur.

9.3.2 Expressing migration

Subjective ambient migration is exactly the same as in MA.

In [MS02] a variation called *Safe Boxed Ambients* (SBA) is presented, in which coactions similar (but not identical) to the ones in [GYY] are introduced: $\overline{\mathsf{in}}\ n$ and $\overline{\mathsf{out}}\ n$, where n is the name of the ambient authorized to cross the boundary. Besides, a behaviour similar to the one of the original SA primitives is recovered by the forms $\overline{\mathsf{in}}\ \star$ and $\overline{\mathsf{out}}\ \star$, which are general authorizations consumable by any ambient. The corresponding reduction rules are:

$$\begin{array}{rcll} n[a?l\langle\tilde{v}\rangle m.P \mid Q] \mid m[\overline{\mathsf{in}}\,\alpha.R \mid S] & \to & m[n[P \mid Q] \mid R \mid S] & \text{for } \alpha \in \{\star, n\} \\ m[n[a!l\langle\tilde{v}\rangle m.P \mid Q] \mid R] \mid \overline{\mathsf{out}}\,\alpha.S & \to & n[P \mid Q] \mid m[R] \mid S & \text{for } \alpha \in \{\star, n\} \end{array}$$

See the next point (9.4) for a simple comparison with ROAM.

9.4 Security

The communication primitives "have immediate and very natural interpretations as access requests: for example, the input prefix $(x)^n$ can be seen as a request to read from (the anonymous channel located into) child ambient n and, dually, $a!\langle\tilde{v}\rangle M^{\uparrow}$ can be interpreted as a write request to the parent ambient" [BCC01].

Classical resource access control mechanisms may thus be directly implemented, while security policies and models may be defined by means of type systems. Like in the case of SA, the main reason for the variation w.r.t. MA is that it allows the definition of more satisfactory type systems for reasoning about ambient and process behaviours.

In the synchronous version we saw that communication between sibling ambients is directly controlled by the parent; in the asynchronous version, on the other hand, this control may be circumvented, since communication may be established using a parent's "shared memory cell" $a!\langle\tilde{v}\rangle M$, without the need of exchanging explicit messages with it:

$$a[(x)^{\uparrow}P] \mid b[a!\langle\tilde{v}\rangle M^{\uparrow}Q] \quad \rightarrow \quad a[(x)^{\uparrow}P] \mid a!\langle\tilde{v}\rangle M \mid b[Q] \quad \rightarrow \quad a[P\{x := M\}] \mid b[Q]$$

For as concerns the SBA calculus [MS02], observe that while the reduction rule for the input is exactly the same as in ROAM [GYY], the exit permit $\overline{\mathsf{out}}\, n$ must be given to n by the (enclosing) ambient where n exits into, not by the one which n leaves. This is rather reasonable, since security hazards usually arise from the fact that a malicious agent may come out of a seemingly legitimate hosted ambient into the hosting surrounding system: such as a virus hidden in a downloaded application or mail message, or, prototypically, such as Odysseus coming out of the seemingly harmless horse into Troy (but compare with a different view, illustrated in relation with MA).

10 The Seal-calculus

10.1 References

[VC98] J. Vitek, G. Castagna, Towards a Calculus of Secure Mobile Computations. In Workshop on Internet Programming Languages, 1998.

[VC99a] J. Vitek, G. Castagna, Mobile Computations and Hostile Hosts. In Proceedings of the 10th JFLA (Journées Francophones des Langages Applicatifs), Avoriaz, France, January 1999.

[VC99b] J. Vitek, G. Castagna, Seal: A Framework for Secure Mobile Computations. In Workshop on Internet Programming Languages, 1999.

10.2 Overview

The SEAL-calculus is a calculus for "large scale distribution", especially conceived to incorporate the essential properties of Internet programs. In this calculus, which is strongly inspired from the π-calculus, the primary focus is on code mobility and resource access control. The main design principles of the calculus are the following: absence of global state, explicit localities, restricted connectivity, dynamic reconfiguration, and access control.

10.3 Distribution

10.3.1 Basic entities

The main entities of the SEAL-calculus are processes, resources and domains. Domains cover both concepts of physical boundary and of logical boundary. Processes capture the flow of control (thread or system process), and consist of a sequence of communication and migration capacities. Resources are supposed to represent both physical resources (memory, peripheral devices) and services (at application level or system level). However the only resources that the calculus explicitly manipulates are communication channels, which are used, as in the π-calculus, for inter-process communication. Channels are named, and explicitly localised by means of a tag attached to their name (this localisation is relative, as explained below).

Making domains apparent

In the SEAL-calculus, a domain is a "seal", which is a named locality containing a number of processes as well as other seals. A seal has the form $n[P]$, where P is a process composed from other processes and seals with the operators of the π-calculus. As in the AMBIENT-calculus, there may be several domains with the same name.

Names and references

Named entities are channels and seals. There may be several receivers on the same channel name. The scope of names is defined, as in the π-calculus, by the restriction operator.

10.3.2 Concept of domain

Topology

Seals are hierarchically structured. A seal may be therefore identified with the tree of its sub-seals. The topology of domains may evolve as a consequence of the mobility of seals.

Semantics

A seal possesses resources (its channels), and controls the access to these resources. In this sense, a seal is a security domain. A seal is also a migration unit, since the calculus supports atomic migration of seals.

10.3.3 Local communication

Local naming

The naming of a channel by a process belonging to the same seal (the channel is called "local" in this case) is made through the name of the channel, decorated by the special symbol $*$ to indicate the local character of the naming. Local naming is therefore explicit in the calculus.

Local communication policy

Local communication, i.e. communication between two processes within the same seal, is synchronous and is only possible via local channels (tagged with the $*$ symbol). For instance:

$$\overline{x}^{\star}(\overrightarrow{z}).P \mid x^{\star}(\overrightarrow{y}).Q \quad \rightarrow \quad P \mid Q\{\overrightarrow{z}/\overrightarrow{y}\}$$

Semantics of reception

Reception is multiple, as in the π-calculus or Nomadic Pict.

10.3.4 Remote communication

Remote naming

The referencing of a distant channel is possible only if this channel is located in an immediately related seal (parent or child). The mode of referencing is completely determined by the relation between the seals of the naming entity and the named entity, that is, relative localisation is made explicit. More precisely, the naming of a channel belonging to some seal by a process belonging to a child seal is made through the use of the channel name decorated by the special symbol $\uparrow$. The naming of a channel belonging to a seal *n* by a process belonging to the parent seal of *n*, is made through the use of the channel name decorated by *n*.

Distant communication

Two processes localised in immediately related seals (that is, seals in the parent-child relationship) may communicate by using a channel located in either of the two seals. This upward or downward communication, which is synchronous like local communication, is the only form of distant communication allowed by the calculus. For instance:

$$n[\overline{x}^{\uparrow}(\overrightarrow{z}).P] \mid x^{\star}(\overrightarrow{y}).Q \quad \rightarrow \quad n[P] \mid Q\{\overrightarrow{z}/\overrightarrow{y}\}$$

In fact, due to its strong security orientation, the calculus requires a further authorisation called a "portal" for distant communication to take place. Roughly speaking, a portal is a linear (usable only once) access permission to a specific channel, which the owner of the channel may give upon request to one of its parent or children seals.

Logical routing

Since only parent-child communication is allowed by the calculus, the routing of messages in the structure of seals must be explicitly programmed.

10.4 Mobility

10.4.1 Migrating entities

The migrating entities are domains, that is seals. As in other calculi, there is another degenerate form of migration which consists in sending a message to another domain.

10.4.2 Expressing migration

In the SEAL-calculus, migration is expressed as a form of communication: sending a seal name on a channel has the effect of moving the seal to the locality where the communication takes place (if this is different from the current locality), that is either to the father's locality or to one of the children localities. Like channel name communication, also seal name communication is explicitly controlled by a "portal" that opens access to the channel. It must be noted that migration in the SEAL-calculus is not limited to a simple movement: it is accompanied by the creation of several copies (possibly none, in which case the seal is destroyed) of the migrating seal, under different names. The migration instruction is objective – it comes from outside the migrating seal, more precisely from the father seal – and asynchronous, since the migrating seal executes independently from the migration order.

10.4.3 Migration during execution

Accordingly to what was just said, migration takes place "in the course of execution".

10.4.4 Migrating an associated state

In the SEAL-calculus there is no notion of state. The "context" of a migrating entity is always fixed, although names may refer to different entities after migration.

10.4.5 Remote references

A migrating seal maintains the knowledge of the channel names for which it possesses a reference, but these names cannot be considered as distant references. In fact, these names may possibly designate entities in the target context of the migrating seal, rather than those they originally referred to. Moreover, the name of a migrating seal may change as an effect of migration.

10.5 Security

One of the objectives of the SEAL-calculus is to allow the user to prove security properties of mobile processes, since these properties are ensured by small protocols that can be verified independently from their context[11]. The calculus views security both as the protection of a host's resources against malicious visitors, and as the protection of visiting agents against their host, the latter aspect being so far rather unexplored.

The SEAL-calculus aims at preventing the following attacks: leaking or unauthorised modification of informations, denial of service, and introduction of Trojan horses into a system.

- The first type of attack is handled through the use of lexical scoping and of linear capacities to protect resources.
- The denial of service attacks are being currently studied. To account for them, the calculus has to be extended with a more general notion of resource including memory and *cpu*-time.

[11] A proof theory in which the notions of observation, test and specification are defined rigourously, still needs to be established.

- Some protection against Trojan horses is offered: seals cannot hide inside another seal that is authorised to enter a system, and then escape this seal once the access has been performed in order to attack the system (”hitch-hiking” is not allowed). Inserting a process in parallel with those contained in a seal is also forbidden, as it could interfere with the way reception is performed.

On the other hand:

- Replay attacks are possible since the calculus allows the duplication of seals[12].
- In spite of the protection offered by the calculus, a hosting structure can falsify the informations it gives to visiting seals, trace their itinerary, trap the seals or steal their identity, and eavesdrop on communications between different hosted seals.

Identification and authentification

Seals are identified by their names. As in the SPI-calculus, it is possible to write authentification protocols in SEAL.

Access control

Access to resources

The calculus does not implement a particular security policy [13] but provides protection mechanisms that allow for a flexible use of different security policies.

Authorisation model

The protection mechanisms concern the following aspects:

- **Control of the use of names:** this is ensured by lexical scoping, which protects channel names (as concerns confidentiality and integrity).
- **Control of the use of resources:** linear capacities [14] called “portals” ensure a fine-grained control over the use of resources. Portals protect a seal against its children as well as against its parent.
- **Control of migration and communication:** the protection model of the calculus is hierarchical: a seal cannot migrate without the explicit permission of its parent seal. Moreover a seal can only step up or down one level at a time in the hierarchy of seals. Finally, portals provide a furher degree of protection as the target seal may decide not to allow the migration. Similarly, for communication, a first degree of protection is given by its restriction to pairsof parent-child seals, and portals add a finer degree of control as they require the owner of the communication channel to agree on its use.

Security domain

The boundary of a seal may represent a protection domain, e.g. a firewall or a sandbox. Unlike the AMBIENT-CALCULUS, the SEAL-CALCULUS does not allow these boundaries to be dissolved. A process executing inside a seal is always protected by the boundary of the enclosing seal: it cannot escape into the environment with malicious intentions; conversely, the environment cannot have access to the resources contained in a seal without its explicit permission.

[12]Although weakenng the security of the calculus, duplication has been allowed in order for applications to be able to use replication, e.g. for fault-tolerance purposes.

[13]An extension of the calculus in this sense is however under development.

[14]These capacities are linear in that they allow a given seal to use a channel for reading or writing just once.

The calculus satisfies the so-called ”perfect-firewall” property, i.e. $(\nu x)x[P] \simeq \mathbf{0}$, because if a seal’s name is not known from any other seal there cannot be any portal open for that name, nor any communication on a channel contained in that seal.

Protection of communication content

Some degree of protection for the content of communication is already offered by the language, through the restriction operator and the rule of lexical scoping, as in the SPI-calculus. The confidentiality of a message may be broken if the environment is capable of guessing the free names inside a seal.

An extension of the calculus with cryptographic primitives as in the SPI-calculus is envisaged. A cryptographic key may be represented as follows: if z is the text of the message and x the key, then the encrypted message $K_x(z)$ may be represented by a seal $y[!\overline{x}^{\uparrow}(z)]$.

11 Mobile Resources

11.1 References

[GHS02a] J. C. Godskesen, T. Hildebrandt and V. Sassone. A Calculus of Mobile Resources. In Proceedings CONCUR 02, to appear, 2002.

[GHS02b] J. C. Godskesen, T. Hildebrandt and V. Sassone. A Calculus of Mobile Resources. TR-2002-16, IT University of Copenhagen, ISBN 87-7949-021-2, august 2002.

11.2 Overview

The calculus of *Mobile Resources* (MR) is inspired by the Mobile Ambient calculus, and bears relationships to Boxed Ambients and the Seal calculus with respect to inter-domain communication. However, it differs from all of these calculi in important ways. Domains are made apparent by introducing the notion of (named) *slots* on top of a simple CCS-like calculus with replication. If p is a process, $n\lfloor p \rfloor$ is a slot named n which contains p as a resource; the notation $n\lfloor \bullet \rfloor$ represents an *empty* slot with name n. Empty slots represent available space and may be occupied by migrating resources. In general, slots may be named by a *set* $\tilde{n}$ of names in order to allow aliasing, and may be decorated by a distinguished name, as for instance m in $\tilde{n}\lfloor p \rfloor_m$, that can be used to delete the slot.

Mobility of resources in MR is objective and controlled by a move capability exercised by processes. In contrast to seals in the Seal calculus, slots are a priori static entities, i.e. a slot remains as an empty slot when its resource is moved, and a resource may only be moved if there exists an empty slot it can be moved to. This is formalised by the reduction rule below.

$$n\lfloor p \rfloor \parallel n \triangleright \overline{m}.q \parallel m\lfloor \bullet \rfloor \searrow n\lfloor \bullet \rfloor \parallel q \parallel m\lfloor p \rfloor$$

Communication is synchronous as in CCS. In addition to local communication, communication is possible across domains, from a process to one of the resources around it. This is realised through *directed* actions of the form δn, where n is a normal CCS-like action and δ is the direction path. In particular, we have the following reduction.

$$mn.q \parallel m\lfloor \overline{n}.p \rfloor \searrow q \parallel m\lfloor p \rfloor$$

Notice that a resource process cannot distinguish between communication with its host (that is a process in an ancestor slot) or a local parallel process.

In general, direction paths δ can be a sequence of names, allowing direct communication with arbitrary deeply nested sub resources, e.g. there is a reduction.

$$m_1 m_2 n.q \parallel m_1\lfloor m_2\lfloor \overline{n}.p \rfloor \parallel p' \rfloor \searrow q \parallel m_1\lfloor m_2\lfloor p \rfloor \parallel p' \rfloor$$

Likewise, resources may be moved from and to arbitrarily deep sub slots by move actions of the form $\delta_1 \triangleright \overline{\delta_2}$.

Finally, slots may be disposed of, that is destroyed together with their entire content. As for mobility, disposing of a slot is controlled objectively by the host, using the dispose prefix: $\natural m$. For instance, we have a reduction

$$n\lfloor p \rfloor_m \parallel \natural m.q \searrow q$$

The complete syntax of MR is as follows

$$\begin{array}{lll} p,q & ::= 0 \mid \lambda.p \mid p \parallel q \mid !p \mid (n)p \mid \tilde{n}\lfloor r \rfloor_m & (P) \\ r & ::= \bullet \mid p & \end{array}$$

where, fixed a universe of names N, we use $n, m, \ldots$ for names, $\tilde{n}, \tilde{m}, \ldots$ for finite sets of names (confusing with abuse of notation singleton sets with their elements). We use λ for actions as given below, for δ in N^+, the set of finite sequences of names.

$$\lambda \quad ::= \quad n \mid \overline{n} \mid \delta n \mid \delta\overline{n} \mid \delta \triangleright \overline{\delta'} \mid \natural n.$$

The reduction rules are defined using an N^*-indexed family of *direction path contexts* defined as follows.

$$\mathscr{C}_\varepsilon ::= (-) \qquad \mathscr{C}_{n\gamma} ::= \tilde{n}\lfloor \mathscr{C}_\gamma \parallel p \rfloor_m \ , \qquad n \in \tilde{n}$$

Observe that for a context $\mathscr{C}_\gamma$, the direction path γ indicates a path under which the context's 'hole' is found. For the special case where the hole is the only content of a slot, we define

$$\mathscr{D}_{\gamma n} ::= \mathscr{C}_\gamma(\tilde{n}\lfloor (-) \rfloor_m) \ , \qquad n \in \tilde{n}$$

The reduction rules that describe the semantics of MR are then the following.

$\gamma\alpha.p \parallel \mathscr{C}_\gamma(\overline{\alpha}.q) \searrow p \parallel \mathscr{C}_\gamma(q)$
$\gamma\delta_1 \triangleright \overline{\gamma\delta_2}.p \parallel \mathscr{C}_\gamma\big(\mathscr{D}_{\delta_1}(q) \parallel \mathscr{D}_{\delta_2}(\bullet)\big) \searrow p \parallel \mathscr{C}_\gamma\big(\mathscr{D}_{\delta_1}(\bullet) \parallel \mathscr{D}_{\delta_2}(q)\big)$
$\natural m.p \parallel \tilde{n}\lfloor r \rfloor_m \searrow p$

Table 2: Reduction rules

The first rule expresses communication across an arbitrary number (include zero) of slot boundaries. The second expresses mobility from an arbitrary (sub)slot to an empty but otherwise arbitrary (sub)slot. The last rule formalises deletion of a slot and its contents.

11.3 Distribution

11.3.1 Concept of domain

The named slots constitute the domains of the MR calculus. As opposed to ambients, they can contain *at most* one process. Formally, an empty slot named by a set of names $\tilde{n}$ is written $\tilde{n}\lfloor \bullet \rfloor$ and a slot named by a set of names $\tilde{n}$ containing a process p is written $\tilde{n}\lfloor p \rfloor$. As in the ambient calculus, different occurrences of the the construct $\tilde{n}\lfloor p \rfloor$ and $\tilde{m}\lfloor q \rfloor$ represent different slots, even if $\tilde{n}$ and $\tilde{m}$ are equal (or not disjoint).

A process in a named slot is a migration unit, i.e. it moves as a block. The migration units are called resources. Slots can be prefixed, allowing new domains (empty or containing resources) to be constructed dynamically, as e.g. in $!\overline{x}.n\lfloor \bullet \rfloor$.

11.3.2 Topology

The topology of the domain space is a forest of trees as in the Mobile Ambients and in the Seal calculus.

11.3.3 Semantics

Mobility in MR is *objective*. A resource is moved by a host process, and moves atomically from a slot to an empty slot of the mover's. The resource to be moved is addressed by a *location path* δ, that is a sequence of names $n_1 n_2 \ldots n_k$ where n_i is a name of the ith slot on the path from the mover to the resource. The fact that slots are named by sets of names allows for aliasing.

11.3.4 Observability and controllability of domains

A domain (slot) is visible and observable by any ancestor (host), except if the name(s) of the slot is explicitly made local by the restriction operator. Aliasing allows a slot to have both public and private (i.e. restricted) access names: having a distinguished name for disposing makes it possible to control by restriction if the slot (and its resources) can be disposed by the environment or not.

The default remote access to domains from ancestors, allows a very fine grained control of the accessibility of domains and communication distance.

11.3.5 Local and Remote Communication

Direct, synchronous communication between a host and one of its sub resources is allowed. For a resource communication with a sibling process, that is a process in its same slot, is totally indistinguishable from hierarchical communication with (a process residing in) an ancestor slot.

Extensions of MR planned for future research include name passing and more general communication primitives, e.g. by allowing upwards addressing in direction paths.

11.4 Mobility

As already remarked, mobility in MR is objective in that it is triggered explicitly by a process wishing to move a resource from one to another of its slots. Migration is enabled provided the destination slot is empty and the source one is not, and regardless of the activity of the latter. It is realised as a synchronous capability exercised by processes. The migrating entity is therefore the resource, that is the contents of a slot, that may in turn contain further slots and resources. Moving resources carry along their active channels and continue their activity at destination, oblivious of the migration.

11.5 Security

Although not specifically designed for security, the calculus embeds a number of features specifically concerned with safety. Above all, MR is designed around the notion that space may or may not be available, and that space constraints cannot possibly be ignored.

Also, resources cannot be copied or forged, as it is the case in MR's intended application domain, where embedded devices such as smartcard are not forgeable, cannot be replicated, and contain secrets that cannot be tampered with.

Finally, aliasing for slot names allows a form of control on slot deletion. Although very prominent in the current MR, in a future version of the calculus with name passing this is likely to have a lesser relevance, as the same effect can be obtained by selectively communicating names and deletion capabilities.

12 The M-calculus

12.1 References

[SS02] A. Schmitt and J.B. Stefani The M-calculus: A Higher-Order Distributed Process Calculus. RR-4361, INRIA, 2002.

12.2 Overview

The M-calculus can be understood as a higher-order extension of the Distributed Join calculus with programmable localities. From the Join calculus, the M-calculus retains the idea of asynchronous communication, definitions with join patterns of messages for synchronization and of hierarchically organized localities. Inpired by Boudol's Blue calculus, the M-calculus has lambda abstraction and functional application constructs. Two key constructs of the M-calculus are the *programmable locality* and the *passivation* construct. The passivation construct allows a locality to freeze its execution and to transform itself into thunks. This basic construct, coupled with higher-order communication, allows for the migration of processes or localities.

The syntax of the M-calculus is given below. A name u can be either a locality name a, a variable x, a resource name r, or an addressed resource name $a.r$. Name patterns μ can take the form of simple names (locality or resource) or of a special wildcard $-$ symbol that matches nay name.

$$\begin{array}{lcl} P & ::= & P \mid nil \mid V \mid \mathbf{a}(P)[P] \mid (P \mid P) \mid PP \mid (P,\ldots,P) \\ & & \mid \nu n.P \mid ([\mu = V]P,P) \mid \langle J \triangleright P \rangle \mid \texttt{pass}\, V \\ V & ::= & () \mid u \mid (V,\ldots,V) \mid \lambda x.P \\ J & ::= & \mathbf{r}\widetilde{x} \mid J \mid J \end{array}$$

12.3 Distribution

12.3.1 Basic entities

Basic entities for distribution are *localities*. In the M-calculus, a locality takes the form $a(P)[Q]$, where a is the name of the locality, P is a process that acts as a filter or controller for the locality, and Q is a process that corresponds to the content of the locality. P and Q may recursively consist of a parallel composition of other processes or localities.

Making domains apparent

The notion of domain is directly manifested in the M-calculus by localities. The behaviour of a locality is determined by its controller process P, which can implement e.g. the fault behaviour of the locality, its access control or message filtering policies.

Names and references

The M-calculus distinguishes between two sorts of names: resource names and locality names. These two kinds of names can be combined in an *addressed resource name* of the form $a.r$ where a is a locality name and r is a resource name, to route messages between localities. To ensure the determinacy of message routing in the M-calculus, a type system is used to ensure the unicity of locality names in a distributed computation. Several resources of the same name r may coexist in the given computation. They may e.g. be residing at different localities, thus providing support for dynamic binding.

12.3.2 Concept of domain

Topology

M-calculus localities are hierarchically organized in a forest (or a tree, if one considers the existence of a virtual root locality that manifests the ether between top-level localities).

Semantics

A locality is a unit of spatial distribution. Two processes placed in different localities, whether in a locality controller or in a locality content, are deemed to be residing at two different places, with possibly different behaviour (e.g. with respect to fault and security). Messages are routed between localities, based on locality names.

Observability/controllability of domains

The overall behavior of a locality in the M-calculus is determined by its controller process since this process intercepts all communications with the locality, whether incoming (from the external environment to the locality) or outgoing (from the locality to its external environment). A controller can passivate its locality into thunks, thus enabling process (and locality) migration. When the passivation operator is evaluated in the controller of a locality $a(\texttt{pass}\,V \mid P)[Q]$, the locality is split into three parts: its name a, its frozen controller $\lambda.P$, and its frozen content $\lambda.Q$. These parts are given as arguments to the function V:

$$a(\texttt{pass}\,V \mid P)[Q] \to V\;a\;(\lambda.P)\;(\lambda.Q)$$

For instance, a function $V = \lambda xpq.x(p())[q()]$ simply recreates the passivated locality.

Key observable actions in the M-calculus are a message entering a locality controller, a message leaving a locality controller, a message entering a locality content, and a message leaving a locality content. The routing semantics of the calculus (see below) guarantees that all messages entering a locality content have been filtered by the locality controller, and that all messages leaving a locality content will be filtered by the locality controller.

12.3.3 Local communication

Local communication in the M-calculus is communication which does not cross a locality boundary, i.e. between non locality processes inside a locality controller or a locality content. The local communication rule is very similar to the JOIN rule of the Join calculus, substituting message arguments for formal arguments in the guarded process (with $J = r_1\widetilde{x_1} \mid \ldots \mid r_n\widetilde{x_n}$):

$$\langle J \triangleright P\rangle \mid r_1\widetilde{V_1} \mid \ldots \mid r_n\widetilde{V_n} \to \langle J \triangleright P\rangle \mid P\{\{\widetilde{x_i}/\widetilde{V_i}\}\}$$

Messages targetting local resources can move freely between controller and content as shown on the following rule, where r belongs to the resources defined in process Q (a symmetric rule allows a local message to pass from content to controller):

$$a(P \mid r\widetilde{V})[Q] \to a(P)[Q \mid r\widetilde{V}]$$

12.3.4 Remote communication

Remote communication in the M-calculus is communication which crosses a locality boundary. Only messages bearing addressed resource names can cross a locality boundary. When corssing a locality boundary, messages are systematically intercepted by the locality controller on predefined resources **i** (for incoming messages) and **o** (for outgoing messages). The routing rule below illustrates the mechanism in the case of incoming messages, i.e. messages entering a locality (b is either a or belongs to active localities within locality a):

$$b.r\widetilde{V} \mid a(P)[Q] \rightarrow a(P \mid \mathbf{i}(b, r, \widetilde{V}))[Q]$$

A message which has successfully entered a locality is turned into a local message, as illustrated by the following rule:

$$a(P \mid a.r\widetilde{V})[Q] \rightarrow a(P \mid r\widetilde{V})[Q]$$

12.4 Mobility

12.4.1 Migrating entities

Migration in the M-calculus is realized by means of higher-order communication, i.e. asynchronous messages bearing process thunks. The unit of migration is thus a process thunk (a lambda abstraction). For instance, locality $Q^m(a)$ below can be moved to a different locality:

$$\begin{aligned} Q^m(a) &= a(Fwd \mid \langle \mathsf{go}\, u \triangleright Go(u) \rangle)[Q] \\ Go(u) &= \mathtt{pass}\, \lambda x\, p\, q.(u.\mathsf{enter}\, \lambda.x(p())[q()]) \end{aligned}$$

A request $\mathsf{go}\, b$, results in the passivation of the locality a and its sending as a thunk to the resource enter of the locality named b. If the request comes from the outside of the locality, the result is an objective form of move. If the request comes from the content of the locality, the result is a subjective form of move. The controller of locality b can contain the process $Enter$ below to allow the insertion of a new locality in its content:

$$Enter = \langle \mathsf{enter}\, f \triangleright \mathtt{pass}\, \lambda x\, p\, q.x(p())[q() \mid f()] \rangle$$

12.4.2 Expressing migration

There is no specific instruction or protocol for process migration in the M-calculus. Higher-order communication does allow to model different forms of migration, however. For instance, locality $Q^m(a)$ below can be moved to a different locality:

$$\begin{aligned} Q^m(a) &= a(Fwd \mid \langle \mathsf{go}\, u \triangleright Go(u) \rangle)[Q] \\ Go(u) &= \mathtt{pass}\, \lambda x\, p\, q.(u.\mathsf{enter}\, \lambda.x(p())[q()]) \end{aligned}$$

A request $\mathsf{go}\, b$, results in the passivation of the locality a and its sending as a thunk to the resource enter of the locality named b. If the request comes from the outside of the locality, the result is an objective form of move. If the request comes from the content of the locality, the result is a subjective form of move. The controller of locality b can contain the process $Enter$ below to allow the insertion of a new locality in its content:

$$Enter = \langle \mathsf{enter}\, f \triangleright \mathtt{pass}\, \lambda x\, p\, q.x(p())[q() \mid f()] \rangle$$

12.4.3 Migration during execution

Migration in the M-calculus implies some form of passivation (i.e. turning a process into a thunk). This is made possible by the passivation construct of the calculus. Passivation can take place at any time during the execution of a locality. Only located processes, i.e. processes executing inside a locality, can be passivated.

12.4.4 Migrating an associated state

Passivated process state in the M-calculus is represented by a thunk, and is the unit of migration. Note that, because of dynamic binding, the context of execution of a resumed process (its set of local resources) may change during a migration.

12.4.5 Remote references

A process can maintain a knowledge of resource definitions in the form of resource names or of addressed resource names. Resource names refer only to resource definitions located in the current locality. Addressed resource names of the form $a.r$ reference resource definitions located in the locality named a.

12.5 Security

Security functions, in the form eg of cryptographic protocols or access control mechanisms can be explicitly programmed in the M-calculus in the form of locality controllers. There are, however, no built-in security primitives. The calculus does maintain a routing invariant that is crucial for security purposes (all messages entering a locality content are guaranteed to have been filtered by the locality controller, and all messages leaving a locality content are guaranteed to be filtered by the locality controller).

13 KLAIM

[BLP02] L. Bettini, M. Loreti, R. Pugliese. An Infrastructure Language for Open Nets. *Proc. of the 2000 ACM Symposium on Applied Computing (SAC'02), Special Track on Coordination Models, Languages and Applications*, pages 373-377, ACM Press, 2002.

[BNP02] L. Bettini, R. De Nicola, R. Pugliese. Klava: a Java Package for Mobile Code. Submitted for pubblication, 2002.

[NFP98] R. De Nicola, G. Ferrari, R. Pugliese. KLAIM: A Kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering*, 24(5):315-330, IEEE Computer Society, 1998.

[NFPV00] R. De Nicola, G. Ferrari, R. Pugliese, B. Venneri. Types for Access Control. *Theoretical Computer Science*, 240:215-254, Elsevier Science, 2000.

13.1 Overview

KLAIM (*kernel language for agents interaction and mobility*) [NFP98, NFPV00, BNP02] is an experimental formalism specifically designed for programming network applications. KLAIM combines a sort of asynchronous higher-order process calculus with the Linda coordination paradigm, and with facilities for process distribution, mobility and security.

The KLAIM communication mechanism, inspired by the Linda one, is based on the concept of *tuple space*. A tuple space is a multiset of *tuples*; these are containers of information items called *fields*. There can be *actual fields* (i.e. expressions, processes, localities, constants, identifiers) and *formal fields* (i.e. variables). Syntactically, a formal field is denoted by $!ide$, where *ide* is an identifier.

Pattern-matching is used to select tuples in a tuple space: two tuples match if they have the same number of fields and corresponding fields match: a formal field matches any value of the same type, and two actual fields match only if they are identical (but two formals never match). For instance, tuple $(\text{“foo”}, \text{“bar”}, 100+200)$ matches with $(\text{“foo”}, \text{“bar”}, !Val)$. After matching, the variable of a formal field gets the value of the matched field: in the previous example, after matching, *Val* (an integer variable) will contain the integer value 300.

In Linda there is only one global shared tuple space; KLAIM extends Linda by handling multiple distributed tuple spaces. Tuple spaces are placed at *nodes* that are part of a *net*. Each node contains a single tuple space and processes in execution; a node can be accessed through its address. There are two kinds of addresses: *sites* are the network references through which nodes can be uniquely identified within a net; *localities* are symbolic names for nodes. A reserved locality, `self`, can be used by processes to refer to their execution node. Sites have an absolute meaning and can be thought of as IP addresses, while localities have a relative meaning depending on the node where they are interpreted; they can be thought of as aliases for network references. Localities are associated to sites through *allocation environments*, represented as partial functions. Each node has its own environment that, in particular, associates `self` to the site of the node.

KLAIM *processes* can be built by using standard operators borrowed from process algebras, such as *action prefixing*, *parallel composition* and *process definition*. Processes may run concurrently, both at the same node or at different nodes, and can perform five *basic operations* (also called *actions*) over nodes. $\mathbf{in}(t)@\ell$ evaluates the tuple t and looks for a matching tuple t' in the tuple space located at ℓ. Whenever the matching tuple t' is found, it is removed from the tuple space. The corresponding values of t' are then assigned to the formal fields of t and the operation terminates. If no matching tuple is found, the operation is suspended until one is available. $\mathbf{read}(t)@\ell$ differs from $\mathbf{in}(t)@\ell$ only because the tuple t', selected by pattern-matching, is not removed from the tuple space located at ℓ. $\mathbf{out}(t)@\ell$ adds the tuple resulting from the evaluation of t to the tuple space located at ℓ. $\mathbf{eval}(P)@\ell$ spawns process P for execution at node ℓ. $\mathbf{newloc}(u)$ creates a new node in the net and binds its site to u. The node can be considered a “private” node that can be accessed by the other nodes only if the creator communicates the value of variable u, which is the only means to access the fresh node.

During tuple evaluation, expressions are computed and localities are translated into sites. Evaluating a process implies substituting it with its *closure* (i.e. the process together with the allocation environment of the node where the evaluation is taking place). The difference between operation $\mathbf{out}(P)@\ell$ and $\mathbf{eval}(P)@\ell$ is that **out** adds the closure of P to the tuple space located at ℓ, while **eval** sends P, not its closure, for execution at ℓ. Therefore, if node s_1 performs an **out** of P to node s_2, when P is executed at s_2, `self` will actually refer to s_1. This means that *static scoping* is used for evaluating localities. On the contrary, if s_1 spawns P at s_2 with **eval**, no closure is sent: P will refer to s_2 when using `self` and *dynamic scoping* is used.

13.2 Distribution

13.2.1 Basic entities

As recalled in the overview, KLAIM nets provide the infrastructure to coordinate processes accessing and sharing a set of resources over a configurable distributed system. Nets are sets of nodes. A node (an active KLAIM abstract machine) consists of a site (network address), an allocation environment, a set of running processes (the active computational entities) and a tuple space (the resources). Processes may access tuple spaces through explicit naming: operations over tuple spaces are indexed with their address. The net primitives are designed to handle all issues related to physical distribution, scoping and mobility of processes: the visibility of localities, the allocation policies of tuple spaces, the scoping disciplines of mobile agents, ...

Making domains apparent

KLAIM has an explicit notion of domain as node of a KLAIM net, that is well-distinguished with respect to the rest of the net. Different domains cannot have the same name (site). New domains can be dynamically created by processes by invoking the operation **eval**.

Names and references

Sites are the concrete names of the nodes and are the main linguistic constructs to provide network references to nodes. Localities are the symbolic names of nodes and programmers need not to know the concrete network references. Allocation environments associate localities to sites.

13.2.2 Concept of domain

A domain is a node of a KLAIM net and consists of a name, an allocation environment, a set of running processes and a multiset of tuples (representing the local resources). Domain names are all different and can be used as the target of process operations.

Topology

In the original KLAIM, the domain space (i.e. a net) is flat and can dynamically evolve only by creation of new domains (i.e. nodes). However, some extensions have been proposed where domains are directly or indirectly (via the interconnection topology) structured into a hierarchy [BLP02].

Semantics

Domain semantics is oriented towards security and permits controlling both process migration and remote communication. Domain names obey a lexical scoping strategy, like in the π-calculus. Processes can dynamically create new (domain) names and then pass such names to other processes via tuple exchange.

When new domain names are created, the semantics guarantees their uniqueness within the whole net via a mechanism similar to the one used for modelling name restriction in the π-calculus. Moreover, as we will see in Section 13.4, for accessing domains and their resources, processes need corresponding access privileges.

Observability/controllability of domains

The control of domains is enforced by associating a security policy to each domain. Security policies are oriented towards access control and, by means of capabilities and privileges, constraint the permissible operations from the associated domain to the others.

13.2.3 Local communication

As recalled in the overview, KLAIM communication mechanism is asynchronous (output operations are non-blocking), anonymous (tuples have no name), associative (tuples are content addressable) and higher order (processes can be exchanged as primary class data). Thus, communication takes place by first adding a tuple to a tuple space and then accessing it. Local, or intra-domain, communication uses the tuple space of the node where both the producer process and the consumer process are (not necessarily concurrently) executed. This tuple space can be referred to either directly by specifying its network address (i.e. site) or indirectly by using a local alias, such as the special locality `self`. Communication can be controlled by means of access control policies.

13.2.4 Remote communication

Extra-domain communication can take place with mechanisms similar to intra-domain communication, provided that access control policies allow the necessary operations. Of course, the names (i.e. sites) of remote tuple spaces need to be known (possibly by using the local allocation environment) by the processes that intend to use them. Tuple delivery is implicit (in the sense that a tuple sent to a tuple space is automatically received by the target space) while the target of a communication must be explicitly specified. Distant communication is global in the sense that distant tuple spaces are reachable as if the were local and the path towards the target node is left implicit. However, in the extension proposed in [BLP02], the node connection topology is explicitly modelled and the path followed by a remote action is made clear.

13.3 Mobility

13.3.1 Migrating entities

KLAIM migrating entities are pieces of process code (domains are immobile), that can be moved before their execution starts. At the moment of migration, the context of the entity is the allocation environment of the original node[15]. KLAIM supports two different mechanisms for mobility.

- **eval** moves a process code to a remote node, and breaks the link to the original context and establish a link with a new context in the target domain. This entails that a *dynamic scoping* discipline for locality evaluation is adopted.

- **out** moves a process closure, i.e. a process code along with a copy of its context. This entails that the bindings of localities to sites are fixed and, then, a *static scoping* discipline for locality evaluation is adopted.

[15]The definitions of the processes (possibly indirectly) invoked by the migrating process are also part of its executable context; such definitions remain implicit in the KLAIM operational semantics, while their management is an important aspect of the language implementation [BNP02].

Moreover, while **eval**$(P)@\ell$ directly spawns process P for (remote) execution at ℓ, processes sent with an **out** have to be explicitly retrieved from the tuple space of the destination node and assigned to process variables. The receiver can then execute the received processes locally, e.g. **in**$(!X)@\texttt{self}.X$, or can spawn them for parallel execution possibly at other sites, e.g. **in**$(!X)@\texttt{self}.$**eval**$(X)@\ell_1.$**eval**$(X)@\ell_2.P'$.

13.3.2 Expressing migration

Migration is invoked by executing operations **eval** and **out**, that explicitly require the name of the target node. Both forms of migrations are examples of *objective migration*.

13.3.3 Migration during execution

Migration of a process code takes place before its execution and follows the *remote evaluation* paradigm.

13.3.4 Migrating an associated state

KLAIM migration falls in the category of *weak migration* thus there is no state of the memory associated to the entity that is moved along with it. This is apparent in the language implementation [BNP02] (indeed, the operational semantics of the language abstracts from any notion of state).

13.3.5 Remote references

While locations are always interpreted locally, sites are network references and represent node absolute addresses. Sites may be kept by a migrating entity during and after migration and always refer to the same nodes.

13.4 Security

KLAIM exploits a *capability-based type system* [NFPV00] to specify and enforce access control policies. KLAIM access policies are oriented to control communication and migration between two domains, also called *computational environments* (CEs) in this section. However, by simply adding type information for tuples, the access control checks could be performed at the level of single tuples.

Capabilities do correspond to the different operations processes can perform: reading/consuming tuples, producing tuples, activating processes, and creating new nodes. Capability-based *access types* are used both to provide information about the intentions of processes relatively to the different nodes of a net and to specify the access control policy of a node with respect to the nodes of a net. In practice, to each process and to each node a *capability list* is associated that defines the list of nodes of the net it intends/can access together with the corresponding set of allowed operations (i.e. capabilities).

Capabilities have a hierarchical structure based on the assumption that a processes authorized to consume a tuple has also the privilege of reading it. The hierarchy of capabilities is reflected in the subtyping relation over access types whose interpretation is as follows: if a process P has access type δ_1, and δ_1 is a subtype of type δ_2, then P can be safely used in all contexts where a process of type δ_2 is expected.

Enforcement of access control policies is performed by the *type checker* which controls that the loaded software modules match the access policies of the nodes. Only processes (stationary or mobile) that have successfully passed the type checking phase can be executed. However, to guarantee the lack of run-time errors, it is necessary to also exploit dynamic type checking. Indeed, communication in KLAIM is anonymous and not based on typed named channels. Thus, the pattern-matching needs to compare the types of its arguments in order to establish if a communication can take place or not, otherwise it would be impossible to guarantee that the type with which a variable is annotated within its binder is actually satisfied.

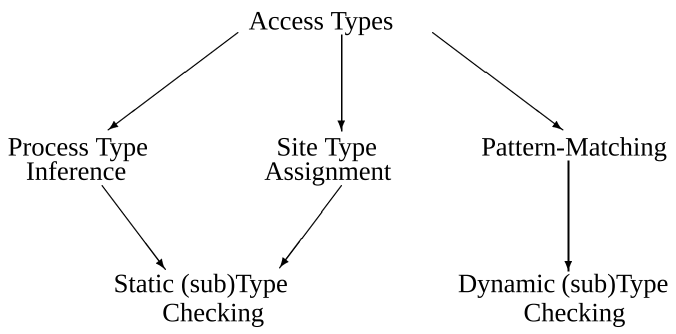

Figure 2: KLAIM type system: main ingredients

The clear distinction between specification and enforcement of access control policies is a key design element of the KLAIM type system. Indeed, the development of KLAIM applications proceeds in two phases. In the first phase, processes are programmed while ignoring the precise physical allocations of tuple spaces and the access rights of processes. By exploiting type annotations and mechanisms for code inspection, a static *type inference system* assigns types to processes and checks whether processes behave consistently with their type declarations. In the second phase, processes are allocated over the nodes of the net. It is in this phase that access control policies are set up as a result of a coordination activity among the nodes of the net. By also exploiting dynamic (sub)type checking when looking for matching tuples, it is possible to guarantee that nets where all processes do respect the security policy of their execution node will never give rise to run-time errors due to the attempt of a process of performing an operation without having the necessary privilege. Figure 2 summarizes the basic ingredients of the KLAIM type system.

14 Obliq

14.1 References

[Car95]L. Cardelli, A language with distributed scope, Computing Systems Vol. 8, No. 1 (1995) 27-59.

14.2 Overview

OBLIQ is one of the first implemented programming languages explicitly featuring mobile code([16]). It is an object-based language (without classes and inheritance), whose semantics relies on the idea of network transparency: the semantics of a program, which is a system of possibly distributed objects, does not depend on the physical location of its components. This is only possible if the migration of a component does not affect the references it may possess on other, possibly remote components. In particular, the languages offers a distributed lexical scoping discipline.

OBLIQ was designed before the advent of the world-wide-web, and it is more suited for LANs, where communication can be assumed to be as reliable as in a single machine, than for WANs like the web, where apparent, transient failures of nodes are part of the behaviour of the network. Nevertheless, OBLIQ is an interesting proposal for distributed programming. In particular, it introduces various forms of migration and has a clear – though not formalized – semantics([17]).

14.3 Distribution

14.3.1 Basic entities

The basic entities in OBLIQ's implementation are the sites, memory locations – that would be called references following ML's terminology –, values and threads. Amongst values are objects and closures. As usual, a closure is a piece of code (normally, a function or a procedure) together with an environment providing the value of its free identifiers. An object is a record of fields whose value mais be a method (with a "self" parameter), an alias or any other kind of value. The operations on objects include field selection, method invocation, field updating (and method overriding), delegation (by which every field is transformed into an alias for the same field of another object), and cloning. Each reference, and also each object, belongs to a unique site. OBLIQ also relies on a notion of network reference, which is a pair of a (name of a) site and a memory location at this site.

Making domains apparent

Sites are completely implicit in OBLIQ, and the only way to (indirectly) refer to sites is to send requests to a name server. A name server registers objects, belonging to some site, and provides on request a network reference to the registered objects. Then the name server provides access to remote objects.

Names and references

Names in OBLIQ are identifiers, in the usual sense. The value of a name may be a network reference, pointing to a location in a site, where the actual value of the name is stored. The references, either local (i.e. memory location) or global (i.e. network reference) are run-time entities, and cannot be manipulated at the programming language level.

[16]a predecessor is EMERALD.
[17]with some flaws, see the work by Kleist, Merro and Nestmann.

14.3.2 Concept of domain

Topology

There is no explicit topology of domains. Sites are, in particular, address spaces, and nothing indicates that they may be nested, so the space of domains seems to be “flat”.

Semantics

A site is an execution unit, where threads are run. It is also the place where objects and the value of memory locations are stored.

Observability/controllability of domains

Not adressed.

14.3.3 Local communication

There is no notion of communication in OBLIQ. However, the usual operations of assigning a value to a variable, calling a procedure or a function, and in particular invoking a method of an object may be regarded as communications.

14.3.4 Remote communication

There is, in the language, no difference between local and remote communication: every operation has a global meaning that, in principle, does not depend on the physical distribution. Any value, except an object, may be transmitted through the network. However, only a part of the value is transmitted in general, since references (memory locations) and objects do not move. Then they are replaced, in the transmitted value, by network references (which can be transmitted).

14.4 Mobility

14.4.1 Migrating entities

The values, and in particular procedure and functions, or more accurately closures, but not objects, can migrate, either as arguments or because they are called for.

14.4.2 Expressing migration

Migration is not explicitly a programming construct in the language. However, there are various kinds of migration that are associated with the primitive operations of the language. For instance, when the value of a field of an object, which is not a method, is selected, or when the value of a reference is required, then this value is copied (according to the process described above, that is up to the embedded parts which are not transmissible, but transformed into network references) and migrates to the calling site. This allows for “code on demand”, when the value is a procedure or a function. Conversely, when a method of an object is invoked, its evaluation is performed in the site hosting the object. When a field of an object or a reference is updated, the new value migrates to the site where the object or the reference is residing. This allows for “code shipping”. When an object is cloned, a copy of it is created in the calling site. Since one may transform the fields of an object into aliases of fields of another object, one may create “surrogates” for objects, and thus one may migrate an object, by first cloning it and then transforming its parent into a surrogate for its clone.

14.4.3 Migration during execution

In the reference cited above, it is said that “*threads may jump from site to site while retaining their conceptual identity*”, but there is apparently no means in the language to express that. Then, since the migrating entity are values, migration during execution does not seem possible.

14.4.4 Migrating an associated state

The memory locations are attached to sites, and cannot move. Therefore there is no migration of the sate associated with a moving entity. However one can for instance program an agent having an explicit state parameter, and send this agent to a remote site together with a copy of this parameter.

14.4.5 Remote references

One of the main features of OBLIQ is that the references that some entity of the language possesses are never broken. They may be replaced by network references upon migration of this entity, when their value is immobile (that is, when it is an object or a memory location).

14.5 Security

Not adressed.

15 Agent Tcl

References

[Gra95] R. Gray. Agent Tcl: Alpha Release 1.1. 1995. Software and documentation available at `http://agent.cs.dartmouth.edu/software/agent1.1/`.

[Gra96] R. Gray. Agent Tcl: A Flexible and Secure Mobile-Agent System. In Proc. *Fourth Annual Tcl/Tk Workshop (TCL'96)*, M. Diekhans and M. Roseman eds, pages 9–23, 1996.

[Gra97] R. Gray. Agent Tcl: A Flexible and Secure Mobile-Agent System. *PhD Thesis*, Dartmouth College, 1997.

[KGN+97] D. Kotz, R. Gray, S. Nog, D. Rus, S. Chawla and G. Cybenko. Agent Tcl: Targeting the Needs of Mobile Computers. In *IEEE Internet Computing*, 4(1), pages 58–67, 1997.

[GKCR98] R. Gray, D. Kotz, G. Cybenko and D. Rus. D'Agents: Security in a Multiple Language Mobile-Agent System. In Proc. *Mobile Agents and Security Workshop*, LNCS 1419, 1998.

[Pro02] The D'Agents Project. Center for Mobile Computing, Dartmouth College, 2002. See also `http://agent.cs.dartmouth.edu/`.

15.1 Overview

AGENT TCL [Gra95, Gra96, Gra97, KGN+97] is an extension of the TCL scripting language to enable the programming of mobile agent systems, where the agents can migrate and communicate while performing computations. The newer version of the system is known as D'AGENTS [Pro02, GKCR98].

15.2 Distribution

15.2.1 Basic entities

AGENT TCL adds to the TCL scripting language a number of primitives to allow the creation and the migration of agents, which are represented as TCL scripts. Apart from the scripts themselves, the basic entities are *agents* and *agent servers* which both possess unique names. An agent comes in the form of a script enclosed by an `agent_begin` and `agent_end` pair of commands, with an optional parameter specifying the network site the agent is to be run on. The agent can be given a symbolic name using the `agent_name` command.

Making domains apparent

In AGENT TCL, domains take the form of agent servers, which are system processes running on network sites and in charge of the execution of a number of agents. The server includes a script interpreter and agent migration mechanisms. At the language-level, an agent server is only accessible through its name.

Names and references

An agent server is identified by the IP address of the underlying site. An agent name includes the name of the agent server running the agent and a unique private name.

15.2.2 Concept of domain

Topology

The topology of domains is flat, each agent server being bound to a single site. A server will only accept an incoming migrating agent if it comes from a trusted site.

Semantics

In AGENT TCL, domains occur as units of execution. Being bound to a site, an agent server is also a failure unit, although that aspect is not addressed by the language.

Observability/controllability of domains

Not addressed.

15.2.3 Local communication

Interactions primarily happen between agents. AGENT TCL provides a single communication mechanism both for local and remote interactions, the latter being the default interaction mechanism. The names necessary to establish a communication comprise the IP address of the agent server running a given agent, and the unique private name of that agent. Two types of primitives are provided for inter-agent communications: by message passing, and by rendez-vous.

- Message sending uses the `agent_send` command which takes as arguments the name of the destination agent, and a message content of type `string`. The send operation is blocking with a default timeout. Reception relies on the `agent_receive` command, which can have either blocking or non-blocking semantics. Communication is monadic and point-to-point. The unicity of receivers is guaranteed since the receiver is always an agent, i.e., the one which executes the `agent_receive` command.

- A rendez-vous occurs by establishing a direct connection (named message stream) between two agents. Synchronization makes uses of the `agent_meet` command which explicitly selects a particular destination agent the source agent wants to synchronize with, and the `agent_accept` command to accept or reject the rendez-vous. The semantics of those commands are similar to the message passing case.

15.2.4 Remote communication

For a message, the name of the destination agent always includes the localization of that agent. It seems that if the destination agent has moved, the source agent is simply blocked, as communication is synchronous.

15.3 Mobility

15.3.1 Migrating entities

The migrating entities are the agents.

15.3.2 Expressing migration

Migration can be expressed using three special commands:

- The `agent_submit` command, taking as parameters the name of an agent server S and a TCL script σ, will provoke the creation of a new agent on S with body σ. The variables and procedures defined in the father agent used by the newly created agent must be specified explicitly. Upon termination of execution, the child agent will return its identifier to its father.
- The `agent_jump` command provokes the migration of the current agent towards a specified agent server. Migration is blocking with respect to the migrating agent, since its execution will resume directly at the point where it was interrupted due to the migration request.
- The `agent_fork` command creates a copy of the current agent on a specified agent server, which will give that "clone" a new identity.

15.3.3 Migration during execution

The `agent_submit` command can be seen as a form of code shipping, while the two other commands allow a strong form of agent migration (mobility of continuations). It should be noted that since AGENT TCL agents are sequential, a given agent cannot simultaneously perform a migration request, and send or receive messages.

15.3.4 Migrating an associated state

The execution of the `agent_submit` command provokes the copy of the variables and procedures specified as arguments to the command. Similarly, the commands `agent_jump` and `agent_fork` provoke a copy of the state of the current agent into the newly created agent, without any further details regarding what exactly is covered by the term "agent state" (what happens in the case of sharing?).

All migration commands induce the agent migration proper, and an update of the table of the migrating agent which holds agent identification information.

15.3.5 Remote references

Although agents are designated in a unique fashion (agent names include localization information), there is no follow-up of a migrating agent, and thus no notion of a remote agent reference transparent to localization.

15.4 Security

The security model of AGENT TCL aims to protect network sites against malicious agents, and to protect agents against one another. When an agent arrives on a site, it will be authenticated by the local agent server, who will transmit the authenticated agent to the appropriate interpreter. Once authentication is completed, AGENT TCL uses *resources manager agents* (RGA) which guard access to critical resources. The security model of SAFE TCL is used to ensure a safe execution of agents within the limits set by the RGAs.

15.4.1 Identification and Authentication

Each agent and agent server possess a well-defined identity. Authentication is based on the PGP (Pretty Good Privacy) [18] protocol. An agent is authenticated by proving his knowledge of a cryptographic key. Authentication occurs in several situations:

[18]Message encryption relies on the IDEA shared key cryptosystem. The shared IDEA session key is first generated randomly by the sender, and encrypted using the public key of the recipient using the RSA algorithm. The encrypted IDEA session key and the ciphertext of the message are then sent to the recipient. An option is available to sign the message using the MD5 algorithm.

- An agent registers with an agent server: the registration request is signed with the private key of the agent and encrypted with the public key of the agent server.
- An agent migrates: the migration request is signed with the private key of the agent server and encrypted with the public key of the recipient. This requires a certain amount of trust between the various agent servers, since the message is signed using the server's private key. The identity of the migrating agent is transmitted in the migration request. If authentication succeeds, the recipient server records the identities of the agent and of its sending server, and the degree of confidence it can place in the validity of the server identity.
- An agent sends a message towards another agent located on a different machine, or spawns an agent remotely: the process is similar.

The AGENT TCL authentication suffers from the following weaknesses: first, there is no distribution mechanism for PGP public keys. Second, the system is vulnerable to replay attacks.

15.5 Access Control

Two types of resources must be distinguished:

- *Built-in resources* are directly accessible through the language primitives. Typical built-in resources include the screen, the file system, or the system clock.
- *Indirect resources* can only be accessed through another agent.

15.5.1 Access Control Policy

It is not explicitly defined, but is contained in the ACLs (Access Control Lists) defined in the trusted interpreter (see below).

15.5.2 Authorization Model

- For indirect resources, a special agent *A* controls the resource by maintaining ACLs of authorized agents which can use that resource. To grant resource access to some agent *B*, *A* takes into account *B*'s identity, the identity of *B*'s agent server, whether *B* and its server could be authenticated, and a confidence level representing the trust *A*'s agent server can place in *B*'s agent server.
- For built-in resources, access control relies on the use of SAFE TCL and on the RGAs. For each agent, SAFE TCL provides both trusted and untrusted interpreters where dangerous commands are forwarded to the trusted interpreter which will authorize or not their execution using ACLs containing the name of the resource, and the authorized number of instances of that resource. Those ACLs are kept by the RGAs. An agent can explicitly ask a RGA for the authorization to use a built-in resource with the keyword `require`, in which case the request is forwarded to the trusted interpreter. Otherwise, access control is performed implicitly.

Future versions of AGENT TCL should support the protection of groups of sites against malicious agents. Solutions could include currency-based resource allocation algorithms, where each agent keeps a finite supply of currency it spends with each access to resources.

15.5.3 Privilege Delegation

Not addressed.

15.5.4 Security Domains

In exchanges between agent servers, the server appears as a domain of trust: a server *S* will trust an external agent *A* provided *S* trusts the agent server currently hosting *A*. In inter-agent communications, the unit of trust is the agent.

An agent server is also a domain of protection in itself: it contains a trusted interpreter, and holds access rights for built-in resources. In particular, the RGAs are members of that protection domain.

15.6 Communications Security

Several situations requires protection of communications:

- Once an agent registered with an agent server, all further communications are encrypted using an IDEA session key.
- When an agent is created remotely, the agent script is signed and encrypted before being sent to the remote site.
- When an agent migrates, its image is signed and encrypted before being sent to the remote site.

15.7 Auditing

Should be addressed in future version of AGENT TCL.

15.7.1 Security Policy Administration

A special `console` agent keeps track of agents which execute on a given network site, and can forbid the migration of new agents, or destroy agents curently under execution.

For the moment, ACLs contained in the trusted interpreter cannot be modified. Future versions of AGENT TCL should allow to define adaptable security policies for an agent server or a group of agent servers.

References

[ABL99] R. Amadio, G. Boudol, and C. Lhoussaine. The receptive distributed pi-calculus. In *Proceedings FST-TCS'99*, volume 1738 of *Lecture Notes in Computer Science*, 1999.

[ABLW92] M. Abadi, M. Burrows, B. Lampson, and E. Wobber. Authentication in Distributed Systems: Theory and Practice. *ACM Transactions on Computer Systems*, 10(4):265–310, November 1992.

[ABvDW96] M. Abadi, M. Burrows, L. van Doorn, and E. Wobber. Secure Network Objects. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, May 1996.

[AG99] M. Abadi and A. Gordon. A calculus for cryptographic protocols: The spi-calculus. *Information and Computation*, 148(1):1–70, 1999.

[Ama97] R. Amadio. An asynchronous model of locality, failure, and process mobility. In *Proceedings COORDINATION 97*, number 1282 in LNCS, 1997.

[Ama00] R. Amadio. On modelling mobility. *Theoretical Computer Science*, 240:147–176, 2000.

[BAN90] M. Burrows, M. Abadi, and R. Needham. A Logic of Authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, February 1990.

[BCC01] M. Bugliesi, G. Castagna, and S. Crafa. Boxed ambients. In *Proceedings TACS 2001*, number 2215 in LNCS, 2001.

[BLP02] L. Bettini, M. Loreti, and R. Pugliese. An infrastructure language for open nets. In *Proc. of the 2000 ACM Symposium on Applied Computing (SAC'02), Special Track on Coordination Models, Languages and Applications*, ACM Press, pages 373–377, 2002.

[BNP99] M. Boreale, R. De Nicola, and R. Pugliese. Symbolic trace analysis of cryptographic protocols. In *Proceedings LICS 99*, IEEE Computer Society Press, 1999. Full version to appear in *SIAM Journal on Computing*.

[BNP02] L. Bettini, R. De Nicola, and R. Pugliese. Klava: a java package for mobile code, 2002. Submitted for publication.

[Bor01] M. Boreale. Symbolic trace analysis of cryptographic protocols. In *Proceedings ICALP 01*, number 2076 in LNCS, 2001.

[Car95] L. Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, 1995.

[Car99] L. Cardelli. Abstractions for mobile computations. In *Proceedings Workshop on Secure Internet Programming*, number 1603 in LNCS, 1999.

[Cas01] I. Castellani. Process algebras with localities. In J. Bergstra, A. Ponse, and S. Smolka, editors, *Handbook of Process Algebra*, pages 945–1045. North-Holland, Amsterdam, 2001.

[CG98] L. Cardelli and A. D. Gordon. Mobile ambients. In *Proceedings FoSSaCS 98*, number 1378 in LNCS, 1998.

[CG99] L. Cardelli and A. D. Gordon. Types for mobile ambients. In *Proceedings POPL 99*, ACM Press, pages 79–92, 1999.

[CGN01] G. Castagna, G. Ghelli, and F. Zappa Nardelli. Typing mobility in the seal calculus. In *Proceedings CONCUR 2001*, number 2154 in LNCS, 2001.

[FG96] C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings POPL 96*, pages 372–385, 1996.

[FGL$^+$96] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *Proceedings CONCUR 96*, number 1119 in LNCS, 1996.

[FL02] F.Germain and M. Lacoste. The K-calculus, 2002. Draft.

[fS88] International Organisation for Standardization. *ISO 7498-2 Basic Reference Model for Open Systems Interconnection (OSI) Part 2: Security Architecture*. Geneva, Switzerland, 1988.

[GC99] A. D. Gordon and L. Cardelli. Equational properties of mobile ambients. In *Proceedings FoSSaCS 99*, number 1578 in LNCS, 1999.

[GHS02a] J. C. Godskesen, T. Hildebrandt, and V. Sassone. A calculus of mobile resources. In *Proceedings CONCUR 02*, 2002.

[GHS02b] Jens Christian Godskesen, Thomas Hildebrandt, and Vladimiro Sassone. A calculus of mobile resources. Technical Report TR-2002-16, IT University of Copenhagen, Glentevej 67, DK-2400 Copenhagen NV, August 2002.

[GKCR98] R. Gray, D. Kotz, G. Cybenko, and D. Rus. D'Agents: Security in a Multiple Language Mobile-Agent System. In *Workshop on Mobile Agents and Security*, number 1419 in LNCS, pages 154–187, 1998.

[Gol99] D. Gollmann. *Computer Security*. Wiley, 1999.

[Gra95] R. Gray. *Agent Tcl: Alpha Release 1.1*, 1995. Software and documentation available at `http://agent.cs.dartmouth.edu/software/agent1.1/`.

[Gra96] R. Gray. Agent Tcl: A Flexible and Secure Mobile-Agent System. In M. Diekhans and M. Roseman, editors, *Fourth Annual Tcl/Tk Workshop (TCL'96)*, pages 9–23, Monterey, CA, 1996.

[Gra97] R. Gray. *Agent Tcl: A Flexible and Secure Mobile-Agent System*. PhD thesis, Dartmouth College, 1997.

[GYY] X. Guan, Y. Yang, and J. You. Making ambients more robust. In *Proceedings Int. Conf. on Software: Theory and Practice*, pages 377–384. Beijing, China, 2000.

[HR98] M. Hennessy and J. Riely. Resource access control in systems of mobile agents. In *Proceedings HLCL 98*, volume 16 of ENTCS, 1998.

[KGN$^+$97] D. Kotz, R. Gray, S. Nog, D. Rus, S. Chawla, and G. Cybenko. Agent Tcl: Targeting the Needs of Mobile Computers. *IEEE Internet Computing*, 1(4):58–67, 1997.

[Lam74] B. Lampson. Protection. *ACM Operating Systems Review*, 8, 1974.

[LFSV00] Luís Lopes, Álvaro Figueira, Fernando Silva, and Vasco T. Vasconcelos. A concurrent programming environment with support for distributed computations and code mobility. In *CLUSTER'00*, pages 297–306. IEEE, 2000.

[LS00] F. Levi and D. Sangiorgi. Controlling interference in ambients. In *Proceedings POPL 2000*. ACM Press, 2000.

[LSV99] Luís Lopes, Fernando Silva, and Vasco T. Vasconcelos. A virtual machine for the TyCO process calculus. In *PPDP'99*, volume 1702 of LNCS, pages 244–260. Springer-Verlag, September 1999.

[Lév97] J.-J. Lévy. Some results in the join-calculus. In *Proceedings TACS 97*, number 1281 in LNCS, 1997.

[Mil83] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:269–310, 1983.

[MPW92] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, Parts 1-2. *Information and Computation*, 100(1):1–77, 1992.

[MS02] M. Merro and V. Sassone. Typing and subtyping mobility in boxed ambients. In *Proceedings CONCUR 2002*, LNCS, 2002.

[NFP98] R. De Nicola, G. Ferrari, and R. Pugliese. Klaim: a kernel language for agents interaction and mobility. *IEEE Trans. on Software Engineering*, 24(5):315–330, 1998.

[NFPV00] R. De Nicola, G. Ferrari, R. Pugliese, and B. Venneri. Types for access control. *Theoretical Computer Science*, 240:215–254, 2000.

[Pie97] B.C. Pierce. Programming in the pi-calculus: A tutorial introduction to pict (pict version 4.0). Technical report, Indiana University, 1997.

[Pro02] D'Agents Project. Center for mobile computing, dartmouth college, 2002. See `http://agent.cs.dartmouth.edu/`.

[PT01] B.C. Pierce and D.N. Turner. Pict: A programming language based on the pi-calculus. In Proof, Language and Interaction: Essays in Honour of Robin Milner, *editor = G. Plotkin, C. Stirling, and M. Tofte, MIT Press*. 2001.

[RH99] J. Riely and M. Hennessy. Trust and partial typing in open systems of mobile agents. In *Proceedings POPL 99*, 1999.

[RMVL02] António Ravara, Ana G. Matos, Vasco T. Vasconcelos, and Luís Lopes. A lexically scoped distributed π-calculus. DI/FCUL TR 02–4, Department of Computer Science, University of Lisbon, 2002.

[San92] D. Sangiorgi. *Expressing mobility in process algebras: first-order and higher order paradigms*. PhD thesis, University of Edinburgh, 1992.

[San93] D. Sangiorgi. From π-calculus to Higher-Order π-calculus — and back. In *Proc. TAP-SOFT'93*, number 668 in LNCS, 1993.

[Sew00] P. Sewell. Applied π: a brief tutorial. Technical Report 498, Computer Laboratory, University of Cambridge, 2000. Based on the Notes from lectures at the MATHFIT Instructional Meeting on Recent Advances in Semantics and Types for Concurrency, Imperial College, 1998.

[SS02] A. Schmitt and J.B. Stefani. The M-calculus: A Higher-Order Distributed Process Calculus. Technical Report RR-4361, INRIA, 2002.

[SWP98] P. Sewell, P.T. Wojciechowski, and B.C. Pierce. Location-independent communication for mobile agents: a two-level architecture, 1998. Technical Report 462, Computer Laboratory, University of Cambridge. Submitted for publication.

[Tho93] B. Thomsen. Plain CHOCS, a second generation calculus for higher-order processes. *Acta Informatica*, 30:1–59, 1993.

[Tur96] N.D. Turner. *The polymorphic pi-calculus: Theory and Implementation*. PhD thesis, Department of Computer Science, University of Edinburgh, 1996.

[VB98] Vasco T. Vasconcelos and Rui Bastos. Core-TyCO, the language definition, version 0.1. DI/FCUL TR 98–3, DIFCUL, March 1998.

[VC98] J. Vitek and G. Castagna. Towards a calculus of secure mobile computations. In *Workshop on Internet Programming Languages*, 1998.

[VC99a] J. Vitek and G. Castagna. Mobile computations and hostile hosts. In *Proceedings of the 10th JFLA (Journés Francophones des Langages Applicatifs*, 1999.

[VC99b] J. Vitek and G. Castagna. Seal: A framework for secure mobile computations. In *Workshop on Internet Programming Languages*, 1999.

[VLS98] Vasco T. Vasconcelos, Luís Lopes, and Fernando Silva. Distribution and mobility with lexical scoping in process calculi. *Electronic Notes in Theoretical Computer Science*, 16(3), 1998.

[VST97] J. Vitek, M. Serrano, and D. Thanos. Security and Communications in Mobile Object Systems. In *J. Vitek (ed.) and C. Tschudin (ed.). Mobile Object Systems: Towards the Programmable Internet*, *Lect. Notes in Comp. Sci. 1222*. Springer-Verlag, April 1997.

[Zil01] S. Dal Zilio. Mobile processes: a commented bibliography. In *Proceedings workshop on Modelling and Verification of Parallel processes*, number 2067 in LNCS, 2001.

[Zim00] P. Zimmer. On the expressiveness of pure mobile ambients. In Preliminary Proceedings of EXPRESS 00, volume NS-00-2 of *BRICS Notes Series*, pages 81–104, 2000.