MIKADO Global Computing Project IST-2001-32222

Mobile Calculi Based on Domains

Requirements for a Global Computing Programming Model

Title :	Requirements for a Global Computing Programming Model
Editor :	J.B. Stefani (INRIA)
Authors :	J.B. Stefani (INRIA)
Classification :	Deliverable D1.1.2, Public
Reference :	RR/WP1/3
Version :	1.0
Date :	January 2003

Abstract

This document presents a set of requirements for a global computing programming model. Requirements are grouped into eight classes covering issues ranging from implementability and effectiveness to support for high-level coordination and atomic activities abstractions.

Contents

1 Introduction

This document gathers a set of (informal) requirements for a global computing programming model. By global computing *programming model* we mean a computational model (calculus or framework of calculi) that can serve as a reference point for distributed and mobile programming programming in much the same way than the λ-calculus serves as a reference point for functional programming or the π-calculus for concurrent programming. This, in turn, means that the model should be both simple and effective. It should be *simple* enough that formal reasoning and proof techniques can reasonably be developed for it. It should be *effective* enough so as to serve as a basis for the development and implementation of efficient and practical programming languages. By *global computing* programming model, we mean a calculus that embodies constructs necessary to deal with key assumptions characteristics of (potentially large scale) distributed systems: asynchrony, unmaskable occurrence of failures (such as network partitions), autonomy of system constituents, spatial and logical system partitioning (for reasons of ownership, administration, trust and security), dynamic reconfiguration (for reasons of system evolution and maintenance, adaptation to environmental changes), and physical mobility (of devices and software agents).

The requirements fall in eight classes, and are documented according to the following template:

Name: This item gives a short name to the requirement.

Description This item provides a short, one or two sentence long, description of the requirement.

Rationale This item provides a detailed explanation of the requirement and the reasons that motivate it.

Discussion: This item provides additional discussion concerning the requirement, such as perceived consequences in term of programming model design, examples of relevant constructs in existing calculi, typical occurrences, relevant references, etc.

The eight classes of requirements are as follows:

Target: This class gathers requirements that pertain to the nature and intent of the targeted programming model.

Effectiveness: This class gathers requirements that pertain to the degree to which the targeted programming model can be effectively (and efficiently) implemented.

Domains: This class gather requirements that pertain to the support of domains as first-class computational abstractions.

Reconfiguration: This class gathers requirements that pertain to the support of dynamic reconfiguration activities.

Communications: This class gathers requirements that pertain to the support of different communication abstractions.

Failures and atomicity: This class gathers requirements that pertain to the support of features necessary for handling faults and failures and for building resilient global computing systems.

Resources: This class gathers requirements that pertain to the support of features necessary for the proper control and management of resources in global computing systems.

Meta-information: This class gathers requirements that pertain to the support of run-time meta-information associated with computational components.

2 Target

2.1 Open model

Description A global computing programming model should be open and extendable, so as to be descriptively complete.

Rationale Large scale distributed systems exhibit a wide variety of features, such as diverse forms of communication, diverse modes of failures, diverse forms of spatial and logical organizations. It is unlikely that a single fixed model can account effectively and at no hidden cost (see requirement 3.1) for this diversity. Failure modes, for instance, can vary greatly from simple fail-stop behavior to arbitrary (Byzantine) failures, including time and omission failures. Likewise, communications means can vary greatly from standard point-to-point asynchronous communication to anonymous broadcast and tuple space communication, with a wide range of different semantics and quality of service characteristics in between. One should thus envisage a protean model, i.e. one that can be extended when required with appropriate operators and constructs to take into account different failure models or communication semantics. This may mean in particular that, to be effective, the definition of the programming model should provide explicit means to introduce extensions, e.g. a defined format for new operators and an associated semantical framework to account for the required extensions.

Discussion An example of a protean specification language is provided by [6], where a process calculus-based specification language can be extended with new operators, provided they can be defined using the GSOS transition system specification format. By analogy, a protean programming model could consist in a principled way to introduce new operators for capturing e.g. failure modes and communication abstractions. To carry this analogy further would require the definition of a semantical framework that would play for candidate operators the role that transition systems play for GSOS-defined operators. As an interesting work in this direction, we can mention Milner's bigraphical reactive systems [24].

2.2 Supporting system programming

Description A global computing programming model should provide a model for low-level system and network programming as well as a model for programming higher-level distributed applications.

Rationale Programming a distributed system can be done at different levels of abstraction, depending on the available infrastructure and environment. For instance, a distributed middleware such as CORBA masks a number of low-level details to the application programmer and provides him/her with a set of uniform communication abstractions that hides the underlying packet-based communication facilities. On the other hand, the programmer of a firewall, for instance, needs to have access to lower-level system abstractions such as packets and protocol headers in order to implement the required filtering mechanisms. In the same vein, the programmer of a system configuration utility needs to be aware of low-level system abstractions such as processes, nodes, and communication channels in order to manage the placement and interconnection of different system components. System-level developments constitute an important target for a distributed programming model, for they are error-prone, involve the use and manipulation of critical system resources, and in general would greatly benefit from the use of formally defined languages and abstractions.

Discussion This requirement is consistent with requirement 2.3 on selective transparency, as well as requirements 3.1 on hidden costs and requirement 4.1 on definable domains. The underlying assumption behind this requirement is that higher-level abstractions can be built from a system-level programming model, but not the other way around (i.e. it is much more difficult to provide access to lower-level resources and abstractions within a model that takes high-level constructs as primitive). That formalizing system-level programming is possible is exemplified by the work done at Cambridge on the TCP calculus [35]. A consequence of this requirement is that one expects to build higher-level abstractions from a core calculus, much as the Pict and Nomadic Pict languages [34] are grown from a π-calculus core.

2.3 Providing selective transparency

Description A global computing programming model should not force given distribution transparencies on application programmers. Instead, it should allow programming distributed applications relying only on selected distribution abstractions.

Rationale Application requirements are extremely varied and unpredictable. Distribution transparencies tend to impose undue abstractions for certain applications. For instance, constructing distributed system management functions requires access to low-level system resources, which particular abstractions, such as communication abstractions, may unduly mask. Application designers can also make different trade-offs, depending on the characteristics and the requirements of their applications. For instance, they can trade consistency in favor of reaction time, or ease of use in favor of security,

in effect requiring less dependability from the supporting infrastructure. As a result, one would require a distributed programming model to allow these different trade-offs to be made, notably by allowing system and application programmers to select and use the key abstractions they require.

Discussion This requirement is consistent with the requirement on system-level programming 2.2, as well as the requirement for an open model 2.1. It is reminiscent of the general trend in distributed system and operating system construction to avoid adding undue abstractions as part of a system's programming interface [12, 13]. It is also reminiscent of the recent rebuttal of transparency in distributed object-oriented programming [23].

3 Effectiveness

3.1 No hidden costs

Description The primitives of a global computing programming model should not provide abstractions masking the basic costs of the supporting distributed infrastructure, notably communication facilities.

Rationale This requirement is key to achieving an efficient implementation of a distributed programming model. In essence it mandates that the primitives of a core programming model be close to the capabilities offered by the infrastructure on which an implementation of this programming model would rely. That way, applications that only require the level of service provided by the basic infrastructure can be efficiently implemented. An example will illustrate this requirement. Suppose, as is the case with the Internet, that the basic infrastructure provides only point-to-point asynchronous communication as a communication primitive. One could think of adopting in a core distributed programming model, a form of communication based on multipoint rendez-vous since it has been shown (e.g. in the course of experiments and specifications with the LOTOS language) that it allows clear and readable design styles (e.g. the constraint-oriented style) for complex coordination patterns. Unfortunately, this choice would be hampered by the following two facts:

1. Multipoint rendez-vous is difficult and expensive to implement in a distributed environment, especially if failures (node, link) can occur[1].

2. A number of applications do not require multipoint rendez-vous to do their job. For instance, basic, read-only access to Web pages only relies on the point-to-point reliable data stream service provided by the TCP protocol; surveillance or monitoring applications can operate efficiently just relying on the simple point-to-point, unreliable asynchronous message service provided by the UDP protocol.

[1] Distributed rendez-vous constitutes a form of distributed consensus which is impossible to implement using only non-randomized algorithms [27].

Choosing multipoint rendez-vous as a communication primitive in this context would impose the implementation costs of multi-way rendez-vous for each communication instance, regardless of the precise application needs.

Discussion Note that the requirement is explicitly *not* phrased in terms of required communication primitives. Indeed, depending on the context, certain forms of communication may or may not be expensive to implement. As a simple example, consider message broadcast. While broadcast is notoriously expensive to implement in the global Internet, broadcast is very cheap to provide in wireless networks. In an open model, i.e. one conforming to requirement 2.1, one would thus expect to be able to use broadcast as a basic communication primitive in contexts where it is unexpensive to do so. The requirement thus has less to say about the basic features of a possible model for global computing than about the effectiveness or cost of implementation of basic primitives of such a model in a given context. This makes it consistent with both the requirement on an open model 2.1 and the requirement on system-level programming 2.2.

C. Fournet [14] has made a detailed case for the basic constructs of the Join calculus (essentially, an asynchronous, localized π-calculus with replicated input [31]) by analyzing the implicit costs associated with the implementation of π-calculus and asynchronous π-calculus communication primitives in an asynchronous point-to-point communication infrastructure. In our view, this is an argument in favor of the no-hidden cost requirement advocated here.

Another example of the importance of this no-hidden cost requirement is provided by the implementation of Mobile Ambients in the Distributed Join calculus [15]. Since the DJoin relies on asynchronous point-to-point communication, the complex protocol required for implementing Mobile Ambients in the Distributed Join reflects the costs associated with Mobile Ambients primitives, in effect demonstrating that Mobile Ambients mobility primitives are too costly to use, in an asynchronous point-to-point communication context, as communication primitives.

3.2 No predefined policy

Description The primitives of a global computing programming model should not encapsulate pre-defined, fixed resource management or security policies.

Rationale In line with the requirements on system-level programming 2.2 and on the "no hidden costs" requirement 3.1, one would expect a global computing model to avoid forcing distributed application programmers to use any particular access control policy. There can be many such policies, depending on the particular goals pursued, and trade-offs made, by application programmers and designers.

Discussion This requirement can be seen as a variant of both the no-hidden cost requirement and the selective transparency requirement. In effect it mandates a basic programming model for global computing to not make specific choices in policies governing access to and the use of resources or information. These should be best left

to specific libraries or derived constructions. Note that there is an implicit codicil attached to this requirement: we expect resource management and security policies to be definable in a global computing model. This is made clear in the requirement on the definability of domains 4.1.

4 Domains

4.1 Supporting multiple domains

Description A global computing programming model should allow the definition of different forms of domains and their associated boundaries.

Rationale A large scale distributed system should really be understood as a system of systems, with different regions of varying characteristics, e.g. with respect to failure behavior, security policies, resource management policies, etc. We call *domain* the region enclosed by such boundaries. A domain is thus primarily a locus of distributed computation, i.e. a place where some computational activity can take place. But in a global computing environment there can be many forms of domains:

- Communication and coordination: a domain can be a region whithin which processes can exchange information and synchronize their activity, according to given communication or synchronization primitives. Examples of communication domains abound. For instance, a single machine, a set of machines interconnected by an Ethernet LAN, or a radio cell in a wireless network constitute different communication domains, characterized by very different communication capabilities.
- Migration: a domain can be a unit of mobility, enclosing activities that can move in a network of physical or logical places (yet other forms of domains).
- Failures: a domain can be a region characterized by a given failure mode, whose failure entails, from the point of view of an external observer, the failure of all computational activities it contains.
- Administration: a domain can be a region where computational activities are subject to certain policies, e.g. for monitoring, resource management and control.
- Security: a domain can be a region protected from the outside world, such as a subnet protected by a firewall, or that exercizes monitoring and control on untrusted activities, such as a sandbox.

A global computing programming model should provide the means to capture the characteristics of (to model) and to define (to program) these different kinds of domains.

Discussion L. Cardelli has argued forcefully [8] for a view of a wide-area network system characterized by the presence of multiple administrative and logical boundaries. Our notion of domain agrees with Cardelli's view of global computations. However, the notion of domain as discussed above goes beyond the view of domain as a migration unit (both as a place to move to and as a piece of computation to be moved) which is propounded in Mobile Ambients [9].

The openness requirement 2.1 reinforces the view that there exists a wide variety of domains in a global computing environment. Thus, definability, in presence of the openness requirement, means that there are principled ways to extend a global computing programming model to encompass new forms of domains and their associated semantics.

From the examples of domains given above, a common thread can clearly be seen: a domain can be understood as a locus of computation which superposes [25] some control behaviour on the computational activities it hosts. This control behaviour entails some form of communication interception (which is required e.g. for the sandbox or firewall behaviour) and at least the ability to interrupt and passivate the contents of a domain (which is required e.g. for mobility or to account for the behaviour of a domain in presence of failures). This is essentially the intuition behind the M-calculus constructs [32].

4.2 Supporting domain graphs

Description A global computing programming model should allow the description of global computations as graphs of domains.

Rationale The spatial organization of computations in a global computing environment can exhibit complex structures. For instance, an enterprise intranet can be seen as a hierarchy or tree-shaped structure of subnets protected and interconnected via routers, gateways and firewalls. A network aware application such as a network management application must have an explicit access to such a structure, e.g. for monitoring or for configuration management purposes. If one considers the various forms of domains which can spatially partition a global computation, one must consider in general domain structures which can form arbitrary graphs, not just trees or forests. For instance, security domains may be organized in a tree structure to reflect different levels of access control and protection, but these domains may run on different machines whose interconnection and failure modes need not reflect the access control hierarchy. More generally, a global computation may involve physical as well as logical domains, which may overlap in various ways.

Discussion C. Fournet gives in its thesis [14] several arguments to justify the presence of a hierarchy of localities in the Distributed Join calculus. First, a hierarchical model of localities allows machines themselves to be represented as localities, yielding a uniform treatment of partial failures and migration. Second, it allows to use localities uniformly both as places other localities can move to, and as units of migration (localities as mobile agents). This, in turn, allows natural atomicity properties to be

maintained in case of migration (in the Distributed Join calculus, the `go` primitive is atomic and stipulates that a locality can migrate in one atomic step to another locality provided it has not failed), something which would be more cumbersome to do in a flat model where the dependency relationships between agents and places are not explicit. One should note that, even in distributed calculi that do not support arbitrary tree-shaped configurations of localities or domains, there are some elements of hierarchy present. For instance, Nomadic Pict [36] makes the distinction between processes, agents and sites: an agent is a unit of migration and a site corresponds to a place where agents can migrate to, whereas processes are units of activity within an agent.

Considering graphs of domains instead of just tree structures proceeds from similar considerations. First, it should allow a uniform treatment of different aspects in a global computation (security, communication, failures, etc), where each aspect can be attached to a particular kind of domain. Second, it should allow more complex and realistic system models to be captured, together with atomicity properties of distributed system structures.

5 Reconfiguration

5.1 Supporting dynamic reconfiguration

Description A global computing programming model should support the design and programming of systems made of interconnected components, and should support the dynamic reconfiguration of such component structures.

Rationale Applications in a global computing environment are likely to be assembled from existing pieces (running processes, servers, software libraries, etc). Providing support for the on-the-fly construction of applications requires some notion of component and the understanding of late binding between such components. Furthermore, considering the long-lived nature of global computing systems, one needs to be able to dynamically reconfigure an existing component structure, in order to cater for changes in the environment, changes in application requirements, control changes (e.g. migrating some server to a different site for load balancing or improved performance), or just maintenance changes (e.g. upgrading or replacing a faulty component).

Discussion Recent research in distributed system support has emphasized the need for highly flexible middleware structure to support the component-based construction of distributed applications (see e.g. [1, 5]). A global computing programming model should provide a formal basis for such component-based construction and natively support basic mechanisms for dynamic reconfiguration. Dynamic reconfiguration, understood in all its variants, constitutes the prime requirement for a computing model with inherent process mobility. Among the basic actions required to support dynamic reconfiguration, one finds: suspending and passivating running components; instantiating new components; activating passivated components; resuming suspended components; adding , removing or replacing a component from a given component structure; connecting (resp. disconnecting) two or more components to enable (resp. disable)

their interaction. Research on configurable distributed systems, software configuration management and software architecture has begun in the last decade to make use of process calculi to define and model component structures (see e.g. [29, 3, 26]).A global computing programming model should at least support the range of configuration and reconfiguration functions investigated in these works.

5.2 Supporting component life-cycle

Description A global computing programming model ought to support the life-cycle of distributed components.

Rationale Software components in a distributed system can appear in different guises, and can exhibit complex life-cycles, going e.g. from software code or template to running process and back, with different installments in between, depending e.g. on the details of deployment, instantiation, activation, de-activation, or removal phases they go through. A global computing programming model should provide support for these different phases and allow different component life-cycles to be supported. This in turn may require support for activities ranging from meta-programming to dynamic assembly and dynamic reconfiguration of software component structures.

Discussion This requirement can be seen as a corollary of the general requirement on dynamic reconfiguration 5.1. The emphasis here is on the ability to deal with activities such as the dynamic assembly of software structures which, traditionally, are not taken into account in a programming model, but are the realm of supporting tools such as pre-compilers, linkers and loaders. In large scale and long-lived systems, however, one is bound to consider a mix of such activities together with standard component executions, as systems evolve through patches, upgrades, additions of new modules and plug-ins, architectural modifications, on-line optimizations, and the like.

As a simple example, consider the loading of (potentially untrusted and unsecure) plug-ins. To model precisely what is taking place in this instance, and to devise appropriate loader or sandbox programs, one must deal with the different forms the plug-ins assume: code, whether in executable or byte-code format for transmission and loading; executable component structure, after successful loading; running process, after activation. With appropriate meta-information on the plug-in, one could imagine programming different kinds of actions or transformations as part of the loading phase (e.g. wrapping, verification, partial evaluation), for security, adaptation or optimization purposes.

6 Communication and coordination

6.1 Supporting multiple communication and binding semantics

Description A global computing programming model should be open to different communication semantics.

Rationale This is a refinement of the open model requirement 2.1, with an emphasis on communication. At the very least, one should expect a global computing programming model to provide the basic forms of communication readily available in current day infrastructures such as asynchronous point-to-point communication, asynchronous broadcast, and asynchronous multicast. More generally, one should expect support for the definition of different forms of binding between distributed components.

Discussion The notion of binding relates to the establishment of communication paths between components in a distributed system (see e.g. [12]). It has long been recognized that multiple forms of bindings can coexist in a distributed system, encompassing many communication semantics. For instance, even in a simple client-server setting, one can have many variations on the standard RPC interaction semantics, including semantical variants of the basic RPC (e.g. at most once, exactly once, timed RPC) and the numerous forms of group RPC between replicated clients or servers. A global computing programming model should be open to extension and allow the definition of arbitrary forms of bindings between distributed components.

6.2 Supporting distributed coordination

Description A global computing programming model should support high-level abstractions for distributed, possibly even large-scale, coordination.

Rationale Current coordination mechanisms in distributed systems are fairly low-level (asynchronous message exchange, RPC, some forms of workflow coordination). One should expect a global computing model to provide higher-level abstractions for coordination among distributed participants.

Discussion Andrade, Fiadeiro et al [4] rightly argue that coordination facilities [19] are key to the orchestration of services in a large scale, dynamic distributed environment such as the Internet. These typically involve a combination of dynamic reconfiguration (see requirement 5.1), superposition (see requirement 4.1), and software architecture techniques (see requirement 5.1) to build dynamically assemblies of services and their supporting components, tied together by means of behavioural contracts. There have been numerous proposals for coordination languages and facilities (see e.g. [2, 16, 17]). Approaching coordination in a global computing environment taking into account the partition of computations into domains, remains as a challenge, however.

7 Failures and Atomicity

7.1 Handling failures

Description A global computing programming model should provide support for handling failures and devising fault-tolerant distributed systems.

Rationale Faults, errors and failures are inevitable in a global computing environment. Site crashes, network partitions, fraudulent intrusions, are examples of the many forms of failures and faults (intentional or not) that may occur in such an environment. It is therefore crucial in a global computing programming model to provide appropriate abstractions to deal with failures.

Discussion Even though, in a large scale distributed setting, site failures can often be undistinguishable from long communication delays, devising practical applications must account for the possibility of failures. Progress is required, especially if a human is in the loop, to at least report suspicions of failures and avoid indefinite waiting. This suggests that, at the very least, a global computing programming model should contain constructs corresponding to basic failure detection capabilities. Indeed, this is the path that has been taken for instance in the π_{1l}-calculus with the `ping` primitive, in the Distributed Join calculus [14] with the `fail` primitive, or in the Oz language [30] with the introduction of specific failure exceptions and asynchronous failure detectors. Elementary failure detectors, in the form of timers, are also the basic means provided by standard computing infrastructure (operating systems and middleware) to deal with failures, as captured in the TCP calculus [35]. Beyond the introduction of exceptions, failure detection primitives or timers, one should consider (as is done in the π_{1l}-calculus and the Distributed Join calculus in the case of fail-stop failures) introducing failure-specific domain abstractions to capture ideas of failure zones and error confinement zones. One should also consider higher-level abstractions involving e.g. notions of atomicity (see requirement 7.2), recovery or replication [18].

7.2 Supporting atomic activities

Description A global computing programming model should provide support for notions of atomic activities.

Rationale Atomic transactions [22, 28] have emerged as a key abstraction for building resilient distributed systems. One should expect a global computing programming model to provide support for such abstractions, either in the form of in-built constructs (see e.g. the ATF calculus [11]) or derived ones (see e.g. the construction of transactions in the Join calculus given in [7]).

Discussion This requirement is strongly related to the requirement on supporting distributed coordination 6.2. Indeed orchestrating and coordinating services in a dynamic distributed environment requires appropriate abstractions for atomic and recoverable pieces of activities. This is reflected e.g. in Microsoft's Biztalk language, formalized in [7].

At the same time, models of (extended) transactions abound, which extend or relax the classical ACID properties of transactions with various atomicity properties, dependencies between transactions, different forms of visibility and conflicts between transactions [10]. To what extent a global computing programming model can be made

to capture or support the vast range of potentially useful transaction models, remains to be seen.

8 Resources

8.1 Supporting resource management

Description A global computing programming model should allow fine-grained resource management.

Rationale Usage of resources in an open environment needs to be carefully controlled and managed, for instance to avoid starvation, to ensure fairness, to allow load-balancing, or to counter denial-of-service attacks. A global computing programming model should provide basic mechanisms to control access to distributed resources and to enable proper resource accounting. Controlling accesss to resources and accounting for the use of resources should be possible at different levels of granularity, to account for the wide range of resources present in a distributed computing environment (from low-level system resources such as memory, processor or energy, to high-level resources such as an e-commerce service).

Discussion Several recent works, including within the MIKADO prokect (e.g. [20, 33]) begin to adress the issue of controlling resource access and accounting for resource usage in a distributed setting. One needs to go beyond these encouraging results, however, if only to deal with the fact that resources may take the form of arbitrary components or processes in a system, and can be managed at different levels of granularity. Furthermore, in accordance with the requirement 3.2 “No predefined policy”, constructs allowing the control and management of resources should avoid determining particular policies and should instead provide for the possibility of dynamically changing ones.

9 Meta-information

9.1 Supporting run-time meta-information

Description A global computing programming model should provide the possibility for executing components to access dynamic meta-information pertaining to characteristics of other components.

Rationale Due to the highly dynamic nature of a global computing environment, it is necessary for components in global computations to obtain information characterizing their environment at a given instant in time. Such information can take many different forms, including for instance interface types of executing components, access rights and security roles, resource capacities and associated thresholds, code and intermediate representations of executing components, security certificates, etc; and it may

also vary dynamically. Meta-information of this kind can be either accessed through repositories or directly attached to components (e.g. as with digital signatures, or proof carrying code). A global computing programming model should therefore account for the presence, at run-time, of such meta-information associated with components in global computations.

Discussion This requirement resonates with requirement 5.2 on the support of component lifecycles. It can also be understood as a general requirement for dynamic typing and dynamic type checking, allowing global computing components to access types born by other components at run-time, typically to gain information on the capabilities of these components and to check that they verify certain properties. A typical example of the use of such information is in the obvious realm of security (see e.g. [21] for an interesting example in MIKADO), with secure component loaders and sandboxes requiring access to component meta-information to enforce their policies.

References

[1] *Proceedings of the First International Workshop on Reflective Middleware (RM 2000) New York, USA*, 2000.

[2] *Proceedings of Coordination 2002, LNCS 2315, F. Arbab and C. Talcott (Eds.), Springer*, 2002.

[3] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology, vol. 6, no 3*, 1997.

[4] L. Andrade, J. Fiadeiro, J. Gouveia, G. Koutsoukos, and M. Wermelinger. Coordination for orchestration. In *Proc. of Coordination 2002, LNCS 2315, F. Arbab and C. Talcott (Eds.), Springer*, 2002.

[5] G. Blair, G. Coulson, A. Andersen, L. Blair, M. Clarke, F. Costa, H. Duran-Limon, T. Fitzpatrick, L. Johnston, R. Moreira, N. Parlavantzas, and K. Saikoski. The Design and Implementation of OpenORB v2. *IEEE Distributed Systems Online, vol. 2 no 6, Special Issue on Reflective Middleware*, 2001.

[6] B. Bloom, A. Cheng, and A. Dsouza. Using a Protean Language to Enhance Expressiveness in Specification. *IEEE Transactions on Software Engineering, Vol. 23, No. 4*, 1997.

[7] R. Bruni, C. Laneve, and U. Montanari. Orchestrating transactions in Join calculus. In *Proceedings of Concur'02, LNCS 2421, Springer*, 2002.

[8] L. Cardelli. Wide area computation. In *Proc. Automata, Languagese and Programming, 26th International Colloquium, (ICALP'99), J. Wiedermann, P. van Emde Boas, M. Nielsen (eds), Lecture Notes in Computer Science, Vol. 1644*. Springer, 1999.

[9] L. Cardelli and A. Gordon. Mobile ambients. In *Foundations of Software Science and Computational Structures, M. Nivat (Ed.), Lecture Notes in Computer Science, Vol. 1378*. Springer Verlag, 1998.

[10] P. Chrysanthis and K. Ramamritham. Synthesis of extended transaction models using ACTA. *ACM Transactions on Database Systems, Vol.19, No 8*, 1994.

[11] D. Duggan. Atomic failure in wide-area computation. In *Proceedings 4th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS), S. Smith and C. Talcott eds*. Kluwer, 2000.

[12] B. Dumant, F. Dang Tran, F. Horn, and J.B. Stefani. Jonathan: an open distributed platform in Java. *Distributed Systems Engineering Journal, vol.6*, 1999.

[13] D.R. Engler, M.F. Kaashoek, and J.W. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 1995 ACM Symposium on Operating Systems Principles*, pages 251–266, Copper Mountain Resort, CO, USA, December 1995.

[14] C. Fournet. *The Join-Calculus*. PhD thesis, Ecole Polytechnique, Palaiseau, France, 1998.

[15] C. Fournet, J.J. Levy, and A. Schmitt. An asynchronous distributed implementation of mobile ambients. In *Proceedings of the International IFIP Conference TCS 2000, Sendai, Japan, Lecture Notes in Computer Science 1872*. Springer, 2000.

[16] N. Francez and I. Forman. *Interacting Processes : A multiparty approach to coordinated distributed programming*. Addison-Wesley, 1996.

[17] S. Frolund. *Coordinating Distributed Objects – An Actor-Based Approach to Synchronization*. MIT Press, 1996.

[18] S. Frolund and R. Guerraoui. X-Ability: a theory of replication. *Distributed Computing, Vol. 14, No 4*, 2001.

[19] D. Gelernter and N. Carriero. Coordination languages and their significance. *Communications of the ACM, vol. 35, no 2*, 1992.

[20] J.C. Godskensen, Thomas Hildebrandt, and Vladimiro Sassone. A calculus of mobile resources. In *Proceedings of Concur'02, LNCS 2421, Springer*, 2002.

[21] D. Gorla and R. Pugliese. Resource access control and dynamic privileges acquisition. Technical report, Dipartimento di Sistemi e Informatica, Univ. Firenze, 2002.

[22] J. Gray and A. Reuter. *Transaction processing: concepts and techniques*. Morgan Kaufmann, 1993.

[23] R. Guerraoui and M. Fayad. Oo distributed programming is not distributed oo programming. *CACM 42(4)*, 1999.

[24] O. Jensen and R. Milner. Bigraphs and Transitions. In *Proceedings 30th Annual ACM Symposium on Principles of Programming Languages (POPL)*, 2003.

[25] S. Katz. A superimposition control construct for distributed systems. *ACM Trans. on Programming Languages and Systems, vol.15, no 2*, 1993.

[26] M. Lumpe, F. Achermann, and O. Nierstrasz. *A Formal Language for Composition*, chapter 4. Cambridge University Press, 2000.

[27] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[28] N. Lynch, M. Merrit, W. Weihl, and A. Fekete. *Atomic transactions*. Morgan Kaufmann, 1994.

[29] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings 5th European Software Engineering Conference, Lecture Notes in Computer Science 989, Springer-Verlag*, 1995.

[30] P. Van Roy. On the separation of concerns in distributed programming: Application to distribution structure and fault tolerance in Mozart. In *Proceedings International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications (PDSIA 99)*, Tohoku University, Sendai, Japan, 1999. World Scientific.

[31] D. Sangiorgi and S. Walker. *The π-calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.

[32] A. Schmitt and J.B. Stefani. The M-calculus: A Higher-Order Distributed Process Calculus. In *Proceedings 30th Annual ACM Symposium on Principles of Programming Languages (POPL)*, 2003.

[33] D. Teller, P. Zimmer, and D. Hirschkoff. Using Ambients to control resources. In *Proceedings of Concur'02, LNCS 2421, Springer*, 2002.

[34] A. Unyapoth and P. Sewell . 2001. Nomadic Pict: Correct Communication Infrastructures for Mobile Computation. In *Proceedings ACM Int. Conf. on Principles of Programming Languages (POPL)*, 2001.

[35] K. Wansbrough, M. Norrish, P. Sewell, and A. Serjantov. Timing udp: Mechanized semantics for sockets, threads, and failures. In *Proceedings ESOP 2002, Lecture Notes in Computer Science 2305, Springer*, 2002.

[36] P. Wojciechowski and P. Sewell. Nomadic Pict: Language and Infrastructure. *IEEE Concurrency, vol. 8, no 2*, 2000.