Modelling global computations with KLAIM

BY ROCCO DE NICOLA* AND MICHELE LORETI

Dipartimento di Sistemi e Informatica, Università di Firenze,
Viale Morgagni 65, 50134 Firenze, Italy

A new area of research, known as Global Computing, is by now well established. It aims at defining new models of computation based on code and data mobility over wide-area networks with highly dynamic topologies, and at providing infrastructures to support coordination and control of components originating from different, possibly untrusted, fault-prone, malicious or selfish sources. In this paper, we present our contribution to the field of Global Computing that is centred on Kernel Language for Agents Interaction and Mobility (KLAIM). KLAIM is an experimental language specifically designed to programme distributed systems consisting of several mobile components that interact through multiple distributed tuple spaces. We present some of the key notions of the language and discuss how its formal semantics can be exploited to reason about qualitative and quantitative aspects of the specified systems.

Keywords: process calculi; mobile code; distributed applications; network awareness; program verification

1. Introduction

Computing technology has become ubiquitous, ranging from global applications to minuscule embedded devices. Computing systems are being challenged by the distributed way in which they operate and the dynamically changing context in which they are situated. The mobility of users, devices and computing processes, the heterogeneity of platforms and the unpredictable patterns of use pose new challenges for the design of these systems. Typically, such systems consist of many nodes linked according to a dynamic topology, and have to maximize efficiency and guarantee performance in the face of changing conditions. Computers do not perform their tasks in isolation, but have to operate and interact in a complex evolving environment that may not be fully specified, consisting of physical devices, software programs, services, virtual agents, etc. These systems that are sometimes referred to as Global Computers, rely on massive networked and dynamically reconfigurable infrastructures interconnecting heterogeneous components, typically autonomous and mobile, that can operate on the basis of incomplete information. Examples of such systems are the Internet, virtual private networks, the World Wide Web, telephone networks and GRID computers.

* Author for correspondence (denicola@dsi.unifi.it).

One contribution of 19 to a Discussion Meeting Issue ‘From computers to ubiquitous computing, by 2020’.
Important requirements for Global Computing applications are as follows.

— *Scalability*. High numbers of users and nodes have to be taken into consideration.
— *Heterogeneity*. Different operating systems and applications have to interoperate.
— *Autonomy*. Independent administration of domains has to be guaranteed.
— *Adaptability*. Dynamic and unpredictable changes have to be taken into account.
— *Mobility*. Migration of processes, code and data has to be considered.

Global Computing research, started roughly 10 years ago, is built on 40 years of research on *concurrent programming*, on theories, models and algorithms for *distributed systems*, and on *networking*. Research on Global Computing has pursued the following main objectives:

— defining new *models of computation* over wide-area networks with highly dynamic topologies;
— providing *linguistic primitives* to program systems at the appropriate level of abstraction and to support coordination of components originating from different, possibly untrusted, fault-prone, malicious or selfish sources; and
— building *tools for qualitative and quantitative evaluation* of systems in order to guarantee their correct behaviour and an appropriate quality of service.

We have witnessed the birth of many calculi and kernel languages intended to support the programming of Global Computing applications and to provide tools for formal reasoning over the modelled systems. We do not have room to provide an overview of all the work that has been done and of the many results that have been achieved. We limit ourselves to mention some closely related work; in particular, we would like to mention the work based on the process algebraic approaches, which arose from calculus of communicating systems (Milner 1989), communicating sequential processes (Hoare 1984) and \( \pi \)-calculus (Milner et al. 1993), and some of the work exploiting the notion of (multiple) tuple spaces, introduced in LINDA (Gelernter 1985), to model the interaction of independent components.

Among the process calculi, we would like to mention the distributed join-calculus (Fournet et al. 1996), the distributed \( \pi \)-calculus (Riely & Hennessy 1998), the ambient calculus (Cardelli & Gordon 2000), the SEAL calculus (Castagna & Vitek 1999) and NOMADIC PICT (Unyapoth & Sewell 2001). Some of these calculi also consider the issue of security, namely *privacy* and *integrity* of data, hosts and agents, and deal with the problem of controlling accesses to resources by distributed and autonomous components. Often, they resort to using suitable notions of types for imposing the appropriate security policy (Bugliesi et al. 2001; Castagna et al. 2001; Hennessy & Riely 2002).

The LINDA-based communication model (Gelernter 1985) has been used for defining a number of libraries and frameworks, developed both in academia and industry, that provide functionalities for programming distributed and mobile systems. JADA (Ciancarini & Rossi 1997) is a coordination toolkit, where coordination and communication among distributed objects are achieved via shared *ObjectSpaces* that implements tuple spaces. MARS (Cabri et al. 1998) is a coordination tool for mobile agents, which defines tuple spaces that may be programmed to react when accessed by agents. JINI (Arnold et al. 1999b) is

*Phil. Trans. R. Soc. A* (2008)
a connection technology that enables many devices to be plugged together to form a community on a network in a scalable way. JINI is developed on the top of JAVASPACES (Arnold et al. 1999a), a tuple-based framework with event notification, blocking operations with timeouts and with the notion of leasing, i.e. granting of resources for a limited period of time. IBM TSPACES (Ford et al. 1998) is a network middleware package that enriches tuple space-based network communication with database capabilities. LIME (Picco et al. 1999) exploits the multiple tuple spaces paradigm (Gelernter 1989) to coordinate mobile agents and adds mobility to tuple spaces; moreover, it allows processes to use private tuple spaces and to share them temporarily.

In the following, we shall present our specific approach that builds on both the experiences outlined above. Kernel Language for Agents Interaction and Mobility (KLAIM; De Nicola et al. 1998) is a formalism specifically designed to describe distributed systems made up of several mobile interacting components. We shall first introduce our calculus, and then consider a simple example that illustrates the potentials of our approach.

2. The KLAIM approach

The KLAIM communication paradigm builds on, and extends, LINDA’s notion of generative communication through a single shared tuple space (Gelernter 1985). A tuple space is a multiset of tuples, which are sequences of information items. Tuples are anonymous and can be picked up from tuple spaces by processes by means of a pattern-matching mechanism (associative selection). Inter-process communication is asynchronous: producer (i.e. sender) and consumer (i.e. receiver) of a tuple do not need to synchronize.

The LINDA communication model was originally proposed for parallel programming on isolated machines. Multiple, possibly distributed, tuple spaces have been advocated later (e.g. Gelernter 1989) to improve modularity, scalability and performance. The resulting communication model has a number of properties that make it appealing for Global Computing. The model permits time uncoupling (data lifetime is independent of the lifetime of the producer process), destination uncoupling (the producer of a datum does not need to know the future use or the final destination of that datum) and space uncoupling (communicating processes need to know a single interface, i.e. the operations over the tuple space).

The KLAIM programming paradigm emphasizes a clear separation between the computational level and the net coordinator/administrator level. Intuitively, programmers design computational units (processes and mobile agents), while coordinators design nets, define the initial distribution of processes and set the security policies for controlling access to resources. Coordinators have complete control over changes of configuration of the network that may be due to addition/deletion of software components and sites, or to transmission of explicit references to nodes. Processes can be placed at different localities of a net, where data can be stored and retrieved.

Thus, in KLAIM the network infrastructure is clearly distinguishable from user processes and explicitly modelled; localities are ‘first-class citizens’ that can be dynamically created and communicated. We argue that this feature permits
a more accurate handling of Global Computing applications. Indeed, structuring these applications in terms of located processes and coordinators provides a clean and powerful abstraction and is instrumental to define security policies and their enforcement mechanisms.

KLAIM has evolved in different directions to deal with most of the issues of Global Computing mentioned in §1.

— A type system for KLAIM has been introduced for controlling process activities, namely their use to resources and their migrations over the net (De Nicola et al. 2000).

— A full-fledged programming language, X-KLAIM, extending KLAIM with a high level syntax for processes, has been defined (Bettini & De Nicola 2005). It has been implemented (Bettini et al. 2002b) in JAVA and has proved to be suitable for programming a wide range of distributed applications with agents and code mobility (Bettini et al. 2002a).

— A number of simpler calculi have been introduced to understand the key nature of Global Computing and to define behavioural equivalences for formally establishing properties of systems (De Nicola et al. 2006a, 2007a,b).

— KLAIM has also been extended with primitives for explicitly modelling connectivity between network nodes and to allow or deny remote operations. Connections are programmable, in that they can be explicitly and dynamically activated and de-activated by processes, and remote operations can take place only if the involved nodes are directly connected (Bettini et al. 2002c; De Nicola et al. 2007a).

— Temporal logics for KLAIM have been defined for specifying and verifying dynamic properties of networks, e.g. resource allocation, access to resources and information disclosure (De Nicola & Loreti 2004, 2008).

— A Markovian extension of KLAIM (STOKLAIM) has been proposed together with a Stochastic Logic that permits addressing issues also related to uncertainty and probabilistic behaviour (De Nicola et al. 2007c).

For the sake of simplicity, we will here describe a simplified version of KLAIM that has been called µKLAIM (e.g. Bettini et al. 2003). The main difference between KLAIM and µKLAIM is that the former allows high-level communication (tuple fields can contain processes) while the latter only permits evaluating processes remotely. Moreover, the simpler language does not make use of the original abstraction mechanism based on allocation environments. These environments are used in KLAIM to map logical localities into physical ones, and permit instantiating at later stages names into actual addresses.

A µKLAIM system, called a net, is a set of nodes, each of which is identified by a locality. Localities can be seen as the addresses of network nodes. Every node has a computational component (a set of processes running in parallel) and a data component (a tuple space). Processes interact with each other either locally or remotely by posting and retrieving tuples to and from a tuple space.

A tuple is a sequence of actual fields. Each actual field can be a locality l, a value v or a variable x. Tuples are retrieved from tuple spaces via pattern matching using templates (T). Templates are sequences of actual and formal fields. The latter are variables that will get a value when a tuple is retrieved. Formal fields are signalled by a ‘!’ before the variable name.
The meaning of the pattern-matching function match is straightforward: a template matches against a tuple if both have the same number of fields and the corresponding fields do match; two values (localities) match only if they are identical, while formal fields match any value of the same type. A successful matching returns a substitution function \( \sigma \) associating the variables contained in the formal fields of the template with the values contained in the corresponding actual fields of the accessed tuple. For instance, if \( T = (l, 4) \) and \( t = (l, 4) \) then \( \text{match}(T, t) = [l/u] \).

Term \( \emptyset \) denotes the empty net, i.e. the net that does not contain any node. Terms \( l :: P \) (located process) and \( l :: \langle t \rangle \) (located tuple) are used to describe basic \( \mu \text{KLAIM} \) nodes: the former states that process \( P \) is running at location \( l \) while the latter states that the tuple space located at \( l \) contains tuple \( \langle t \rangle \). \( \mu \text{KLAIM} \) nets are obtained by parallel composition (\( || \)) of located processes and tuples.

The following term:

\[
\langle \text{act} \rangle :: P_1 || \langle l_1 :: P_1 \rangle || \langle l_2 :: (Q_1 \mid Q_2) \rangle || \langle l_2 :: \langle t_1 \rangle \rangle || \langle l_2 :: \langle t_2 \rangle \rangle
\]

denotes a net consisting of two nodes, named \( l_1 \) and \( l_2 \), where processes \( P_1 \) and \( P_2 \) are running at \( l_1 \) while \( Q_1 \) and \( Q_2 \) are running at \( l_2 \). The tuple space located at \( l_2 \) contains tuples \( \langle t_1 \rangle \) and \( \langle t_2 \rangle \) while that located at \( l_1 \) is empty.

The syntax of \( \mu \text{KLAIM} \) processes is similar to that of other process algebras: \text{nil} stands for the process that cannot perform any actions, \( P_1 \parallel P_2 \) denotes parallel composition of \( P_1 \) and \( P_2 \), \( P_1 + P_2 \) denotes the non-deterministic choice between \( P_1 \) and \( P_2 \) and \text{act}.\( P \) stands for the process that executes action \text{act} and then behaves like \( P \). Possible actions are \text{out}(t)@l, \text{eval}(P)@l, \text{in}(T)@l, \text{read}(T)@l \) and \text{newloc}(u).

Action \text{out}(t)@l \) adds tuple \( t \) to the tuple space located at \( l \). Action \text{eval}(P)@l \) spawns process \( P \) at locality \( l \). Action \text{in}(T)@l \) is used to retrieve tuples from tuple spaces. Different from \text{out}, action \text{in} \) is a blocking operation: the computation is blocked until a tuple matching template \( T \) is found in the tuple space located at \( l \). When such a tuple \( t \) is found, it is removed from the tuple space and the continuation process is closed with substitution \( \sigma = \text{match}(T, t) \) that replaces the formal fields in \( T \) with the corresponding values in \( t \). For this reason, \text{in}(T)@l.\( P \) acts as a binder for variables in the formal fields of \( T \). Action \text{read} \) is similar to \text{in} but the matched tuple is not removed from the tuple space. Finally, action \text{newloc}(u) \) permits creating a new node that is identified by a fresh locality \( l \) that will be used in place of \( u \) in the continuation process. Freshness of names is enforced by operator \( v_l \)—used for guaranteeing that \( l \) is a private locality of nets or processes.

### 3. A Global Computing scenario in \( \mu \text{KLAIM} \)

To give a flavour of how to use \( \text{KLAIM} \) to specify systems and verify their properties, we will consider a typical scenario of Global Computing. This is a system where mobile agents migrate over the nodes of a network for executing specific tasks such as, for instance, CPU-intensive computations. Execution of tasks may require resources that are not available locally. For this reason, processes migrate over the net to find the appropriate resources.
It is assumed that each node of the network is equipped with a set of computational environments that can host tasks for execution. This set of nodes can change dynamically: new computational environments can be added to nodes while existing ones can become unavailable. Since agents’ tasks can be computationally intensive, the system might want to distribute the work between available environments in order to get optimal resource use.

A simple algorithm that guarantees load balancing is the following: when a new task has to be executed, a request is sent to all the available computational environments at a given node; the first to reply to the request is then assigned the task. If no computational environment reacts, the agent tries with other nodes. The underlying assumption is that overloaded computational environments will be slower in replying than underused ones.

This simple scenario permits considering four of the distinguished requirements on applications for Global Computers described in §1. The number of agents and nodes involved in the scenario does not influence the overall system architecture (scalability). Each agent is programmed without knowing the specific environment where it will be executed (autonomy). Node selection takes into account the changes occurring relative to the computational power of a node (adaptability). Processes migrate over the network in order to find the appropriate computational resources (mobility).

The system outlined above can be modelled in µKlaim as a net consisting of a number of nodes, \( n_0, n_1, \ldots \), hosting mobile agents.

When an agent \( \text{Proc}(n_i) \) running at \( n_i \) needs executing a task, it stores a message in the local tuple space containing string ‘\text{REQ}’ and a nonce representing the task id (a private name \( l \)). After issuing the request, \( \text{Proc}(n_i) \) waits until one of the computational environments deposits a tuple containing message ‘\text{OK}’, name \( l \) and a locality \( u \), that will indicate where the task can be executed. Finally, \( \text{Proc}(n_i) \) spawns task \( \text{Task}(l, u) \) at \( u \) for execution. Pending execution requests can be revoked in case they are not promptly serviced. In that case, the agent has to select an alternative node reference among those that are stored at \( n_i \). A request is revoked by withdrawing tuple ‘\text{REQ}', \( l \) while the selection of the alternative for \( n_i \) is performed by \( \text{read}(!\text{nxt}@n_i) \).

The behaviour of \( \text{Proc}(n_i) \) is formally modelled in µKlaim as

\[
\text{Proc}(n_i) \overset{\text{def}}{=} \text{vl.out}(\text{\textquote{REQ}}, l)@n_i.
\]

\[
\text{in}(\text{\textquote{OK}}, l, !u)@n_i, \text{eval}(\text{Task}(l, u))@u
\]

\[
+ \text{in}(\text{\textquote{REQ}}, l)@n_i, \text{read}(!\text{nxt}@n_i, \text{eval}(\text{Proc}(\text{nxt}))@\text{nxt.nil})
\]

Since µKlaim does not provide an explicit mechanism for handling time (which is instead available in X-Klaim), time outs are modelled by means of a non-deterministic choice between waiting or considering the exceptional situation. In our case, process \( \text{Proc}(n_i) \) has thus the possibility of revoking the pending request until it is not serviced.

It is assumed that before terminating, \( \text{Task}(l, u) \) notifies termination to the hosting execution environment by putting tuple ‘\text{FINISH}', \( l \) in the tuple space located at \( u \). Hence,

\[
\text{Task}(l, u) \overset{\text{def}}{=} \ldots \text{out}(\text{\textquote{FINISH}}, l)@u.\text{nil}
\]
Computational environments are modelled as locations $s_i$ executing as many processes $\text{Slot}(n_j, s_i)$ as the number of tasks that can be concurrently executed at $s_i$ from agents hosted at $n_j$. The behaviour of these processes can be defined as

$$\text{Slot}(n_j, s_i) \overset{\text{def}}{=} \text{in}(\text{‘REQ’}, l)@n_j, \text{out}(\text{‘OK’}, l, s_i)@n_j, \text{in}(\text{‘FINISH’}, l)@s_i$$

These processes wait for a request at locality $n_j$ and after receiving the request send tuple $\text{‘OK’}, l, s_i$ to $n_j$ and wait for a termination signal from locally executed tasks.

Note that new computation environments can be associated to a node $n_i$ affecting neither the agent implementation nor the network structure. Indeed, to associate a computational environment $s_i$ to a node $n_j$ it is sufficient to execute $\text{Slot}(n_j, s_i)$ at $s_i$. Furthermore, a computational environment $s_i$ is not available anymore for node $n_j$ when all processes $\text{Slot}(n_j, s_i)$ running at $s_i$ are shut down. The system is able to handle these failures, and guarantees that unavailable environments are not selected for executing tasks.

The following $\mu$KLAIM term describes a relevant part of the modelled system:

$$\cdots \parallel n_1 :: \text{nil} \parallel s_1 :: (\text{Slot}(n_1, s_1) \mid \text{Slot}(n_1, s_1) \mid \text{Slot}(n_1, s_1)) \parallel n_1 :: (n_3) \parallel n_1 :: (n_2)$$

$$n_2 :: \text{Proc}_i \parallel n_2 :: (n_1) \parallel s_2 :: (\text{Slot}(n_2, s_2) \mid \text{Slot}(n_2, s_2)) \parallel s_3 :: \text{Slot}(n_2, s_3) \parallel \cdots$$

It contains two nodes, named $n_1$ and $n_2$, each of which can execute three tasks concurrently. However, while $n_1$ provides a single computational environment ($s_1$), $n_2$ provides two computational environments ($s_2$ and $s_3$). Moreover, $n_1$ is the alternative for $n_2$ while $n_2$ and $n_3$ are the alternatives for $n_1$. In the considered term, process $\text{Proc}_i$ is running at $n_2$ while no agent is running at $n_1$.

In order to deal with performance and dependability issues that are of utmost importance for ‘network-aware’ computing, we have extended $\mu$KLAIM by adding distribution rates to its actions (De Nicola et al. 2006b). In the proposed extension, referred to as $\text{StoKLAIM}$, actions are assumed to have a random duration governed by a negative exponential distribution. By using the stochastic semantics of

\[\text{Phil. Trans. R. Soc. A} \ (2008)\]
StoKlaim, a continuous time Markov chain corresponding to the given specification can be generated. Moreover, by using the logic and the model checking technique presented in De Nicola et al. (2007c), quantitative properties can be checked and established.

Relative to the computational scenario described above, figure 1a shows how the steady-state probability of a task waiting for execution is influenced by the task execution rate. Figure 1b, instead, describes how the probability that a node gets overloaded changes with an increase in the task execution rates. A node is overloaded if there is a task that is waiting for the execution, while no associated computational environment is able to serve the request.

We are greatly indebted to Lorenzo Bettini, Gianluigi Ferrari and Rosario Pugliese with whom Klaim was originally conceived, designed and implemented. We also wish to thank the friends who have worked with us on Klaim and its extensions: Viviana Bono, Daniele Gorla, Joost-Pieter Katoen, Diego Latella, Mieke Massink, Eugenio Moggi, Emilio Tuosto and Betti Venneri. Our project has also benefited from the research of many other colleagues all around the world who have worked on similar topics and/or commented on our contribution. Unfortunately, the space available here is not sufficient to mention all of them and to include proper references to their work in the references section.

This work has been partially supported by EU FET—GC II Project SENSORIA IST-2005-16004.

References


*Phil. Trans. R. Soc. A* (2008)