X-Klaim Is Back

Lorenzo Bettini[1(✉)], Emanuela Merelli[2], and Francesco Tiezzi[2]

[1] Dipartimento di Statistica, Informatica, Applicazioni, Università di Firenze, Firenze, Italy
lorenzo.bettini@unifi.it

[2] School of Science and Technology, Computer Science Division, Università di Camerino, Camerino, Italy
{emanuela.merelli,francesco.tiezzi}@unicam.it

Abstract. Klaim is a coordination language specifically designed to model and program distributed systems consisting of mobile components interacting through multiple distributed tuple spaces. The Klaim's theoretical foundations provided a solid ground for the implementation of the Klaim's programming model. To practically program Klaim-based applications, the X-Klaim programming language has been proposed. It extends Klaim with enriched primitives and standard control flow constructs, and is compiled in Java to be executed. However, due to the limits of X-Klaim in terms of usability and the aging of the technology at the basis of its compiler, X-Klaim has been progressively neglected. Motivated by the success that Klaim has gained, the popularity that still has in teaching distributed computing, and its possible future exploitations in the development of modern ICT systems, in this paper we propose a renewed and enhanced version of X-Klaim. The new implementation, coming together with an Eclipse-based IDE tooling, relies on recent powerful frameworks for the development of programming languages.

Keywords: Network-aware programming · Coordination language · Klaim · X-Klaim · Eclipse IDE

1 Introduction

In the mid-90s Rocco De Nicola, to whom this LNCS volume is dedicated, came up with the idea of combining the work on process algebras, to which he had turned his research interest so far, with Linda's notion of asynchronous generative communication. Linda is a coordination paradigm providing a set of primitives for decoupling communicating processes both in space and time [41]. Communication is achieved via a shared data repository, called *tuple space*, where processes *insert*, *read* and *withdraw* tuples (i.e., sequences of data items). The data retrieving mechanism uses *pattern-matching* to find the required data in the tuple space.

The first attempt is PAL (Process Algebra based on Linda, [35]), a process algebra obtained by embedding the Linda primitives for interprocess communication in a CSP-like process description language. Then, this language was

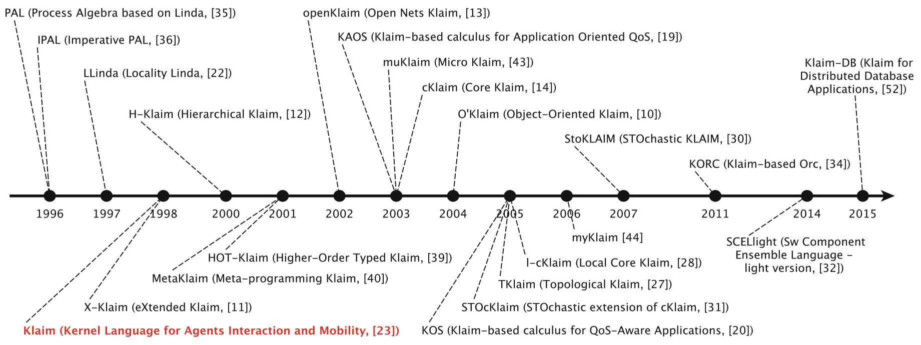

PAL (Process Algebra based on Linda, [35])
IPAL (Imperative PAL, [36])
LLinda (Locality Linda, [22])
H-Klaim (Hierarchical Klaim, [12])
openKlaim (Open Nets Klaim, [13])
KAOS (Klaim-based calculus for Application Oriented QoS, [19])
muKlaim (Micro Klaim, [43])
cKlaim (Core Klaim, [14])
O'Klaim (Object-Oriented Klaim, [10])
StoKLAIM (STOchastic KLAIM, [30])
KORC (Klaim-based Orc, [34])
Klaim-DB (Klaim for Distributed Database Applications, [52])
1996
1997
1998
2000
2001
2002
2003
2004
2005
2006
2007
2011
2014
2015
myKlaim [44]
SCELlight (Sw Component Ensemble Language - light version, [32])
l-cKlaim (Local Core Klaim, [28])
HOT-Klaim (Higher-Order Typed Klaim, [39])
MetaKlaim (Meta-programming Klaim, [40])
TKlaim (Topological Klaim, [27])
X-Klaim (eXtended Klaim, [11])
STOcKlaim (STOchastic extension of cKlaim, [31])
Klaim (Kernel Language for Agents Interaction and Mobility, [23])
KOS (Klaim-based calculus for QoS-Aware Applications, [20])

Fig. 1. The KLAIM family.

extended with *localities* (i.e., network addresses) as first-class citizens, which can be dynamically created and communicated. This capability is essential for achieving the so-called *network-aware programming*, where processes of a distributed application can explicitly refer and control the spatial structure of the network where they are currently deployed. The resulting formalism, LLINDA (Locality Linda, [22]), considers multiple tuple spaces that are distributed over a collection of network nodes, and uses localities to distribute/retrieve data over/from these nodes. As the code of processes is itself data, higher-order communication is enabled in order to support the definition of applications with *mobile* components. Syntax and semantics of LLINDA were later revised and cleaned up, thus obtaining the coordination language KLAIM (Kernel Language for Agents Interaction and Mobility, [23]). It allows one to design distributed systems consisting of stationary and mobile components interacting through multiple distributed tuple spaces.

Since then, a lot of effort has been made on KLAIM. On the one hand, several variants of KLAIM have been proposed to face the new challenges posed by the continuously evolving scenario of network-based technology. We show in Fig. 1 a timeline reporting the significant results on this research line. On the other hand, the theoretical foundations of KLAIM enabled the definition of several verification techniques (e.g., type systems [21,24–26,29,42,43], behavioral equivalences [27], flow logic [37], model checking [30,33,38]), as well as they provided a solid ground for the implementation of the KLAIM's programming model. As a further evidence of the success and influence that KLAIM has gained, we report here the number of citations that the seminal paper [23] has received at the time of writing: in Scopus it is cited by 362 documents, in Web of Science by 225, and in Google Scholar by 666.

In this paper, we focus on KLAIM's implementation. In order to program applications according to the KLAIM's paradigm, the toolchain depicted in Fig. 2 was initially developed. Since KLAIM was originally conceived as a formalism rather than as a full-fledged programming language, it had been extended with high-level process constructs to make the programming task more friendly.

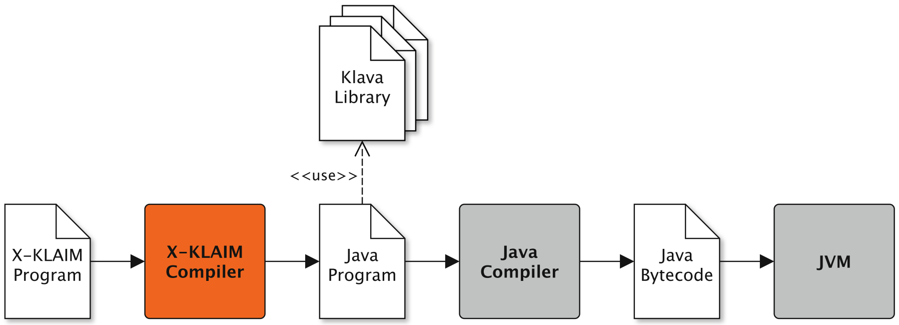


Fig. 2. X-KLAIM toolchain.

The resulting programming language, called X-KLAIM (eXtended Klaim, [11]), provides variable declarations, enriched communication primitives, assignments, conditionals, sequential and iterative process composition. The X-KLAIM compiler translates X-KLAIM programs into Java programs that exploit the Java package KLAVA (KLAIM in Java, [8]), which provides the runtime environment for X-KLAIM operations. The produced Java code can be then compiled and executed in the standard way.

KLAVA has evolved over the years and is still a maintained framework used for directly programming in Java according to the KLAIM paradigm. Instead, the X-KLAIM compiler has been progressively neglected, due to the aging of its underlying technologies, the lack of an IDE supporting the programming and debugging activity, and the limitations of X-KLAIM on exchangeable data and supported expressions. These deficiencies undermined the usability of the language and, hence, its usage by the coordination community. To fill this gap, in this paper we propose a renewed and enhanced version of X-KLAIM, by relying on powerful modern frameworks for the development of domain-specific programming languages. This is not only motivated by the success of KLAIM, as shown above, but also by the fact that KLAIM is still a popular language for teaching distributed computing in academia[1]. Moreover, we also envisage possible exploitations of the renewed X-KLAIM as coordination language for developing modern ICT systems, in such domains as IoT, Smart Cities, e-Health, etc.

The new version of X-KLAIM is available as an open source project. Sources and links to Eclipse update site and to complete Eclipse distributions are available from: https://github.com/LorenzoBettini/xklaim.

The rest of the paper is organized as follows. Section 2 provides an informal overview of KLAIM, and introduces a simple running example concerning a leader election algorithm. Section 3 describes the renewed version of X-KLAIM we propose, together with details on the implementation and the related Eclipse-based

[1] KLAIM has been and is still taught on courses about coordination and distributed computing at, e.g., Università di Firenze, Università di Camerino, Università di Pisa, IMT Scuola Alti Studi Lucca, and Danmarks Tekniske Universitet.

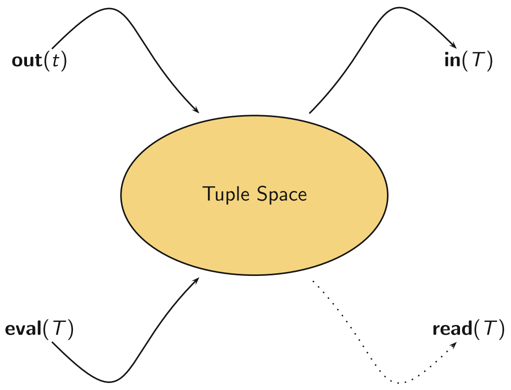


Fig. 3. Tuple space and Linda primitives.

IDE tooling. Finally, Sect. 4 concludes the paper by touching upon directions for future works.

2 KLAIM

In this section, we summarize the key features of KLAIM. It is a formal language specially devised to design distributed applications consisting of several (possibly mobile) components deployed over the nodes of network infrastructure. Although KLAIM is based on process algebras, it makes use of Linda-like asynchronous communication and supports distributed data management via multiple shared tuple spaces. A *tuple space* is a multiset of tuples, the latter consisting of sequences of data items. Processes interact by inserting, reading and withdrawing tuples to/from tuple spaces. The tuple retrieving mechanism relies on *pattern-matching* to find the required data in the tuple space. KLAIM enriches Linda primitives (see Fig. 3) with information about the network *localities* where processes and tuples are allocated. Localities can be explicitly referred and exchanged, thus supporting *network-aware programming*.

KLAIM syntax is shown in Table 1. We use the following disjoint sets: the set of *physical localities* (ranged over by l), the set of *logical localities* (ranged over by u), the set of *locality variables* (ranged over by r), the set of *value variables* (ranged over by x), the set of *process variables* (ranged over by X), and the set of *process identifiers* (ranged over by A). We also use a set of *expressions* (ranged over by e), whose exact syntax is omitted; we assume that expressions contain, at least, values (ranged over by V) and value variables. We shall use ℓ to denote a locality, either physical or logical, or a locality variable.

Nets N are finite collections of nodes where processes and data can be located (see Fig. 4). Nets are formed by composing nodes by means of the parallel operator $N_1 \parallel N_2$.

A computational node $l::_\rho P$ is characterized by its physical locality l, a running process P and an *allocation environment* ρ. The latter acts as a name solver

Table 1. Klaim syntax.

(Nets)

$N \ ::= \ l ::_{\rho} P \quad | \quad l :: \langle et \rangle \quad | \quad N_1 \parallel N_2$

(Processes)

$P \ ::= \ \mathbf{nil} \quad | \quad a.P \quad | \quad P_1 \mid P_2 \quad | \quad X \quad | \quad A(\bar{p})$

(Actions)

$a \ ::= \ \mathbf{out}(t)@\ell \quad | \quad \mathbf{in}(T)@\ell \quad | \quad \mathbf{read}(T)@\ell \quad | \quad \mathbf{eval}(P)@\ell \quad | \quad \mathbf{newloc}(r)$

(Tuples)

$t \ ::= \ e \quad | \quad \ell \quad | \quad P \quad | \quad t_1, t_2$

(Evaluated tuples)

$et \ ::= \ V \quad | \quad l \quad | \quad P \quad | \quad et_1, et_2$

(Templates)

$T \ ::= e \quad | \quad \ell \quad | \quad P \quad | \quad !x \quad | \quad !r \quad | \quad !X \quad | \quad T_1, T_2$

binding logical localities, occurring in the processes hosted in the corresponding node, into specific physical localities. The distinguished logical locality **self** is used by processes to refer to the physical locality of their current hosting node. The term l::$\langle et \rangle$ indicates that the evaluated tuple et is located to the physical locality l. The *tuple space* for a given locality consists of all the evaluated tuples located there.

Processes P are the active computational units of Klaim. They can be executed concurrently, either at the same physical locality or at different localities. Processes are built up from the empty process **nil** (which does nothing), basic actions a, process variables X, and process calls $A(\bar{p})$, by means of the action prefixing operator $a.P$ and the parallel composition $P_1 \mid P_2$. Recursive behaviors are modeled via process definitions; it is assumed that each process identifier A has a single defining equation $A(\bar{f}) \triangleq P$, where $\bar{f}$ and $\bar{p}$ denote lists of formal and actual parameters, respectively. Hereafter, we do not explicitly represent process definitions (and their migration to make migrating processes complete), and assume that they are available at any node of a net. Process variables support *higher-order* communication, namely the capability to exchange (the code of) a process and possibly execute it. It is realized by first adding a tuple containing the process to a tuple space and then retrieving/withdrawing this tuple while binding the process to a process variable.

During their execution, processes perform some basic *actions* (see Fig. 5). Action $\mathbf{out}(t)@\ell$ adds the tuple resulting from the evaluation of t to the tuple

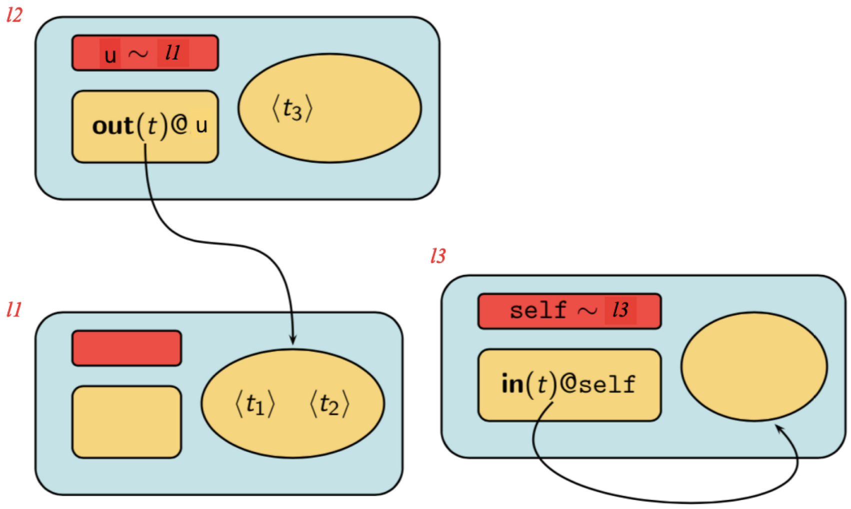


Fig. 4. The KLAIM net.

space of the target node identified by ℓ. A tuple is a sequence of the actual field, i.e., expressions, localities, locality variables, or processes. The evaluation of a tuple amounts to computing the values of its expressions. Action $\mathbf{in}(T)@\ell$ (resp. $\mathbf{read}(T)@\ell$) permits to withdraw (resp. read) tuples from the tuple space hosted at the (possibly remote) locality ℓ. If matching tuples are found, one is non-deterministically chosen, otherwise, the process is blocked. These retrieval actions exploit templates as patterns to select tuples in a tuple space. *Templates* are sequences of actual and formal fields, where the latter are written $!\,x$, $!\,r$ or $!\,X$ and are used to bind variables to values, physical localities, or processes, respectively. Templates must be evaluated before they can be used for retrieving tuples; their evaluation is like that of tuples, where formal fields are left unchanged by the evaluation. Intuitively, an evaluated template matches against an evaluated tuple if both have the same number of fields and 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 associating the variables contained in the formal fields of the template with the values contained in the corresponding actual fields of the accessed tuple; such substitution is applied to the continuation process of the executed action. Action $\mathbf{eval}(P)@\ell$ sends the process P for execution to the (possibly remote) node identified by ℓ. Finally, action $\mathbf{newloc}(r)$ creates a new network node with physical locality bound to the locality variable r. Differently, from all the other actions, this latter action is not indexed with a target locality because it always acts locally.

We conclude the section with a simple example (inspired by those in [16]) aiming at showing KLAIM at work on the specification of a leader election algorithm.

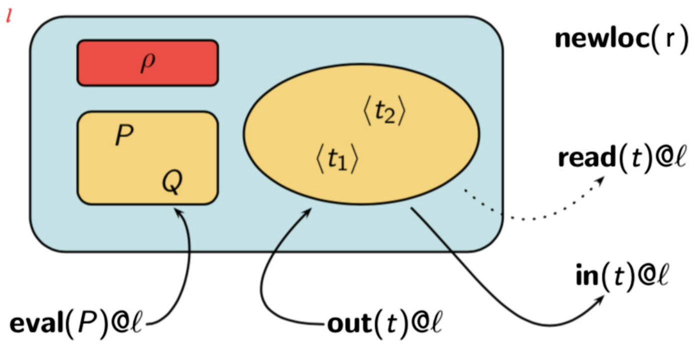


Fig. 5. The Klaim node.

Example 1 (*Running example in* Klaim)**.** We consider a system where n participants distributed on the nodes of a network have to elect a leader. The system is rendered in Klaim as the following net:

$$l_0 ::_{[\mathbf{self} \mapsto l_0, u_{next} \mapsto l_1]} P \parallel l_1 ::_{[\mathbf{self} \mapsto l_1, u_{next} \mapsto l_2]} P \parallel \ldots \parallel l_{n-1} ::_{[\mathbf{self} \mapsto l_{n-1}, u_{next} \mapsto l_0]} P$$
$$\parallel$$
$$l_{rg} ::_{[\mathbf{self} \mapsto l_{rg}]} \langle \text{"}ID\text{"}, 0\rangle \parallel l_{rg} ::_{[\mathbf{self} \mapsto l_{rg}]} \langle \text{"}ID\text{"}, 1\rangle \parallel \ldots \parallel l_{rg} ::_{[\mathbf{self} \mapsto l_{rg}]} \langle \text{"}ID\text{"}, n-1\rangle$$

The topology of the network is a ring, which is a common assumption for leader election algorithms. Thus, the allocation environment of node l_i, in addition to the standard mapping **self** $\mapsto l_i$, maps the logical locality u_{next} to the physical locality $l_{i+1 \ mod \ n}$ of the next node in the ring. The node identified by l_{rg} acts as a random generator: it provides different identifiers, retrieved by the participants at the outset. In this way, each participant will be uniquely identified by an identifier selected randomly. The leader will be the participant with the smallest identifier.

The process P deployed in each participant node is defined as follows:

$$\begin{array}{l} \mathbf{in}(\text{"}ID\text{"}, !\, x_{id})@l_{rg}. \\ \mathbf{out}(\text{"}ID\text{"}, x_{id})@\mathbf{self}. \\ \mathbf{eval}(A_{checker}(x_{id}))@u_{next}.\mathbf{nil} \end{array}$$

Once a participant has retrieved an identifier, it spawns a mobile *checker* process to the next node. This process will travel along the ring to determine if the source node has to be the leader.

The checker process is defined as follows:

$$\begin{array}{rl} A_{checker}(myId) \triangleq & \mathbf{read}(\text{"}ID\text{"}, !\, x)@\mathbf{self}. \\ & \mathbf{if}\ myId < x\ \mathbf{then} \\ & \quad \mathbf{eval}(A_{checker}(myId))@u_{next}.\mathbf{nil} \\ & \mathbf{else\ if}\ myId > x\ \mathbf{then} \\ & \qquad \mathbf{eval}(A_{notifier}(myId))@u_{next}.\mathbf{nil} \\ & \qquad \mathbf{else} \\ & \qquad \mathbf{out}(\text{"}LEADER\text{"})@\mathbf{self}.\mathbf{nil} \end{array}$$

The process carries the identifier of the source node (parameter *myId*) and compares it with the identifier of the node where it is running (retrieved via a **read** action and stored in the variable x). If the source identifier is smaller than the current one, the currently hosting node is not the leader: the process moves to the next node and restarts. Instead, if the source identifier is greater than the current one, the source node is not the leader: the process activates the *notifier* process that crosses the rest of the ring to come back to the source node and insert this information in the local tuple space. If the two identifiers are identical, the process is back on the source node (thus no node with a smaller identifier has been found in the ring) and inserts the information that this is the leader in the local tuple space.

The notifier process is defined as follows:

$$\begin{aligned} A_{\mathit{notifier}}(\mathit{myId}) \triangleq\ & \mathbf{read}(\text{“}\mathit{ID}\text{”}, !\,x)@\mathbf{self}.\\ & \mathbf{if}\ x = \mathit{myId}\ \mathbf{then}\\ & \quad \mathbf{out}(\text{“}\mathit{FOLLOWER}\text{”})@\mathbf{self}.\mathbf{nil}\\ & \mathbf{else}\\ & \quad \mathbf{eval}(A_{\mathit{notifier}}(\mathit{myId}))@u_{\mathit{next}}.\mathbf{nil} \end{aligned}$$

It simply looks for the node with identifier *myId*; when it finds this node, it inserts in the local tuple space the information that this node is a follower.

Notably, for the sake of simplicity, we resort in this example of the conditional construct **if** e_{bcond} **then** P **else** Q. This is a macro that can be expressed here by exploiting pattern-matching and parallel composition as follows:

$$\begin{aligned} & \mathbf{out}(\text{“}\mathit{ITE}\text{”}, e_{bcond})@\mathbf{self}.\\ & (\mathbf{in}(\text{“}\mathit{ITE}\text{”}, \mathit{true})@\mathbf{self}.P\\ & \ \mid\ \mathbf{in}(\text{“}\mathit{ITE}\text{”}, \mathit{false})@\mathbf{self}.Q) \end{aligned}$$

3 X-Klaim 2.0

In this section we present the new version of X-Klaim. In particular, we first briefly recap the limitations of the old implementation of X-Klaim. Then, we illustrate the main features of the new version of X-Klaim by showing the implementation of the leader election example, which has been presented in Klaim in Example 1, Sect. 2. Finally, we present a few interesting additional features of the new version of X-Klaim, including its debugging mechanism integrated in the Eclipse IDE.

3.1 The Old Implementation

As mentioned in the Introduction, X-Klaim programs are compiled into Java programs that make use of the Java library Klava, which provides the runtime environment for X-Klaim operations. Klava is a Java library with some classes and methods to develop Java programs according to the Klaim programming model.

Klava is also meant to be used directly for programming in Java according to the Klaim primitives and mechanisms. It allows the programmer to fully exploit Java mechanisms and the libraries of its huge ecosystem, while using the Klaim programming model. While using Klava in Java, the programmer can benefit from IDE tooling, such as content assist, code navigation and debugging. However, this also implies that the programmer will have to deal with the verbosity of Java, which also makes it hard to directly use Klaim primitives. For example, using Klaim tuple space operations with Klava requires some additional Java instructions to set up the tuple (in particular, its formal fields if any) and to update possible variables representing formal fields with the values retrieved from pattern-matching. Klava strives for making Java programmers life easy but it can only do that by obeying the rules of Java. Originally, X-Klaim was designed to give the programmers a language as close as possible to the Klaim programming model, while still providing typical programming features such as variable declarations, control structures, etc. Thus, with X-Klaim, the programmer could easily write Klaim tuple space operations without additional boilerplate code. However, X-Klaim programs could not rely on the Java ecosystem and making X-Klaim program and Java program communicate with each other required too much programming effort. Moreover, no IDE mechanisms for the X-Klaim compiler were implemented, forcing the X-Klaim programmer to write an X-Klaim program with a text editor, without any assistance from any IDE, explicitly call the X-Klaim command line compiler waiting for possible compilation errors, and finally manually compile the generated Java code. Finally, debugging X-Klaim programs was not possible: the programmer had to debug the generated Java code, and debugging automatically generated code is known to be quite hard. Summarizing, the benefits of the X-Klaim programming language were evident only for very small prototype programs.

On the other hand, Klava kept on evolving during the years. For example, starting from our experience in implementing Klava network and code mobility mechanisms, we proposed a general framework for implementing Java network applications with code mobility, called IMC (Implementing Mobile Code) [7]. Then, we refactored Klava completely, implementing it in terms of IMC. Due to the limitations of X-Klaim, though, we decided it was not worthwhile to port its compiler to the new version of Klava. This decision was also due to the limitations of the compilation technologies at that time and to the programming effort required to implement IDE mechanisms for the compiler, e.g., on top of Eclipse.

3.2 The New Implementation

Compiler and IDE technologies have evolved since then. In particular, the framework Xtext quickly gained popularity. Xtext [9] is an Eclipse framework for the development of programming languages and domain-specific languages (DSLs). Starting from a grammar definition, Xtext generates a parser, an abstract syntax tree, and a complete IDE support based on Eclipse (e.g., editor with syntax highlighting, code completion, error reporting and incremental

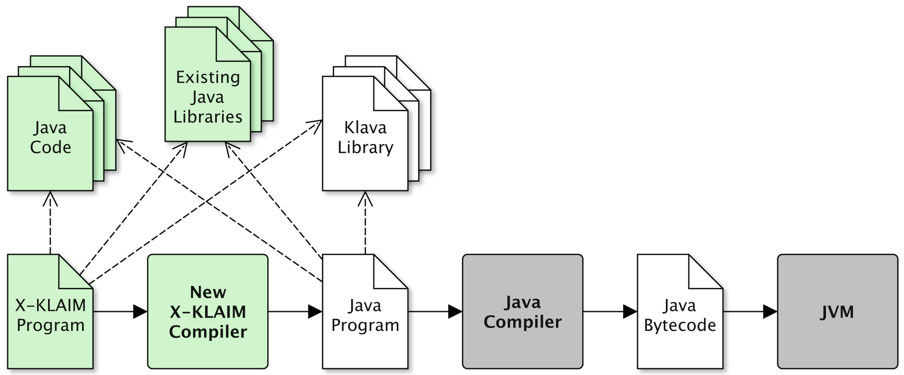


Fig. 6. X-Klaim 2.0 toolchain (green color highlights the new elements, dashed arrows represent 'use' relationships and solid arrows represent 'input-output' relationships). (Color figure online)

building). Xtext comes with good defaults for all the above mechanisms and the language developer can easily customize all such mechanisms.

Thus, we decided to re-implement X-Klaim, targeting the new version of Klava. Since we implemented this new version of X-Klaim from scratch, we also took the chance to make its syntax similar to mainstream languages, in particular, we gave it a Java-like shape. This means that programs written in the previous version of X-Klaim are not compliant with this new version (however, we do not think that it is a considerable problem). Concerning the integration into Eclipse, we used Xtext with all its powerful and useful mechanisms, mentioned above, which help the programmer. Furthermore, we also rely on another mechanism provided by Xtext, that is, Xbase. Xbase is an extensible and reusable expression language, which provides a Java-like syntax and which is meant to be embedded in your own Xtext DSL. By using Xbase in X-Klaim, besides inheriting Xbase rich Java-like syntax, we also inherit its interoperability with Java and its type system. This means that an X-Klaim program can seamlessly access any Java type available in the classpath of the project. This allows us to get rid of one of the worst drawbacks mentioned above of the previous implementation: Java and X-Klaim programs can now interoperate automatically, and X-Klaim programs can reuse the whole Java ecosystem. This also implies that one can write a Java application where some parts are written directly in Java using Klava, and other parts are written in X-Klaim (using the parts written in Java).

The syntax of Xbase is similar to Java, but it removes much "syntactic noise" from Java (for example, terminating semicolons are optional, as well as other syntax elements like parenthesis when invoking a method without arguments). Xbase should be easily understood by Java programmers. Moreover, Xbase comes with a powerful type inference mechanism, compliant with the Java type

```
proc InitialProc(String nodeName) {
   val rg = getPhysical(logloc("rg"))
   val next = logloc("next")
   in("ID", var Integer xid)@rg
   out("ID", xid)@self
   eval(new CheckerProc(xid))@next
   in(var String result)@self
   println(nodeName + ": result is " + result)
}
```

Fig. 7. The process P of Example 1 deployed in each participant node implemented in X-Klaim.

```
proc CheckerProc(Integer myId) {
   val next = logloc("next")
   read("ID", var Integer x)@self
   if (myId < x) {
      eval(new CheckerProc(myId))@next
   } else if (myId > x) {
      eval(new NotifierProc(myId))@next
   } else {
      out("LEADER")@self
   }
}
```

```
proc NotifierProc(Integer myId) {
   val next = logloc("next")
   read("ID", var Integer x)@self
   if (x == myId) {
      out("FOLLOWER")@self
   } else {
      eval(new NotifierProc(myId))@next
   }
}
```

Fig. 8. The processes $A_{checker}$ and $A_{notifier}$ of Example 1 implemented in X-Klaim.

system, that allows the programmer to avoid specifying types in declarations when they can be inferred from the context.

The X-Klaim compiler implemented with Xtext/Xbase is now completely integrated into Eclipse. Thus, IDE mechanisms like content assist and code navigation are available in the X-Klaim editor. Moreover, the compiler is now integrated in the automatic building mechanism of Eclipse: saving an X-Klaim file automatically triggers the Java code generation, which in turns triggers the generation of Java byte-code. This avoids the manual compilation tasks of the previous implementation. Finally, it is now possible to debug an X-Klaim program while the generated Java code is executed (as shown in Sect. 3.5). We show the renewed X-Klaim toolchain in Fig. 6.

3.3 The Leader Election Example in X-Klaim

We will now describe the main features of the new version of X-Klaim by showing the implementation of the leader election example, which has been presented in Klaim in Example 1, Sect. 2.

First of all, the process P deployed in each participant node is defined in X-Klaim as shown in Fig. 7. Note that the types such as `String` and `Integer` are actually Java types, since, as mentioned above, X-Klaim programs can refer

```
net LeaderElectionNet {
  node L1 [next −> L2] {
    eval(new InitialProc("L1"))@self
  }
  node L2 [next −> L3] {
    eval(new InitialProc("L2"))@self
  }
  node L3 [next −> L1] {
    eval(new InitialProc("L3"))@self
  }
  node RG logical "rg" {
    out("ID", 0)@self
    out("ID", 1)@self
    out("ID", 2)@self
  }
}
```

Fig. 9. The net of Example 1 implemented in X-Klaim.

directly to Java types. Expressions and statements in X-Klaim are based on the Xbase syntax. Variable declarations in Xbase start with `val` or `var`, for final and non-final variables, respectively. The type of the variable can be omitted if it can be inferred from the initialization expression. Xbase syntax has been extended with Klaim operations. Formal fields in a tuple are specified as variable declarations, since, just like in Klaim, formal fields implicitly declare variables that are available in the code after `in` and `read` operations. Boolean non-blocking versions of `in` and `read` are also available: `in_nb` and `read_nb`, respectively. `logloc` (and `phyloc`, not shown in the example) are syntactic sugar for creating instances of localities. Finally, `getPhysical` and `println` are Java methods available in the runtime library of X-Klaim, which, of course, includes Klava. Notably, since X-Klaim aims at being a programming language, localities, which in Klaim can be used without explicit declarations, must be explicitly declared and initialized in X-Klaim. The only exception is `self`, which is a predefined locality also in X-Klaim. Since we are in a Java-like context, process invocation corresponds to the creation of an instance of the process (using the `new` operator); we did not use the same "invocation" syntax of Klaim since that would conflict with the standard Java-like syntax for method invocation.

The processes $A_{checker}$ and $A_{notifier}$ of Example 1 are defined in X-Klaim as shown in Fig. 8.

Finally, the net of Example 1 is defined in X-Klaim as shown in Fig. 9 (here we fix the number of the nodes to 3). Note that the mapping for `self` is implicit in every node, so it does not have to be defined. Explicit locality mappings (corresponding to Klaim allocation environments, Sect. 2) are specified for each node with the syntax `[ l1 -> l2 ]`. For example, the node `L1` maps `next` to `L2`. A node can specify the logical locality with which it will be known in the containing net, with the `logical` clause, as in the node `RG`. If this clause is not specified, a node is automatically known to the net with a logical locality corresponding to its name (like `L1`, `L2` and `L3`).

3.4 Additional Features

The syntax of net and nodes shown above allows the programmer to quickly specify a "flat" net, where all nodes are at the same level. However, X-Klaim implements the hierarchical version of the Klaim model as presented in [12,13]. This implies that, if a node is not able to resolve a logical locality into a physical locality then it delegates it to the "parent" node, that is, to the containing `net`. However, the programmer can also define a `node` outside a `net` element and explicitly use the operations of `login` and `accept` (or the versions dealing explicitly with logical localities, `subscribe` and `register`). This will allow X-Klaim programs to define a custom hierarchical net. For example, this is an X-Klaim program defining a node accepting remote connections from other nodes, which will then be part of its network (note how physical localities are expressed in terms of the standard TCP syntax `host:port`):

```
node Receiver physical "localhost:9999" {
  while (true) {
    val remote = new PhysicalLocality
    accept(remote)
  }
}
```

and this is a possible client node connecting to this network and evaluating a process remotely:

```
node Sender [server −> phyloc("localhost:9999")] {
  login(server)
  val myLoc = getPhysical(self)
  eval({
    println(String.format("Hello %s...", server))
    println("...from a process coming from " + myLoc)
    out("DONE")@myLoc
  })@server
  in("DONE")@self
  logout(server)
  System.exit(0)
}
```

The above example also shows how X-Klaim code can access Java code, like the static methods `String.format` and `System.exit`. Moreover, it also shows how X-Klaim allows the programmer to specify anonymous processes, e.g., for remote evaluation with `eval` (just like Klaim):

```
in(var String s)@self
eval( in(s)@self )@l
```

In the code snippet above, the process `in(s)@self` will be evaluated at the remote locality `l`. Note that when a process migrates, it is closed with respect to the variables of the original enclosing scope, like the `s` in the example. Anonymous processes with several statements must be enclosed in a code block `{...}`, like in the previous `Sender` example. In order to insert an anonymous process, with code `p_code`, into a tuple space, the syntax `proc { p_code }` must be used. This is required to disambiguate with a code block that would be evaluated to produce a value to be part of the tuple:

```
in(var String s)@self
out( proc { in(s)@self } )@l
```

A process can be retrieved from a tuple space with a formal field of type `KlavaProcess` (defined in the Klava library), e.g.,

```
in(var KlavaProcess X)@self
eval(X)@self
```

The above X-Klaim code corresponds to the Klaim process

$$\mathbf{in}(!X)@\mathbf{self}.\,\mathbf{eval}(X)@\mathbf{self}.\mathbf{nil}$$

Code mobility is completely delegated to Klava and IMC, which automatically collect the Java classes of the migrating process so that they can be loaded at the remote destination (as described in details in [5]). However, in this new version of X-Klaim, *strong mobility* is not supported yet. We will implement this feature in the compiler according to the transformation described in [6].

As a further improvement, the new version of X-Klaim allows the programmer to fully exploit the recursive nature of processes in a way that was not possible in the previous version nor in Klaim itself. In fact, a process can refer to itself with the Java keyword `this`, which has the same semantics as in Java. It allows a process to spawn itself to a remote site. This can also be used in anonymous processes. This mechanism allows one to write complex (possibly anonymous) recursive processes. For example, the process in Fig. 10 implements the leader election example without additional process definitions, showing how `this` correctly refers to the current anonymous process, even in the presence of nesting. This also shows how X-Klaim automatically deals with the closure of the enclosing scope (e.g., the `myId` and `next` used by the anonymous migrating processes).

3.5 Debugging X-Klaim Programs

As already anticipated, thanks to Xtext/Xbase, the new version of X-Klaim, and in particular its integration in Eclipse, allows the programmer to debug an X-Klaim program, as shown in Fig. 11. In this example, based on the X-Klaim code of Fig. 10, we set a breakpoint in the X-Klaim program, and during the execution, we can see the current values of variables, either in the "Variables" Eclipse view or by hovering over a variable in the program (like `myId`).

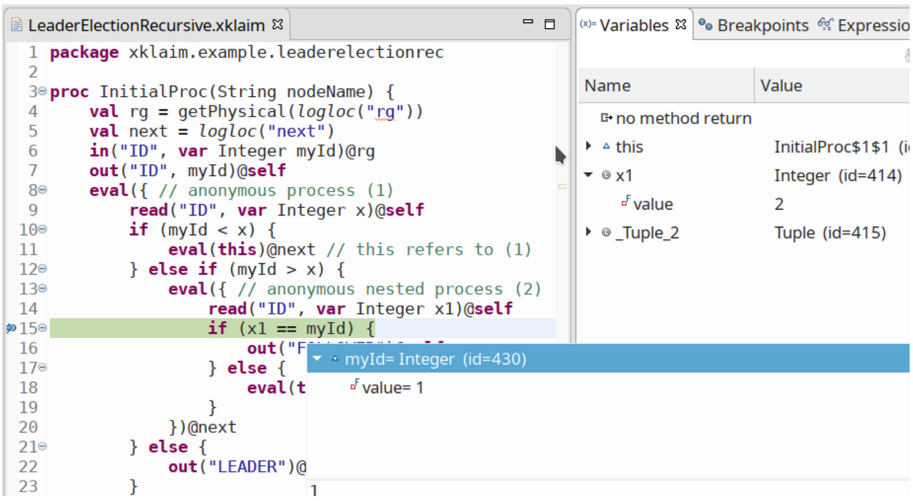
```
proc InitialProc(String nodeName) {
  val rg = getPhysical(logloc("rg"))
  val next = logloc("next")
  in("ID", var Integer myId)@rg
  out("ID", myId)@self
  eval({ // anonymous process (1)
    read("ID", var Integer x)@self
    if (myId < x) {
      eval(this)@next // this refers to (1)
    } else if (myId > x) {
      eval({ // anonymous nested process (2)
        read("ID", var Integer x1)@self
        if (x1 == myId) {
          out("FOLLOWER")@self
        } else {
          eval(this)@next // this refers to (2)
        }
      })@next
    } else {
      out("LEADER")@self
    }
  })@next
  in(var String result)@self
  println(nodeName + ": result is " + result)
}
```

Fig. 10. The recursive process in X-Klaim with migrating operations, implementing altogether the processes of Figs. 7 and 8.

We believe that being able to debug an X-Klaim program directly is a crucial feature when programming distributed applications accessing remote tuple spaces and dealing with code mobility. The debugging mechanisms of X-Klaim are as powerful as the standard Java debugging mechanism of Eclipse. For example, during an X-Klaim debugging session, we can evaluate expressions on the fly. For example, as shown in Fig. 12, we can retrieve the current physical locality where the debugged process is executing, by calling the Klava method `getPhysical`.

Of course, the current debugging mechanism allows the developer to debug only a local running process. Currently, it is not possible to debug a process that runs on a remote node. In order to achieve also this mechanism, a dedicated debugging protocol should be implemented in the Klava runtime library. It will be interesting to investigate this feature as a future work.

Note that the X-Klaim Eclipse support also includes the ability to directly run or debug an X-Klaim file, with dedicated context menus: there's no need to run the generated Java code manually.



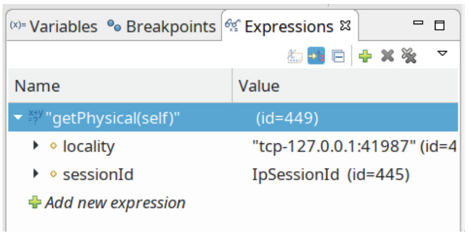
```
package xklaim.example.leaderelectionrec

proc InitialProc(String nodeName) {
    val rg = getPhysical(logloc("rg"))
    val next = logloc("next")
    in("ID", var Integer myId)@rg
    out("ID", myId)@self
    eval({ // anonymous process (1)
        read("ID", var Integer x)@self
        if (myId < x) {
            eval(this)@next // this refers to (1)
        } else if (myId > x) {
            eval({ // anonymous nested process (2)
                read("ID", var Integer x1)@self
                if (x1 == myId) {
                    out("F
                } else {
                    eval(t
                }
            })@next
        } else {
            out("LEADER")@
        }
```

Name	Value
no method return	
this	InitialProc$1$1 (i
x1	Integer (id=414)
value	2
_Tuple_2	Tuple (id=415)

```
myId= Integer (id=430)
    value= 1
1
```

Fig. 11. Debugging an X-Klaim program.



Name	Value
"getPhysical(self)"	(id=449)
locality	"tcp-127.0.0.1:41987" (id=4
sessionId	IpSessionId (id=445)
Add new expression	

Fig. 12. Evaluating expressions while debugging an X-Klaim program.

4 Concluding Remarks

Motivated by the success that the Klaim language gained over the last years, and believing that still nowadays it can provide further contributions to the coordination research field and application area, we have brought X-Klaim back to life. In doing that, by resorting to modern compiler and IDE technologies, we have enhanced X-Klaim making it a usable and effective coordination language.

The fundamental novelties of this renewed version of X-Klaim are:

– Java-like syntax, which should be easily understood by programmers;
– full interoperability with Java, so that X-Klaim code can access the whole Java ecosystem;
– type inference mechanism, allowing programmers to avoid specifying types that can be inferred from the context;
– IDE support and debugging facilities;
– recursive definition of processes.

The support of these features clarifies how the contribution of this work represents a significant advancement with respect to the previous version of X-Klaim introduced in [11]. Comparisons with other implementations of Linda-like languages and code mobility frameworks are provided in [8,11], such as Jada [17], MARS [15], Jini [3], JavaSpaces [48], IBM T Spaces [53], IBM Aglets [46], μCODE [49], Lime [50], Sumatra [2]. More recently, other implementations of the Linda paradigm have been proposed. GigaSpaces [1] is a commercial implementation of tuple spaces mainly used for big data analytics. Differently, from X-Klaim, GigaSpaces supports database-like features, such as complex queries, transactions, and replication. This could be obtained in X-Klaim by using its interoperability mechanisms to access Java code. Tupleware [4] is a framework providing a scalable (both distributed and centralized) tuple space. It is based on distributed hash tables, similar to other distributed implementations of tuple space like Blossom [51] and DTuples [45]. The focus of these frameworks is on the performance of the search in the distributed tuple space, rather than on the programming facilities to support the development of tuple-space-based applications. Differently from X-Klaim, they do not consider code mobility features. Instead, LuaTS [47] provides a reactive event-driven tuple space system that also supports code mobility. While X-Klaim is based on the mainstream Java technology, LuaTS relies on Lua. Finally, we refer to [18] for a recent survey of coordination tools, including both those based on Linda and the ones relying on different coordination models.

As future work, we plan to assess the effectiveness of X-Klaim in programming distributed, possibly mobile, applications in different domains, such as IoT and Bioinformatics. To this aim, we will use X-Klaim to implement different case studies from academia and industry. We also intend to appropriately validate the usability of the language, by involving students of BSc and MSc in Computer Science, as well as developers from different industrial settings.

Acknowledgements. The authors would like to take the opportunity to thank Rocco in this Festschrift contribution.

Lorenzo Bettini: Rocco was one of my professors at the University and the supervisor of my Master Thesis and of my PhD. He's the main person who introduced me to the charming world of research. He drove me, helped me and supported me for most of my academic career. He taught me so many things that I could hardly list them here. X-Klaim was the main subject of my Master and PhD thesis and the starting point of my research cooperation with Rocco, thus I'm very happy to take this opportunity to "remove the dust" from X-Klaim and to revive it in a modern shape. For all these reasons, it is an extreme pleasure (and honor) to take part to this Festschrift contribution.

Emanuela Merelli: While I was doing my Master's thesis at CNR in Pisa, in Via Santa Maria, Rocco De Nicola worked in a nearby building. From there I heard about his fame as a brilliant researcher. At that time, I never imagined that I could be adopted by one of his doctoral students. When Flavio arrived in Camerino he changed my scientific life and the evolution of the local Computer Science group. He generously introduced me to Rocco and to the whole scientific family of Italian researchers working on formal methods. I am very grateful to Rocco for having understood what it meant

for me to be alone in Camerino coping with the difficulties that arose at that time. I am very grateful to him also for having supported my scientific growth, my career and having listened to my odd scientific thoughts, even if they were often not so rigorous as the community usually expects. He always encouraged me to go ahead without limits to creativity. Many important events happened since our first meeting, some beautiful and some painful. I thank Rocco for having contributed to the growth of part of the scientific family also in this side of Italy that sometimes quakes. I'm very proud to be part of Rocco's scientific family!

Francesco Tiezzi: I first met Rocco when I was a student at the University of Florence, where he introduced me to the realm of formal methods. He then gave me the opportunity of working with his research group, at Florence during my PhD and at IMT Lucca later. During these years, his guidance was fundamental for my professional and personal growth. Apart from what he taught me of a technical nature, for which I will always be grateful, the most important lesson that I have learnt from Rocco his the attitude he always had in his job. Everyday he shows the curiosity and the interest of a young researcher for the continuously evolving scenarios of computer science. What I appreciate most about Rocco is that he does with levity a 'serious' job like that of an academician. All this allows Rocco to be able to have a direct dialogue with everybody, from a student to a Turing-awarded researcher or a minister. For all of this, and for having supported me in all occasions, I would sincerely say "Grazie Rocco!".

References

1. GigaSpaces XAP v14.0 Documentation. https://docs.gigaspaces.com/xap/14.0/
2. Acharya, A., Ranganathan, M., Saltz, J.H.: Sumatra: a language for resource-aware mobile programs. In: Vitek, J., Tschudin, C. (eds.) MOS 1996. LNCS, vol. 1222, pp. 111–130. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-62852-5_10
3. Arnold, K., Scheifler, R., Waldo, J., O'Sullivan, B., Wollrath, A.: Jini Specification. Addison-Wesley, Boston (1999)
4. Atkinson, A.: Tupleware: A Distributed Tuple Space for the Development and Execution of Array-Based Applications in a Cluster Computing Environment. University of Tasmania, School of Computing and Information Systems thesis (2010)
5. Bettini, L.: A Java package for transparent code mobility. In: Guelfi, N., Reggio, G., Romanovsky, A. (eds.) FIDJI 2004. LNCS, vol. 3409, pp. 112–122. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31869-9_11
6. Bettini, L., De Nicola, R.: Translating strong mobility into weak mobility. In: Picco, G.P. (ed.) MA 2001. LNCS, vol. 2240, pp. 182–197. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45647-3_13
7. Bettini, L., De Nicola, R., Falassi, D., Lacoste, M., Loreti, M.: A flexible and modular framework for implementing infrastructures for global computing. In: Kutvonen, L., Alonistioti, N. (eds.) DAIS 2005. LNCS, vol. 3543, pp. 181–193. Springer, Heidelberg (2005). https://doi.org/10.1007/11498094_17
8. Bettini, L., De Nicola, R., Pugliese, R.: Klava: a Java package for distributed and mobile applications. Softw. Pract. Exp. **32**(14), 1365–1394 (2002)
9. Bettini, L.: Implementing Domain-Specific Languages with Xtext and Xtend, 2nd edn. Packt Publishing, Birmingham (2016)
10. Bettini, L., Bono, V., Venneri, B.: O'Klaim: a coordination language with mobile mixins. In: De Nicola, R., Ferrari, G.-L., Meredith, G. (eds.) COORDINATION 2004. LNCS, vol. 2949, pp. 20–37. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24634-3_5

11. Bettini, L., De Nicola, R., Pugliese, R., Ferrari, G.L.: Interactive mobile agents in X-Klaim. In: WETICE, pp. 110–117. IEEE Computer Society (1998)
12. Bettini, L., Loreti, M., Pugliese, R.: Structured nets in KLAIM. In: SAC, pp. 174–180. ACM (2000)
13. Bettini, L., Loreti, M., Pugliese, R.: An infrastructure language for open nets. In: SAC, pp. 373–377. ACM (2002)
14. Bettini, L., et al.: The klaim project: theory and practice. In: Priami, C. (ed.) GC 2003. LNCS, vol. 2874, pp. 88–150. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-40042-4_4
15. Cabri, G., Leonardi, L., Zambonelli, F.: Reactive tuple spaces for mobile agent coordination. In: Rothermel, K., Hohl, F. (eds.) MA 1998. LNCS, vol. 1477, pp. 237–248. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0057663
16. Calzolai, F., Loreti, M.: Simulation and analysis of distributed systems in Klaim. In: Clarke, D., Agha, G. (eds.) COORDINATION 2010. LNCS, vol. 6116, pp. 122–136. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13414-2_9
17. Ciancarini, P., Rossi, D.: Jada: coordination and communication for Java agents. In: Vitek, J., Tschudin, C. (eds.) MOS 1996. LNCS, vol. 1222, pp. 213–226. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-62852-5_16
18. Ciatto, G., Mariani, S., Louvel, M., Omicini, A., Zambonelli, F.: Twenty years of coordination technologies: state-of-the-art and perspectives. In: Di Marzo Serugendo, G., Loreti, M. (eds.) COORDINATION 2018. LNCS, vol. 10852, pp. 51–80. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-92408-3_3
19. De Nicola, R., Ferrari, G., Montanari, U., Pugliese, R., Tuosto, E.: A formal basis for reasoning on programmable QoS. In: Dershowitz, N. (ed.) Verification: Theory and Practice. LNCS, vol. 2772, pp. 436–479. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-39910-0_21
20. De Nicola, R., Ferrari, G., Montanari, U., Pugliese, R., Tuosto, E.: A process calculus for QoS-aware applications. In: Jacquet, J.-M., Picco, G.P. (eds.) COORDINATION 2005. LNCS, vol. 3454, pp. 33–48. Springer, Heidelberg (2005). https://doi.org/10.1007/11417019_3
21. De Nicola, R., Ferrari, G.L., Pugliese, R.: Coordinating mobile agents via blackboards and access rights. In: Garlan, D., Le Métayer, D. (eds.) COORDINATION 1997. LNCS, vol. 1282, pp. 220–237. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-63383-9_83
22. De Nicola, R., Ferrari, G.L., Pugliese, R.: Locality based Linda: programming with explicit localities. In: Bidoit, M., Dauchet, M. (eds.) CAAP 1997. LNCS, vol. 1214, pp. 712–726. Springer, Heidelberg (1997). https://doi.org/10.1007/BFb0030636
23. De Nicola, R., Ferrari, G.L., Pugliese, R.: KLAIM: a kernel language for agents interaction and mobility. IEEE Trans. Softw. Eng. **24**(5), 315–330 (1998)
24. De Nicola, R., Ferrari, G.L., Pugliese, R.: Types as specifications of access policies. In: Vitek, J., Jensen, C.D. (eds.) Secure Internet Programming. LNCS, vol. 1603, pp. 117–146. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48749-2_6
25. De Nicola, R., Ferrari, G.L., Pugliese, R.: Programming access control: the Klaim experience. In: Palamidessi, C. (ed.) CONCUR 2000. LNCS, vol. 1877, pp. 48–65. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44618-4_5
26. De Nicola, R., Ferrari, G.L., Pugliese, R., Venneri, B.: Types for access control. Theor. Comput. Sci. **240**(1), 215–254 (2000)

27. De Nicola, R., Gorla, D., Pugliese, R.: Basic observables for a calculus for global computing. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 1226–1238. Springer, Heidelberg (2005). https://doi.org/10.1007/11523468_99
28. De Nicola, R., Gorla, D., Pugliese, R.: On the expressive power of klaim-based calculi. Electr. Notes Theor. Comput. Sci. **128**(2), 117–130 (2005)
29. De Nicola, R., Gorla, D., Pugliese, R.: Confining data and processes in global computing applications. Sci. Comput. Program. **63**(1), 57–87 (2006)
30. De Nicola, R., Katoen, J., Latella, D., Loreti, M., Massink, M.: Model checking mobile stochastic logic. Theor. Comput. Sci. **382**(1), 42–70 (2007)
31. De Nicola, R., Latella, D., Massink, M.: Formal modeling and quantitative analysis of KLAIM-based mobile systems. In: SAC, pp. 428–435. ACM (2005)
32. De Nicola, R., et al.: Programming and verifying component ensembles. In: Bensalem, S., Lakhneck, Y., Legay, A. (eds.) ETAPS 2014. LNCS, vol. 8415, pp. 69–83. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54848-2_5
33. De Nicola, R., Loreti, M.: A modal logic for mobile agents. ACM Trans. Comput. Log. **5**(1), 79–128 (2004)
34. De Nicola, R., Margheri, A., Tiezzi, F.: Orchestrating tuple-based languages. In: Bruni, R., Sassone, V. (eds.) TGC 2011. LNCS, vol. 7173, pp. 160–178. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30065-3_10
35. De Nicola, R., Pugliese, R.: A process algebra based on Linda. In: Ciancarini, P., Hankin, C. (eds.) COORDINATION 1996. LNCS, vol. 1061, pp. 160–178. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61052-9_45
36. De Nicola, R., Pugliese, R.: Testing semantics of asynchronous distributed programs. In: Dam, M. (ed.) LOMAPS 1996. LNCS, vol. 1192, pp. 320–344. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-62503-8_15
37. De Nicola, R., et al.: From flow logic to static type systems for coordination languages. Sci. Comput. Program. **75**(6), 376–397 (2010)
38. Eckhardt, J., Mühlbauer, T., Meseguer, J., Wirsing, M.: Semantics, distributed implementation, and formal analysis of KLAIM models in maude. Sci. Comput. Program. **99**, 24–74 (2015)
39. Ferrari, G.L., Moggi, E., Pugliese, R.: Global types and network services. Electr. Notes Theor. Comput. Sci. **54**, 35–48 (2001)
40. Ferrari, G., Moggi, E., Pugliese, R.: MetaKlaim: meta-programming for global computing. In: Taha, W. (ed.) SAIG 2001. LNCS, vol. 2196, pp. 183–198. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44806-3_11
41. Gelernter, D.: Generative communication in Linda. ACM Trans. Program. Lang. Syst. **7**(1), 80–112 (1985)
42. Gorla, D., Pugliese, R.: Enforcing security policies via types. In: Hutter, D., Müller, G., Stephan, W., Ullmann, M. (eds.) Security in Pervasive Computing. LNCS, vol. 2802, pp. 86–100. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-39881-3_10
43. Gorla, D., Pugliese, R.: Resource access and mobility control with dynamic privileges acquisition. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 119–132. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-45061-0_11
44. Hansen, R.R., Probst, C.W., Nielson, F.: Sandboxing in myKlaim. In: ARES, pp. 174–181. IEEE (2006)
45. Jiang, Y., Xue, G., Jia, Z., You, J.: DTuples: a distributed hash table based Tuple space service for distributed coordination. In: GCC, pp. 101–106. IEEE (2006)

46. Lange, D.B., Mitsuru, O.: Programming and Deploying Java Mobile Agents Aglets. Addison-Wesley, Boston (1998)
47. Leal, M.A., de La Rocque Rodriguez, N., Ierusalimschy, R.: LuaTS - a reactive event-driven tuple space. J. UCS **9**(8), 730–744 (2003)
48. Mamoud, Q.H.: Getting Started With JavaSpaces Technology: Beyond Conventional Distributed Programming Paradigms (2005). https://www.oracle.com/technetwork/articles/java/javaspaces-140665.html
49. Picco, G.P.: μCODE: a lightweight and flexible mobile code toolkit. In: Rothermel, K., Hohl, F. (eds.) MA 1998. LNCS, vol. 1477, pp. 160–171. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0057656
50. Picco, G.P., Murphy, A.L., Roman, G.: LIME: Linda meets mobility. In: ICSE, pp. 368–377. ACM (1999)
51. van der Goot, R.: High performance Linda using a class library. Ph.D. thesis, Erasmus University Rotterdam (2001)
52. Wu, X., Li, X., Lafuente, A.L., Nielson, F., Nielson, H.R.: Klaim-DB: a modeling language for distributed database applications. In: Holvoet, T., Viroli, M. (eds.) COORDINATION 2015. LNCS, vol. 9037, pp. 197–212. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19282-6_13
53. Wyckoff, P., McLaughry, S.W., Lehman, T.J., Ford, D.A.: T spaces. IBM Syst. J. **37**(3), 454–474 (1998)