Revealing the Trajectories of KLAIM Tuples, Statically

Chiara Bodei[1], Pierpaolo Degano[1], Gian-Luigi Ferrari[1], and Letterio Galletta[2(✉)]

[1] Dipartimento di Informatica, Università di Pisa, Pisa, Italy
[2] IMT Institute for Advanced Studies Lucca, Lucca, Italy
letterio.galletta@imtlucca.it

Abstract. KLAIM (Kernel Language for Agents Interaction and Mobility) has been devised to design distributed applications composed by many components deployed over the nodes of a distributed infrastructure and to offer programmers primitive constructs for communicating, distributing and retrieving data. Data could be sensitive and some nodes could not be secure. As a consequence it is important to track data in their traversal of the network. To this aim, we propose a Control Flow Analysis that over-approximates the behaviour of KLAIM processes and tracks how tuple data can move in the network.

1 Introduction

Premise. About twenty years ago Rocco De Nicola contributed to the introduction of KLAIM, a Kernel Language for Agents Interaction and Mobility – as the name suggests – designed for specifying the behaviour of distributed and coordinated processes at a suitable level of abstraction. As it is often the case, this line of work changed with the times, by always evolving to deal with the challenges posed by the new programming paradigms. Starting by our common interest in languages and process algebras, we decided to honour Rocco on his 65th birthday and our long friendship, by working on KLAIM and exploiting our previous experience with static analysis techniques.

Contribution. Modern distributed systems are extremely difficult to model, specify and verify because they are inherently concurrent, asynchronous, and non deterministic. Furthermore, computing nodes in a distributed system are loosely coupled and exhibit a high level of autonomy. These features provide several benefits. For instance, scaling is simplified since each computing node can be scaled independently from the other nodes. Moreover, decoupling enables the design of new mechanisms for orchestrating the overall behaviour. Designing secure and

The first three authors have been partially supported by Università di Pisa PRA_2018_66 *DECLWARE: Metodologie dichiarative per la progettazione e il deployment di applicazioni*; the last author by IMT project *PAI VeriOSS.*

M. Boreale et al. (Eds.): De Nicola-Festschrift, LNCS 11665, pp. 437–454, 2019.
https://doi.org/10.1007/978-3-030-21485-2_24

safe distributed systems is of paramount importance given the vast attack surface presented by them. We cannot address these issues without a solid formal model of system security offering advantages of two different kinds. On the one hand such a model permits to evaluate *a priori* how to prevent security breaches and, on the other hand, it provides the machinery for identifying the techniques one might adopt to achieve the goal of securing distributed systems.

In previous works [3–5], we proposed a *Security by Design* development methodology, consisting of a kernel programming language to describe both the structure of the system and its interactive capabilities. The kernel language is equipped with a suitable *static analysis* that approximates the evolution of the system by providing an *abstract model* of behaviour. These abstractions allow predicting how (abstract) data may flow inside the system. Hence, designers can detect *a priori* the occurrence of unsafe data and possible security breaches, inspecting the "abstract simulation" and intervene as early as possible during the design phase. This methodology has been extended in [6] by introducing a *data path analysis* that supports tracking of the propagation of data, thus identifying their possible trajectories among the computing nodes.

In this paper we apply our methodology to support the design of distributed systems modelled using KLAIM, *Kernel Language for Agents Interaction and Mobility* [11]. This language has been specifically devised to design distributed applications made up of several loosely coupled components deployed over the nodes of a distributed infrastructure. The KLAIM programming model relies on tuples and tuple spaces to coordinate component communications and data management. The language builds on Linda's notion of *generative communication* through a single shared tuple space [17] and generalises it with multiple tuple spaces.

A distinguishing feature of the KLAIM model is the so-called *network awareness*. It indicates the ability of the software components of a distributed application to directly manage a sufficient amount of knowledge about the network environment where they are currently deployed. This capability allows components to have a highly dynamic behaviour and manage unpredictable changes of the network environment over time. Crucial to network awareness are *localities* associated with the network nodes, which are a first order feature of KLAIM.

In this paper we introduce a *control flow analysis* that extends the one proposed in [5,6], and that handles network awareness and coordination via multiple tuple spaces. Our static analysis can be used to detect where and how data are manipulated and how messages flow among the nodes of a KLAIM network. More in detail, the results of the analysis enable us to reason about

- the path in the KLAIM network through which (a value in) a tuple of a specific node reaches another one; and about
- which transformations are applied to a selected datum along those paths.

The proposed analysis permits to identify possible security breaches in the data workflow of a distributed application. For instance, it may keep the safe paths that data inside a tuple can traverse apart from those that pass through a possible untrusted node.

Plan of the Paper. The next section briefly recalls the main aspects of KLAIM. Section 3 illustrates a simple example that is used along the paper. Section 4 defines the static analysis and shows how to inspect its results for checking specific properties. The last section concludes.

2 KLAIM: A Kernel Language for Agents Interaction and Mobility

KLAIM [11] has been specifically devised to design distributed applications consisting of several components (both stationary and mobile) deployed over the nodes of a distributed infrastructure. Its programming model relies on a unique interface (i.e. set of operations) supporting component communications and data management.

The basic building blocks of KLAIM for guaranteeing network awareness are the *locations*. They are the linguistic abstraction to manage addresses (i.e. network references) of nodes and are referred to through identifiers. Locations can be exchanged among the computational components and obey to sophisticated scoping rules. They provide the naming mechanism to identify network resources and to represent the notion of administrative domain: computations at a given location are under the control of a specific authority. In this way, locations provide a natural abstraction to structure and support programming of spatially distributed applications.

KLAIM has multiple distributed tuple spaces. A tuple space is a multiset of tuples. Tuples are *anonymous* sequences of data items and are retrieved from tuple spaces by means of an *associative selection*. Interprocess communication occurs through *asynchronous* exchange of tuples via tuple spaces: there is no need for producers (i.e. senders) and consumers (i.e. receivers) of a tuple to synchronise.

The obtained communication model has a number of properties that make it appealing for distributed computing in general (see, e.g., [7,9,15,18]). It supports *time uncoupling* (data life time is independent of the producer process life time), *destination uncoupling* (the producer of a datum needs not to know the future use or the final destination of that datum) and *space uncoupling* (programmers need to know a single interface only to operate over the tuple spaces, regardless of the node where the action will take place).

2.1 Syntax and Semantics

In this section, we introduce a dialect of KLAIM in the style of a process calculus whose syntax is presented in Table 1. The set of *locations* Loc consists of three disjoint entities:

- the absolute locations $\ell \in \mathscr{L}$;
- the symbolic locations $p \in LSym$;
- the location variables $u \in LVars$.

Table 1. KLAIM syntax

NETS:

N	$::=$	$\ell ::_\rho P$	(computational node)
	$\mid$	$\ell :: \langle t \rangle$	(located tuple, t closed)
	$\mid$	$N_1 \parallel N_2$	(net composition)

PROCESSES:

P	$::=$	$\mathbf{nil}$	(null process)
	$\mid$	$\mathbf{out}(t)@l.P$	(output)
	$\mid$	$\mathbf{in}(t)@l.P$	(input)
	$\mid$	$\mathbf{read}(t)@l.P$	(read)
	$\mid$	$P_1 \mid P_2$	(parallel composition)
	$\mid$	X	(process variable)
	$\mid$	A	(process invocation)

TUPLES:

$t ::= \langle tf_1, \ldots, tf_r \rangle \quad r \geq 1$

TUPLE FIELDS:

$tf ::= E \mid l$

LOCATIONS: $l \in Loc$

$l ::= \ell \in \mathscr{L} \mid p \in LSym \mid u \in LVars$

EXPRESSIONS $E \in \mathscr{E}$:

$E ::= v \in Values \mid x \in Vars \mid f(E_1, \ldots, E_r)$

Absolute locations are used to denote network addresses, through names already assigned to absolute addresses of network components. Symbolic locations, instead, provide the mechanism to support symbolic addressing. They are keywords that refer to specific entities of which the currently running code is a part. The entity referred to by these keywords thus depends on the execution context. For instance, the symbolic location *self* will always refers to the current absolute address of the current execution environment. Since locations are denotable entities we also need location variables.

NETS are finite collections of nodes where processes and data can be placed. A *computational node* takes the form $\ell ::_\rho P$, where ρ is an *allocation environment* and P is a process. Since processes may refer to location variables or symbolic locations, the allocation environment acts as a *name solver* that binds locations variables and symbolic locations to absolute locations. TUPLES are sequences of fields, i.e. of expressions and of locations. The precise syntax of EXPRESSIONS is deliberately not specified; it is just assumed that they contain, at least, *basic values* V, and *value variables*, ranged over by x. The tuple space of a node consists of all the tuples that do not contain variables and that are located there (we will sometimes refer to them as evaluated tuples). We will use $[\![t]\!]$ to denote the result of evaluating the expression t, possibly applying also the allocation environment ρ.

PROCESSES are the active computational units of KLAIM. Their syntax is standard. Recursive behaviours are modelled via process definitions. For that we assume a set of *process identifiers*, ranged over by A. A process definition has the standard form $A \triangleq P$, but we additionally assume each identifier A has a *single* defining equation. The actions processes perform provide the programming abstractions that support data management. Three primitive behaviours are provided: adding (**out**), withdrawing (**in**) and reading (**read**) a tuple to/from a tuple space. Input and output actions are *mutators*: their execution modifies the

Table 2. Structural congruence

(Com)	$N_1 \parallel N_2 \equiv N_2 \parallel N_1$	(Assoc)	$(N_1 \parallel N_2) \parallel N_3 \equiv N_1 \parallel (N_2 \parallel N_3)$
(Abs)	$\ell :: P \equiv \ell :: (P \mid \mathbf{nil})$	(PrInv)	$\ell :: A \equiv \ell :: P \quad \textit{if } A \triangleq P$
(Clone)	$\ell :: (P_1 \mid P_2) \equiv \ell :: P_1 \parallel \ell :: P_2$		

tuple space. The read action is an *observer*: it checks the availability and takes note of the content of a certain tuple without removing it from the tuple space. Actions are tagged with the (possibly remote) location where they will take place. Note that, in principle, each network node can provide its own implementation of the action interface. This feature can be suitably exploited to sustain different policies for data handling as done, e.g. in MetaKlaim [16].

Names occurring in processes and nets can be *bound*. For example, the action prefix $\mathbf{in}(u)@l.P$ binds u in P, which is the scope of the bindings made by the action. A name that is not bound is *free*. The sets of free and bound names of a process/net term are defined in the standard way. As usual, we say that two terms are α*-equivalent*, written $\equiv_\alpha$, if one can be obtained from the other by renaming bound names. Hereafter, we shall work with terms whose bound names are all distinct and different from the free ones.

A *pattern-matching* mechanism is used for associatively selecting (evaluated) tuples from tuple spaces. Intuitively, a tuple matches against an evaluated one if both have the same number of fields and corresponding fields do match; two values (locations) match only if they are identical, while variables match any value of the same type. A successful matching returns a substitution associating the variables contained in the fields of the tuples with the values contained in the corresponding fields of the evaluated tuple. We will use σ to range over *substitutions*. As usual, substitution application may require α-conversion to avoid capturing of free names.

We will use the notation $match(\llbracket\, t \,\rrbracket, et) = \sigma$ to indicate that σ is the substitution resulting from the pattern matching of tuple t with the evaluated tuple et.

The operational semantics is given in terms of a structural congruence $\equiv$ and of a reduction relation $\succ\!\!\longrightarrow$ over nets. The *structural congruence* is defined as the smallest congruence relation over nets that satisfies the laws in Table 2. These relate nets that intuitively behave the same, stating that $\parallel$ is commutative and associative, that the null process can always be safely removed/added, that a process identifier can be replaced with the body of its definition, and that it is always possible to transform a parallel of co-located processes into a parallel over nodes. Indeed, rule (Struct) says that all structural congruent nets can make the same reduction steps.

The *reduction relation* is the least relation induced by the rules in Table 3. All the rules for (possibly remote) process actions require the target node to exist. In addition, the rule (In) requires the chosen datum to occur in the target node. Moreover, the rule says that action $\mathbf{in}(u)@l'$ looks for any name ℓ'' at ℓ'

Table 3. Operational semantics of KLAIM

$$\text{(OUT)}\quad \frac{\rho(l) = l' \qquad [\![\, t \,]\!] = et}{l :: \mathbf{out}(t)@l'.P \parallel l' :: P' \rightarrowtail l :: P \parallel l' :: P' \parallel l' :: \langle et \rangle}$$

$$\text{(IN)}\quad \frac{\rho(l) = l' \qquad match([\![\, t \,]\!], et) = \sigma}{l :: \mathbf{in}(t)@l'.P \parallel l' :: \langle et \rangle \rightarrowtail l :: P\sigma \parallel l' :: \mathbf{nil}}$$

$$\text{(READ)}\quad \frac{\rho(l) = l' \qquad match([\![\, t \,]\!], et) = \sigma}{l :: \mathbf{read}(t)@l'.P \parallel l' :: \langle et \rangle \rightarrowtail l :: P\sigma \parallel l' :: \langle et \rangle}$$

$$\text{(PAR)}\quad \frac{N_1 \rightarrowtail N_1'}{N_1 \parallel N_2 \rightarrowtail N \vdash_1' \parallel N_2}$$

$$\text{(STRUCT)}\quad \frac{N \equiv N_1 \qquad N_1 \rightarrowtail N_2 \qquad N_2 \equiv N'}{N \rightarrowtail N'}$$

that is then used to replace the free occurrences of u in the continuation of the process performing the input, while action $\mathbf{in}(\ell'')@\ell'$ looks exactly for the name ℓ'' at ℓ'; in both cases, the matched datum is consumed. With abuse of notation, we use **nil** to replace the consumed data.

Rule (PAR) says that if part of a net makes a reduction step, the whole net reduces accordingly. Process interaction is asynchronous: no synchronisation takes place between sender and receiver processes (only existence of target nodes is checked). Moreover, communication is anonymous, because data do not include the name of the sender, and associative, because data are accessed via pattern matching.

3 Example: A Microservice Architecture

Microservices have been recently introduced as a software architecture pattern used to build distributed applications composed of small, independent and highly decoupled services. A microservice is equipped with a dedicated data storage support (e.g. a data base) and provides basic (simple) services by computing certain functionalities (e.g. querying a database). A microservice-based application usually takes the form of a structured protocol composed by multiple phases. Each phase is implemented by a specific microservice. Microservices interact by exchanging messages. Since all the components of the software architecture are microservices, the overall behaviour is derived by the coordination of its components via message exchange. As an example, the Netflix service uses around 700 microservices to control each of its many parts.

Microservice software architectures present many security challenges, not new, since they apply to the Service-Oriented paradigm. However, they become

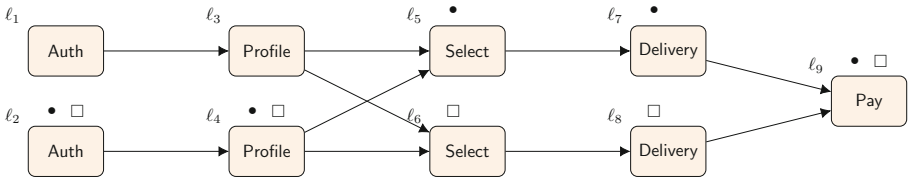

Fig. 1. A network of microservices. The same datum following the trajectory with bullets • is at risk, while it is safe along the trajectory with boxes □.

even more challenging in this context since service requests are routed among the multiple independent services. For instance, it may happen that a single microservice controlled by a malicious entity may corrupt the coordination of the service requests and therefore the overall behaviour of the application is compromised.

We outline the main features of the design of a (simplified) *Microservice Application* for delivering digital artefacts or contents (e.g. movies) to registered users. The underlying structured protocol basically consists of several stages. The first provides an authentication/authorisation facility. Registered users may select one or more products to buy. In the second phase, the selected item is sent to the users. Finally, the user pays, which requires the execution of an entire sub-protocol, involving also a digital bank. In a monolithic architecture this will be implemented as a stateful application. This is not the case with microservices since one has to route the requests to multiple independent services. Figure 1 illustrates the structure of the application together with the underlying workflow of messages. We comment on the architecture:

- The AUTH microservice provides facilities for authenticating registered user; it also grants her/him some specific interactions;
- The PROFILE microservice determines user's profile taking into account all what was stored by the application about the registered users;
- The SELECT microservice supports the user in making the choice, possibly suggesting the user the items she/he will like;
- The DELIVER microservice sends the required digital artefact to the user;
- The PAY microservice deduces the monthly fee from the user's account.

We assume that each stage of the application is split over and implemented by groups of microservices. For instance, the AUTH service is distributed over a pair of microservices independent from each other. This also implies that the application has multiple entry points to control users' access. Similarly multiple PROFILE microservices will be dedicated to manage user requests by providing the suitable context of user preferences. Note that each PROFILE micro service may be built over different database schemata storing different data. This sort of

decentralised governance is applied to all the stages of the application. Figure 1 also illustrates a possible workflow of service requests with indicators of risk level.

The software architecture briefly discussed above is rendered here by making each microservice a KLAIM node. For simplicity, we will focus on the coordination among the microservices via tuple-based messaging. With an abuse of notation we will freely exploit certain suitable processes without showing their detailed implementation. The main processes of the AUTH microservice level are given below:

$$\begin{aligned} H \triangleq\ & \mathbf{in}(usr, psw, req)@\mathsf{self}.\mathbf{out}(usr, psw)@\mathsf{self}.\mathbf{in}(usr, token)@\mathsf{self}. \\ & \{\mathbf{for}\ l \in Policy(usr, token) \\ & \quad \mathbf{out}(usr, req, token)@l\}.H \\ C \triangleq\ & \mathbf{in}(usr, psw)@\mathsf{self}.I.\mathbf{out}(usr, Check(usr, psw))@\mathsf{self}C \end{aligned}$$

The handler process H receives the authentication request, obtained by sensing in the tuple space the tuple (usr, psw, req), activates one of the processes checking user credentials by emitting in the tuple space the tuple (usr, psw) and finally generates the authentication token by inspecting the tuple space. The authorisation token is made available by the checking user credential process. We abstract from the detailed description of checking user credential process C. We simply assume that the process is activated by the presence of the tuple (usr, psw) in the tuple space and yields as result the tuple $(usr, token)$, where the value $token$ is the authorisation information associated to the specific user usr. The authorisation token is computed, after having executed some internal activities I, by applying the function $Check$, which takes as input the values usr, psw, making clear that the authorisation token strictly depends on the user information. The result of the authentication is then forwarded to the PROFILE microservices hosted in the locations l, depending on a certain $Policy$ function that implements the workflow of messages in accordance with the multistage pattern of the application.

To conclude the description of the authorisation stage, we present the KLAIM nodes that realise the AUTH microservices.

$$AUTH \triangleq (l_1 ::_{\rho_1} H \mid C \mid T_1) \parallel (l_2 ::_{\rho_2} H \mid C \mid T_2)$$

The authorisation microservices consist of two KLAIM nodes located at l_1 and l_2 respectively. Intuitively, registered users can open more than one session of the application at the same time and, therefore, using more than one microservice of the application at the same time. Each node hosts the handler processes H and the process C checking user credentials as discussed above together with the local tuple spaces, represented by the suitable process T_1 and T_2. Each microservice stores and manages its own data within the local tuple space. It is worth noting the exploitation of tuple spaces to coordinate the behaviour of the processes deployed in the nodes.

We now move our attention to the PROFILE stage that computes the personal data associated to the specific registered user. Note that the user's profile depends on the location where the microservice is located, because, in a

microservice-based architecture, each microservice owns and controls its own database that is not shared with others to avoid conflicts. The main processes of the PROFILE stage are given below

$$\begin{aligned} D &\triangleq \mathbf{in}(usr, req, token)@\mathsf{self}.\mathbf{out}(usr, \mathsf{self}, token)@\mathsf{self}. \\ &\quad\ \mathbf{in}(usr, profile)@\mathsf{self}.\mathbf{out}(usr, token, req, profile)@next.D \\ P &\triangleq \mathbf{in}(usr, u, token).@\mathsf{self}.I.\mathbf{out}(usr, \mathit{UserProfile}(usr, u))@\mathsf{self}.P \end{aligned}$$

The driver process D receives the user request, obtained by sensing in the tuple space the tuple $(usr, req, token)$. Note that each user request is tagged with the authorisation token to identify the specific user's session. The driver activates the process that has the task of calculating the user's profile by emitting in the tuple space the tuple $(usr, profile)$. Finally, the next step in the workflow begins with the generation of the tuple $(usr, token, req, profile)$ and its transmission to the remote node identified by the symbolic location *next* (that will be instantiated by the allocation environment of the nodes where processes will be deployed). The behaviour of the process P is straightforward. We only emphasise the role of the function *UsrProfile*. This function abstracts the activity of computing user's profile taking into account the information available locally. This feature also implies a certain amount of autonomy of the microservice. The awareness of the locality where information is taken transforms the tuple space into a bounded context: each local tuple space may have its own understanding of what a "user" is (e.g. maybe in a certain tuple space the "user" is characterised by several tuples while in a different tuple space a single tuple is enough).

The KLAIM nodes that implement the PROFILE stage are the following

$$\mathit{PROFILE} \triangleq (l_3 ::_{\rho_3} D \mid P \mid T_3) \parallel (l_4 ::_{\rho_4} D \mid P \mid T_4)$$

Each node hosts the drive processes D and the process P computing user's profile as discussed above, together with the local tuple spaces, represented by the suitable processes T_3 and T_4.

This third stage of the application is characterised by the SELECT microservice. Two processes drive the behaviour of the microservice. Both processes are activated by sensing in the tuple space the tuple $(usr, token, req, profile)$. The first process S_1 prompts a list of suggestions based on the user's profile taking advantage of the information made available by the auxiliary process C_S, with the obvious meaning. We only comment on the function *CheckProfile* that abstracts the activities for computing the list of suggestions, according to the user's request and profile. The second process S_2 simply shows to the user her/his requests of the session at hand.

$$\begin{aligned} S_1 &\triangleq \mathbf{in}(usr, token, req, profile)@\mathsf{self}.\mathbf{out}(usr, req, profile)@\mathsf{self}. \\ &\quad\ \mathbf{in}(usr, suggestion)@\mathsf{self}.\mathbf{out}(usr, token, req, suggestion)@\mathsf{self}.S_1 \\ C_S &\triangleq \mathbf{in}(usr, req, profile)@\mathsf{self}.\mathbf{out}(usr, \mathit{CheckProfile}(req, profile)) \\ S_2 &\triangleq \mathbf{in}(usr, token, req, profile)@\mathsf{self}.\mathbf{out}(usr, token, req)@\mathsf{self}.S_2 \end{aligned}$$

The KLAIM nodes that implement the SELECT stage are the followings

$$\mathit{SELECT} \triangleq (l_5 ::_{\rho_5} S_1 \mid S_2 \mid C_S \mid F \mid T_5) \parallel (l_6 ::_{\rho_6} S_1 \mid S_2 \mid C_S \mid F \mid T_6)$$

Each node hosts the drive processes S_1, S_2, the auxiliary process C_S discussed above, and the process F, the detailed description of which omitted here. This process takes the user's confirmation, sends the user digital rights for the purchase (via the DELIVERY microservice) and activates the payment microservice.

4 Control Flow Analysis

Below, we first introduce regular tree grammars that will be used to abstractly represent KLAIM data; then we present our control flow analysis; and finally we show that the results of the analysis can be used to check how data are manipulated and how they traverse the network of processes.

4.1 Abstract Representation of Data

In the following we represent the data populating and traversing a net of KLAIM processes in an abstract form. Since a system is designed to be continuously active and may contain feedback loops, data can grow unboundedly, while we insist on having finite representation. We resort then to set of regular trees and we associate with data regular tree grammars [8] as *finite* abstractions. The leaves of a tree in the language of a regular grammar represent basic values v and locations ℓ. Instead, its nodes represent functions applied to data, tuple constructions and transfer from the tuple space of a specific computational node to another one. A brief survey on regular tree grammars follows.

A *regular tree grammar* is a quadruple $\widehat{G} = (\mathbb{N}, \mathbb{T}, Z, R)$ where

- $\mathbb{N}$ is a set of non-terminals (with rank 0),
- $\mathbb{T}$ is a ranked alphabet, whose symbols have an associated arity,
- $Z \in \mathbb{N}$ is the starting non-terminal,
- R is a set of productions of the form $A \to t$, where t is a tree composed from symbols in $\mathbb{N} \cup \mathbb{T}$ according to their arities.

In the following we denote the language generated by a given grammar $\widehat{G}$ with $Lang(\widehat{G})$.

Given a net of processes, the grammars we use will have the alphabet $\mathbb{T}$ consisting of the following set of ranked symbols

- ℓ (with arity 0) for each $\ell \in \mathscr{L}$
- v^ℓ (with arity 0) for each value $v \in \mathit{Value}$ and $\ell \in \mathscr{L}$
- t^ℓ (with arity r) to represent a tuple with arity r in $\ell \in \mathscr{L}$
- f^ℓ (with arity r) for each function f in $\ell \in \mathscr{L}$ with arity r
- s^ℓ (with arity 1) to represent an output from $\ell \in \mathscr{L}$

The non-terminals $\mathbb{N}$ of our grammars include a symbol for each terminal, and carry the label of the relevant computational node. Just for readability we shall capitalize the ranked symbols above and use them as non-terminals, i.e. L^ℓ, V^ℓ, T^ℓ, F^ℓ, and S^ℓ. When irrelevant, we shall omit the labels ℓ, and we shall use a capital letter for a generic non-terminal. For example, a r-tuple is abstractly

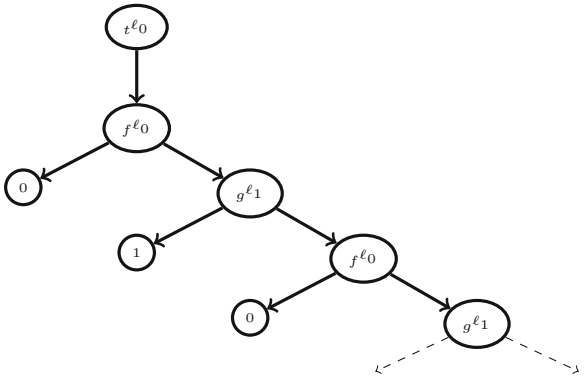

Fig. 2. An infinite abstract tree

represented by a grammar with the production $T^\ell \to t^\ell(A_1, ..., A_r)$ and the productions for A_i, that generates the tree rooted in t^ℓ and children generated by $A_1, ..., A_r$.

It is convenient introducing some notation. For brevity and when not ambiguous, we will simply write $\hat{v} = (Z, R)$ for the grammar $\widehat{G} = (\mathbb{N}, \mathbb{T}, Z, R)$ with starting non-terminal Z and regular productions in R, without explicitly listing the terminals and the non-terminals. Then, we denote with $\mathbb{R}$ the set of all possible productions over $\mathbb{N}$ and $\mathbb{T}$.

As an example of a possible infinite abstract tree, consider two computational nodes P_{ℓ_0} and P_{ℓ_1} and two binary functions f and h. Suppose that P_{ℓ_0} applies f to 0 and to a value taken in the tuple space of P_{ℓ_1}. Similarly, P_{ℓ_1} applies h to 1 and the value taken in the tuple space of P_{ℓ_0}. The resulting value in the tuple space of P_{ℓ_0} is abstracted as the set of binary trees of unbounded depth. The following grammar represents them all (an element of its language is in Fig. 2):

$$(T^{\ell_0}, \{T^{\ell_0} \to t^{\ell_0}(F^{\ell_0}), F^{\ell_0} \to f^{\ell_0}(I_0^{\ell_0}, G^{\ell_1}), I_0^{\ell_0} \to 0, G^{\ell_1} \to g^{\ell_1}(I_1^{\ell_1}, F^{\ell_0}), I_1^{\ell_1} \to 1\})$$

Now we are ready to introduce the *abstract terms* that belong to the set

$$\widehat{\mathscr{V}} = 2^{\mathbb{N} \times \mathbb{R}}$$

4.2 Specification of the Analysis

The result or *estimate* of our CFA is a pair $(\Sigma_\ell, \Theta_\ell)$ for tuple fields tf, a triple $(\Sigma_\ell, \Theta_\ell, \kappa)$ for processes P, and a triple (Σ, Θ, κ) for nets of processes. The components of an estimate are the following *abstract domains* (we omit labels ℓ for brevity):

– *abstract enviroment* $\Sigma : (LVar \cup Vars) \to \widehat{\mathscr{V}}$ is an abstract environment that associates symbolic locations and variables with a set of abstract values;

– *abstract data collection* $\Theta : \mathscr{L} \to \widehat{\mathscr{V}}$ approximates the values that a node hosted at ℓ can manipulate;
– *abstract tuple space* $\kappa : \mathscr{L} \to \widehat{\mathscr{V}}$ approximates the tuple space of a node.

The syntax directed rules of Tables 4 and 5 specify when an analysis estimate is valid and they are almost in the format of AFPL, which is a logic used to specify static analyses and which allows systematically deriving analysis algorithms [20]. For each tuple t (and its fields tf), the judgement $(\Sigma_\ell, \Theta_\ell) \models^\rho_\ell t : \vartheta$ (and $(\Sigma_\ell, \Theta_\ell) \models^\rho_\ell tf : \vartheta$) expresses that $\vartheta \in 2^{\widehat{\mathscr{V}}}$ approximates the set of tuples that t (tf) may evaluate to, given the abstract environment Σ_ℓ. An actual location and a value evaluate to the set ϑ, provided that their abstract representations belong to ϑ (rules (Loc) and (Val)). This abstract representation is a grammar made of a non-terminal symbol whose production generates a tree with a single node. For example, the abstract value for an actual location ℓ' is $(L^\ell, \{L^\ell \to \ell'\})$ that represents a grammar with the initial symbol is L^ℓ that only generates the tree ℓ'. The rule (L-sym) takes care of symbolic locations and resolves them through ρ. The rules (L-var) and (E-var) for variables require the binding for them to be included in ϑ. The rule (E-fun) analyses the application of an r-ary function f to produce the set ϑ. To do that (i) for each term E_i, it finds the sets ϑ_i, and (ii) for all sequences of r values (Z_i, R_i) in ϑ_i, it checks if ϑ includes the grammars with distinct symbol F^ℓ generating the trees rooted in f^ℓ with subtrees generated by Z_i. The rule (Tuple) is similar. Note that in all the rules above, we require that the abstract data collection $\Theta(\ell)$ includes all the abstract values in ϑ.

Table 4. Analysis of tuples $(\Sigma_\ell, \Theta_\ell) \models^\rho_\ell t : \vartheta$ and of tuple fields $(\Sigma_\ell, \Theta_\ell) \models^\rho_\ell tf : \vartheta$.

(Loc)
$$\frac{(L^\ell, \{L^\ell \to \ell'\}) \in \vartheta \subseteq \Theta_\ell}{(\Sigma_\ell, \Theta_\ell) \models^\rho_\ell \ell' : \vartheta}$$

(L-sym)
$$\frac{(L^\ell, \{L^\ell \to \rho(p)\}) \in \vartheta \subseteq \Theta_\ell}{(\Sigma_\ell, \Theta_\ell) \models^\rho_\ell p : \vartheta}$$

(L-var)
$$\frac{\Sigma_\ell(u) \subseteq \vartheta \subseteq \Theta_\ell}{(\Sigma_\ell, \Theta_\ell) \models^\rho_\ell u : \vartheta}$$

(Val)
$$\frac{(V^\ell, \{V^\ell \to v^\ell\}) \in \vartheta \subseteq \Theta_\ell}{(\Sigma_\ell, \Theta_\ell) \models^\rho_\ell v : \vartheta}$$

(E-var)
$$\frac{\Sigma_\ell(x) \subseteq \vartheta \subseteq \Theta_\ell}{(\Sigma_\ell, \Theta_\ell) \models^\rho_\ell x : \vartheta}$$

(Fun)
$$\frac{\bigwedge_{i=1}^{r} (\Sigma_\ell, \Theta_\ell) \models^\rho_\ell E_i : \vartheta_i \wedge \quad \forall (Z_1, R_1), \cdots, (Z_r, R_r) : \bigwedge_{i=1}^{r} (Z_i, R_i) \in \vartheta_i \Rightarrow (F^\ell, \{F^\ell \to f^\ell(Z_1, \cdots, Z_r)\} \cup \bigcup_{i=1}^{r} R_i) \in \vartheta \subseteq \Theta_\ell}{(\Sigma_\ell, \Theta_\ell) \models^\rho_\ell f(E_1, \cdots, E_r) : \vartheta}$$

(Tuple)
$$\frac{\bigwedge_{i=1}^{r} (\Sigma_\ell, \Theta_\ell) \models^\rho_\ell tf_i : \vartheta_i \wedge \quad \forall (Z_1, R_1), \cdots, (Z_r, R_r) : \bigwedge_{i=1}^{r} (Z_i, R_i) \in \vartheta_i \Rightarrow (T^\ell, \{T^\ell \to t^\ell(Z_1, \cdots, Z_r)\} \cup \bigcup_{i=1}^{r} R_i) \in \vartheta \subseteq \Theta_\ell}{(\Sigma_\ell, \Theta_\ell) \models^\rho_\ell \langle tf_1, \cdots, tf_r \rangle : \vartheta}$$

Some further auxiliary definitions may help keeping the logical specification of the analysis of nets and processes less intricate. In particular, they simplify handling the grammars and extracting the needed information from them. The function *put* constructs a grammar that records that a tuple, approximated by (Z, R), may be inserted in the tuple space of the computational node at ℓ. The function @ takes a set of grammars ϑ and returns the set of actual locations in those grammars with starting symbol L. The function *get* recursively visits a grammar to find a tuple that has been acquired by a process at ℓ; its base cases exhibit, if any, the tuple built by a process at ℓ' approximated by the grammar with starting symbol $T^{\ell'}$.

Definition 1 (Auxiliary definitions for the analysis).
Let P be a process, ℓ be an absolute location, and (Z, R) be an abstract value in the following three auxiliary functions.

- $put(\ell, (Z, R)) = (S^\ell, \{S^\ell \to s^\ell(Z)\} \cup R)$
- $\widehat{@}\vartheta = \{\ell \mid (L^\ell, \{L^\ell \to \ell\}) \in \vartheta\}$
- $$\begin{cases} get((S^\ell, \{S^\ell \to s^\ell(T^{\ell'}), T^{\ell'} \to t^{\ell'}(Z_1, \ldots, Z_r)\} \cup R)) = \\ \qquad \langle (S^\ell, \{S^\ell \to s^\ell(Z_1)\} \cup R_1), \ldots, (S^\ell, \{S^\ell \to s^\ell(Z_r)\} \cup R_r)\rangle \\ \qquad\qquad \textit{where } R_i \textit{ are the productions concerning } Z_i \\ get((S^\ell, \{S^\ell \to s^\ell(S_0^{\ell'})\} \cup R)) = get((S_0^{\ell'}, R)) \\ get((A, R)) = \langle\rangle \qquad \textit{if } A \neq S^\ell \end{cases}$$

The specification of the analysis of nets $(\Sigma, \Theta, \kappa) \models N$, and of processes $(\Sigma_\ell, \Theta_\ell, \kappa) \models^\rho_\ell P$ is in Table 5. The rules (N-node) and (N-tuple) lift a process and a tuple in a specific location where they have been analysed; note that the approximation of the tuple is included in the abstract tuple space of the node ℓ. The rule (N-par) says that the estimate of the parallel composition is also valid for the components.

The remaining rules are for processes. The rules for the inactive node (N-nil) and for parallel composition (N-par) are standard, as well as those for process definition and invocation, where to save notation, we assumed that each variable X is uniquely bound to the body P. The rule (P-out) (i) approximates the tuple t with ϑ and the symbolic location l with ϑ'; and (ii) for all grammar $\hat{v}$ in ϑ and for all locations ℓ' extracted from ϑ', it checks if the tuple space approximation $\kappa(\ell')$ contains a grammar that records that the (approximation $\hat{v}$ of the) tuple t has been inserted by the node ℓ; and finally that P has a valid approximation.

The rule (P-in) for input and read is the most complex, with a premise made of two conjuncts that imply three. The first conjunct of the condition finds the approximation ϑ of the symbolic location l. The second one extracts the actual locations ℓ' possibly bound to l and looks for the grammars approximating the tuples in the space of those locations. If there are any (non-empty) such tuples $\hat{v}_1, \cdots, \hat{v}_r$ and if a component tf_i of the input/read tuple is an actual location ℓ'', then ℓ'' must also occur in the same position in the approximation, i.e. in $\hat{v}_i$. This statically implements pattern matching on tuple, but on locations only. The first

Table 5. Analysis of nets $(\Sigma, \Theta, \kappa) \models N$, and of processes $(\Sigma_\ell, \Theta_\ell, \kappa) \models^\rho_\ell P$.

$$\text{(N-NODE)}\ \frac{(\Sigma_\ell, \Theta_\ell, \kappa) \models^\rho_\ell P}{(\Sigma, \Theta, \kappa) \models \ell ::_\rho P} \qquad \text{(N-TUPLE)}\ \frac{(\Sigma_\ell, \Theta_\ell) \models^\rho_\ell t : \vartheta \ \wedge \vartheta \subseteq \kappa(\ell)}{(\Sigma, \Theta, \kappa) \models \ell :: t} \qquad \text{(N-PAR)}\ \frac{(\Sigma, \Theta, \kappa) \models N_1 \wedge (\Sigma, \Theta, \kappa) \models N_2}{(\Sigma, \Theta, \kappa) \models N_1 \parallel N_2}$$

$$\text{(P-NIL)}\ \frac{}{(\Sigma_\ell, \Theta_\ell, \kappa) \models^\rho_\ell \mathbf{nil}} \qquad \text{(P-PAR)}\ \frac{(\Sigma_\ell, \Theta_\ell, \kappa) \models^\rho_\ell P_1 \ \wedge \ (\Sigma_\ell, \Theta_\ell, \kappa) \models^\rho_\ell P_2}{(\Sigma_\ell, \Theta_\ell, \kappa) \models^\rho_\ell P_1 \mid P_2}$$

$$\text{(P-DEF)}\ \frac{(\Sigma_\ell, \Theta_\ell, \kappa) \models^\rho_\ell P}{(\Sigma_\ell, \Theta_\ell, \kappa) \models^\rho_\ell A}\ \text{if } A \triangleq P \qquad \text{(P-VAR)}\ \frac{}{(\Sigma_\ell, \Theta_\ell, \kappa) \models^\rho_\ell A}$$

$$\text{(P-OUT)}\ \frac{(\Sigma_\ell, \Theta_\ell) \models^\rho_\ell t : \vartheta \wedge \ (\Sigma_\ell, \Theta_\ell) \models^\rho_\ell l : \vartheta' \ \wedge \left(\forall \hat{v} \in \vartheta, \forall \ell' \in \widehat{@}\vartheta' \Rightarrow put(\ell, \hat{v}) \in \kappa(\ell')\right) \wedge (\Sigma_\ell, \Theta_\ell, \kappa) \models^\rho_\ell P}{(\Sigma_\ell, \Theta_\ell, \kappa) \models^\rho_\ell \mathbf{out}(t)@l.P}$$

$$\text{(P-IN/READ)}\ \frac{\begin{array}{c}(\Sigma_\ell, \Theta_\ell) \models^\rho_\ell l : \vartheta \ \wedge \forall \ell' \in \widehat{@}\vartheta. \forall \hat{v} \in \kappa(\ell'). \left(get(\hat{v}) = \langle \hat{v}_1, \ldots, \hat{v}_r \rangle \ \wedge (\bigwedge_{i=1}^{r} tf_i \in \mathscr{L} \Rightarrow tf_i \ \widehat{in}\ \hat{v}_i)\right) \Rightarrow \\ \left(\bigwedge_{i=1}^{r} tf_i \in (LVar \cup Vars) \Rightarrow \hat{v}_i \in \Sigma_\ell(tf_i) \ \wedge \bigwedge_{i=1}^{r} tf_i \in \mathscr{E} \setminus Vars \Rightarrow (\Sigma_\ell, \Theta_\ell) \models^\rho_\ell tf_i : \vartheta_i \ \wedge \ (\Sigma_\ell, \Theta_\ell, \kappa) \models^\rho_\ell P\right)\end{array}}{(\Sigma_\ell, \Theta_\ell, \kappa) \models^\rho_\ell \mathbf{in}(\langle tf_1, \ldots, tf_r \rangle)@l.P \,/\, \mathbf{read}(\langle tf_1, \ldots, tf_r \rangle)@l}$$

conjunct of the conclusion of the implication requires that abstract environment includes the abstract values for each identifier and location variables occurring in the read tuple. The second and the third conjuncts require that the analysis validates the other elements of the tuple and the continuation P.

4.3 Checking Data Manipulation and Trajectories

We now illustrate how the outcome of the analysis can be used to detect where and how data are manipulated and how messages flow in a system. More precisely, the results of the analysis enable us to reason about (i) the path in the network through which (a value in) a tuple of a specific node reaches another one, and about (ii) which transformations are applied to a selected datum along those paths.

In our example of Sect. 3 a designer could be interested in imposing a policy that forbids serving a request coming from a certain geographic area while the user is associated with a different area by the profile. This situation occurs, e.g. when the user is travelling, the microservices reside in different areas and the user connects to the closest such microservice. Suppose that the two AUTH microservices serve each a different region. In our terms, one has to check whether a certain request authorised in ℓ_2 does not reach the SELECT microservice in ℓ_5. For brevity, we consider below only the parts of the analysis that check this property, while we do not consider data manipulations.

From now onwards, assume that all the symbolic names l have been bound to the corresponding absolute locations by the environments. The analysis of AUTH requires that the following holds

$$(\Sigma_\ell, \Theta_\ell, \kappa) \models^{\rho_2}_{\ell_2} \mathrm{H}$$

One has to analyse first $\mathbf{in}(usr, psw, req)@\ell_2$. We skip this step and for simplicity assume that the following holds, where (Z_i, R_i) are suitable tree grammars

$$(\Sigma_{\ell_2}, \Theta_{\ell_2}) \models^{\rho_2}_{\ell_2} req : \vartheta = \{(Z_1, R_1) \dots (Z_n, R_n)\}$$

The process H terminates by sending the relevant tuple to ℓ_4 and its analysis

$$(\Sigma_\ell, \Theta_\ell, \kappa) \models^{\rho_2}_{\ell_2} \mathbf{out}(usr, req, token)@\ell_4.\mathrm{H}$$

enriches the grammars $\{(Z'_1, R'_1) \dots (Z'_k, R'_k)\}$ of the tuple with the information about this transit, performed by the function *put*, yielding $\{(S^{\ell_2}, \{S^{\ell_2} \to s^{\ell_2}(Z'_1)\} \cup R'_1) \dots (S^{\ell_2}, \{S^{\ell_2} \to s^{\ell_2}(Z'_k)\} \cup R'_k)\}$.

Now the analysis of SELECT requires that of its actions, but we only concentrate on

$$(\Sigma_\ell, \Theta_\ell, \kappa) \models^{\rho_4}_{\ell_4} \mathbf{out}(usr, token, req, suggestion)@\ell_5.S_2$$

Again, the output tuple is enriched with the information represented by s^{ℓ_4}.

Before discussing how to use the analysis results of this example, we define the following notions, formalising the inspection of results. We start by defining a function that, given an abstract value $\hat{v}$ (i.e. a tree grammar) returns a finite set of finite sequences of labels, ending with either a value in *Value* or a location in $\mathscr{L}$. Below we assume as given a network N and the result of its analysis.

Definition 2 (Extracting trajectories). *Let $\hat{v}$ be a tree grammar, the set of* trajectories *of the values and locations represented by $\hat{v}$ is*

$$TRJ(\hat{v}) = {}^{\emptyset}trj^{\emptyset}(\hat{v})$$

where ${}^{I}trj^{J}$ is inductively defined on the shape of $\hat{v}$ as follows

- ${}^{I}trj^{J}(A^\ell, \{A^\ell \to c^\ell(Z_1, \dots, Z_r)\} \cup R) = \bigcup_{i=1, Z_i \notin I}^{r} \{\ell \cdot {}^{I'}trj^{J \cup A}((Z_i, R_i))\}$
- ${}^{I}trj^{J}(L^\ell, \{L^\ell \to \ell'\}) = \ell \cdot \ell'$
- ${}^{I}trj^{J}(V^\ell, \{V^\ell \to v^\ell\}) = \ell \cdot v$

and

- $\ell \cdot X = \{\ell \cdot x \mid x \in X\}$
- $I' = \begin{cases} I \cup A & \text{if } A \notin J \\ I & \text{otherwise} \end{cases}$

As expected, the auxiliary function ${}^{I}trj^{J}$ extracts a trajectory from an abstract value, by accumulating on each sequence the location of a traversed node. The trajectories are kept finite because the sets I and J keep track of the visited nodes, which are not visited more than twice.

Now we define when a value or a label does not traverse a node that the designer considers malicious, and thus that trajectory violates the policy.

Definition 3 (Datum reaches). *A datum* $d \in \mathit{Value} \cup \mathscr{L}$ reaches *a node* ℓ_k *without passing through a node* ℓ *if and only if*

$$\forall \hat{v} \in \Theta_{\ell_k}.\ \ell_0 \cdots \ell_k \cdot d \in TRJ(\hat{v}) \ \Rightarrow\ \forall j.\, \ell \neq \ell_j$$

We turn our attention to data manipulations. In particular, we describe how a designer can check where data originates and which functions transform them.

Definition 4 (Data manipulation). *A datum* $d \in \mathit{Value} \cup \mathscr{L}$*, originated from the node with label* ℓ_0*, is an* ingredient *of a node* ℓ_k *if and only if*

$$\exists \hat{v} \in \Theta_{\ell_k}.\ \ell_0 \cdots \ell_k \cdot d \in TRJ(\hat{v})$$

Furthermore, a function f may manipulate *a value* v *reaching a node* ℓ_k *if and only if there exists an abstract value* $(A, R) \in \Theta_{\ell_k}$ *such that* R *contains a production* $F^{\ell'} \rightarrow f^{\ell'}(R_1, \ldots, R_n)$*, for some* ℓ'*.*

The first part of the above definition is straightforward since inspecting the Θ_{ℓ_k} suffices to understand if a value may be stored in the tuple space of the node ℓ_k. The second part checks if the function f may be applied in any node along the path traversed by the value v. Again, this information can be extracted from the grammars inside Θ_{ℓ_k}.

Back to our example, applying the function TRJ to an element $\hat{v}$ of Θ_{ℓ_5} gives the trajectory $\ell_2 \cdot \ell_4 \cdot \ell_5 \cdot v$. The requirement that a user can only access a service within his geographic area is therefore detected.

5 Conclusions

We have introduced a static analysis, technically a contol flow analysis, for a variant of KLAIM that provides an abstract simulation model that tracks the propagation of tuples and identifies their possible trajectories within a KLAIM net. We have illustrated our approach on a microservice-based software architecture, showing that one can detect when a datum can safely traverse a path in the network, and when passing through a specific node may be dangerous. Our variant of KLAIM includes no primitive mechanism for code mobility, e.g. the **eval** action, which however can be managed with some additional technicalities. As future work, we intend to study when nodes continue to behave in a reasonable way even in the presence of not completely reliable data, by linking our approach to that in [21]. There, the authors use the Quality Calculus to program software components with a sort of backup plan in case of partly unreliable communication or data. Finally, we plan to consider one of the available implementations of the KLAIM model, e.g. [1,2], to instrument them with our static analysis and to perform experimental evaluation on some case studies.

Related Work. Several verification techniques have been defined for KLAIM and its variants. An important effort has been devoted to exploit behavioural type systems for security [12,14,19]. By exploiting static and dynamic checks, type checking guarantees that only those processes are allowed to proceed, the intentions of which match the rights granted to them. An expressive language extension, METAKLAIM [16] integrates METAML (an extension of SML for multi-stage programming) and KLAIM, to permit interleaving of meta-programming activities (such as assembly and linking of code fragments), dynamic checking of security policies at administrative boundaries, and traditional computational activities. METAKLAIM exploits a powerful type system (including polymorphic types *á la* system F) to deal with highly parameterised mobile components and to enforce security policies dynamically: types are metadata that are extracted from code at run-time and are used to express trustiness guarantees. The dynamic type checking ensures that the trustiness guarantees of wide area network applications are maintained also when computations interoperate with potentially untrusted components.

A framework based on temporal logic [10] has been developed for specifying and verifying dynamic properties of mobile processes specified in KLAIM. This framework provides support for establishing deadlock freedom and liveness properties as well as security properties such as resource access and information disclosure. A different approach to control accesses to tuple spaces and mobility of processes is introduced in [13]. Like ours, this approach is based on Flow Logic (so also enabling to design a fully static type system) and considers a version of KLAIM slightly different from ours. The abstract domains differ, because theirs contain tuples only made by localities, while ours also have values. Since access control is of interest, their domains also record possible policies and violations.

References

1. Bettini, L., De Nicola, R., Pugliese, R.: KLAVA: a Java package for distributed and mobile applications. Softw. Pract. Exper. **32**(14), 1365–1394 (2002)
2. Bettini, L., De Nicola, R., Pugliese, R., Ferrari, G.L.: Interactive mobile agents in X-Klaim. In: Proceedings of 7th Workshop on Enabling Technologies (WETICE 1998), Infrastructure for Collaborative Enterprises, pp. 110–117. IEEE Computer Society (1998)
3. Bodei, C., Degano, P., Ferrari, G.L., Galletta, L.: A step towards checking security in IoT. In: Proceedings of ICE 2016, EPTCS, vol. 223, pp. 128–142 (2016)
4. Bodei, C., Degano, P., Ferrari, G.-L., Galletta, L.: Where do your IoT ingredients come from? In: Lluch Lafuente, A., Proença, J. (eds.) COORDINATION 2016. LNCS, vol. 9686, pp. 35–50. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-39519-7_3
5. Bodei, C., Degano, P., Ferrari, G.L., Galletta, L.: Tracing where IoT data are collected and aggregated. Log. Methods Comput. Sci. **13**(3), 1–38 (2017)
6. Bodei, C., Galletta, L.: Tracking data trajectories in IoT. In: Mori, P., Furnell, S., Camp, O. (eds.) Proceedings of the 5th International Conference on Information Systems Security and Privacy, ICISSP, vol. 1, pp. 572–579. SCITEPRESS (2019). https://doi.org/10.5220/0007578305720579, ISBN 978-989-758-359-9

7. Castellani, S., Ciancarini, P., Rossi, D.: The ShaPE of ShaDE: a coordination system. Technical report UBLCS 96-5, Dip. di Scienze dell'Informazione, Univ. Bologna (1996)
8. Comon, H., et al.: Tree automata techniques and applications. http://www.grappa.univ-lille3.fr/tata (2007). Released 12 Oct 2007
9. Davies, N., Wade, S., Friday, A., Blair, G.: L^2imbo: a tuple space based platform for adaptive mobile applications. In: Rolia, J., Slonim, J., Botsford, J. (eds.) ICODP/ICDP. IFIPAICT, pp. 291–302. Springer, Boston (1997). https://doi.org/10.1007/978-0-387-35188-9_22
10. De Nicola, R., Loreti, M.: A modal logic for mobile agents. ACM Trans. Comput. Log. **5**(1), 79–128 (2004)
11. 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)
12. De Nicola, R., Ferrari, G.L., Pugliese, R., Venneri, B.: Types for access control. Theor. Comput. Sci. **240**(1), 215–254 (2000)
13. De Nicola, R., et al.: From flow logic to static type systems for coordination languages. Sci. Comput. Program. **75**(6), 376–397 (2010)
14. De Nicola, R., Gorla, D., Pugliese, R.: Confining data and processes in global computing applications. Sci. Comput. Program. **63**(1), 57–87 (2006)
15. Deugo, D.: Choosing a mobile agent messaging model. In: ISADS, pp. 278–286. IEEE (2001)
16. Ferrari, G.L., Moggi, E., Pugliese, R.: MetaKlaim: a type safe multi-stage language for global computing. Math. Struct. Comput. Sci. **14**(3), 367–395 (2004)
17. Gelernter, D.: Generative communication in Linda. ACM Trans. Program. Lang. Syst. **7**(1), 80–112 (1985)
18. Gelernter, D.: Multiple tuple spaces in Linda. In: Odijk, E., Rem, M., Syre, J.-C. (eds.) PARLE 1989. LNCS, vol. 366, pp. 20–27. Springer, Heidelberg (1989). https://doi.org/10.1007/3-540-51285-3_30
19. Gorla, D., Pugliese, R.: Dynamic management of capabilities in a network aware coordination language. J. Log. Algebraic Program. **78**(8), 665–689 (2009)
20. Nielson, F., Nielson, H.R., Seidl, H.: A succinct solver for ALFP. Nordic J. Comput. **9**(4), 335–372 (2002)
21. Nielson, H.R., Nielson, F., Vigo, R.: A calculus of quality for robustness against unreliable communication. J. Log. Algebraic Meth. Program. **84**(5), 611–639 (2015)