A Formal Approach to the Engineering of Domain-Specific Distributed Systems

Rocco De Nicola, Gianluigi Ferrari, Rosario Pugliese, Francesco Tiezzi

A Formal Approach to the Engineering of Domain-Specific Distributed Systems

Rocco De Nicola[1], Gianluigi Ferrari[2], Rosario Pugliese[3], and Francesco Tiezzi[4]

[1] IMT Institute for Advanced Studies Lucca, Italy
[2] Università di Pisa, Italy
[3] Università degli Studi di Firenze, Italy
[4] Università di Camerino, Italy

Abstract. We review some results regarding specification, programming and verification of different classes of distributed systems which stemmed from the research of the Concurrency and Mobility Group at University of Firenze. More specifically, we review distinguishing features of network-aware programming, service-oriented computing, autonomic computing, and collective adaptive systems programming. We then present an overview of four different languages, namely KLAIM, COWS, SCEL and AbC. For each language, we discuss design choices, present syntax and informal semantics, show some illustrative examples, and describe programming environments and verification techniques.

1 Introduction

Since the mid-90s, we have witnessed an evolution of distributed computing towards increasingly complex systems formed by several software components featuring asynchronous interactions, and operating in open-ended and non-deterministic environments. Such transformation, initially induced by the spreading of internetworking technologies, led to a paradigm shift making software components *aware* of the underlying network infrastructure. Such awareness, on the one hand, constrained the remote access to distributed resources and, on the other hand, enabled computation mobility, to support different kinds of optimisations.

On top of these networked systems, software components have been then deployed to provide *services* accessible by end-users and other system components. This fostered the development of sophisticated applications built by reusing and composing simpler elements. Such service-based compositional approach required to overcome the interaction challenges posed by the heterogeneity of the involved components; interoperability was then achieved through the definition of standard protocols and suitable run-time support for programming languages.

Later on, the need arose of reducing the maintenance cost of these systems, whose size was becoming bigger and bigger, and of extending their applicability to interact with and control the physical world, possibly in scenarios where human intervention was difficult or even impossible. It was then advocated to rely on *autonomic* components, which are capable of continuously monitoring their internal status and the working environment, and to adapt their behaviour accordingly.

More recently, a growing interest emerged in a new class of computational systems consisting of a large number of interacting components featuring complex behaviour that are usually distributed, heterogeneous, decentralised and interdependent, and operate in dynamic and possibly unpredictable environments. The components form *collectives* by combining their behaviours to achieve specific goals, or to contribute to an emerging behaviour of the global system. Collectives abstract from the identity of the single components to guarantee scalability.

The evolution of distributed computing described above corresponds to the emergence of classes of systems that characterise specific programming domains. Correspondingly, dedicated programming paradigms have been proposed, namely *network-aware programming*, *service-oriented computing*, *autonomic computing*, and *collective adaptive systems programming*. Besides dealing with the distinctive aspects of each of such domains, the main challenge in engineering these classes of distributed systems is to coordinate the overall behaviour resulting from the involved components while ensuring trustworthiness of the whole system. To meet this goal, many researchers have adopted language-based approaches that combine the use of formal methods techniques with model-driven software engineering. The key ingredients of the resulting methodology can be summarised as follows:

1. a specification language equipped with a formal semantics, which associates mathematical models to each term of the language to precisely establish the expected behaviour of systems;
2. a set of techniques and tools, built on top of the models, to express and verify properties of interest for the considered class of systems;
3. a programming framework together with the associated runtime environment to actually execute the specified systems.

When specialising this methodology, a major challenge for (specification or programming) language designers is to devise appropriate abstractions and linguistic primitives to deal with the specificities of the domain under investigation. Indeed, including the distinctive aspects of the domain as first-class elements of the language makes systems design more intuitive and concise, and their analysis more effective. In fact, when the outcome of a verification activity is expressed by considering the high level features of a system, and not its low-level representation, system designers can be provided with more direct feedbacks.

This paper reviews some of the efforts, to which the authors have contributed, in applying the outlined methodology to the classes of distributed systems mentioned above by taking as starting point process algebras and some of the verification techniques and tools developed for them. The approach was initially applied to network-aware programming and the main result was the definition of the KLAIM language [23] (Section 2). Afterwards, the approach was applied to service-oriented computing resulting in the design of COWS [57] (Section 3), to autonomic computing obtaining as a result SCEL [27] (Section 4), and to collective adaptive systems programming to obtain AbC [2] (Section 5). For each of these domain-specific languages, we discuss design choices, present syntax and informal semantics, show some simple but illustrative example specifications, and describe programming environments and verification

techniques. We want to stress that all languages have been equipped with a formal operational semantics, based on labelled transition systems, that is omitted here for the sake of space; the interested reader is referred to the relevant papers in the bibliography. Moreover, for the sake of readability and understandability, the examples are presented at the level of the specification language; of course they can be refined in order to be implemented by means of the proposed programming environments, but currently they are not. The paper ends with a summary of distinguishing features of the presented languages and with a few considerations about the lessons learnt (Section 6).

2 KLAIM: a Kernel language for Agents Interaction and Mobility

Network awareness indicates the ability of the software components of a distributed application to manage directly 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. This is of great importance when programming mobile components capable of disconnecting from one node of the underlying infrastructure and of reconnecting to a different node. Programmers are usually supported with primitive constructs that enable components to communicate, distribute and retrieve data to and from the nodes of the underlying infrastructure.

KLAIM (*Kernel Language for Agents Interaction and Mobility*, [23]) has been specifically devised to design distributed applications consisting of several components (both stationary and mobile) deployed over the nodes of a distributed infrastructure. The KLAIM programming model relies on a unique interface (i.e. set of operations) supporting component communications and data management.

Localities are the basic building blocks of KLAIM for guaranteeing network awareness. They are symbolic addresses (i.e. network references) of nodes and are referred by means of identifiers. Localities can be exchanged among the computational components and are subjected 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 locality are under the control of a specific authority. This way, localities naturally support programming spatially distributed applications.

KLAIM builds on Linda's notion of *generative communication* through a single shared tuple space [36] and generalizes it to 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., [37, 19, 14, 31]). It supports *time uncoupling* (data life time is independent of the producer process life time), *destination uncoupling* (the producer of a datum does not need to know the future use or the final destination of that datum) and *space uncoupling* (programmers need to know a single interface only to operate over the tuple spaces, regardless of the network node where the action will take place).

NETS:		
N	$::= \; l ::_\rho P$	(computational node)
	$\mid \; l :: \langle et \rangle$	(located tuple)
	$\mid \; N_1 \parallel N_2$	(net composition)
PROCESSES:		
P	$::= \; \mathbf{nil}$	(null process)
	$\mid \; a.P$	(action prefixing)
	$\mid \; P_1 \mid P_2$	(parallel composition)
	$\mid \; X$	(process variable)
	$\mid \; A$	(process invocation)
ACTIONS:		
a	$::= \; \mathbf{out}(t)@\ell$	(output)
	$\mid \; \mathbf{in}(T)@\ell$	(input)
	$\mid \; \mathbf{read}(T)@\ell$	(read)
	$\mid \; \mathbf{eval}(P)@\ell$	(migration)
	$\mid \; \mathbf{newloc}(u)$	(creation)

TUPLES:	
t	$::= \; f \mid f, t$
TUPLE FIELDS:	
f	$::= \; e \mid \ell \mid u \mid P$
EVALUATED TUPLES:	
et	$::= \; ef \mid ef, et$
EVALUATED TUPLE FIELDS:	
ef	$::= \; V \mid l \mid P$
TEMPLATES:	
T	$::= \; F \mid F, T$
TEMPLATE FIELDS:	
F	$::= \; f \mid !\,x \mid !\,u \mid !\,X$
EXPRESSIONS:	
e	$::= \; V \mid x \mid \ldots$

Table 1. Klaim syntax

2.1 Syntax

The syntax of KLAIM is presented in Table 1. We assume existence of two disjoint sets: the set of *localities*, ranged over by l, and the set of *locality variables*, ranged over by u. Their union gives the set of *names*, ranged over by ℓ. We also assume three other disjoint sets: a set of *value variables*, ranged over by x, a set of *process variables*, ranged over by X, and a set of *process identifiers*, ranged over by A.

NETS are finite collections of nodes where processes and data can be placed. A *computational node* takes the form $l ::_\rho P$, where ρ is an *allocation environment* and P is a process. Since processes may refer to locality variables, the allocation environment acts as a *name solver* binding locality variables to specific localities.

PROCESSES are the active computational units of KLAIM. Their syntax is standard and specifies the ACTIONS to be executed. Recursive behaviours are modelled via process definitions; it is assumed that each identifier A has a *single* defining equation $A \triangleq P$.

The tuple space of a node consists of all the EVALUATED TUPLES located there. TUPLES are sequences of *actual* fields, i.e. expressions, localities or locality variables, or processes. The precise syntax of EXPRESSIONS is deliberately not specified; it is just assumed that they contain, at least, basic values, ranged over by V, and variables, ranged over by x. TEMPLATES are sequences of actual and formal fields, and are used as patterns to select tuples in a tuple space. *Formal* fields are identified by the !-tag (e.g. $!\,x$) and are used to bind variables to values.

2.2 Informal semantics

NETS aggregate nodes through the *composition* operator $\_ \parallel \_$, which is both commutative and associative. PROCESSES are concurrently executed in an *interleaving* fashion, either at the same computational node or at different nodes. They can perform operations borrowed from a unique interface which provides two categories of actions. The

first one consists of the programming abstractions supporting 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 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. The second category of actions refers to network awareness: the migration action (**eval**) activates a new process over a network node, while the creation action (**newloc**) generates a new network node. The latter action is the only one not indexed by a locality because it acts locally; all the other actions are tagged with the (possibly remote) locality 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 [34].

Only evaluated tuples can be added to a tuple space and templates must be evaluated before they can be used for retrieving tuples. Tuple and template evaluation amounts to computing the values of their expressions. Localities and formal fields are left unchanged by such evaluation.

A *pattern-matching* mechanism is used for associatively selecting (evaluated) tuples from tuple spaces according to (evaluated) templates. 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.

Process variables support *higher-order* communication, namely the capability to exchange (the code of) a process and possibly execute it. This is realised 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.

Finally, KLAIM offers two forms of process mobility. One is based on *static scoping*: by exploiting higher-order communication, a process moves along the nodes of a net with a fixed binding of resources determined by the allocation environments of the nodes from where, from time to time, it is going to move. The other form of mobility relies on *dynamic scoping*: when migrating, a process breaks the local links to resources and inherits those of the destination node.

2.3 Example: a street light controller

We outline here the main features of the design of a (simplified) *Street Light Controller* working on a one-way street, inside a restricted traffic zone. It consists of several integrated components. Smart lamp post components are cyber-physical entities (battery powered). They can sense their surrounding environment and can communicate with their neighbours to share information. For instance if (a sensor of) the lamp post perceives a pedestrian and there is not enough light in the street it turns its light on and communicates the presence of the pedestrian to the lamp posts nearby. A further component of the street light controller uses the information provided by the electronic access point to the street. When a car crosses the checkpoint, a message is sent to the supervisor of the street accesses, that in turn notifies the presence of the car to a further

component of the system: the supervisor of the street. A notice is also sent to the node that hosts the cloud service of the police department. This service checks whether the car is enabled to enter that restricted zone, through automatic number plate recognition. The street supervisor, as a result of this coordinated behaviour, is in charge of sending the authorisation message to the lamp post closest to the checkpoint that starts a forward chain till the end of the street, thus completing the overall cooperative behaviour. For simplicity, here we assume that each sensor has a unique name and the sensed values are modelled as tuples containing the name of the sensor and the detected value. Since every cyber-physical node has a fixed number of sensors, the tuple space of the node is designated to store the values read by sensors.

The process running at checkpoint node is the driver of the visual sensor S_{cp}, defined below. The driver takes a picture of the car detected in the street and stores it in the tuple space:

$$S_{cp} \triangleq \mathbf{in}(probe, !v)@\mathsf{self}.\mathbf{out}(picture, v)@\mathsf{self}.S_{cp}$$

where $probe$ is the unique identifier of the sensor and the tuple tagged by $picture$ identifies the collected picture of the car. Then, the picture is enhanced (by using the function $noiseRed$ for reducing noise) by the process P_{cp} and sent to the supervisor:

$$P_{cp} \triangleq \mathbf{in}(picture, !z)@\mathsf{self}.\mathbf{out}(enPicture, noiseRed(z))@controller.P_{cp}$$

The checkpoint node N_{cp} is defined as $l_{cp} ::_{\rho_{cp}} (P_{cp} \mid S_{cp} \mid B_{cp})$, where ρ_{cp} is the allocation environment binding the locality variable $controller$ to the locality l_a where the access controller node is deployed, and B_{cp} abstracts other components we are not interested in, among which the tuple space at l_{cp}. The access controller node N_a receives the picture and communicates the presence of the car to the lamp posts supervisor and to the police department. The behaviour of the driver process running at node N_a is as follows

$$P \triangleq \mathbf{in}(enPicture, !x)@\mathsf{self}.\mathbf{out}(car, x)@supervisor.\mathbf{out}(car, x)@pdept.P$$

and the node is defined as $l_a ::_{\rho_a} (P \mid B_a)$, where ρ_a binds the locality variable $supervisor$ and $pdept$ to the localities where the street supervisor and the police department are deployed. The process B_a abstracts other components we are not interested in, among which the tuple space at l_a. The supervisor node N_s contains the process P_s that receives the picture from N_a and sends a message to the lamp node closest to the checkpoint; its behaviour is straightforward.

In our smart street light control system there is a node N_p for each lamp post. Each lamp post is equipped with four sensors to sense (1) the environment light, (2) the solar light, (3) the battery level and (4) the presence of a pedestrian. These sensors are the interface towards the cyber-physical world and their asynchronous behaviour simply inserts the acquired information in the tuple space of the node. The drivers of sensors share the same structure; hence we only show that for the battery level:

$$S_{battery} \triangleq \mathbf{in}(probeBatteryLevel, !v)@\mathsf{self}.\mathbf{out}(batteryLevel, v)@\mathsf{self}.S_{battery}$$

The control process reads the current values from the sensors and stores the resulting values in a local tuple consisting of four terms, i.e. environment light, solar light,

battery level and presence of pedestrian, by means of the action $\mathbf{out}(el, sl, bl, p)@\mathsf{self}$. Action $\mathbf{read}(!el, !sl, !bl, !p)@\mathsf{self}$ is used to access such information in order to detect the actual state of affair: *(i)* a pedestrian is in the street ($p = true$), *(ii)* the intensity of environment and solar lights are greater than, or equal to, the given thresholds, $el \geq th_1$ and $sl \geq th_2$, and *(iii)* there is enough battery (at least $bl \geq th_3$). The presence of the pedestrian is communicated to the lamp posts nearby, whose locality is obtained from the allocation environment ($\mathbf{out}(pedestrian, p)@next$). In case the battery level is insufficient, an error message is sent to the supervisor node ($\mathbf{out}(failure)@supervisor$).

The overall intelligent controller of the street lights is then described as the parallel composition of the checkpoint node N_{cp}, the supervisor nodes N_a and N_s, the nodes of lamp posts N_p, with $p \in [1, k]$, and the police department node N_{pd}:

$$N_{cp} \parallel N_a \parallel N_s \parallel N_1 \parallel \cdots \parallel N_k \parallel N_{pd}$$

2.4 Programming environment

X-KLAIM (*eXtended* KLAIM, [11]) is an experimental programming language that extends KLAIM with a high level syntax for processes. It provides variable declarations, enriched operations, assignments, conditionals, sequential and iterative process composition. The implementation of X-KLAIM is based on KLAVA[5] (KLAIM *in Java*, [12]), a Java package that provides the run-time system for X-KLAIM operations, and on a compiler, which translates X-KLAIM programs into Java programs that use KLAVA. X-KLAIM can be used to write the higher layer of distributed applications while KLAVA can be seen both as a middleware for X-KLAIM programs and as a Java framework for programming according to the KLAIM paradigm. By using KLAVA directly, the programmer is able to exchange, through tuples, any kind of Java object, and to implement a finer grained type of mobility.

2.5 Verification techniques

Many verification techniques have been defined for KLAIM and variants thereof. Due to lack of space, here we only mention a few of them. In [26] a temporal logics is proposed for specifying and verifying dynamic properties of mobile processes specified in KLAIM. The inspiration for the proposal was the Hennessy-Milner Logics, but it needed significant adaptations due to the richer operating context of components. The resulting logic provides tools for establishing not only deadlock freedom, liveness and correctness with respect to given specifications (which are crucial properties for process calculi and similar formalisms), but also properties concerned with resource allocation, resource access and information disclosure (which are important issues for processes involving different actors and authorities).

An important topic deeply investigated for KLAIM is the use of type systems for security [24, 25, 40], devoted to control accesses to tuple spaces and mobility of processes. In these type systems, traditional types are generalised to *behavioural types*. These are

[5] X-KLAIM and KLAVA are available on line at http://music.dsi.unifi.it.

abstractions of process behaviours that provide information about *processes capabilities*, namely the operations that processes can execute at a specific locality (downloading/consuming a tuple, producing a tuple, activating a process, and creating a new node). When using behavioural types, each KLAIM node is equipped with a security policy, determined by a net coordinator, that specifies the execution privileges; the policy of a node describes the actions processes there located can execute. By exploiting static and dynamic checks, type checking guarantees that only processes whose intentions match the rights granted to them by coordinators are allowed to proceed.

An alternative approach to control accesses to tuple spaces and mobility of processes is introduced in [28]. It is based on Flow Logic and permits statically checking absence of violations. Starting from an existing type system for KLAIM with some dynamic checks, the insights from the Flow Logic approach are exploited to construct a type system for statically guaranteeing secure access to tuple spaces and safe process migration for a smooth extension of KLAIM. This is the first completely static type system for controlling accesses devised for a tuple space-based coordination language.

Finally, an expressive language extension, called METAKLAIM, and a powerful type system are described in [34]. METAKLAIM is a higher order distributed process calculus equipped with staging mechanisms. It 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 on a wide area network (such as remote communication and code mobility). 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.

3 COWS: Calculus for Orchestration of Web Services

Since the early 2000s, the increasing success of e-business, e-learning, e-government, and other similar systems, has led the World Wide Web, initially thought of as a system for human use, to evolve towards an architecture for Service-Oriented Computing (SOC) supporting automated use. The SOC paradigm, that finds its origin in object-oriented and component-based software development, aims at enabling developers to build networks of distributed, interoperable and collaborative applications, regardless of the platform where the applications run and of the programming language used to develop them. The paradigm is based on the use of independent computational units, called *services*. They are loosely coupled reusable components, that are built with little or no knowledge about clients and about other services involved in their operating environment.

One successful instantiation of the general SOC paradigm is given by the Web Service technology [62], which exploits the pervasiveness of the Internet and related standards. Traditional software engineering technologies, however, do not neatly fit with

SERVICES:			RECEIVE-GUARDED CHOICE:		
s	$::= u \bullet u'!\bar{\epsilon}$	(invoke)	g	$::= \mathbf{0}$	(nil)
	$\mid \mathbf{kill}(k)$	(kill)		$\mid p \bullet o?\bar{w}.s$	(request processing)
	$\mid g$	(receive-guarded choice)		$\mid g + g$	(choice)
	$\mid s \mid s$	(parallel composition)			
	$\mid \{\!\mid s \mid\!\}$	(protection)			
	$\mid [e]\, s$	(delimitation)			
	$\mid * s$	(replication)			

Table 2. COWS syntax

SOC, thus hindering its full realisation in practice. The challenges come from the necessity of dealing at once with such issues as asynchronous interactions, concurrent activities, workflow coordination, business transactions, resource usage, and security, in a setting where demands and guarantees can be very different for the many involved components.

COWS (*Calculus for Orchestration of Web Services*, [44, 57]) is a process calculus whose design has been influenced by the OASIS standard WS-BPEL [54] for orchestration of web services. In COWS, *services* are computational entities capable of generating multiple instances to concurrently handle different client requests. Inter-service communication occurs through *communication endpoints* and relies on pattern-matching for logically correlating messages to form an interaction session by means of their identical contents. Differently from most process calculi, receive activities in COWS bind neither names nor variables, and this is crucial for allowing concurrent service instances to share (part of) the state. The calculus also supports service fault and termination handling by providing activities to force termination of labelled service instances and to protect service activities from a forced termination.

3.1 Syntax

The syntax of COWS is presented in Table 2. We use three countable disjoint sets: the set of *values* (ranged over by v), the set of 'write once' *variables* (ranged over by x), and set of *killer labels* (ranged over by k). The set of values is left unspecified; however, we assume that it includes the set of partner and operation *names* (ranged over by n, p, o) mainly used to represent communication endpoints. We also use a set of *expressions* (ranged over by ϵ), whose exact syntax is deliberately omitted; we just assume that expressions contain values and variables, and do not contain killer labels. As a matter of notation, w ranges over values and variables, u ranges over names and variables, and e ranges over *elements*, i.e. killer labels, names and variables. Notation $\bar{\cdot}$ stands for tuples, e.g. $\bar{x}$ means $\langle x_1, \ldots, x_n \rangle$ (with $n \geq 0$), where variables in the same tuple are all distinct.

Services are structured activities built from basic activities, i.e. the empty activity $\mathbf{0}$, the invoke activity $\_ \bullet \_!\_$, the receive activity $\_ \bullet \_?\_$, and the kill activity $\mathbf{kill}(\_)$, by means of prefixing $\_.\_$, choice $\_ + \_$, parallel composition $\_ \mid \_$, protection $\{\!\mid \_ \mid\!\}$, delimitation $[\_]\,\_$ and replication $* \_$. We write $I \triangleq s$ to assign a name I to the term s.

3.2 Informal semantics

Invoke and *receive* are the communication activities. The former permits invoking an operation (i.e., a functionality like a method in object-oriented programming) offered by a service, while the latter permits waiting for an invocation to arrive. Besides output and input parameters, both activities indicate an endpoint through which communication should occur.

An *endpoint* $p \bullet o$ can be interpreted as a specific implementation of operation o provided by the service identified by the logic name p. The names composing an endpoint can be dealt with separately, as in an asynchronous request-response interaction, where usually the service provider statically knows the name of the operation for sending the response, but not the partner name of the requesting service it has to reply to. Partner and operation names can be exchanged in communication, thus enabling many different interaction patterns among service instances. However, dynamically received names cannot form the endpoints used to receive further invocations (as in *localised* π*-calculus* [51]). In other words, endpoints of receive activities are identified statically because the syntax only allows using names and not variables for them. This design choice reflects the current (web) service technologies that require endpoints of receive activities be statically determined.

An invoke $p \bullet o!\langle \epsilon_1, \ldots, \epsilon_n \rangle$ can proceed as soon as all expression arguments are successfully evaluated. A receive $p \bullet o?\langle w_1, \ldots, w_n \rangle.s$ offers an invocable operation o along with a given partner name p, thereafter the service continues as s. An inter-service communication between these two activities takes place when the tuple of values $\langle v_1, \ldots, v_n \rangle$, resulting from the evaluation of the invoke argument, matches the template $\langle w_1, \ldots, w_n \rangle$ argument of the receive. This causes a substitution of the variables in the receive template (within the scope of variables declarations) with the corresponding values produced by the invoke.

Communication is asynchronous, as in KLAIM. This results from the syntactic constraints that invoke activities cannot be used as prefixes and choice can only be guarded by receive activities (as in *asynchronous* π*-calculus* [6]). Indeed, in service-oriented systems, communication is usually asynchronous, in the sense that (i) there may be an arbitrary delay between the sending and the receiving of a message, (ii) the order in which messages are received may differ from that in which they were sent (iii) a sender cannot determine if and when a sent message will be received.

The *empty* activity does nothing, while *choice* permits selecting for execution one between two alternative receives.

Execution of *parallel* services is interleaved. However, if more matching receives are ready to process a given invoke, only one of the receives that generate a substitution with smallest size (in terms of number of variable-value replacements) is allowed to progress (namely, execution of this receive takes precedence over that of the others). This mechanism permits to model the precedence of a service instance over the corresponding service specification when both of them can process the same request (see [57] for detailed examples), and enables a sort of blind-date conversation joining strategy [16].

Delimitation is the only binding construct: $[e]\, s$ binds the element e in the scope s. According to its first argument, delimitation is used for three different purposes: (i) to

regulate the range of application of substitutions produced by communication, when the delimited element is a variable; (ii) to generate fresh names, when the delimited element is a name; (iii) to confine the effect of a kill activity, when the delimited element is a killer label. The scope of names can be dynamically extended, in order to model the communication of private names, as done with the restriction operator in π-calculus [52]. Instead, killer labels cannot be dynamically extended, because the activities whose termination would be forced by the execution of a kill need to be statically determined.

The *kill* activity forces immediate termination of all the concurrent activities not enclosed within the *protection* operator. To faithfully model fault and termination handling of SOC applications, kill activities are executed eagerly with respect to the communication activities enclosed within the delimitation of the corresponding killer label.

Finally, the *replication* construct $* s$ permits to spawn in parallel as many copies of s as necessary. This, for example, is exploited to implement recursive behaviours and to model business process definitions, which can create multiple instances to serve several requests simultaneously.

3.3 Example: a travel agency scenario

We report here a few examples aimed at illustrating the main COWS features. We consider a typical SOC scenario, where a travel agency exposes a service to automatically book a hotel and a flight according to customers' requests.

At a high level of abstraction, the travel agency service is rendered in COWS as:

$$\mathit{TravelAgency} \triangleq * [x_{cust}, x_{dates}, x_{dest}]\, p_{ta} \bullet o_{req}?\langle x_{cust}, x_{dates}, x_{dest}\rangle. \\ x_{cust} \bullet o_{resp}!\langle \mathit{book}(x_{dates}, x_{dest})\rangle$$

The replication operator $*$ is used here to specify that the service is *persistent*, i.e. capable of creating multiple instances to serve several requests simultaneously. The delimitation operator specifies the scope of the variables arguments of the subsequent receive activity on operation o_{req}, used to receive a request message from a customer. Besides dates and destination of the travel, this message contains the partner name that the customer will use to receive the response, which will be sent by the service by means of the invoke activity on operation o_{resp}. Booking of hotel and flight is here abstracted by the (unspecified) expression $\mathit{book}(x_{dates}, x_{dest})$.

A customer of the travel agency is specified as follows:

$$\mathit{Customer} \triangleq p_{ta} \bullet o_{req}!\langle p_c, v_{dates}, v_{dest}\rangle \mid [x_{travel}]\, p_c \bullet o_{resp}?\langle x_{travel}\rangle.s$$

The customer behaviour is specular to that of the travel agency: it starts with an invoke and then waits for a response message containing the travel data.

The overall specification of the scenario is simply the parallel composition of the two components: $(\mathit{Customer} \mid \mathit{TravelAgency})$. Whenever prompted by a client request, the travel agency service creates an instance to serve that specific request, and is immediately ready to concurrently serve other possible requests. Therefore, the resulting COWS term after such a computational step is the following:

$$[x_{travel}]\, p_c \bullet o_{resp}?\langle x_{travel}\rangle.s \mid \mathit{TravelAgency} \mid p_c \bullet o_{resp}!\langle \mathit{book}(v_{dates}, v_{dest})\rangle$$

The created service instance (highlighted by a grey background) is represented as a service running in parallel with the other terms. Notably, the variables of the invoke activity are instantiated (i.e., replaced) by the corresponding values exchanged in the communication. This invoke activity can now synchronise with the receive activity of the customer, whose execution will then continue as s with x_{travel} replaced by the value resulting from the evaluation of the $book$ expression.

Let us now consider a more refined specification, where the role of the $book$ expression is played by the interactions with services for flights and hotels searching:

$$
\begin{aligned}
\mathit{TravelAgency}' \triangleq * [x_{cust}, x_{dates}, x_{dest}]\, p_{ta} \bullet o_{req}?\langle x_{cust}, x_{dates}, x_{dest}\rangle.\\
[p, o, x_{\mathit{flight}}, x_{hotel}]\\
(\ (p_{\mathit{flight}} \bullet o_{book}!\langle p_{ta}, x_{cust}, x_{dates}, x_{dest}\rangle\\
\quad \mid\ p_{ta} \bullet o_{fRes}?\langle x_{cust}, x_{dates}, x_{dest}, x_{\mathit{flight}}\rangle.\ (p \bullet o!\langle end\rangle \mid s_f))\\
\mid\ (p_{hotel} \bullet o_{book}!\langle p_{ta}, x_{cust}, x_{dates}, x_{dest}\rangle\\
\quad \mid\ p_{ta} \bullet o_{hRes}?\langle x_{cust}, x_{dates}, x_{dest}, x_{hotel}\rangle.\ (p \bullet o!\langle end\rangle \mid s_h))\\
\mid\ p \bullet o?\langle end\rangle.\ p \bullet o?\langle end\rangle.\ x_{cust} \bullet o_{resp}!\langle x_{\mathit{flight}}, x_{hotel}\rangle\ )
\end{aligned}
$$

After the reception of a customer request, the service contacts in parallel the two searching services (by invoking the operation o_{book}). When the responses from both services are available, the travel agency service combines them and replies to the customer. To this aim, a private endpoint $p \bullet o$ is exploited: the reception of a message from a searching service triggers an end signal (i.e., an internal message) along the private endpoint, and two of such signals are necessary to trigger the invoke the activity for replying to the customer. Notice that the scope of variable x_{flight} (resp. x_{hotel}) includes not only the continuation s_f (resp. s_h) of the service performing the receive, but also the activity for sending the response to the customer. This is different from most process calculi and accounts for easily expressing variables shared among parallel activities within the same service instance, which is a feature typically supported in SOC.

The behaviour of the above service is of particular interest when it is included in a scenario with multiple customers (the specifications of customers and searching services are omitted, we just assume that they follow the communication protocol established by the travel agency specification):

$$\mathit{Customer}_1 \mid \mathit{Customer}_2 \mid \mathit{TravelAgency}' \mid \mathit{FlightBooking} \mid \mathit{HotelBooking}$$

After a certain number of computational steps have taken place, we can obtain a system configuration where one instance of the travel agency service is created per each customer, and both instances have sent their requests to the searching services and are waiting for replies. Now, to send the values resulting from the processing of the request of the first customer, the flight searching service has to perform an invoke activity of the form $p_{ta} \bullet o_{fRes}!\langle p_{c1}, v_{dates}, v_{dest}, v_{\mathit{flight}}\rangle$. However, the travel agency service has two instances waiting for such message along the endpoint $p_{ta} \bullet o_{fRes}$. In order to deliver the message to the proper instance, i.e. the one serving the request of the first customer, the *message correlation* mechanism is used. In fact, in SOC, it is up to each single message to provide a form of context that enables services to associate the message with the appropriate instance. This is achieved by embedding values, called *correlation data*, in the message itself. Pattern-matching is the mechanism used by the COWS's semantics for

locating correlation data. In our example, these data are the customer's partner name, the travel dates and the destination, which have instantiated the corresponding variables in the receive activity $p_{ta} \bullet o_{fRes}?\langle p_{c1}, v_{dates}, v_{dest}, x_{flight}\rangle$ within $Customer_1$. While the receive of the first customer is enabled, the one within the second customer instance is not, as it has been instantiated with unmatchable values.

Finally, let us provide further details of the travel agency specification, in order to add fault and compensation handling activities (highlighted by a grey background):

$$
\begin{aligned}
TravelAgency'' \triangleq\ & *\,[x_{cust}, x_{dates}, x_{dest}]\; p_{ta} \bullet o_{req}?\langle x_{cust}, x_{dates}, x_{dest}\rangle. \\
& [p, o, x_{flight}, x_{hotel}, \boxed{k}\,] \\
& (\ (p_{flight} \bullet o_{book}!\langle p_{ta}, x_{cust}, x_{dates}, x_{dest}\rangle \\
& \quad \mid\ p_{ta} \bullet o_{fRes}?\langle x_{cust}, x_{dates}, x_{dest}, x_{flight}\rangle. \\
& \qquad (p \bullet o!\langle end\rangle \mid s_f \\
& \qquad\quad \mid\ \boxed{\{\!|p \bullet o?\langle comp\rangle.\ p_{flight} \bullet o_{cancel}!\langle x_{cust}, x_{dates}, x_{dest}\rangle|\!\}}\ ) \\
& \qquad + p_{ta} \bullet o_{fFault}?\langle x_{cust}, x_{dates}, x_{dest}\rangle. \\
& \qquad\quad \boxed{(\mathbf{kill}(k) \mid \{\!|p \bullet o!\langle comp\rangle \mid p \bullet o!\langle fault\rangle|\!\})}\ ) \\
& \quad \mid\ (p_{hotel} \bullet o_{book}!\langle p_{ta}, x_{cust}, x_{dates}, x_{dest}\rangle \\
& \quad\quad \mid\ p_{ta} \bullet o_{hRes}?\langle x_{cust}, x_{dates}, x_{dest}, x_{hotel}\rangle. \\
& \qquad (p \bullet o!\langle end\rangle \mid s_h \\
& \qquad\quad \mid\ \boxed{\{\!|p \bullet o?\langle comp\rangle.\ p_{hotel} \bullet o_{cancel}!\langle x_{cust}, x_{dates}, x_{dest}\rangle|\!\}}\ ) \\
& \qquad + p_{ta} \bullet o_{hFault}?\langle x_{cust}, x_{dates}, x_{dest}\rangle. \\
& \qquad\quad \boxed{(\mathbf{kill}(k) \mid \{\!|p \bullet o!\langle comp\rangle \mid p \bullet o!\langle fault\rangle|\!\})}\ ) \\
& \quad \mid\ p \bullet o?\langle end\rangle.\ p \bullet o?\langle end\rangle.\ x_{cust} \bullet o_{resp}!\langle x_{flight}, x_{hotel}\rangle \\
& \quad \mid\ \boxed{\{\!|p \bullet o?\langle fault\rangle.\ x_{cust} \bullet o_{fault}!\langle\rangle|\!\}}\ \ )
\end{aligned}
$$

Now, when a positive response from a searching service is received, a compensation handler is installed. This consists of an invoke activity on operation o_{cancel}, triggered by a *comp* signal, devoted to cancel the booking. If a negative response on o_{fFault} (resp. o_{hFault}) is received, the normal execution of the service is immediately terminated (by means of the **kill** activity), the activity compensating the hotel (resp. flight) booking is activated, if installed, and a *fault* signal is emitted. This last signal triggers the execution of the fault handler, consisting of an invoke activity for notifying the customer that the request booking is failed. Notably, fault and compensation activities are enclosed within protection blocks, in order to protect them from the killing effect of the **kill** activities.

3.4 Programming environment

To effectively program SOC applications, COWS, originally conceived as a process calculus, has been extended with high-level features, such as standard control flow constructs (i.e., sequentialization, assignment, conditional choice, iteration) and a scope activity explicitly defining fault and compensation handlers. The implementation of the resulting orchestration language, called B*lite* [46], is based on a software tool [15] supporting a rapid and easy development of SOC applications via the translation of service orchestrations written in B*lite* into executable WS-BPEL programs. More specifically,

a B*lite* program given as input to this tool also includes a declarative part, containing the variable types and the physical service bindings, necessary for generating the corresponding WSDL document and the process deployment descriptor. These files, together with the one containing the WS-BPEL code, are organised in a package that can be deployed and executed in a WS-BPEL engine.

3.5 Verification techniques

The main verification techniques devised for COWS specifications are the following: (i) a type system for checking confidentiality properties [45], which uses types to express and enforce policies for regulating the exchange of data among services; (ii) a bisimulation-based observational semantics [58], which permits to check interchangeability of services and conformance against service specifications; (iii) a verification methodology for checking functional properties specific of SOC systems [33].

Concerning the third technique, the properties are described by means of SocL, a logic specifically designed to express in a convenient way distinctive aspects of services, such as, e.g., acceptance of a request, provision of a response, and correlation among service requests and responses. The verification of SocL formulae over COWS specifications is assisted by the on-the-fly model checker CMC. This approach has been used in [33, 49, 38] to verify some properties of interest of an automotive scenario, an e-Health authentication protocol, and a finance case study, respectively.

4 SCEL: Software Component Ensemble Language

Developing massively distributed and highly dynamic computing systems which interact with and control the physical world is a major challenge in todays software engineering. Difficulties arise from the open-ended and dynamic nature of large-scale systems, the non-deterministic and unpredictably changing external environment, the often limited or even impossible human intervention, and the need of ensembles of components to interact and collaborate for achieving specific goals, while hiding the complexity to end-users. A possible answer to the problems posed by such systems is to make them *self-aware*, by continuously monitoring their behaviour and their working environment, and able to *self-adapt* their behaviour or structure, by selecting the actions to perform for dealing with the current status of affairs. These and other *self-management* capabilities, like self-configuration, self-healing, self-optimisation, and self-protection, characterise *autonomic computing* [43] systems.

SCEL (*Software Component Ensemble Language*, [22, 27]) is a formal language providing a set of linguistic abstractions for specifying the behaviour of (autonomic) components, the interaction among them, and the formation of their ensembles. In SCEL, components are computational entities that have assigned dedicated knowledge repositories and behavioural policies. They also have an interface exposing characterising attributes. Ensembles, in turn, are aggregations of interacting partner components dynamically determined by means of predicates validated by each component on the basis of its attributes.

SCEL linguistic abstractions support programming self- and context-awareness, adaptation and autonomicity. Indeed, through the knowledge repositories, components can gain information on their status (self-awareness) and environment (context awareness). By exploiting awareness and higher-order features (i.e. the capability to store/retrieve processes in/from components knowledge repositories and to dynamically activate new processes), components can trigger self-adaptation and/or initiate self-healing actions for reacting to faults or activate optimization strategies by, e.g., including or replacing processes and other components. By integrating SCEL with suitable policy languages, it is possible to guarantee self-protection against, e.g., unauthorized accesses or denial-of-service attacks.

4.1 Syntax

SCEL syntax is reported in Table 3. Five countable disjoint sets are used: the set of *names* (ranged over by n, n', ...), the set of *predicate names* (ranged over by p, ...), the set of *variables for names* (ranged over by x, x', ...), the set of *variables for processes* (ranged over by $X, Y, \ldots$), and the set of parameterised *process identifiers* (ranged over by $A, \ldots$). self is a distinguished variable standing for the name of a component.

SYSTEMS result from the aggregation of COMPONENTS which, in turn, result from the aggregation of KNOWLEDGE and PROCESSES, according to some POLICIES. PROCESSES specify the flow of the ACTIONS that can be performed. ACTIONS can have a TARGET to determine the other components, in addition to the subject one, that are involved in that action.

SCEL is parametric with respect to some syntactic categories, namely POLICIES, KNOWLEDGE, TEMPLATES and ITEMS (with the last two determining the part of KNOWLEDGE to be retrieved/removed or added, respectively). This choice permits integrating different approaches to policy specification and knowledge handling within SCEL, like, e.g., the *access control policies* of [48] and the *constraint stores* of [53]. A simple, yet expressive, instance of SCEL, named SCELIGHT, has been introduced in [29] where policies are absent (equivalently, any process action is authorised) and knowledge repositories are implemented as tuple spaces *á la* KLAIM.

4.2 Informal semantics

SYSTEMS aggregate COMPONENTS through the *composition* operator $\_ \parallel \_$, which is both commutative and associative. It is also possible to restrict the scope of a name, say n, by using the name *restriction* operator $(\nu n)\_$. In a system of the form $S_1 \parallel (\nu n)S_2$, the effect of the operator is to make name n invisible from S_1.

A COMPONENT $\mathcal{I}[\mathcal{K}, \Pi, P]$ consists of:

- An *interface* $\mathcal{I}$ publishing and making available information about the component itself in the form of *attributes*, i.e. names acting as references to information stored in the component's knowledge repository. Among them, attribute id is mandatory and is bound to the name of the component.
- A *knowledge repository* $\mathcal{K}$ managing both *application data* and *awareness data*, together with the specific handling mechanism. Application data are used for enabling

SYSTEMS:		
S	::= C	(component)
	$\mid S_1 \parallel S_2$	(composition)
	$\mid (\nu n)S$	(name restriction)
COMPONENTS:		
C	::= $\mathcal{I}[\mathcal{K}, \Pi, P]$	(single component)
PROCESSES:		
P	::= **nil**	(inert)
	$\mid a.P$	(action prefixing)
	$\mid P_1 + P_2$	(choice)
	$\mid P_1 \mid P_2$	(composition)
	$\mid X$	(process variable)
	$\mid A(\bar{p})$	(invocation)

ACTIONS:		
a	::= $\mathbf{get}(T)@c$	(withdraw)
	$\mid \mathbf{qry}(T)@c$	(retrieve)
	$\mid \mathbf{put}(t)@c$	(addition)
	$\mid \mathbf{fresh}(n)$	(scope)
	$\mid \mathbf{new}(\mathcal{I}, \mathcal{K}, \Pi, P)$	(new)
TARGETS:		
c	::= n	(name)
	$\mid x$	(variable)
	$\mid$ self	(self)
	$\mid \mathcal{P}$	(predicate)
	$\mid p$	(pred. name)

Table 3. SCEL syntax (POLICIES Π, KNOWLEDGE $\mathcal{K}$, TEMPLATES T, and ITEMS t are parameters of the language)

the progress of components' computations, while awareness data provide information about the environment in which the components are running (e.g. monitored data from sensors) or about the status of a component (e.g. its current location).

- A set of *policies* Π regulating the interaction between the different parts of a single component and the interaction between components.
- A *process* P, together with a set of process definitions that can be dynamically activated.

PROCESSES are the active computational units. Each process is built up from the *inert* process **nil** via *action prefixing* ($a.P$), nondeterministic *choice* ($P_1 + P_2$), controlled *composition* ($P_1 \mid P_2$), *process variable* (X), and parameterised process *invocation* ($A(\bar{p})$). The semantics of the construct $P_1 \mid P_2$ is another parameter of SCEL. It can be instantiated so as to capture various forms of *parallel composition* commonly used in process calculi. For example, in SCELIGHT, it corresponds to the standard *interleaving* execution of the two involved processes. Communication can be *higher-order*, as in KLAIM. We assume that A ranges over a set of parameterised *process identifiers* that are used in (possibly recursive) process definitions. We also assume that each process identifier A has a *single* definition of the form $A(\bar{f}) \triangleq P$. Lists of actual and formal parameters are denoted by $\bar{p}$ and $\bar{f}$, respectively.

Processes can perform five different kinds of ACTIONS. Actions $\mathbf{get}(T)@c$, $\mathbf{qry}(T)@c$ and $\mathbf{put}(t)@c$ are used to manage shared knowledge repositories by *withdrawing/retrieving/adding* information items from/to the knowledge repository identified by target c. These actions exploit templates T as patterns to select knowledge items t in the repositories. They heavily depend on the chosen kind of knowledge repository (a parameter of SCEL, as we have already noticed) and are implemented by invoking the knowledge handlers it provides. Action $\mathbf{fresh}(n)$ introduces a *scope* restriction for the name n so that this name is guaranteed to be *fresh*, i.e., different from any other name previously used. Action $\mathbf{new}(\mathcal{I}, \mathcal{K}, \Pi, P)$ creates a *new* component $\mathcal{I}[\mathcal{K}, \Pi, P]$.

Action **get** may cause the process executing it to wait for the expected element, in case it is not (yet) available in the knowledge repository. Action **qry**, exactly like **get**, may suspend the process executing it if the knowledge repository does not (yet) contain

or cannot "produce" the expected element. The two actions differ for the fact that **get** removes the found item from the knowledge repository while **qry** leaves the target repository unchanged. Actions **put**, **fresh** and **new** are instead immediately executed (provided that their execution is allowed by the policies in force).

Different entities may be used as the target c of an action. In addition to names and variables for names, the distinguished variable self can be used by processes to refer to the name of the component hosting them. The possible targets could be also singled out via a *predicate* $\mathcal{P}$ (or the name p of a predicate). Predicates are boolean-valued expressions obtained by logically combining relations between attributes and value expressions. When the target of a communication action is a predicate, this predicate acts as a "guard" specifying the *ensemble* of all those components with which the process performing the action intends to interact. Thus, e.g., actions $\mathbf{put}(t)@n$ and $\mathbf{put}(t)@\mathcal{P}$ give rise to two different primitive forms of communication: the former is a *point-to-point* communication, while the latter is a sort of *group-oriented* communication.

It is worth noticing that the group-oriented variant of action **put** is used to insert a knowledge item in the repositories of *all* components belonging to the ensemble identified by the target predicate. Differently, the group-oriented variants of actions **get** and **qry** withdraw and retrieve, respectively, an item from a *single* component nondeterministically selected among those satisfying the target predicate.

4.3 Example: a collection of service components

SCEL has proved to be suitable for modelling autonomic systems from different application scenarios such as, e.g., collective robotic systems [17, 27], cooperative e-vehicles [13], service provision and cloud-computing [22, 48, 50]. Here, we consider a scenario, borrowed from [29] and modelled in SCELIGHT, consisting of m provider components $\mathcal{I}_{p_j}[\mathcal{K}_{p_j}, A_{p_j}]$, offering a variety of services, and n client components $\mathcal{I}_{c_h}[\mathcal{K}_{c_h}, P_{c_h}]$:

$$\mathcal{I}_{p_1}[\mathcal{K}_{p_1}, A_{p_1}] \parallel \ldots \parallel \mathcal{I}_{p_m}[\mathcal{K}_{p_m}, A_{p_m}] \parallel \mathcal{I}_{c_1}[\mathcal{K}_{c_1}, P_{c_1}] \parallel \ldots \parallel \mathcal{I}_{c_n}[\mathcal{K}_{c_n}, P_{c_n}] \,.$$

Each service component manages and elaborates service requests with different requirements, roughly summarised by the following three quality levels: *gold*, *silver* and *base*. These levels are defined via a combination of predicates on the hardware configuration and the runtime state of the provider components. To this aim, we assume that attributes named hw and $load$ are provided by each service component. The former can take an integer value from 0 to 10 that gives an indication of the capacity of the hardware configuration of the component, while the latter can take an integer value from 0 to 100 that estimates the actual computational load of the component. The three quality of service levels are then characterised by following predicates:

$$\begin{array}{l} \mathcal{P}_g \triangleq (7 \leq hw \quad\;\;) \\ \mathcal{P}_s \triangleq (4 \leq hw < 7) \vee (\mathcal{P}_g \wedge load < 40) \\ \mathcal{P}_b \triangleq (\quad hw < 4) \vee (\mathcal{P}_s \wedge load < 40) \vee (\mathcal{P}_g \wedge load < 20) \end{array}$$

identifying, respectively, three ensembles of service components that

- *Gold*: have a high level of hardware configuration, i.e. a hardware level greater or equal to 7;

– *Silver*: provide a hardware configuration with a level that is at least 4 and, whenever the hardware level is over 7, the computational load is less than 40%; this latter condition guarantees that gold components can handle requests at silver level only when their computational load is under 40%;
– *Base*: have any hardware level, however if they are also gold or silver components then their computational load is under 20% or 40%, respectively.

Each service component also stores in its knowledge repository a collection of items indicating the provided services, together with the component identifier. For example, the provider p_j offering the $factorial$ service stores in its local repository the item $\langle service, factorial, i_{p_j} \rangle$. Note that including the identifier in the tuple publishing the service is fundamental as the group-oriented communication primitives are completely anonymous, i.e., the actual targets of a group-oriented communication action are not known to the subject.

Finally, each service component p_j runs the process A_{p_j} defined as:

$$\begin{aligned} A_{p_j} \triangleq\ & \mathbf{get}(invoke, factorial, ?x, ?y)@\mathsf{self}. \\ & \mathbf{get}(load, ?z)@\mathsf{self}. \\ & \mathbf{put}(load, (z + 20))@\mathsf{self}. \\ & (A_{p_j} \mid Q(x, y)) \end{aligned}$$

The process is triggered by a client request. Whenever this happens, the computational load is updated; we assume that each service instance uses 20% of the sever's capacity. Then, the $factorial$ service becomes again ready to serve other client requests, and the process Q, which actually computes the result of the invoked service for the current request, is executed. We assume that, before its termination, process Q updates the value of attribute $load$, and puts the result of the computation in the repository of the client.

We remark that components dynamically and transparently leave or enter an ensemble when their computational load changes. For instance, a *gold* component leaves a *silver* ensemble when its computational load becomes higher than 40%.

Each client component c_h runs the process P_{c_h}, that takes care of the interaction with the *factorial* service and is of the form

$$\begin{aligned} & \mathbf{qry}(service, factorial, ?x)@\mathcal{P}_k. \\ & \mathbf{put}(invoke, factorial, v, i_{c_h})@x. \\ & \mathbf{get}(result, factorial, ?y)@\mathsf{self}.\ P'_{c_h} \end{aligned}$$

for some service level k in $\{b, s, g\}$ and some argument v for the factorial function the client would like the server to execute. Intuitively, such process first searches among the components belonging to the ensemble identified by predicate $\mathcal{P}_k$, via a **qry** action, an item matching the template $(service, factorial, ?x)$. In this way, by taking advantage of group-oriented communication, the client is able to dynamically identify a component x that provides the $factorial$ service at the desired service level k. If more than one provider component meets these requirements, one of them will be non-deterministically selected. Then, via a **put** action, the process invokes the selected service, in a point-to-point fashion, by providing the actual parameter v of the request. After issuing the invocation, the process waits for the result (recall that action **get** is

blocking). Whenever the result of the service invocation is made available, the process can withdraw it from the local repository and continue as process P'_{c_h}.

4.4 Programming environment

SCEL systems can be executed and simulated in jRESP[6] (Java Runtime Environment for SCEL Programs), which offers specific software tools to develop and support SCEL systems. In particular, jRESP provides an API that permits enriching Java programs with the SCEL's linguistic constructs. The API is instrumental to assist programmers in the implementation of autonomic systems, which thus turns out to be simplified with respect to using "pure" Java. Moreover, jRESP provides a set of classes enabling execution of *virtual components* on top of a simulation environment that can control component interactions and collect relevant simulation data.

4.5 Verification techniques

A prototype framework for statistical model-checking has been developed [30] by relying on the jRESP simulation environment. The tool is parameterised with respect to a given *tolerance* ε and *error probability* p, thus allowing one to verify whether the implementation of a system satisfies a given property with a certain degree of confidence. The underlying randomised algorithm guarantees that the difference between the computed value and the exact one is greater than ε with a probability that is less than p.

Qualitative properties of SCELIGHT specifications have been verified through the Spin model checker [42]. The verification relies on a preliminary translation from SCELIGHT into Promela, i.e., the input language of Spin. This approach has been used in, e.g., [29] to verify some properties of interest of the application scenario illustrated in Section 4.3, like absence of deadlock, server overload and responsiveness, and in [30] to verify similar properties for a swarm robotics scenario.

SCEL's operational semantics has also been implemented by using the Maude framework [18]. The outcome, named MISSCEL (Maude Interpreter and Simulator for SCEL), focuses on SCELIGHT and exploits the rich Maude toolset to perform, among other things, qualitative analysis via Maude's invariant and LTL model checkers, and statistical model checking via MULTIVESTA [59] (as done in [10] for a robotic collision avoidance scenario). A further advantage of MISSCEL is that SCEL specifications can be intertwined with (very expressive) raw Maude code. This permits to obtain sophisticated specifications in which SCEL is used to model behaviours, aggregations, and knowledge handling, while scenario-specific details are specified with Maude.

5 AbC: Attribute-based communication

Collective-Adaptive Systems (CAS) [35] are new emerging computational systems, consisting of a massive number of components, featuring complex interaction mechanisms. These systems are usually distributed, heterogeneous, decentralised and interdependent, and are operating in dynamic and often unpredictable environments. CAS

[6] jRESP website: http://jresp.sourceforge.net/.

components combine their behaviours, by forming collectives, to achieve specific goals depending on their attributes, objectives, and functionalities. CAS are inherently scalable and their boundaries are fluid in the sense that components may enter or leave the collective at any time; so they need to dynamically adapt to their environmental conditions and contextual data.

AbC (Attribute-based Communication calculus, [4, 2]) is a process calculus specifically designed to deal with CAS. It has been heavily inspired by SCEL, but has been designed to reduce complexity and keep the set of linguistic primitives to a minimum. Indeed, it was originally designed as a trimmed version of SCEL that was obtained by ignoring the parts relative to policies and knowledge and concentrating only on behaviours and interfaces. In this respect, AbC has similar aims to SCELIGHT, but the underling communication paradigm is very different; explicit message passing for the former and shared memory *à la* KLAIM for the latter.

Indeed, the original aim of AbC was to assess the impact of the new message passing paradigm based on attributes and compare it with more classical ones that handle the interaction between distributed components by relying on identities (Actors [5]), or channels (π-calculus), or broadcast (B-π-calculus [55]). In all these formalisms, messages exchanges rely on names or addresses of the involved components and are independent of their status and capabilities. This makes it hard to program, coordinate, and adapt complex behaviours that highly depend on run-time changes of components.

In AbC, the attribute-based system is however more than just the parallel composition of interacting partners; it is also parametrised with respect to the environment or the space where system components are executed. The environment has a great impact on how components behave and provides a new means of indirect communication, that allows components to mutually influence each other, possibly unintentionally.

5.1 Syntax

Table 4 contains the syntax of AbC. The top-level entities of the calculus are COMPONENTS. A component, $\Gamma :_I P$, is a process P associated with an attribute environment Γ, and an interface I. The *attribute environment* provides a collection of attributes whose values represent the status of the component and influence its run-time behaviour. Formally, $\Gamma : \mathcal{A} \twoheadrightarrow \mathcal{V}$ is a partial map from attribute identifiers ($a \in \mathcal{A}$) to values ($v \in \mathcal{V}$), i.e., to numbers, strings, tuples, ... The *interface* $I \subseteq \mathcal{A}$ contains the *public attributes* of a component (the attributes in $dom(\Gamma) - I$ being *private*). Composed components $C_1 \| C_2$ are built by using the parallel operator.

A PROCESS P can be: the *inactive* process $\mathsf{0}$; an *action-prefixed* process, $act.U$, where act is a communication action and the UPDATE U is a process possibly preceded by *attribute updates*; a *context aware* process, $\langle \Pi \rangle P$, where Π is a PREDICATE built from boolean constants and from atomic predicates, based on EXPRESSIONS over attributes, by using standard boolean operators; a *nondeterministic choice* between two processes, $P_1 + P_2$; a *parallel composition* of two processes, $P_1 | P_2$; or a process call with an identifier K used in a unique process definition $K \triangleq P$.

COMPONENTS:			PREDICATES:		
C	::= $\Gamma :_I P$	(component)	Π	::= true	(true)
	$\mid C_1 \| C_2$	(composition)		$\mid \mathrm{false}$	(false)
PROCESSES:				$\mid p(\tilde{E})$	(atomic predicate)
P	::= $\mathsf{0}$	(inaction)		$\mid \Pi_1 \wedge \Pi_2$	(conjunction)
	$\mid \Pi(\tilde{x}).U$	(attribute-based input)		$\mid \Pi_1 \vee \Pi_2$	(disjunction)
	$\mid (\tilde{E})@\Pi.U$	(attribute-based output)		$\mid \neg\Pi$	(negation)
	$\mid \langle \Pi \rangle P$	(context awareness)	EXPRESSIONS:		
	$\mid P_1 + P_2$	(choice)	E	::= v	(value)
	$\mid P_1 \mid P_2$	(parallel composition)		$\mid x$	(variable)
	$\mid K$	(process identifier)		$\mid a$	(attribute identifier)
UPDATES:				$\mid \mathsf{this}.a$	(local reference)
U	::= $[a := E]U$	(attribute update)		$\mid op(\tilde{E})$	(operator)
	$\mid P$	(process)			

Table 4. The syntax of the AbC calculus

5.2 Informal Semantics

Attribute-based actions for sending and receiving messages permit to establish communication links between different components according to specific predicates over their attributes.

Specifically, *attribute-based output* $(\tilde{E})@\Pi$ sends the result of the evaluation of the sequence of expressions $\tilde{E}$ to the components whose attributes satisfy the predicate Π. Notably, together with the computed values, also the portion of the attribute environment of the sending component that can be perceived by the context is sent; this is obtained from the local environment by limiting its domain to the attributes in the component interface. This information is needed to allow receivers to determine whether they are interested in the sent message.

Instead, *attribute-based input* $\Pi(\tilde{x})$ specifies receipt of messages from a component satisfying predicate Π; the sequence $\tilde{x}$ acts as a placeholder for received values. A message can be received when two *communication constraints* are satisfied: the public local attribute environment satisfies the predicate used by the sender to identify potential receivers, and the sender environment satisfies the receiving predicate. In this case, attribute updates are performed under the generated substitution. An *attribute update* $[a := E]$ assigns the value of E to the attribute identifier a. This action is used to change the values of the attributes according to contextual conditions and to adapt component's behaviour. Notice that the execution of a communication action and the following update(s) is atomic.

The *awareness construct* $\langle \Pi \rangle P$ blocks the execution of P until predicate Π is satisfied when using the local attribute environment, possibly after a change of state by a component. This construct permits to collect awareness data and take decisions based on the changes in the attribute environment.

5.3 Example: a TV broadcaster scenario

We now illustrate the features of AbC by considering a simple scenario borrowed from the paper where AbC was originally introduced [4]. In this scenario, we consider a TV

broadcaster (e.g., CNN) represented by the process CNN, and two receivers represented by the processes RcvA and RcvB:

$$\begin{aligned}
\texttt{CNN} &\triangleq (v_s)@\Pi_{sport}.\texttt{CNN} + (v_n)@\Pi_{news}.\texttt{CNN} \\
&\quad + ()@\text{false}.[\texttt{Qbrd} := \texttt{LD}]\texttt{CNN} + ()@\text{false}.[\texttt{Qbrd} := \texttt{HD}]\texttt{CNN} \\
\texttt{RcvA} &\triangleq (\texttt{Qbrd} = \texttt{HD})(x).\texttt{RcvA} \\
&\quad + ()@\text{false}.[\texttt{Genre} := \texttt{Sport}]\texttt{RcvA} + ()@\text{false}.[\texttt{Genre} := \texttt{News}]\texttt{RcvA} \\
\texttt{RcvB} &\triangleq (\text{true})(x).\texttt{RcvB} + \ldots
\end{aligned}$$

where

$$\begin{aligned}
\Pi_{sport} &= (\texttt{Genre} = \texttt{Sport}) \wedge (\texttt{CNN-Sub} = \texttt{tt}) \\
\Pi_{news} &= (\texttt{Genre} = \texttt{News})
\end{aligned}$$

The overall system is expressed as the parallel composition below, where the dots refer to other possible broadcasters or receivers:

$$\Gamma_{cnn} :_C \texttt{CNN} \mid \Gamma_a :_A \texttt{RcvA} \mid \ldots \mid \Gamma_b :_B \texttt{RcvB}.$$

CNN periodically broadcasts Sport or News and targets different groups of receivers based on the predicates Π_{sport} and Π_{news}. Π_{sport} targets the group of receivers who want to watch Sport ($\texttt{Genre} = \texttt{Sport}$) provided that those receivers have subscribed to CNN ($\texttt{CNN-Sub} = \texttt{tt}$). On the other hand, Π_{news} targets the group of receivers who want to watch News ($\texttt{Genre} = \texttt{News}$).

The quality of the broadcasted multimedia varies according to different factors (e.g., low bandwidth). CNN channel non-deterministically chooses to broadcast low-definition ($[\texttt{Qbrd} := \texttt{LD}]$) or high-definition ($[\texttt{Qbrd} := \texttt{HD}]$) multimedia. The receiving processes RcvA and RcvB almost have the same behaviour except that RcvA is only interested in high quality broadcasts while RcvB is willing to accept broadcasts of any quality. So they either accept the broadcast that their attributes in Γ_a and Γ_b satisfy, or change the genre.

A fragment of the possible interactions in this scenario is reported below; we use $\longrightarrow_b$ and $\longrightarrow_\tau$ to denote the computational steps induced by a broadcast action and by an attribute update action, respectively, and also use the grey-shaded box to indicate the components involved in the evolution.

$$\begin{aligned}
&\Gamma_{cnn} :_C \texttt{CNN} \mid \Gamma_a :_A \texttt{RcvA} \mid \ldots \mid \Gamma_b :_B \texttt{RcvB} \\
\longrightarrow_b \quad &\boxed{\Gamma_{cnn} :_C \texttt{CNN}} \mid \boxed{\Gamma_a :_A \texttt{RcvA}} \mid \ldots \mid \boxed{\Gamma_b :_B \texttt{RcvB}} \\
&\vdots \\
\longrightarrow_\tau \quad &\boxed{\Gamma_{cnn}\,[\texttt{Qbrd} \mapsto \texttt{LD}] :_C \texttt{CNN}} \mid \Gamma_a :_A \texttt{RcvA} \mid \ldots \mid \Gamma_b :_B \texttt{RcvB} \\
\longrightarrow_b \quad &\boxed{\Gamma'_{cnn} :_C \texttt{CNN}} \mid \Gamma_a :_A \texttt{RcvA} \mid \ldots \mid \boxed{\Gamma_b :_B \texttt{RcvB}}
\end{aligned}$$

We assume that the initial attribute environments Γ_{cnn}, Γ_a and Γ_b are: $\Gamma_{cnn} = \{(\texttt{Qbrd}, \texttt{HD}), \ldots\}$, $\Gamma_a = \{(\texttt{Genre}, \texttt{News}), \ldots\}$ and $\Gamma_b = \{(\texttt{Genre}, \texttt{News}), \ldots\}$.

The interfaces of CNN, RcvA, and RcvB are defined as follows: $C = \{\texttt{Qbrd}\}$ and $A = B = \{\texttt{Genre}\}$. Assume also that CNN initiates the interaction by broadcasting high quality News. As shown above, both RcvA and RcvB can join the collective

and receive the broadcast because their attributes satisfy the condition of the broadcast (based on predicate Π_{news}). After a while CNN chooses to lower the quality of multimedia (indeed, its environment is updated with Qbrd $\mapsto$ LD) to cope with some situations, such as low bandwidth, and CNN can evolve independently. Finally, CNN continues broadcasting News and in this case RcvA chooses to leave the collective because the quality of the broadcast does not satisfy its receiving predicate, while RcvB stays because it has no requirement for the input quality.

5.4 Programming Environment

Basing the interaction on the values of run-time attributes is indeed a nice idea, but it needs to be supported by a middleware that provides efficient ways for distributing messages, checking attribute values, and updating them. A typical approach is to rely on a centralised broker that keeps track of all components, intercepts every message and forwards it to registered components. It is then the responsibility of each component to decide whether to receive or discard the message. This is the solution proposed in [3] where a Java implementation of AbC is provided, that however suffers of serious performance problems. Two additional implementations of AbC have thus been considered, which are built on the top of two well-established programming languages largely used for concurrent programming, namely Erlang and Go, guaranteeing better scalability. The two implementations are called $AErlang$, for Attribute based Erlang, and *GoAt*, for Go with attributes.

$AErlang$ [21] is a middleware enabling attribute-based communication among programs in Erlang [32], a concurrent functional programming language originally designed for building telecommunication systems and recently successfully adapted to broader contexts, such as large-scale distributed messaging platforms like Facebook and WhatsApp. $AErlang$ lifts Erlang's send and receive communication primitives to attribute-based reasoning. In Erlang, the send primitive requires an explicit destination address while in $AErlang$ processes are not aware of the presence and identity of each other, and communicate using predicates over attributes. $AErlang$ has two main components: (i) a process registry that keeps track of process details, such as the process identifier and the current status, and (ii) a message broker that undertakes the delivery of outgoing messages. The *Process registry* is a generic server that accepts requests regarding process (un)registration and internal updates. It stores process identifiers and all the information used by the message broker to deliver messages. The *Message broker* is responsible for delivering messages between processes. It is implemented as an Erlang server process listening for interactions from attribute-based send. To address potential bottlenecks arising in the presence of a very large number of processes, the message broker can be set up to run in multiple parallel threads. Like the Java implementation for AbC presented in [3], the message broker is still centralised, however, to avoid broadcasts, the broker has an attribute registry where components register their attribute values and the broker is now responsible for message filtering. Different distribution policies have been implemented that can be used by taking into account dynamicity of attributes and of predicates.

GoAt[7] extends Go [39], the language introduced by Google to handle massive computation clusters, and to make working in these environments more productive. Go has an intuitive and lightweight concurrency model with a well-understood semantics and extends the CSP model [41] with channel mobility, like in π-calculus. It also supports buffered channels, to provide mailboxes *à la* Erlang. The *Attribute-based API* for Go offers the possibility of using the AbC primitives to program the interaction of CAS applications directly in Go. The actual implementation faithfully models the formal semantics of AbC and it is parametric with respect to the infrastructure that mediates interactions. The *GoAt* API offers the possibility of using three different distributed coordination infrastructures for message exchange, namely cluster, ring, and tree. For all three infrastructures, it has been proved that the message delivery ordering is the same as the one required by the original formal semantics of AbC [1]. An Eclipse plugin permits programming in a high-level syntax, which can be analysed via formal methods by relying on the operational semantics of AbC. Once the code has been analysed, the *GoAt* plugin will generate formally verifiable Go code. Examples available from *GoAt*'s site permit to appreciate how intuitive it is to program a complex variant of the well-known problem of Stable Allocation in Content Delivery Network [47].

5.5 Verification techniques

Some work has now started to verify properties of AbC programs. On the one hand, it is under investigation the use of the generic tools that have been designed for verifying properties of Erlang and Go programs. On the other hand, tools are under development to prove directly properties of the AbC specifications. The second alternative is under consideration because in some cases the correspondence between the actual AbC specifications and the running programs may not be immediate, and the difference would reduce the effectiveness of the effort.

A novel approach to the analysis of concurrent systems modelled as AbC terms has been introduced in [20]. It relies on the UMC model checker, a tool based on modelling concurrent systems as communicating UML-like state machines [61]. A structural translation from AbC specifications to the UMC internal format is used as the basis for program analysis. This permits identifying emerging properties of systems and unwanted behaviours.

Recent work considers a variant of AbC and proposes a technique to prove properties of the system by translating the specifications into symbolic C programs to be analysed with SAT-based approaches.

6 Concluding remarks

This paper surveyed four domain-specific coordination languages supporting the engineering of different classes of modern distributed systems. These languages have been developed in the last twenty years by the authors (three of which have been working for quite a while in the Concurrency and Mobility Group at University of Florence)

[7] *GoAt* codes and examples can be retrieved from https://giulio-garbi.github.io/goat/.

and other collaborators. Within the coordination community other research groups have followed a similar methodology, however relying on different specification models, e.g. coalgebras [7], actors [60] or automata [8], rather than process algebras.

Below, we summarise the programming abstractions introduced with the different formalisms and the lessons learned when designing and using languages for

1. Network-Aware Programming,
2. Service-Oriented Computing,
3. Autonomic Computing,
4. Collective Adaptive Systems Programming.

The design of KLAIM has shown that network awareness in distributed systems can be achieved by the explicit use of localities as first-order citizens of the language. Localities, indeed, identify network nodes, where computation takes place and data is stored. Network awareness relies on the notion of (multiple) tuple spaces, which can be accessed via a unique interface to insert and retrieve data. Communication is thus asynchronous, anonymous and associative, pattern matching plays a crucial role and guarantees high expressive power. Network awareness also supports computation mobility, thus paving the way for different kinds of optimisations.

From COWS, we learnt that SOC applications typically abstract from the structure of the underlying network and from distribution of data, which become transparent to the programmer. Pattern-matching still plays a key role in supporting communication, as it is at the basis of the message correlation mechanism. Novel distinguishing features are service persistence, state sharing among concurrent service instances, and service fault and termination handling. We have shown that the modelling of the first one can rely on the standard process replication operator. Instead, the modelling of the second one relies on the combined use of suitable binder operators, and non-standard receive activities binding neither names nor variables. Similarly, the modelling of the third feature requires a combination of some ingenious constructs to either force termination or protect activities in case of termination of other processes.

In SCEL the central notion is that of ensemble of components, which can be dynamically created in an opportunistic and transparent way. Indeed, the formation of an ensemble and the establishment of interactions among its members rely on the information exposed as attributes in the interface of the involved components. This enables an effective group-oriented communication model. Ensemble components are equipped with knowledge repositories that generalise KLAIM's tuple spaces by supporting different knowledge representations and handling mechanisms. Self- and context-awareness make these components capable to adapt their behaviour to evolving needs and environmental changes.

Finally, AbC refines the group-oriented communication model of SCEL, in order to convey in a distilled form the attribute-based communication paradigm exploited to model and program Collective Adaptive Systems. The result of this synthesis effort is a compact calculus, suitable for studying the theoretical impact of the novel communication paradigm and for obtaining new programming frameworks by the new paradigm in different well-established programming languages, such as Java, Erlang and Go.

To recap, we think that the engineering methodology we presented, as witnessed by the four instantiations we have illustrated, provides a uniform linguistic approach,

based on formal methods techniques, for ensuring the trustworthiness of the considered classes of systems and possibly of the other ones that will emerge in the near future. In this respect, we plan to consider the Aggregate Programming [9] domain, where the abstraction level in designing distributed systems further increases. In such an engineering approach, data and devices are aggregated via 'under-the-hood' coordination mechanisms. Although these aggregations resemble the notions of ensemble and collectives discussed in this paper, they mainly focus on distributed computation rather than on communication mechanisms.

As a final disclaimer we would like to say that obviously a section dedicated to related work is missing. Given the time span and the different programming domains covered by the development of our four languages, it would have needed a paper on his own. Thus the only thing we can do is to refer the interested reader to the bibliography sections of the papers that have introduced and developed KLAIM, COWS, SCEL and AbC. Moreover, for references on network-aware programming and relation with KLAIM we refer to [56], for service-oriented computing and COWS to [63] and for autonomic computing and SCEL to [64].

Acknowledgements. This work would not have been possible without the contribution of our collaborators that have helped us in shaping the four languages we have introduced. They are too many to be listed, but their names could be inferred from the bibliography below. However, we would like to make an exception and explicitly thank Michele Loreti. Michele has been a driving force for most of the results we have presented, he is not among the authors only because he is one of the PC chairs of the conference to which the work was submitted.

References

1. Y. Abd Alrahman, R. De Nicola, G. Garbi, and M. Loreti. A distributed coordination infrastructure for attribute-based interaction. In *FORTE*, LNCS. Springer, 2018. To appear.
2. Y. Abd Alrahman, R. De Nicola, and M. Loreti. On the power of attribute-based communication. In *FORTE*, volume 9688 of *LNCS*, pages 1–18. Springer, 2016. Full technical report can be found on http://arxiv.org/abs/1602.05635.
3. Y. Abd Alrahman, R. De Nicola, and M. Loreti. Programming of CAS systems by relying on attribute-based communication. In *ISoLA*, volume 9952 of *LNCS*, pages 539–553. Springer, 2016.
4. Y. Abd Alrahman et al. A calculus for attribute-based communication. In *SAC '15*, pages 1840–1845. ACM, 2015.
5. G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.
6. R. Amadio, I. Castellani, and D. Sangiorgi. On Bisimulations for the Asynchronous pi-Calculus. *Theor. Comput. Sci.*, 195(2):291–324, 1998.
7. F. Arbab and J. J. M. M. Rutten. A coinductive calculus of component connectors. In *WADT*, volume 2755 of *LNCS*, pages 34–55. Springer, 2002.
8. C. Baier, M. Sirjani, F. Arbab, and J. J. M. M. Rutten. Modeling component connectors in reo by constraint automata. *Sci. Comput. Program.*, 61(2):75–113, 2006.
9. J. Beal and M. Viroli. Aggregate programming: From foundations to applications. In *SFM*, volume 9700 of *LNCS*, pages 233–260. Springer, 2016.

10. L. Belzner, R. De Nicola, A. Vandin, and M. Wirsing. Reasoning (on) service component ensembles in rewriting logic. In *Specification, Algebra, and Software*, volume 8373 of *LNCS*, pages 188–211. Springer, 2014.
11. L. Bettini, R. De Nicola, G. Ferrari, and R. Pugliese. Interactive Mobile Agents in X-Klaim. In *WETICE*, pages 110–115. IEEE Computer Society Press, 1998.
12. L. Bettini, R. De Nicola, and R. Pugliese. KLAVA: a Java package for distributed and mobile applications. *Software – Practice and Experience*, 32(14):1365–1394, 2002.
13. T. Bures et al. A life cycle for the development of autonomic systems: The e-mobility showcase. In *SASOW*, pages 71–76. IEEE, 2013.
14. S. Castellani, P. Ciancarini, and D. Rossi. The ShaPE of ShaDE: a coordination system. Technical Report UBLCS 96-5, Dip. di Scienze dell'Informazione, Univ. Bologna, 1996.
15. L. Cesari, R. Pugliese, and F. Tiezzi. A tool for rapid development of WS-BPEL applications. *SIGAPP Applied Computing Review*, 11(1):27–40, 2010.
16. L. Cesari, R. Pugliese, and F. Tiezzi. Blind-date conversation joining. *Service Oriented Computing and Applications*, 11(3):265–283, 2017.
17. L. Cesari et al. Formalising Adaptation Patterns for Autonomic Ensembles. In *FACS*, volume 8348 of *LNCS*, pages 100–118. Springer, 2013.
18. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott. *All About Maude*, volume 4350 of *LNCS*. Springer, 2007.
19. N. Davies, S. Wade, A. Friday, and G. Blair. L^2imbo: a tuple space based platform for adaptive mobile applications. In *ICODP/ICDP*, pages 291–302. Springer, 1997.
20. R. De Nicola, T. Duong, O. Inverso, and F. Mazzanti. Verifying properties of systems relying on attribute-based communication. In *ModelEd, TestEd, TrustEd*, volume 10500 of *LNCS*, pages 169–190. Springer, 2017.
21. R. De Nicola, T. Duong, O. Inverso, and C. Trubiani. AErlang: Empowering erlang with attribute-based communication. In *COORDINATION*, volume 10319 of *LNCS*, pages 21–39. Springer, 2017.
22. R. De Nicola, G. Ferrari, M. Loreti, and R. Pugliese. A Language-based Approach to Autonomic Computing. In *FMCO*, LNCS 7542, pages 25–48. Springer, 2012.
23. R. De Nicola, G. L. Ferrari, and R. Pugliese. Klaim: A Kernel Language for Agents Interaction and Mobility. *IEEE Trans. Software Eng.*, 24(5):315–330, 1998.
24. R. De Nicola, G. L. Ferrari, R. Pugliese, and B. Venneri. Types for access control. *Theor. Comput. Sci.*, 240(1):215–254, 2000.
25. R. De Nicola, D. Gorla, and R. Pugliese. Confining data and processes in global computing applications. *Sci. Comput. Program.*, 63(1):57–87, 2006.
26. R. De Nicola and M. Loreti. A Modal Logic for Mobile Agents. *ACM Trans. Comput. Log.*, 5(1):79–128, 2004.
27. R. De Nicola, M. Loreti, R. Pugliese, and F. Tiezzi. A formal approach to autonomic systems programming: The SCEL language. *TAAS*, 9(2):7, 2014.
28. R. De Nicola et al. From flow logic to static type systems for coordination languages. *Sci. Comput. Program.*, 75(6):376–397, 2010.
29. R. De Nicola et al. Programming and verifying component ensembles. In *FPS*, volume 8415 of *LNCS*, pages 69–83. Springer, 2014.
30. R. De Nicola et al. The SCEL language: Design, implementation, verification. In *Software Engineering for Collective Autonomic Systems - The ASCENS Approach*, volume 8998 of *LNCS*, pages 3–71. Springer, 2015.
31. D. Deugo. Choosing a Mobile Agent Messaging Model. In *ISADS*, pages 278–286. IEEE, 2001.
32. Ericsson Computer Science Laboratory. The Erlang programming language. https://www.erlang.org/. Last Access April 12, 2018.

33. A. Fantechi et al. A logical verification methodology for service-oriented computing. *ACM Trans. Softw. Eng. Methodol.*, 21(3):16:1–16:46, 2012.
34. G. L. Ferrari, E. Moggi, and R. Pugliese. Metaklaim: a type safe multi-stage language for global computing. *Mathematical Structures in Computer Science*, 14(3):367–395, 2004.
35. A. Ferscha. Collective adaptive systems. In *UbiComp/ISWC*, pages 893–895. ACM, 2015.
36. D. Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.
37. D. Gelernter. Multiple tuple spaces in linda. In *PARLE*, volume 366 of *LNCS*, pages 20–27. Springer, 1989.
38. S. Gnesi, R. Pugliese, and F. Tiezzi. The sensoria approach applied to the finance case study. In *Rigorous Software Engineering for Service-Oriented Systems*, volume 6582 of *LNCS*, pages 698–718. Springer, 2011.
39. Google. The Go programming language. https://golang.org/doc/. Last Access February 20, 2018.
40. D. Gorla and R. Pugliese. Dynamic management of capabilities in a network aware coordination language. *J. Log. Algebr. Program.*, 78(8):665–689, 2009.
41. C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.
42. G. J. Holzmann. The model checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.
43. J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. *Computer*, 36:41–50, 2003.
44. A. Lapadula, R. Pugliese, and F. Tiezzi. A Calculus for Orchestration of Web Services. In *ESOP*, volume 4421 of *LNCS*, pages 33–47. Springer, 2007.
45. A. Lapadula, R. Pugliese, and F. Tiezzi. Regulating data exchange in service oriented applications. In *FSEN*, volume 4767 of *LNCS*, pages 223–239. Springer, 2007.
46. A. Lapadula, R. Pugliese, and F. Tiezzi. Using formal methods to develop WS-BPEL applications. *Sci. Comput. Program.*, 77(3):189–213, 2012.
47. B. M. Maggs and R. K. Sitaraman. Algorithmic nuggets in content delivery. *SIGCOMM Comput. Commun. Rev.*, 45(3):52–66, July 2015.
48. A. Margheri, R. Pugliese, and F. Tiezzi. Linguistic abstractions for programming and policing autonomic computing systems. In *UIC/ATC*, pages 404–409. IEEE, 2013.
49. M. Masi, R. Pugliese, and F. Tiezzi. On secure implementation of an IHE XUA-based protocol for authenticating healthcare professionals. In *ICISS*, volume 5905 of *LNCS*, pages 55–70. Springer, 2009.
50. P. Mayer et al. The Autonomic Cloud: A vision of voluntary, peer-2-peer cloud computing. In *SASOW*, pages 89–94. IEEE, 2013.
51. M. Merro and D. Sangiorgi. On asynchrony in name-passing calculi. *Mathematical Structures in Computer Science*, 14(5):715–767, 2004.
52. R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, I and II. *Inf. Comput.*, 100(1):1–40, 41–77, 1992.
53. U. Montanari, R. Pugliese, and F. Tiezzi. Programming autonomic systems with multiple constraint stores. In *Software, Services, and Systems*, volume 8950 of *LNCS*, pages 641–661. Springer, 2015.
54. OASIS WSBPEL TC. Web Services Business Process Execution Language Version 2.0. Technical report, OASIS, April 2007.
55. K. V. S. Prasad. A calculus of broadcasting systems. *Sci. Comput. Program.*, 25(2-3):285–327, 1995.
56. C. Priami and P. Quaglia, editors. *Global Computing*, volume 3267 of *LNCS*. Springer, 2005.
57. R. Pugliese and F. Tiezzi. A calculus for orchestration of web services. *J. Applied Logic*, 10(1):2–31, 2012.

58. R. Pugliese, F. Tiezzi, and N. Yoshida. On Observing Dynamic Prioritised Actions in SOC. In *ICALP*, volume 5556 of *LNCS*, pages 558–570. Springer, 2009.
59. S. Sebastio and A. Vandin. MultiVeStA: statistical model checking for discrete event simulators. In *ValueTools*, pages 310–315. ICST/ACM, 2013.
60. M. Sirjani, A. Movaghar, A. Shali, and F. S. de Boer. Modeling and verification of reactive systems using rebeca. *Fundamenta Informaticae*, 63(4):385–410, Dec. 2004.
61. M. H. ter Beek, A. Fantechi, S. Gnesi, and F. Mazzanti. A state/event-based model-checking approach for the analysis of abstract system properties. *Sci. Comput. Program.*, 76(2):119–135, 2011.
62. W3C. Web services activity. https://www.w3.org/2002/ws/. Last Access 20 February, 2018.
63. M. Wirsing and M. M. Hölzl, editors. *Rigorous Software Engineering for Service-Oriented Systems - Results of the SENSORIA Project on Software Engineering for Service-Oriented Computing*, volume 6582 of *LNCS*. Springer, 2011.
64. M. Wirsing, M. M. Hölzl, N. Koch, and P. Mayer, editors. *Software Engineering for Collective Autonomic Systems - The ASCENS Approach*, volume 8998 of *LNCS*. Springer, 2015.