TLP **18** (5-6): 846–873, 2018. © Cambridge University Press 2018
doi:10.1017/S1471068418000364 First published online 18 July 2018

Logic programming as a service

ROBERTA CALEGARI and ENRICO DENTI
Alma Mater Studiorum—Università di Bologna, Bologna, Italy
(*e-mails:* roberta.calegari@unibo.it, enrico.denti@unibo.it)

STEFANO MARIANI
Università di Modena e Reggio Emilia, Reggio Emilia, Italy
(*e-mail:* stefano.mariani@unimore.it)

ANDREA OMICINI
Alma Mater Studiorum—Università di Bologna, Cesena, Italy
(*e-mail:* andrea.omicini@unibo.it)

submitted 31 March 2017; revised 31 December 2017; accepted 14 June 2018

Abstract

New generations of distributed systems are opening novel perspectives for logic programming (LP): On the one hand, service-oriented architectures represent nowadays the standard approach for distributed systems engineering; on the other hand, pervasive systems mandate for situated intelligence. In this paper, we introduce the notion of *Logic Programming as a Service* (LPaaS) as a means to address the needs of pervasive intelligent systems through logic engines exploited as a distributed service. First, we define the abstract architectural model by re-interpreting classical LP notions in the new context; then we elaborate on the nature of LP interpreted as a service by describing the basic LPaaS interface. Finally, we show how LPaaS works in practice by discussing its implementation in terms of distributed tuProlog engines, accounting for basic issues such as interoperability and configurability.

KEYWORDS: logic programming, distributed systems, service-oriented architecture, pervasive systems, intelligent systems, LPaaS, situatedness

1 Introduction

Computation is moving towards pervasive, ubiquitous environments where devices, software agents, and services are expected to seamlessly integrate and cooperate in support of human users, anticipating their needs and more generally acting on their behalf, delivering services in an 'anywhere, anytime' fashion (Finin *et al.* 2001; Zambonelli *et al.* 2015). Even more, software agents, robots, sensors, etc. could work together with people for a common goal, with the same level of efficiency and expertise as human-only teams. Such systems could face important challenges in several fields—from military network-centric operations, to gaming technologies, simulation, computer security, transportation and logistics, and others (Parker 2008).

The above scenarios naturally fit a distributed approach: tasks are often distributed in space, time, or functionality, and their completion can clearly benefit

from the chance of solving subproblems modularly and concurrently. At the same time, the same scenarios inherently call for intelligence—namely, *distributed situated intelligence* (Parker 2008) —to exploit domain knowledge, understand the local context, and share information in support of intelligent applications and services (Chen *et al.* 2003; Smart 2017).

Logic programming (LP henceforth) boasts a long-respected reputation in supporting intelligence: Originally conceived for single solvers and later extended towards concurrency and parallelism, LP has the potential to fully support pervasive computing scenarios once it is suitably *re-interpreted*. Re-interpretation of LP should develop along three main lines: (i) *architecture*—that is, the need to go beyond the (originally monolithic) structure of LP systems, which is unsuitable for distributed contexts such as IoT mobility/cloud ecosystems, typically grounded upon the service-oriented computing paradigm (Erl 2005); (ii) *situatedness*—that is, enabling logic theories, queries, and resolutions to be context-aware w.r.t. the (computational) environment, space, and time; (iii) *interaction*—that is, the opportunity to re-think the interaction patterns used by clients to query logic engines, which should lean towards on-demand computation.

At the same time, LP declarativeness and explicit knowledge representation enable knowledge sharing at the most adequate level of abstraction, while supporting modularity and separation of concerns (Oliya and Pung 2011), which are specially valuable in open and dynamic distributed systems (*serendipitous interoperability*, Niezen 2013). As a further element, LP soundness and completeness straightforwardly enable agents' intelligent reasoning. Finally, LP extensions or logic-based computational models—such as meta-reasoning about situations (Loke 2004) or labelled variables systems (Calegari *et al.* 2018)—could be incorporated so as to enable complex behaviours tailored to the situated components.

Although LP languages and technologies represent in principle a natural candidate for injecting intelligence within computational systems (Brownlee 2011), and despite the many practical application developed over the years—see Palù and Torroni (2010), Martelli (1995) for a survey—the adoption of LP in pervasive contexts has been historically hindered by technological obstacles—efficiency, integration issues—as well as by some cultural resistance towards LP-based approaches outside the academy. However, technology advancements, on the one hand, and the emergence of the IoT context, on the other, are drastically changing such a scenario, possibly allowing LP to unleash its full potential in real-world applications.

Along this line, in this paper, we present *Logic Programming as a Service* (LPaaS), a novel approach intended as the natural evolution of distributed LP in pervasive systems, explicitly designed to exploit context-awareness so as to promote the distribution of situated intelligence within smart environments. As the name suggests, the basic idea is to deliver LP-based intelligence *as a service*, granting ubiquitous access to knowledge and on-demand reasoning via LP services, spread over the network and configured to respond to specific local needs. Accordingly, some classical LP notions need to be revised and extended: For instance, client/service interaction is no longer bound to the traditional console-based query/response loop, and is instead redesigned to provide the dynamism, flexibility, and expressiveness

https://doi.org/10.1017/S1471068418000364 Published online by Cambridge University Press

required by the targeted application scenarios—e.g., IoT. Similarly, time and space situatedness promotes novel forms of client/service interaction, enabling clients to submit 'situated' queries where the notions of time and locus explicitly affect the computation.

The remainder of the paper is organised as follows. After Section 2 reviews the main works about the evolution of distributed LP, Section 3 introduces the vision behind LPaaS, by discussing how the service perspective and the new situated dimension of computation mandate for a re-interpretation of some basic LP concepts. Section 4 shows how such a re-interpretation affects LP at the architectural level, by discussing more practically the logic-based service-oriented architecture (SOA) supporting LPaaS. Section 5 defines the LPaaS service interface, and elaborates on the interaction patterns. Section 6 presents a prototype implementation developed on the top of the tuProlog system, while Section 7 discusses a case study in the Smart House field. Related works are reported in Section 8.

2 Distributed LP: Evolution

Research on distributed intelligence has gained increasing popularity over the years (Parker 2008). Starting from the seminal work of Clark and Gregory (1981), concurrency, parallelism, and several approaches for distributing intelligence have been explored—from LP languages specifically designed for distribution, to pure logic-based models, rule-based systems, probabilistic graphical models, and ontologies. In the following, we organise and describe some of the most relevant contributions to the field and to our approach, motivating the need for further advancement.

Implicit Parallelism. The first efforts to advance beyond sequential LP start from the programming schemes for the interpretation of logic programs—in particular, towards implicit parallel evaluation, leading to explore AND-parallelism, OR-parallelism, Search parallelism, and Stream-AND-parallelism.

Clark (1978) introduces a scheme that allows negative literals in queries; some years later, the Naish scheme (Naish 1988) introduces co-routing among procedure calls. Meanwhile, Wolfram *et al.* (1984) focus on AND-parallel evaluation: Their asynchronous version corresponds to the execution models of parallel LP languages. These schemes perform and adapt well to different forms of parallelism: However, they are not meant to face distributed programming. Also, it is worth noting that implicit parallelism lacks two important control mechanisms: synchronisation of logic processes and control over the non-determinism of schedulers.

Explicit Parallelism. Later approaches focus on 'extraction' of parallelism via explicit language constructs.

A first line of research moves from concurrent logic languages, rooted in the Relational Language (Clark and Gregory 1981), generally acknowledged as the first concurrent LP language. In Concurrent Prolog (Shapiro 1987), Guarded Horn Clauses (Ueda 1986), and Parlog (Clark 1987), goal evaluation is carried out by a network of fine-grained logic processes (i.e., atomic goals) that are executed in

https://doi.org/10.1017/S1471068418000364 Published online by Cambridge University Press

parallel: Processes communicate via shared streams, i.e., bi-directional channels on which data items flow.

An alternative research line follows the idea of extending Prolog with special features for distributed execution, like message passing. This approach preserves the operational semantics of sequential Prolog, augmenting the language with ad-hoc communication primitives. One of the major references in this field is Delta Prolog (Brogi and Gorrieri 1989), where Prolog is extended with constructs for sequential and parallel composition of goals, inter-process communication and synchronisation, and external non-determinism. Delta Prolog programs (Cunha *et al.* 1989) using concurrency mechanisms do not lend themselves to the usual declarative interpretation as Horn clauses, and are grounded instead on the theory of Distributed Logic (Monteiro 1984). This approach extends Horn clause logic with the notion of time-dependent events, on which process communication and synchronisation are based, making distributed logic a special kind of temporal logic.

Besides enabling inter-process communication for logic programs, orthogonal aspects such as their deployment are not considered, neither the issues brought along by distribution—such as validity in time of logic theories and their global consistency—are taken into account.

Agents, Communication, and Coordination for Distributed LP. Further steps towards distributed LP come with Shared Prolog (Brogi and Ciancarini 1991), based on parallel *agents* that are Prolog programs extended with a guard mechanism. The programmer controls the granularity of parallelism, coordinating agents' communication and synchronisation via a centralised data structure, the *blackboard*, inspired to the model defined in (Nii 1986) as well as to the LINDA coordination model (Gelernter 1985). The main idea is to exploit the blackboard within the logic framework to coordinate logic processes. However, the inference engine is not situated in time and space, i.e., the query result is independent from the entities' position, the time flow, and context/situation changes.

LP in Pervasive, Context-aware Systems. More recently, LP has been explored as a promising solution to bring intelligence into pervasive context-aware systems.

Ranganathan and Campbell (2003) show that using first-order logic is a very effective and powerful way of dealing with context, promoting an approach to develop a flexible and expressive model supporting context-awareness, enabling deduction of higher-level situations from perceptions about basic contexts—via rule-based approaches. A key advantage of formally modelling the context is that the expressiveness of the model itself can be clearly specified and automatically verified. Loke (2004) emphasises that LP is generally useful for context reasoning, as well as for supporting rule-based (meta)programming in context-aware applications, enabling, i.e., hierarchical description of complex situations in terms of other situations. This approach encourages a high level of abstraction for representing and reasoning about situations, and supports building context-aware systems incrementally through modularity and separation of concerns. The focus on context-awareness of both contributions is at the base of our choice of re-interpreting distributed LP by

targeting especially context-aware systems, as pervasive ones usually are—being the IoT a prominent example.

Other works take different approaches: from pure logic-based models to rule-based systems and probabilistic graphical models, up to ontologies.

Rule-based systems (Salber *et al.* 1999; Dey 2001; Etter *et al.* 2006; Wang *et al.* 2011) have been in use for decades for both model representation and reasoning in context-aware applications. More recently, Nalepa and Bobek (2014) have proposed a rule-based, *learning* middleware for storage and reasoning in a distributed scenario. The idea is to delegate context acquisition to middleware, that is, a rule-based context reasoning platform tailored to the needs of intelligent distributed mobile computing devices. The need for a dedicated middleware layer is apparent in the aforementioned works, further strengthening the idea that distributed LP is not confined to context manipulation, and deserves instead general attention.

In Ranganathan *et al.* (2004), fuzzy and probabilistic logic is exploited to handle the uncertainty of the environment and deal with the imperfections of data. Probabilistic graphical models (Bettini *et al.* 2010) can be exploited to support the modelling of, and the reasoning about, uncertain information in pervasive systems, even if exact inference in complex probabilistic models can be a NP-hard task. Description logic, usually used in combination with ontologies, is another LP extension effective for modelling concepts, roles, individuals, and their relationships, as well as to provide simple reasoning capabilities (Hu *et al.* 2012). However, only simple classification tasks can be solved, and no mechanisms are provided to infer more complex information from existing data. Also, design and implementation are typically more difficult and time consuming than with other approaches. Since uncertainty of information is the natural enemy of global consistency, our approach moves from the choice of abandoning the idea of globally consistent (in terms of both time and space) logic theories (or, knowledge bases—KB) in favour of *locally* consistent ones.

3 The LPaaS vision

The evolution of LP in parallel, concurrent, and distributed scenarios is the main motivation for re-interpreting the notion of *distribution* of LP in today's context. Since SOA is the *de facto* standard for distributed application development in both the academia and the industry (Erl 2005), Section 3.1 focuses on how LP can be re-interpreted in the *service* perspective. This perspective further emphasises the role of *situatedness*, already brought along by distribution in itself: Thus, Section 3.2 discusses how being situated in space, time, and context affects LP computation. The two novel perspectives are merged together in Section 3.3, which develops the idea of LP as a *situated service*.

3.1 The service perspective

The service-oriented perspective deeply affects the way in which LP engines are conceived, designed, and used—in particular, as far as the very nature of LP

https://doi.org/10.1017/S1471068418000364 Published online by Cambridge University Press

encapsulation is concerned, the way in which clients interact (requiring *statelessness*), and the assumptions about the surrounding context (*locality*) are concerned.

Encapsulation. A service hides both data representation and the computational mechanisms behind a public interface exposed to its clients. In the context of LP engines, this means that both the logic theory (the data) and the resolution process (the computational mechanism) are *inaccessible*—and, in general, *not observable*—from outside the boundary of the service interface. As a consequence, theory manipulation mechanisms, such as `assert/retract`, are no longer directly applicable from the client perspective: Since the logic theory is encapsulated by the service, dedicated mechanisms are required for its handling. For instance, in an IoT scenario, this may happen via a separate 'sensor API' through which sensor devices regularly update the KB of the LP service according to their perception of the surrounding environment.

Accordingly, the logic theory of a LPaaS service can be either *static* or *dynamic* (which are mutually exclusive configurations). The way in which the LP service can be accessed obviously depends on that: Time is an issue for dynamic KB, not for static ones.

Statelessness. Encapsulation makes it irrelevant *how* the encapsulated behaviour is implemented: What actually matters are the inputs triggering, and the outputs resulting from, that behaviour. Furthermore, in the SOA perspective, services are usually redundantly distributed over a network of hosts for enhancing the service availability and reliability: thus, it does not really matter *who* actually carries out the encapsulated behaviour. In the context of LP, this means that interactions with clients should be also allowed to be *stateless*—that is, include all the information required by the resolution process, since a different component may serve a different request. Notably, statelessness is the default for RESTful web services (WS), too.

It is worth emphasising here that statelessness does not contrast with the above encapsulation property, since the former regards the *invocation* of LPaaS services—hence the interaction between clients and servers—whereas the latter concerns LPaaS services themselves—that is, their inner nature. In other words, statelessness implies that servers are not supposed to track the state of interactions, so that a service request should not assume or rely on previous interactions, whereas encapsulation means that only the selected properties of the service are visible and modifiable from the outside.

At the same time, in order to cope with data-intensive applications, where stateless interaction may become cumbersome, LPaaS also supports *stateful* interaction—yet, at the clients' convenience and will. This is particularly handy for scenarios where reasoning and inference should be based on continuous and possibly unbounded streams of data, such as those coming from sensors in IoT deployments.

Locality. The distributed nature of the system drastically changes the perspective over consistency: maintaining *globally consistent* information is typically unfeasible in such systems. Furthermore, when pervasive systems enter the picture, even *globally available* information is usually not a realistic assumption: For instance, in IoT scenarios, heterogeneous data streams are continuously made available by sensor

https://doi.org/10.1017/S1471068418000364 Published online by Cambridge University Press

devices scattered in specific portions of the physical environment. As a consequence, encapsulation is inevitably bound to a specific, (*local*) portion of the system—with a notion of locality extending up to when/where availability and/or consistency are inevitably lost.

In the context of LP, this means first of all resorting to a *multi-theory* logical framework, exploiting the typical approach to *modularity* adopted in traditional LP in order to allow for parallel and concurrent computation (Bugliesi *et al.* 1994). Also, locality implies that each logic theory describes just what is *locally* true—which basically means leaving aside in principle the global acceptation of the *closed world* assumption (Reiter 1978) in favour of a more realistic *locally closed world* assumption. Accordingly, every LP service is to be queried about *what is locally known to be true*, with no need to resort to global knowledge of any sort—and with no need to distribute the resolution process in any way.

3.2 The situatedness perspective

The distribution of LP service instances directly calls for *situatedness*, intended as the property of the LP service to be immersed in the surrounding computational/physical environment, whose changes may affect its computations (Mariani and Omicini 2015). As an example, new sensor data may change the replies of an LP service to queries. Situatedness adds three new dimensions to LP computations: *space*, *time*, and *context*.

Space. To be situated in space means that the *spatial context* where the LP service is executing may affect its properties, computations, and the way it interacts with clients.

Distribution *per se* constitutes a premise to spatial situatedness: Each LP instance runs on a different device, thus on a different network host, therefore accessing the different computational and network resources that are *locally* available. Moreover, since LP services encapsulate the logic theory for their resolution process, the locally gathered knowledge affects the result, once it is represented in terms of logic axions.

Also, more articulated forms of spatial situatedness may be envisioned: For instance, *mobile* clients may request LP services from different locations at each request, possibly even *while* moving, which means that the LP service must be able to coherently identify and track clients so as to reply to the correct network address. Finally, it is possible in principle to conceive logic theories—or even individual axioms therein—with spatially bound validity, that is, that are true only in specific points or regions in space—analogously to *spatial tuples* in Ricci *et al.* (2017).

Time. Complementarily, being situated in time means that the *temporal context* when the LP service is executed may affect its properties, computations, and interactions with clients. Yet again, distribution alone already brings about temporal issues: Moving information in a network takes time; thus, aspects such as expiration of requests, obsolescence of logic theories, and timeliness of replies should be taken into account when designing the LP service.

Furthermore, since reconstructing a *global* notion of time in pervasive systems is either unfeasible or non-trivial, each LP service should operate on its own local time—that is, computing deadlines, leasing times, and the like according to its *local perception* of time. Also, in the same way as for spatial situatedness, temporal situatedness may also imply that logic theories or individual axioms may have their time-bounded validity—e.g., holding true up to a certain instant in time, and no longer since then.

Context. Besides the space/time fabric, situatedness also regards the generic *environment* within which LP services execute—that is, the computational and physical context which may affect their working cycle: For instance, it may depend on the available CPUs and RAM, whether an accelerometer is available on the current hosting device, etc.

A basic level of *contextual situatedness* is already embedded in the very nature of the LP service: In fact, locality of the resolution process implies that the logic theory for goal resolution belongs to the context of the LP service, thus straightforwardly affecting its behaviour. However, especially in the IoT scenarios envisioned for LPaaS, the computational and physical contexts may both impact the LP service: For example, sensor devices may continuously update the service KB with their latest perceptions, while actuators may promptly provide feedback on success/failure of physical actions.

3.3 Towards LP as a situated service

The above perspectives promote a radical re-interpretation of a few facets of LP, moving LP itself towards the notion of LPaaS envisioned in this paper—that is, in terms of a *situated service*. Such a notion articulates along four major lines:

- The preservation (with re-contextualisation) of the SLD resolution process.
- Stateless interactions.
- Time-sensitive computations.
- Space-sensitive computations.

The re-contextualisation of the SLD resolution process. The SLD resolution process (Robinson 1965) remains a staple in LPaaS: Yet, it is re-contextualised in the situated nature of the specific LP service. This means that, given the precise *spatial*, *temporal*, and *general* contexts within which the service is operating *when the resolution process starts*, the process follows the usual rules of SLD resolution: Situatedness is accounted for through the service abstraction with respect to such three contexts.

With respect to the *spatial context*, the resolution process obviously takes place in the hosting device *where* the LP service is running, thus taking into account the specific properties of the computational and physical environment therein available—CPU, RAM, network facilities, GPS positioning, etc.—there included the specific logic theory the LP service relies on. As mentioned in Section 3.2, more articulated forms of spatial situatedness—e.g., involving mobility of clients (and LP services, in principle), or, virtual/physical regions of validity for logic axioms—could be envisioned.

https://doi.org/10.1017/S1471068418000364 Published online by Cambridge University Press

The *temporal context* refers to the resolution process taking place on a *frozen snapshot* of the LP service state—there including its KB, which stays unaffected to external stimuli (possibly affecting the resolution process) until the process itself terminates. This way, despite the dynamic nature of the KB—encapsulated by the service abstraction—which could change, e.g., due to sensors' perceptions—the resolution process is guaranteed to operate on a consistent stable state of the logic theory.

Finally, the resolution process depends on the *general context* of the specific device hosting the LP service instance—thus considering the state of the KB therein available, as assembled by, e.g., the set of sensors devices therein available, the service agents gathering new local information, and so on.

Stateless interactions. A first change brought by LPaaS concerns interaction with clients of the LP service.

In classical LP, interactions are necessarily *stateful*: The user first sets the logic theory, then defines the goal, and then asks for one or more solutions, iteratively. This implies that the LP engine is expected to store the logic theory to exploit as its KB, to memorise the goal under demonstration, and to track how many solutions have been already provided to the user—and all these items become part of the state of the LP engine.

Instead, in LPaaS interactions are first of all (even though not exclusively) *stateless*: Coherently with SOA, the LP service instance that actually serves each request may be different at each time, e.g., due to redundancy of distributed software components aimed at improving availability and reliability of the LP service. In such a perspective, each client query (interaction) should be possibly self-contained, so that it does not matter which specific service instance responds—because there is no need for it to track the state of the interaction session.

It is worth emphasising that in the case of stateful interaction, adequate measures need be taken to prevent possible problems related to different LPaaS service instances serving repeated requests, requests from mobile clients, and similar situations. Two possible solutions could be considered for this concern: (i) the LPaaS middleware could simply *lock* a service in case of stateful interaction, ensuring that the client always interacts with the same service instance (this is essentially the problem of *conversational continuity*, well documented in the literature; Gelernter, 1985); (ii) alternatively, the LPaaS middleware could take care of the *hand-off* of the interaction from instance to instance, ensuring proper sharing of the information needed to preserve statefulness across service instances.

Time-local computation. Another change stemming from the situated nature of LPaaS is concerned with the relationship between the resolution process and the flow of time.

In pure LP, the logic theory is simply assumed to be always valid, and time-related aspects do not affect the resolution process; for instance, assertion/retraction mechanisms are most typically regarded as extra-logic ones. As discussed above, in LPaaS the consistency of the resolution process is guaranteed by the fact that the possibly ever-changing KB encapsulated by the service is *frozen* in time when the

resolution process itself begins: Nevertheless, time situatedness requires by definition that time affects the LP service computation in some way.

Accordingly, in LPaaS each axiom in the KB is decorated with a *time interval*, indicating the time validity of each clause. Every time a new resolution process starts in order to serve a LPaaS request, the logic theory used is the one containing all and only the axioms holding true at the *timestamp* associated with the resolution process itself. In the simplest case, such a timestamp is implicitly assigned by the LP server as the current local time when the request for goal demonstration is first served. Otherwise, it could also be explicitly assigned by clients along with the request—e.g., defining a specific time when asking for a goal demonstration.

Space-local computation. Analogously, classical LP has no notion of space situatedness: Be it a virtual or a physical space, the LP engine is a monolithic component providing its 'services' only locally, to its co-located 'clients' working on the same machine.

The LPaaS interpretation stems again from the very nature of service in modern SOA-based applications—a computational unit providing its functionalities through a network-reachable endpoint. Therefore, the resolution process in LPaaS is naturally and inherently affected by the specific *computational locus* where a given LP service instance is executing at a given moment—there including the locally available resources.

4 The LPaaS architecture

LPaaS is a logic-based, service-oriented approach for *distributed situated intelligence*, conceived and designed as the natural evolution of LP in nowadays pervasive computing systems. Its purpose is to enable situated reasoning via explicit definition of the spatio-temporal structure of the environment where situated entities act and interact.

Along the lines traced in Section 3.3, we now elaborate more practically on how encapsulation, statelessness, and locality—that is, the *service perspective* (Section 3.1)—are exploited in LPaaS according to the three dimensions of *situatedness* described in Section 3.2—that is, time, space, and context. Then, we briefly describe microservices (Familiar 2015) as a key enabler architecture for LPaaS.

4.1 Service architecture

Encapsulation. As it straightforwardly stems from SOA principles, encapsulation is exploited in LPaaS so as to define a standard API that shields LPaaS clients from the inner details of the service while providing suitable means of interaction.

Accordingly, each LP server node exposes its LP service to clients via two interfaces, depicted in Figure 1:

Client Interface exposes methods for *observation* and *usage*. *Client* refers to any kind of users, either individuals (humans, software agents) or groups entitled to exploit the LPaaS services.

https://doi.org/10.1017/S1471068418000364 Published online by Cambridge University Press

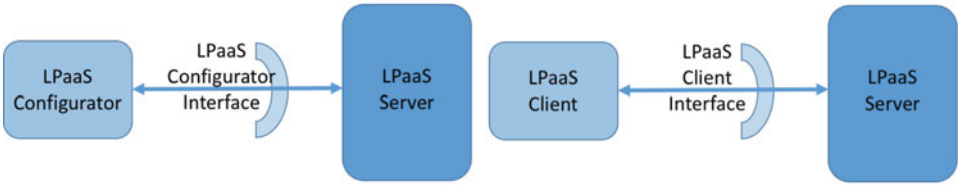

Fig. 1. LPaaS configurator service Architecture (left) and client service architecture (right).

Configurator Interface enables service *configuration* and requires proper access credentials. *Configurator* refers to service managers—privileged agents with the right of enforcing control and policies for that local portion of the system.

Applications can access the service as either *Clients* or *Configurators*, via the corresponding interfaces. The service is initialised at deployment-time on the server machine: Once started, it can be dynamically re-configured at run-time by any configurator.

Locality. Situatedness is exploited as a means to consistently handle *locality* w.r.t. context, time, and space.

In fact, dealing with situated logic theories means to give up with the idea of global consistency in a closed world: In LPaaS, multiple KB are spread throughout a network infrastructure, likely geographically distributed, executing within different computational contexts, and possibly either fed by sensors or manipulated by service agents perceiving the physical context. By allowing distributed access and reasoning over its own locally situated knowledge base, each LPaaS node actively contributes to the overall availability of the global knowledge.

Accordingly, pervasive application scenarios where logic theories represent local knowledge inherently call for *dynamic KB*, *autonomously* evolving during the service lifetime[1]. As such, each situated KB of a LPaaS service can be seen as representing what is known to be true and relevant in a given location in space at a given time, thus possibly changing over time—e.g., due to data streams coming from sensor devices—and potentially different from any other KB located elsewhere—as depicted in Figure 2.

Accordingly,

- each LPaaS clause has a lifetime, expressed as a time interval of validity—as in the case of the current temperature in a room;
- as a result, at any point t in time a LPaaS service has precisely one logic theory made of all and only the clauses that hold true at time t;
- each LPaaS resolution process is either implicitly (by the LPaaS server) or explicitly (by the LPaaS client) labelled with a *timestamp*, used to determine the KB to be used for the resolution itself—which then works as the standard LP resolution.

[1] Here, 'autonomously' means that in the LPaaS perspective the logic KB may evolve over time with no need for a client to invoke `assert/retract`, or equivalent methods—which, in fact, are not included in the LPaaS standard API detailed in Section 5.1—but, e.g., due to sensor devices' perceptions transparently feeding the LP service KB.

https://doi.org/10.1017/S1471068418000364 Published online by Cambridge University Press

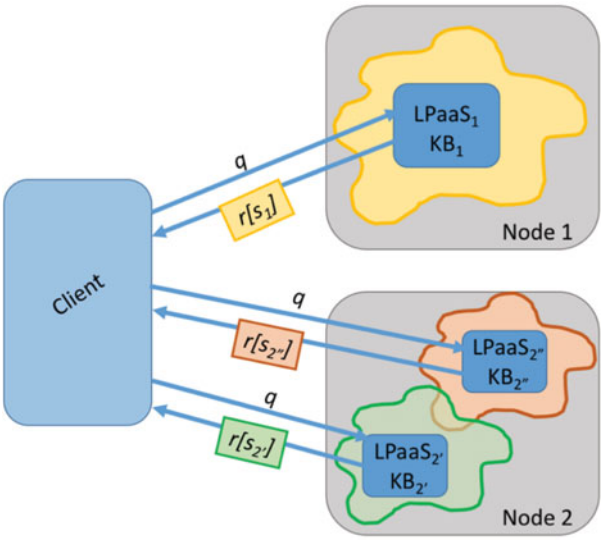

Fig. 2. Situatedness of LPaaS: The same query (q) by the same client may be resolved differently ($r[s_1], r[s_{2'}], r[s_{2''}]$) by distinct LPaaS services ($LPaaS_1, LPaaS_{2'}, LPaaS_{2''}$) based on their local computational, physical, and spatio-temporal context ($S_1, S_{2'}, S_{2''}$).

Statelessness. Uncoupling is one of the main requirements for interaction in distributed systems: That is why LPaaS provides stateless client–server interaction as one of its main features. The same holds true in particular for pervasive systems, where *instability* is one of the main issues, as well as for mobile systems, with any sort of mobility: physical mobility of users and devices; users who change their computing device while using applications; service instances migrating from machine to machine—as in a cloud-based environment.

The need for uncoupling promotes *stateless interaction* in LPaaS. Thus, for instance, both LPaaS clients and service instances can freely move with no concerns for requests tracking and identity/location bookeeping.

In order to balance the effect of statelessness on data-intensive interactions between LP service and users, LPaaS also provides clients with the ability to ask for more than one solution at a time, and even all of them, with a single request. Nevertheless, LPaaS also makes it possible to obtain a *stream of solutions* from the resolution process, rather than a single solution at a time in an individual interaction session, to better meet the needs of fast-paced dynamic scenarios in which clients want to be constantly updated by the LP service about some situation.

Accordingly, LPaaS provides clients with the means to obtain both stateless and stateful client–server interaction:

Stateful once the logic theory to consider is settled, and the goal stated, the client should be able to ask for any amount of solutions—possibly iteratively, possibly at different times and from different places—with the service being responsible to guarantee consistency and validity of solutions by keeping track of the related interaction sessions with the same client.

https://doi.org/10.1017/S1471068418000364 Published online by Cambridge University Press

Stateless in this case, no session state is tracked by the server, so each client request should contain all the information required to serve the request itself.

It is worth highlighting that nothing prevents the service from being stateful and stateless simultaneously, because the LP server can manage multiple kinds of requests concurrently; instead, of course, each client request in LPaaS is either stateful or stateless.

4.2 Microservices as technology enablers

SOAs represent nowadays the standard approach for distributed system engineering (Erl 2005): So, LPaaS adopts the *Software as a Service* (SaaS) architecture as its reference (Cusumano 2010).

Accordingly, information technology resources are conceived as continuously provided services: SaaS applications are supposed to be available 24/7, scale up and down elastically, support resiliency to changes (i.e., in the form of suitable fault-tolerance mechanisms), provide a responsive user experience on all popular devices, and require neither user installation nor application updates.

In particular, LP services in LPaaS can be fruitfully interpreted as *microservices* (Familiar 2015). Microservices are a recent architectural style for SaaS applications promoting usage of self-contained units of functionally with loosely coupled dependencies on other services: As such, they can be designed, developed, tested, and released independently. Thanks to their features, microservices are deserving increasing attention also in the industry—pretty much like SOA in the mid 2000s—where fast and easy deployment, fine-grained scalability, modularity, and overall agility are particularly appreciated (Richards 2016).

Technically speaking, microservices are designed to expose their functionality through standardised network-addressable APIs and data contracts, making it possible to choose the programming language, operating system, and data store that best fit the service needs and the developers' skills set, without worrying about interoperability. Microservices should also be dynamically *configurable*, possibly in different forms and with different configuration levels. Obviously, actual support to interoperability requires multiple levels of standardisation: To this end, LPaaS defines its own *interfaces* for both *configuration* and *exploitation*, while relying on widely adopted standards as far as the representation formats (i.e., JSON 2017) and interaction protocols (i.e., REST over HTTP, or MQTT 2017) are concerned.

5 The LPaaS service

Following the reference architecture above, designing LPaaS amounts first of all at defining the Configurator Interface and the Client Interface—as in Figure 1.

Generally speaking, the LP service should support (i) *observational methods* to provide configuration and contextual information about the service, (ii) *usage methods* to trigger computations and reasoning, as well as to ask for solutions, and (iii) *configuration methods* to allow the configurator to set the LP service configuration.

https://doi.org/10.1017/S1471068418000364 Published online by Cambridge University Press

Table 1. *LPaaS Configurator Interface*

```
setConfiguration(+ConfigurationList)
getConfiguration(-ConfigurationList)
       resetConfiguration()

        setTheory(+Theory)
        getTheory(-Theory)
       setGoals(+GoalList)
       getGoals(-GoalList)
```

Observational methods make it possible to query the service about its configuration (stateful/stateless, static/dynamic), the state of the knowledge base, and the admissible goals: As such, they belong to the Client Interface, but can be made available also in the Configurator Interface for convenience. Usage methods, instead, belong uniquely to the Client Interface: They allow clients to ask for one or more solutions—one solution, *n* solutions, or all solutions available, for stateful or stateless requests as well. Configurator methods belong uniquely to the Configurator Interface, and are intended to set the service configuration, KB nature, and admissible goals.

5.1 Service interfaces

Adopting the Prolog notation for input/output (Deransart *et al.* 1996), the actual Configurator methods are detailed in Table 1, while the Client Interface is detailed in Table 2. Since the first is rather self-explanatory, we focus on the Client Interface.

The first thing worth noting is that usage predicates for stateless and stateful requests are slightly different from each other. In the case of stateless requests, the `solve` operation is conceptually atomic and self-contained—so, e.g., the `Goal` to solve is always one of its arguments; instead, in the case of stateful requests it is up to the server to keep track of the request state, so the goal is to be set only once by the client before the first `solve` request is issued.

The second key aspect is the threefold impact of *time awareness*: Regardless of whether the server is either computing or idle, time flows anyway, so predicates have to be time-sensitive. Accordingly,

- `solve` predicates can also contain a `Timeout` parameter (server time) for the resolution, so as to avoid blocking the server indefinitely: If the resolution process does not complete within the given time, the request is cancelled, and a negative response is returned;
- for *stateful requests*, the client could also ask for a *stream* of solutions, which is particularly useful in IoT scenarios exploiting sensor devices, or monitoring processes: To this end, `solve` takes a `time` argument (server time), meaning that each new solution should be returned not faster than every `time` milliseconds;
- when the KB is *dynamic*, all predicates take an additional `Timestamp` argument, meaning that each theory has a *time-bounded validity*: This feature can be used

https://doi.org/10.1017/S1471068418000364 Published online by Cambridge University Press

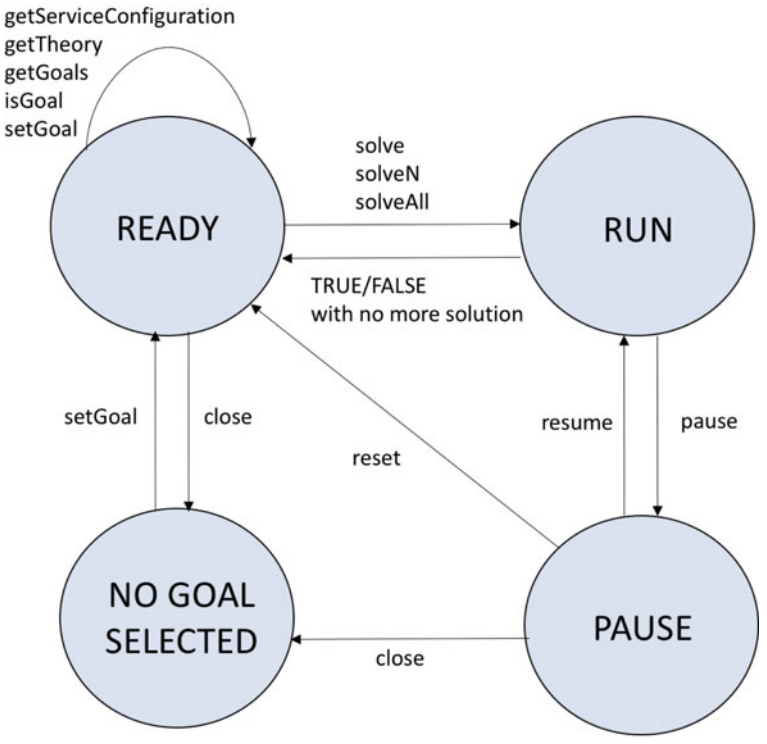

Fig. 3. The LPaaS Finite State Machine.

during the proof of a goal to ensure that only the clauses valid at the given `Timestamp` are taken into account in that resolution process.

For the sake of convenience, `solveAfter` methods are provided for mimicking the LP stateful interaction on a stateless request channel, fast-forwarding to the `N+1` solution `AfterN`.

Finally, the `reset` primitive resets the resolution process, with no need to reconfigure the service (i.e., re-select the goal); in contrast, the `close` primitive actually closes the communication with the server, so the goal must be re-set before re-querying the server.

5.2 Computational model

The computational model of the service is depicted by the Finite State Machine in Figure 3, made of four states:

- *ready* (initial state) – where the service is started and the engine is configured;
- *run* – where the service is undergoing some resolution process triggered by queries;
- *pause* – representing the temporary suspension of computations;
- *no goal selected* (final state) – when the client connection is closed.

In the *ready* state, the service can be queried about its properties and a new goal can be set, thus defining a new resolution process. When a new query is submitted, the

https://doi.org/10.1017/S1471068418000364 Published online by Cambridge University Press

Table 2. *LPaaS client interface*

STATIC KNOWLEDGE BASE

Stateless	Stateful
getServiceConfiguration(-ConfigList)	
getTheory(-Theory)	
getGoals(-GoalList)	
isGoal(+Goal)	
	setGoal(template(+Template))
	setGoal(index(+Index))
solve(+Goal, -Solution)	solve(-Solution)
solveN(+Goal, +NSol, -SolutionList)	solveN(+N, -SolutionList)
solveAll(+Goal, -SolutionList)	solveAll(-SolutionList)
solve(+Goal, -Solution, within(+Time))	solve(-Solution, within(+Time))
solveN(+Goal, +NSol, -SolutionList, within(+Time))	solveN(+NSol, -SolutionList, within(+Time))
solveAll(+Goal, -SolutionList, within(+Time))	solveAll(-SolutionList, within(+Time))
solveAfter(+Goal, +AfterN, -Solution)	
solveNAfter(+Goal, +AfterN, +NSol, -SolutionList)	
solveAllAfter(+Goal, +AfterN, -SolutionList)	
	solve(-Solution, every(@Time))
	solveN(+N, -SolutionList, every(@Time))
	solveAll(-SolutionList, every(@Time))
	pause()
	resume()
reset()	
close()	

DYNAMIC KNOWLEDGE BASE

Stateless	Stateful
getServiceConfiguration(-ConfigList)	
getTheory(-Theory, ?Timestamp)	
getGoals(-GoalList)	
isGoal(+Goal)	
	setGoal(template(+Template))
	setGoal(index(+Index))
solve(+Goal, -Solution, ?Timestamp)	solve(-Solution, ?Timestamp)
solveN(+Goal, +NSol, -SList, ?TimeStamp)	solveN(+N, -SolutionList, ?TimeStamp)
solveAll(+Goal, -SList, ?TimeStamp)	solveAll(-SolutionList, ?TimeStamp)
solve(+Goal, -Solution, within(+Time), ?TimeStamp)	solve(-Solution, within(+Time), ?TimeStamp)
solveN(+Goal, +NSol, -SList, within(+Time), ?TimeStamp)	solveN(+NSol, -SList, within(+Time), ?TimeStamp)
solveAll(+Goal, -SList, within(+Time), ?TimeStamp)	solveAll(-SList, within(+Time), ?TimeStamp)
solveAfter(+Goal, +AfterN, -Solution, ?TimeStamp)	
solveNAfter(+Goal, +AfterN, +NSol, -SList, ?TimeStamp)	
solveAllAfter(+Goal, +AfterN, -SList, ?TimeStamp)	
	solve(-Solution, every(@Time), ?TimeStamp)
	solveN(+N, -SList, every(@Time), ?TimeStamp)
	solveAll(-SList, every(@Time), ?TimeStamp)
	pause()
	resume()
reset()	
close()	

https://doi.org/10.1017/S1471068418000364 Published online by Cambridge University Press

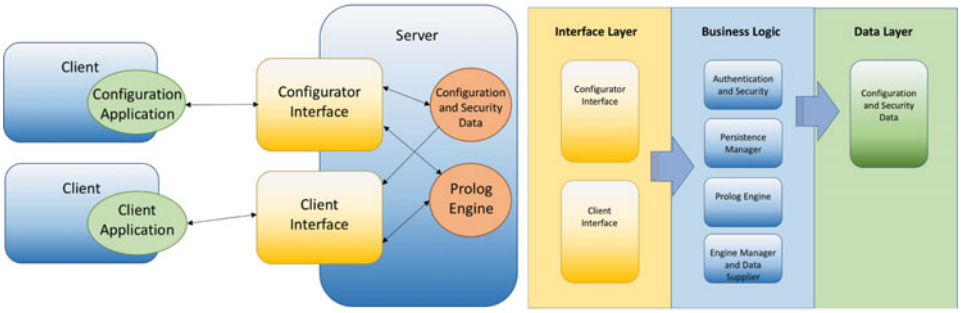

Fig. 4. The LPaaS RESTful Web Service (left) and Server architecture (right).

service moves to the *run* state, indicating that a resolution process is taking place. Computation may then be paused several times, causing the service to move back and forth from the *pause* state: From there, resolution can also be reset (coming back to the initial state), or closed (moving to state *no goal selected*).

6 LPaaS in tuProlog

To test the effectiveness of the proposed model and architecture, we implement a first prototype of LPaaS as a RESTful WS (Fielding and Taylor 2002), embracing the *Web of Things* (WoT) vision (Heuer *et al.* 2015). Accordingly, our approach follows the WoT perspective in re-interpreting the 'things' (as well as their functionalities and data streams) as RESTful resources accessible through WS protocols, addressing the need to harmonically exploit all the components of the IoT system by virtualising individual things in some sort of software abstraction. There, each interaction session starts with a client request conveying the so-called 'method' information (i.e., how the receiver has to process the request) and the 'scope' information (i.e., which is the target data). Then, computations occur on the receiving side, where the target resource applies the method to the scope. The result is a response conveying an optional representation of the requested resource (functionality or data).

The computational model of the prototype reflects the state machine described in Figure 3. We reuse and adapt patterns commonly used for the REST architectural style, and introduce a novel architecture which supports the embedding of Prolog engines into WS. Figure 4 (left) shows the general architecture of the server side and its components (access interfaces, Prolog engine, and data store), as well as some exemplary client applications interacting via HTTP requests and JSON objects.

Figure 4 (right) shows the server inner architecture, composed of three logical layers: the interface, the business logic, and the data layer. The interface layer encapsulates the Configurator and Client interfaces. The business logic layer wraps the Prolog engine with the aim of managing incoming requests consistently. The data layer is responsible for managing the data store tracking, i.e., all the configuration options necessary to restore the service in case of unpredictable shutdown (i.e., operating parameters and security metadata such as clients' role, username, and password).

https://doi.org/10.1017/S1471068418000364 Published online by Cambridge University Press

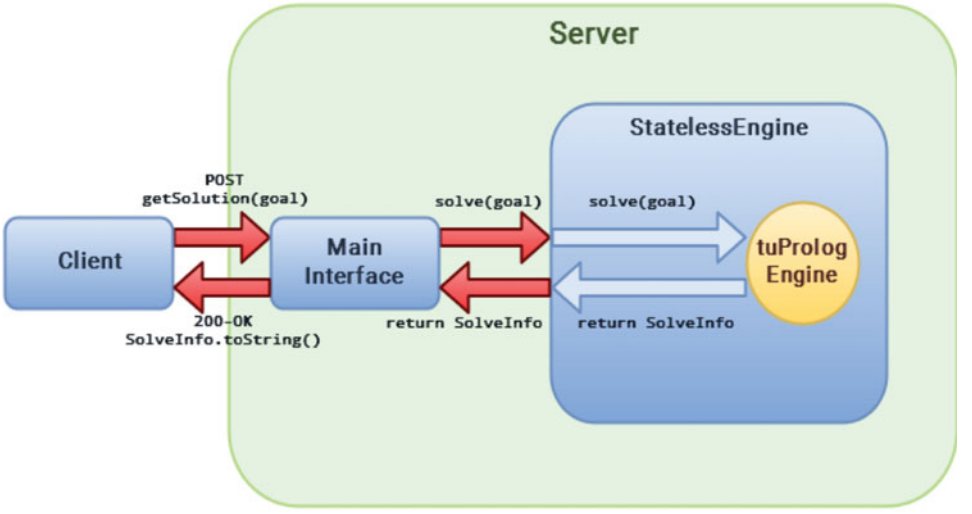

Fig. 5. Client–server interaction – inner calls.

Since those data are expected to be rather limited in size for most scenarios, we choose to keep them in the server application so as to offer a light-weight, self-contained service: However, they could be easily moved to a separate persistence layer on, i.e., an external DB application, if necessary.

The server implementation is realised by exploiting a plurality of technologies that are commonly found in the SOA field: The business logic is realised on the J2EE framework (J2EE 2017), exploiting EJB (EJB 2017), while the database interaction is implemented on top of JPA (Java Persistence API 2017).

The Prolog engine is implemented on top of the tuProlog system (Denti *et al.* 2001), which provides not only a light-weight engine, particularly well-suited for the envisioned pervasive computing scenarios, but also a multi-paradigm and multi-language working environment, paving the way towards further forms of interaction and expressiveness. Also, tuProlog 3.2 supports JSON serialisation natively, ensuring the interoperability required by a WS. The tuProlog engine, distributed as a Java JAR or Microsoft .NET DLL, is easily deployable and exploitable by applications as a *library service*—that is, from a software engineering standpoint, a suitably encapsulated set of related functionalities.

Figure 5 shows a client–server interaction in case of a stateless request. The StatelessEngine component, realised exploiting a Stateless Bean, wraps the Prolog engine object to manage the concurrent requests transparently.

The service interfaces exploit the EJB architecture, but can also be accessed as RESTful WS, realised using JAX-RS Java Standard implemented in the Jersey library (Jersey 2017). Security is based on JOSE (jose.4.j 2017), an open source (Apache 2.0) implementation of JWT and the JOSE specification suite. The application is deployed using the Payara Application Server (Payara 2017), a Glassfish open-source fork.

https://doi.org/10.1017/S1471068418000364 Published online by Cambridge University Press

6.1 tuProlog-as-a-service in action

The tuProlog-as-a-Service prototype is freely available on Bitbucket (LPaaS 2018) with the corresponding installation guide.

Two different prototype implementations are provided: LPaaS as a *RESTful WS* and LPaaS as an *agent in an agent society* (MAS), both built on top of the tuProlog system that provides the required interoperability and customisation. The first aims to emphasise how LPaaS can effectively support REST, probably the most typical IoT paradigm, while the second means to highlight the LPaaS effectiveness in supporting and promoting distributed situated intelligence.

The concrete implementations are discussed in detail in Calegari *et al.* (2017), Calegari *et al.* (2018). In Calegari *et al.* (2017), a Smart Bathroom is supposed to monitor physiological functions to deduce symptoms and diseases, alerting the user via an Android app as appropriate: Local sensors could perform situated reasoning, applying their local knowledge to aggregate the raw data and produce higher level synthesised information. Calegari *et al.* (2018), instead, discusses a Smart Kitchen where devices provide information about food supply and users' preferences, generating high-level knowledge used to coordinate and collaborate with other entities in the system.

7 Case study and discussion

The following section is meant to illustrate the LPaaS approach and its benefits by means of a running example in the Smart House field: Then, we compare the result with a more traditional LP approach.

The case study concerns the automatic assembly of home furniture by a domestic robot. The robot is in charge of the assembly operation, and the furniture pieces are supposed to be augmented with some form of computational capability—from simple RFID tags for being discoverable, up to embedded chips to store data and perform simple inference tasks. The key aspect is that the installation instruction, the location, and the assembly constraints (such as avoid putting heavy things on fragile walls) are not known in advance by the robot itself, and have to be derived by suitably exploiting situated knowledge.

As shown in Figure 6, the envisioned system features the following actors:

- The Designer agent, hosted in the Cloud, owner of and responsible for the house design project.
- The Assembler robotic agent, responsible for assembling the furniture.
- SmartFurniture LPaaS agents, owners of and responsible for storing and making available their own assembly instructions (i.e., the 'Billy' bookshelf provides installation instructions for itself only).
- SmartWall LPaaS agents, each responsible for knowing the structural properties of a given wall (i.e., materials of constructions, maximum allowable weight, and so on).

The Assembler acts like a human with the goal of assembling all the furniture in the house according to the envisioned design, but conscious of the unexpected

https://doi.org/10.1017/S1471068418000364 Published online by Cambridge University Press

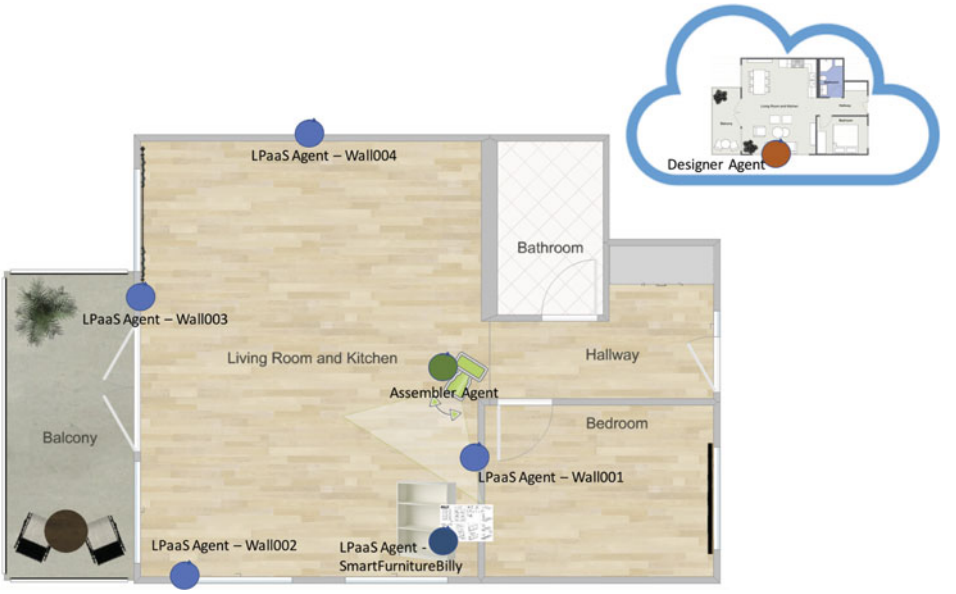

Fig. 6. IoT home: Case study.

contingencies that may arise (fragile walls, wrong measures, etc.). Moreover, exactly like a human, it does not know how to assemble a given piece of furniture in advance—it needs the installation instructions. Accordingly, the Assembler first acquires the *procedural knowledge* it needs—essentially, the set of plans for assembling the furniture of interest—by exploiting the intelligence embedded in the surrounding environmental structures (the walls, the ceiling, and the floor) and furniture. To this end, the Assembler needs not to be a full-fledged LPaaS agent, but can be a much simpler LPaaS client, with just the capability of requesting the LPaaS service. Its (normal) workflow is thus as follows:

- Once in a room, it selects a wall and asks the Designer which pieces of furniture are to be positioned against that wall.
- Then, it starts discovering the LPaaS agents representing such pieces, and asks them the pre-conditions they need for a successful assembly—i.e., about the structural properties the wall should have, or any other relevant property.
- Then, the Assembler interacts with the targeted SmartWall LPaaS agent to check if such pre-conditions are satisfied—notice that this check is delegated to the wall itself, which is the only one bearing the situated knowledge needed to effectively evaluate the feasibility of the design project.

In case of unforeseen situations, the SmartWall agent proposes an alternative disposition of the furniture that would be possibly implemented by the Assembler if the Planner agrees. Yet again, the Assembler simply exploits the situated intelligence of the LPaaS services in its surroundings. We would like to emphasise that this is the only way in which the same simple robot may be able to assemble (in principle) an unbounded number of heterogeneous pieces of furniture in all sorts of different

https://doi.org/10.1017/S1471068418000364 Published online by Cambridge University Press

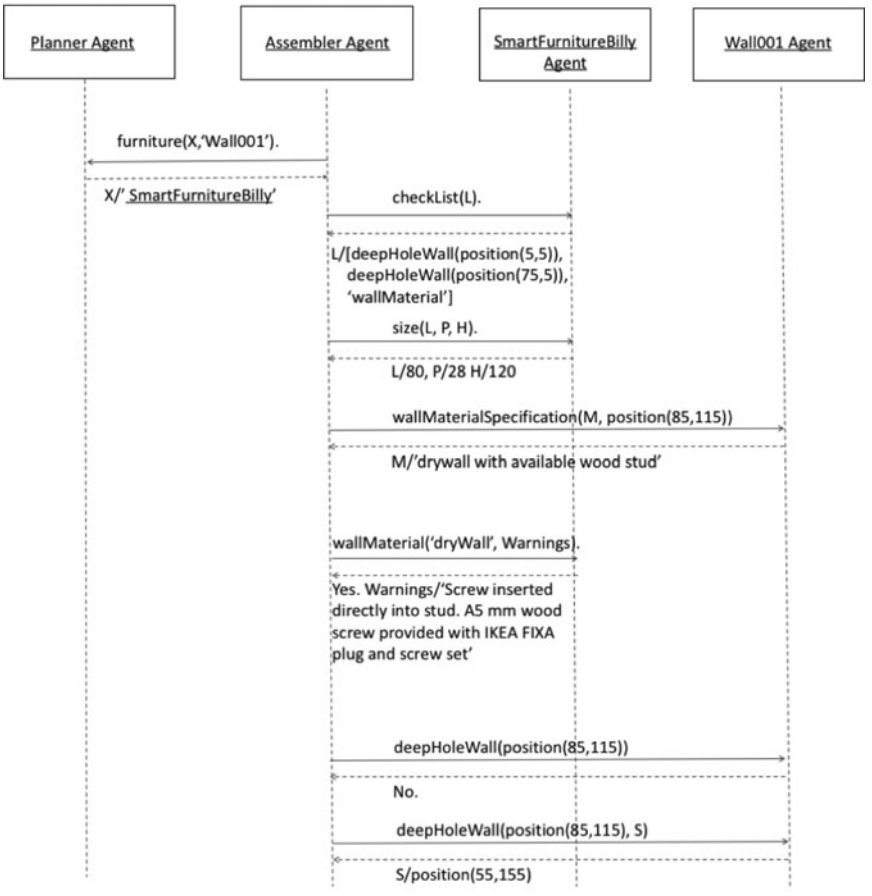

Fig. 7. Example of agents interaction.

walls, ceilings, floors, without hard-coding the instructions in its knowledge base, design update/patch mechanisms, or resort to code-on-demand features.

This is the kind of situated scenarios that LPaaS is most suited for, especially if compared with traditional LP approaches. There, in fact, the robot would be in need of storing its own knowledge base—the logic theory—describing how to build the furniture, how to match the single pieces of furniture against walls, ceilings, etc., and overall the whole home project. Besides leading to undesired centralisation and to a monolithic design, the most negative effect of such an approach is that the robot would be unable to work with new, unknown kinds of furniture—ultimately hindering flexibility, extensibility, and maintainability over time.

An example run is shown in Figure 7:

- Fixing the library to the wall requires two deep holes, but one of them cannot be done due to a chimney (detected by the SmartWall agent).
- The SmartWall then, undertaking an inference step on its own knowledge base, proposes an alternative solution (i.e., to move the position of the hole

https://doi.org/10.1017/S1471068418000364 Published online by Cambridge University Press

of 30 cm), which is then implemented by the Assembler after the Designer's approval.

It is worth noting that all the SmartFurniture and SmartWall agents implement, respectively, the same service—that is, all SmartFurniture agents answer the same queries, and all SmartWall agents do the same for their own set of queries—; however, the answer, unlike traditional LP, could be different because of the surrounding situation. For instance, given the query `wallMaterialSpecification(M, position(X,Y))`, agent `SmartWall001`, responsible for wall #1, based on its wall material could reply `M/'drywall with available wood stud'`, while agent `SmartWall002`, responsible for a different wall with different characteristics, could reply `M/'masonry'`.

Despite its simplicity, the case study above highlights the effectiveness of the LPaaS approach in spreading intelligence in pervasive systems enabling ubiquitous intelligence. The approach turns out to be particularly interesting when dealing with different contexts, because taking into account local knowledge, situated in time and space, enables the system to take autonomous real time decisions based on the specific situation. Moreover, relying mostly on locally available information reduces both the bandwidth consumption and the need for reliable communications between the distributed components, which are highly desirable features in IoT scenarios.

Also, modularity and encapsulation of LPaaS improve *scalability* of the LPaaS deployment: In traditional LP, there would be a single LP engine to scale up/down depending on the demand coming from clients, which translates to scaling a singleton monolithic entity as a whole, even if the actual need for scaling only concerns a portion of its functionality—i.e., only queries regarding a given portion of its knowledge base. In LPaaS instead—provided that the overall LP system functionality has been appropriately designed by distributing sub-functionalities to a set of distributed situated LPaaS services—whenever any given portion of the LP inference service suffers from excessive demand, only that portion of the system needs to be scaled up—namely, only that LPaaS service instance.

Another benefit related to modularity and encapsulation is that LPaaS lends itself to application in real-time scenarios. First of all, splitting out the LP service in multiple, smaller instances, responsible for a well-defined portion of the knowledge needed by the application at hand, helps achieving greater performance while doing inference, compared to traditional LP. Second, time-awareness of LPaaS helps dealing with time-related aspects, such as discarding obsolete knowledge, ignore old requests, setting temporal bounds on the resolution process, etc. Third, the IoT-oriented perspective according to which LPaaS clients do not need assert/retract mechanisms for manipulating the LP service knowledge base, because that functionality is envisioned for situated sensors and actuators directly interacting with the service through a dedicated API, helps ensuring that each LP service instance is always up-to-date with the state of the local world as per sensors' perceptions.

Nevertheless, we would like to point out that a real-time deployment of the currently available LPaaS prototypes—the RESTful and MAS-oriented tuProlog

https://doi.org/10.1017/S1471068418000364 Published online by Cambridge University Press

implementations—has not yet been experimented, thus the above discussion is mostly based on speculation.

8 Related work

The SOA paradigm is widely used in IoT scenarios (Cannata *et al.* 2010; Guinard *et al.* 2010b; Guinard *et al.* 2010a; Pontelli *et al.* 2008; Karnouskos *et al.* 2012; Messina *et al.* 2017). Moreover, communication via REST enables the direct integration of SOA-ready devices (i.e., devices hosting native WS).

MobIoT (Hachem *et al.* 2014) provides efficient service discovery, composition, and access in heterogeneous, dynamic, mobile IoT contexts, revisiting the standard SOA approach by providing probabilistic registration, look-up and thing-based composition based on comprehensive ontologies. However, it does not support runtime interaction with users to let them specify their goals, and still needs proper validation from the scalability viewpoint when the number of registered services is very large.

Pontelli *et al.* (2008) present a comprehensive LP framework designed to support intelligent composition of WS. The work proposes a theoretical framework for reasoning with heterogeneous knowledge bases, which can be combined with logic programming-based planners for WS composition. The framework makes a step towards the interoperation between knowledge bases encoded using different rule markup languages and the integration of different components that reason about knowledge bases. Unlike our framework, the system is not focused on situated reasoning: Rather, it is mainly concerned about dealing with heterogeneous WS in the context of the WS composition problems.

A novel approach for engineering IoT systems is proposed by Alkhabbas *et al.* (2017), where a set of things with their functionalities and services is connected and led to cooperate temporarily so as to achieve a given goal. Moreover, many research work deal with event-driven SOA (EDA-SOA) (Schulte and Natis 2003; Michelson 2006)—where communication between users, applications and services is carried out by events, rather than using remote procedure calls. In particular, Prado *et al.* (2017) propose an event-driven SOA that provides context awareness in the scope of IoT, whereby the generation of an event can trigger the concurrent execution of one or more services. When a given event occurs, different services can be triggered automatically, endowing the system with the capability of real-time sensing and rapid response to events in a loosely coupled, distributed computing environment. In general, pure SOA and EDA have their own limitations, but could complement each other; that is, some degree of service coordination can be achieved among mutually independent services through the event mechanism. As mentioned by Cheng *et al.* (2017), such a complementarity suits well the features of IoT, requiring high autonomy inside a domain and efficient coordination across domains; furthermore, it both improves the real-time response to constantly changing business requirements and minimises the impact on the existing application system to allow a large-scale, distributed IoT service application to be easily developed and maintained.

Many aspects developed in the aforementioned works have worked as sources of inspiration for LPaaS, in particular in how the model and architecture are conceived and designed. Following the SOA principles, LPaaS aims at modelling ubiquitous intelligence in a dynamic context by promoting portability and interoperable interaction over a network (via proper standards) and emphasising the separation of the service interface from its implementation. While following the EDA principles, LPaaS goes beyond the state-of-the-art mainly as far as context-awareness is concerned, in particular by supporting the injection of intelligence within existing services/agents via the awareness of the context, thus promoting their adaptivity.

9 Conclusion and future work

In this paper, we propose the LPaaS approach for *distributed situated intelligence* as the natural evolution of LP in the context of nowadays pervasive computing systems. We discuss its properties and its computational and architectural models by relating and comparing them to the notions and development of LP over the years, tracked in Section 8.

The main advantages of exploiting an LP-based approach in pervasive systems amount at (i) writing declaratively complex rules involving the context, (ii) assessing provable statements about the expressive power and decidability of the context model, and (iii) actually supporting light-weight reasoning and cooperation among distributed components. We also present a first prototype implementation built on top of the tuProlog system, to demonstrate and test the effectiveness of the LPaaS approach.

Our service-based approach, in particular, (i) encourages representing and reasoning with situations using a declarative language, providing a high level of abstraction; (ii) supports the incremental construction of context-aware systems by providing modularity and separation of concerns; (iii) promotes the cooperation and interoperation among the different entities of a pervasive system; and (iv) enables reasoning over data streams, like those collected by sensors.

Of course, a number of enhancements are still possible, both to the model and to the infrastructure. From the model viewpoint, specific *space-awareness* methods could be defined and added: For instance, a `solveNeighbours` primitive to deal with the space around either the client or the server, making it possible to opportunistically federate LP engines upon need as a form of dynamic service composition. From the infrastructure viewpoint, we plan to focus on the design and implementation of a specialised LP-oriented middleware, dealing with heterogeneity of platforms as well as with distribution, life-cycle, interoperability, and coordination of multiple situated Prolog engines—possibly based on the existing tuProlog technology and TuCSoN middleware (Omicini and Zambonelli 1999)—so as to explore the full potential of logic-based technologies in IoT scenarios and applications. Also, providing some sort of distributed service directories enabling dynamic discovery of LPaaS services—in turn promoting opportunistic interactions with clients or other services (for service composition)—is surely a promising path to follow to further widen the applicability of the LPaaS approach in pervasive scenarios.

https://doi.org/10.1017/S1471068418000364 Published online by Cambridge University Press

References

Alkhabbas, F., Spalazzese, R. and Davidsson, P. 2017. Architecting emergent configurations in the internet of things. In *Proc. of IEEE International Conference on Software Architecture (ICSA)*. IEEE, Los Alamitos, CA, USA, 221–224.

Bettini, C., Brdiczka, O., Henricksen, K., Indulska, J., Nicklas, D., Ranganathan, A. and Riboni, D. 2010. A survey of context modelling and reasoning techniques. *Pervasive and Mobile Computing 6,* 2, 161–180.

Brogi, A. and Ciancarini, P. 1991. The concurrent language, Shared Prolog. *ACM Transactions on Programming Languages and Systems 13,* 1, 99–123.

Brogi, A. and Gorrieri, R. 1989. A distributed, net oriented semantics for Delta Prolog. In *International Joint Conference on Theory and Practice of Software Development (TAPSOFT '89)*, J. Díaz and F. Orejas, Eds. Lecture Notes in Computer Science, vol. 351. Springer, Berlin Heidelberg, 162–177.

Brownlee, J. 2011. *Clever Algorithms: Nature-Inspired Programming Recipes*. Lulu Press, Morrisville, NC, USA.

Bugliesi, M., Lamma, E. and Mello, P. 1994. Modularity in logic programming. *Journal of Logic Programming 19–20*, 443–502. Special Issue: Ten Years of Logic Programming.

Calegari, R., Denti, E., Dovier, A. and Omicini, A. 2018. Extending logic programming with labelled variables: Model and semantics. *Fundamenta Informaticae 161*, 53–74.

Calegari, R., Denti, E., Mariani, S. and Omicini, A. 2017. Logic programming as a service (LPaaS): Intelligence for the IoT. In *Proc. of IEEE 14th International Conference on Networking, Sensing and Control (ICNSC 2017)*, G. Fortino, M. Zhou, Z. Lukszo, A. V. Vasilakos, F. Basile, C. Palau, A. Liotta, M. P. Fanti, A. Guerrieri and A. Vinci, Eds. IEEE, Los Alamitos, CA, USA, 72–77.

Calegari, R., Denti, E., Mariani, S. and Omicini, A. 2018. Logic programming as a service in multi-agent systems for the Internet of Things. *International Journal of Grid and Utility Computing (in press)*. URL: http://www.inderscience.com/info/ingeneral/forthcoming.php?jcode=ijgu

Cannata, A., Karnouskos, S. and Taisch, M. 2010. Evaluating the potential of a service oriented infrastructure for the factory of the future. In *Proc. of 8th IEEE International Conference on Industrial Informatics (INDIN 2010)*. IEEE, Los Alamitos, CA, USA, 592–597.

Chen, H., Finin, T. and Joshi, A. 2003. An ontology for context-aware pervasive computing environments. *The Knowledge Engineering Review 18,* 3, 197–207.

Cheng, B., Wang, M., Zhao, S., Zhai, Z., Zhu, D. and Chen, J. 2017. Situation-aware dynamic service coordination in an IoT environment. *IEEE/ACM Transactions on Networking 25,* 4, 2082–2095.

Clark, K. L. 1978. Negation as failure. *Logic and Data Bases*. Springer, Boston, MA, USA, 293–322.

Clark, K. L. 1987. PARLOG: The language and its applications. In *Proc. of PARLE Parallel Architectures and Languages Europe. Volume II: Parallel Languages. Eindhoven, The Netherlands, 15–19 June 1987*, J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, Eds. Lecture Notes in Computer Science, vol. 259. Springer, Berlin, Heidelberg, 30–53.

Clark, K. L. and Gregory, S. 1981. A relational language for parallel programming. In *1981 Conference on Functional Programming Languages and Computer Architecture (FPCA '81)*. ACM, New York, NY, USA, 171–178.

Cunha, J. C., Ferreira, M. C. and Pereira, L. M. 1989. Programming in Delta Prolog. In *Proc. of the 6th International Conference on Logic Programming (ICLP 1989)*, Lisbon, Portugal, 19–23 June 1989, G. Levi and M. Martelli, Eds. MIT Press, Cambridge, MA, USA, 487–504.

Cusumano, M. 2010. Cloud computing and SaaS as new computing platforms. *Communications of the ACM 53,* 4, 27–29.

Denti, E., Omicini, A. and Ricci, A. 2001. tuProlog: A light-weight Prolog for Internet applications and infrastructures. In *Proc. of 3rd International Symposium Practical Aspects of Declarative Languages (PADL 2001)*, Las Vegas, NV, USA, 11–12 Mar. 2001, I. V. Ramakrishnan, Ed. Lecture Notes in Computer Science, vol. 1990. Springer, Berlin, Heidelberg, 184–198.

Deransart, P., Ed-Dbali, A. and Cervoni, L. 1996. *Prolog: The Standard. Reference Manual.* Springer, Berlin, Heidelberg.

Dey, A. K. 2001. Understanding and using context. *Personal and Ubiquitous Computing 5,* 1, 4–7.

EJB. Home Page. URL: http://www.oracle.com/technetwork/java/javaee/ejb/. Accessed 10 May 2018.

Erl, T. 2005. *Service-Oriented Architecture: Concepts, Technology, and Design.* Prentice Hall/Pearson Education International, Upper Saddle River, NJ, USA.

Etter, R., Costa, P. D., and Broens, T. 2006. A rule-based approach towards context-aware user notification services. In *Proc. of 2006 ACS/IEEE International Conference on Pervasive Services (ICPS 2006)*. IEEE, Lyon, France, 281–284.

Familiar, B. 2015. *Microservices, IoT, and Azure: Leveraging DevOps and Microservice Architecture to Deliver SaaS Solutions*, 1st ed. Apress, Berkely, CA, USA.

Fielding, R. T. and Taylor, R. N. 2002. Principled design of the modern Web architecture. *ACM Transactions on Internet Technology 2,* 2, 115–150.

Finin, T., Joshi, A., Kagal, L., Ratsimore, O., Korolev, V. and Chen, H. 2001. Information agents for mobile and embedded devices. In *Proc. of 5th InternationalWorkshop Cooperative Information Agents V (CIA 2001)*, Modena, Italy, 6–8 May 2001, M. Klusch and F. Zambonelli, Eds. Springer, Berlin, Heidelberg, 264–286.

Gallaire, H. and Minker, J., Eds. 1978. *Logic and Data Bases.* Springer, Boston, MA, USA.

Gelernter, D. 1985. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems 7,* 1, 80–112.

Guinard, D., Trifa, V., Karnouskos, S., Spiess, P. and Savio, D. 2010a. Interacting with the SOA-based Internet of Things: Discovery, query, selection, and on-demand provisioning of Web Services. *IEEE Transactions on Services Computing 3,* 3, 223–235.

Guinard, D., Trifa, V. and Wilde, E. 2010b. A resource oriented architecture for the web of things. In *Proc. Internet of Things (IOT)*. IEEE, Los Alamitos, CA, USA, 1–8.

Hachem, S., Pathak, A. and Issarny, V. 2014. Service-oriented middleware for large-scale mobile participatory sensing. *Pervasive and Mobile Computing 10*, 66–82. Selected Papers from the Eleventh Annual IEEE International Conference on Pervasive Computing and Communications (PerCom 2013).

Heuer, J., Hund, J. and Pfaff, O. 2015. Toward the web of things: Applying web technologies to the physical world. *Computer 48,* 5, 34–42.

Hu, B., Wang, Z. and Dong, Q. 2012. A modeling and reasoning approach using description logic for context-aware pervasive computing. In *Proc. of Emerging Research in Artificial Intelligence and Computational Intelligence: International Conference, AICI 2012*, Chengdu, China, October 26–28, 2012, J. Lei, F. L. Wang, H. Deng and D. Miao, Eds. Communications in Computer and Information Science, vol. 315. Springer, Berlin, Heidelberg, 155–165.

J2EE. 2017. Home Page. URL: http://www.oracle.com/technetwork/java/javaee/. Accessed 10 May 2018.

Java Persistence API. 2017. Home Page. URL: http://docs.oracle.com/javaee/6/tutorial/doc/bnbpz.html. Accessed 10 May 2018.

Jersey. 2017. Home Page. URL: http://jersey.java.net. Accessed 10 May 2018.

jose.4.j. 2017. Home Page. URL: http://bitbucket.org/b_c/jose4j/. Accessed 10 May 2018.

JSON. 2017. Home Page. URL: http://www.json.org. Accessed 10 May 2018.

Karnouskos, S., Colombo, A. W., Bangemann, T., Manninen, K., Camp, R., Tilly, M., Stluka, P., Jammes, F., Delsing, J. and Eliasson, J. 2012. A SOA-based architecture for empowering future collaborative cloud-based industrial automation. In *Proc. of 38th Annual Conference on IEEE Industrial Electronics Society (IECON 2012)*. IEEE, Los Alamitos, CA, USA, 5766–5772.

Loke, S. W. 2004. Representing and reasoning with situations for context-aware pervasive computing: a logic programming perspective. *The Knowledge Engineering Review 19,* 3, 213–233.

LPaaS. 2018. Home page. URL: http://lpaas.apice.unibo.it. Accessed 10 May 2018.

Mariani, S. and Omicini, A. 2015. Coordinating activities and change: An event-driven architecture for situated MAS. *Engineering Applications of Artificial Intelligence 41*, 298–309.

Martelli, M. 1995. Constraint logic programming: Theory and applications. In *1985–1995: Ten years of Logic Programming in Italy*, M. Sessa, Ed. Palladio Editrice, Salerno, Italy, 137–166.

Messina, F., Mikkilineni, R. and Morana, G. 2017. Middleware, framework and novel computing models for grid and cloud service orchestration. *International Journal of Grid and Utility Computing 8*, 71.

Michelson, B. M. 2006. Event-driven architecture overview: Event-driven SOA is just part of the EDA story. Report, Patricia Seybold Group. Feb.

Monteiro, L. 1984. A proposal for distributed programming in logic. In *Implementations of Prolog*, J. A. Campbell, Ed. Artificial Intelligence. Ellis Horwood Limited, Chicester, UK, 329–340.

MQTT. 2017. Home Page. URL: http://mqtt.org. Accessed 10 May 2018.

Naish, L. 1988. Parallelizing NU-Prolog. In *Proc. of the 5th International Conference and Symposium on Logic Programming*, Seattle, Washington, 15–19 Aug. 1988, R. A. Kowalski and K. A. Bowen, Eds. MIT Press, Cambridge, MA, USA, 1546–1564.

Nalepa, G. J. and Bobek, S. 2014. Rule-based solution for context-aware reasoning on mobile devices. *Computer Science and Information Systems 11,* 1, 171–193.

Niezen, G. 2013. Ontologies for interaction: Enabling serendipitous interoperability in smart environments. *Journal of Ambient Intelligence and Smart Environments 5,* 1, 135–137.

Nii, H. P. 1986. The blackboard model of problem solving and the evolution of blackboard architectures. *The AI Magazine 7,* 2, 38–106.

Oliya, M. and Pung, H. K. 2011. Towards incremental reasoning for context aware systems. In *Proc. of Advances in Computing and Communications: 1st International Conference, ACC 2011*, Kochi, India, July 22-24, 2011, Part I, A. Abraham, J. Lloret Mauri, J. F. Buford, J. Suzuki and S. M. Thampi, Eds. Communications in Computer and Information Science, vol. 190. Springer, Berlin, Heidelberg, 232–241.

Omicini, A. and Zambonelli, F. 1999. Coordination for Internet application development. *Autonomous Agents and Multi-Agent Systems 2,* 3, 251–269. Special Issue: Coordination Mechanisms for Web Agents.

Palù, A. D. and Torroni, P. 2010. 25 years of applications of logic programming in Italy. In *A 25-Year Perspective on Logic Programming*, A. Dovier and E. Pontelli, Eds. Springer, Berlin, Heidelberg, 300–328.

Parker, L. E. 2008. Distributed intelligence: Overview of the field and its application in multi-robot systems. *Journal of Physical Agents 2,* 1, 5–14.

Payara. 2017. Home Page. URL: http://www.payara.fish. Accessed 10 May 2018.

Pontelli, E., Cao Son, T. and Baral, C. 2008. A logic programming based framework for intelligent Web Service composition. In *Managing Web Service Quality: Measuring Outcomes and Effectiveness*. IGI Global, Hershey, PA, USA, 193–221.

Prado, A. G. D., Ortiz, G. and Boubeta-Puig, J. 2017. CARED-SOA: A context-aware event-driven service-oriented architecture. *IEEE Access 5*, 4646–4663.

Ranganathan, A., Al-Muhtadi, J. and Campbell, R. H. 2004. Reasoning about uncertain contexts in pervasive computing environments. *IEEE Pervasive Computing 3,* 2, 62–70.

Ranganathan, A. and Campbell, R. H. 2003. An infrastructure for context-awareness based on first order logic. *Personal and Ubiquitous Computing 7,* 6, 353–364.

Reiter, R. 1978. On closed world data bases. *Logic and Data Bases*. Springer, Boston, MA, USA, 55–76.

Ricci, A., Viroli, M., Omicini, A., Mariani, S., Croatti, A. and Pianini, D. 2017. Spatial Tuples: Augmenting physical reality with tuple spaces. In *Intelligent Distributed Computing X. Proc. of the 10th International Symposium on Intelligent Distributed Computing – IDC 2016*, Paris, France, October 10–12 2016, C. Badica, A. El Fallah Seghrouchni, A. Beynier, D. Camacho, C. Herpson, K. Hindriks and P. Novais, Eds. Studies in Computational Intelligence, vol. 678. Springer, Berlin, Heidelberg, 121–130.

Richards, M. 2016. *Microservices AntiPatterns and Pitfalls*. O'Reilly, Sebastopol, CA, USA.

Robinson, J. A. 1965. A machine-oriented logic based on the resolution principle. *Journal of the ACM 12,* 1, 23–41.

Salber, D., Dey, A. K. and Abowd, G. D. 1999. The context toolkit: Aiding the development of context-enabled applications. In *Proc. of SIGCHI Conference on Human Factors in Computing Systems (CHI '99)*. ACM, New York, NY, USA, 434–441.

Schulte, R. W. and Natis, Y. V. 2003. Event-driven architecture complements SOA. Research note, Gartner. 8 July.

Shapiro, E. Y. 1987. *Concurrent Prolog – Vol. 1: Collected Papers*. Logic Programming. The MIT Press, Cambridge, MA, USA.

Smart, P. 2017. Situating machine intelligence within the cognitive ecology of the Internet. *Minds and Machines 27,* 2, 357–380.

Ueda, K. 1986. Guarded Horn clauses. In *Proc. of the 4th Conference on Logic Programming '85*, Tokyo, Japan, 1–3 July 1985, E. Wada, Ed. Lecture Notes in Computer Science, vol. 221. Springer, Berlin, Heidelberg, 168–179.

Wang, H., Mehta, R., Supakkul, S. and Chung, L. 2011. Rule-based context-aware adaptation using a goal-oriented ontology. In *Proc. of 2011 International Workshop on Situation Activity & Goal Awareness (SAGAware '11)*. ACM, New York, NY, USA, 67–76.

Wolfram, D. A., Maher, M. J. and Lassez, J.-L. 1984. A unified treatment of resolution strategies for logic programs. In *Proc. of 2nd International Conference on Logic Programming (ICLP 1984)*, S.-Å. Tärnlund, Ed. Association for Logic Programming, Uppsala, Sweden, 263–276.

Zambonelli, F., Omicini, A., Anzengruber, B., Castelli, G., DeAngelis, F. L., Di Marzo Serugendo, G., Dobson, S., Fernandez-Marquez, J. L., Ferscha, A., Mamei, M., Mariani, S., Molesini, A., Montagna, S., Nieminen, J., Pianini, D., Risoldi, M., Rosi, A., Stevenson, G., Viroli, M. and Ye, J. 2015. Developing pervasive multi-agent systems with nature-inspired coordination. *Pervasive and Mobile Computing 17*, 236–252. Special Issue "10 years of Pervasive Computing" In Honor of Chatschik Bisdikian.