Logic Programming in Space-Time: The Case of Situatedness in LPaaS

Roberta Calegari*, Giovanni Ciatto*, Stefano Mariani†, Enrico Denti*, Andrea Omicini*
*Department of Computer Science and Engineering (DISI)
ALMA MATER STUDIORUM–Università di Bologna, Italy
Email: roberta.calegari@unibo.it, giovanni.ciatto@unibo.it, enrico.denti@unibo.it, andrea.omicini@unibo.it
†Department of Sciences and Methods for Engineering (DISMI)
Università degli Studi di Modena e Reggio Emilia, Italy
Email: stefano.mariani@unimore.it

***Abstract*—Situatedness is a fundamental requirement for today's complex software systems, as well as for the computational models and programming languages used to build them. Spatial and temporal situatedness, in particular, are essential features for AI, enabling actors of the system to take autonomous decisions contextual to the space-time they live in. To support spatio-temporal awareness in distributed pervasive systems, we adopt the standpoint of Logic Programming (LP) by focussing on the Logic Programming as a Service (LPaaS) approach, promoting the distribution of *situated intelligence*. Accordingly, we provide an interpretation about what it means to make LP span across space and time, then we extend the LPaaS model and architecture towards spatio-temporal situatedness, by identifying a set of suitably-expressive spatio-temporal primitives.**

***Index Terms*—LPaaS, Situatedness, Logic Programming, SOA, space-time programming**

I. INTRODUCTION

The widespread diffusion of mobile and wearable technologies, along with the pervasiveness of the Internet, opens the way towards a wide range of distributed, location- and context-aware applications, where both the system's goals and its computational behaviour depend on the users' position in the space [1], [2], [3], [4], and on time passing by.

There, one of the main technical challenges is to design and develop context-aware services that anticipate users' situation, proactively serve their information needs, and personalise suggestions, starting from the agreement that most cognitive processes are *contextual* in that they depend on the environment inside which they are carried on [5], [6]. Contextual information such as location, time, activity, physical conditions, and social interaction, in fact, is acknowledged to generate trustworthy and accurate reasoning and recommendations [7].

Logic Programming (LP) can play a key role by enabling the widespread diffusion of *distributed intelligence*. Nevertheless, although LP languages and technologies represent a natural candidate for injecting intelligence within computational systems [8], and despite the many practical applications developed over the years – see [9], [10] for a survey –, the adoption of LP in pervasive contexts has been historically hindered by technological obstacles like efficiency and integration issues, as well as by some cultural resistance towards LP-based approaches outside the academy.

However, technology advancements and the emergence of the Internet of Intelligent Things context [11], [12] are drastically changing such a scenario, eventually enabling LP to unveil its full potential in real-world applications. Along this line, in this paper we extend the *Logic Programming as a Service* (LPaaS) approach, intended as the natural evolution of distributed LP in pervasive systems, towards spatio-temporal awareness, to promote the distribution of *situated intelligence*.

Accordingly, in Section II we introduce LPaaS focussing on the most relevant features for context-aware domains, then (Section III) we discuss spatio-temporal issues for LP and LPaaS, and (Section IV) we identify a suitably-expressive set of spatio-temporal primitives. A case study is presented in Section V, and conclusions are drawn in Section VI.

II. BACKGROUND: LPAAS KEY FEATURES

LPaaS [13], [14] is the evolution of LP in parallel, concurrent, distributed scenarios according to the *Service-Oriented Architecture* (SOA) paradigm [15], concretising the *micro-intelligence* vision in terms of *micro-services*.

This perspective emphasises the role of *situatedness*, already brought along by distribution in itself. The resolution process [16] remains a staple in LPaaS, yet it is re-contextualised according to the situated nature of the specific LP service.

Thus, given the precise spatial, temporal, and general context within which the service is operating when the resolution process starts, the process follows the usual rules of resolution. At the same time, this context is exactly what can make resolution come up with different solutions to the same queries: indeed, the situatedness of the resolution process (Section IV) is the first novelty of the LPaaS approach.

The second novelty concerns *interaction* with the clients of the LP service. In classical LP, interactions are necessarily *stateful*: users set the logic theory, define the goal, then ask for solutions, iteratively. This implies that the LP engine is expected *i)* to store the logic theory, *ii)* to memorise the goal under demonstration, and *iii)* to track how many solutions have been already provided: altogether, this information composes the *state* of the LP engine.

In LPaaS, instead, interactions are mainly *stateless*: coherently with the SOA approach, the LP service instance that

serves each request may be different each time, e.g. due to the redundancy of the distributed software components to improve availability and reliability of the LP service. Each client query (interaction) should then be self-contained, so that it does not matter which specific service instance actually responds.

Another key feature is the possibility of choosing between *dynamic* or *static* knowledge bases (KBs): from the client viewpoint, a static KB is immutable, while a dynamic KB can evolve during the service lifetime due to assertion/retraction of clauses. This implies that clauses in a dynamic KB have a lifetime, representing mutable knowledge about the world.

The last key novelty regards the *software engineering* process: since LPaaS embraces the SOA perspective and its most recent evolutions, such as the micro-services paradigm and the Web of Things perspective, LPaaS is both built as a micro-service and exploits micro-services for its inner architecture. LPaaS functionalities are made available as a standardised RESTful web service, exploiting technologies such as JSON, JWT, and Docker for the sake of interoperability, and a fully automated continuous integration/delivery pipeline for an agile development and deployment (details in [17]).

III. LP in Space-Time: the LPaaS Perspective

A. LP in Space-Time

Making LP aware of space-time essentially means making the resolution process sensitive *i)* to the passing of time, and *ii)* to the topology of space—that is, aware of the existence of different points in time/space and of the possibility of moving between them. Fig. 1 illustrates this idea by representing LP approaches in a bidimensional time/space chart.

In classical LP, a goal is basically proven against a logic theory that is considered true independently of space and time—that is, regardless of when and where the resolution process takes place. LPaaS promotes a broader vision: the resolution process becomes sensitive to the time and space dimension. Hence, a goal is demonstrated against a logic theory that is considered true within a (possibly open) *interval of time*, and within a *region of space*.

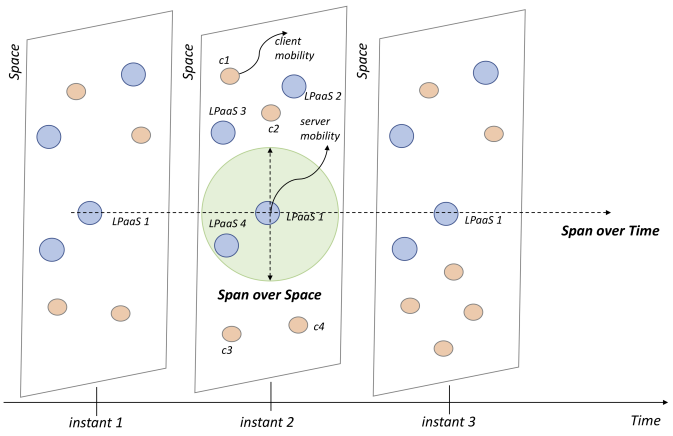

Fig. 1. LPaaS in the space-time.

In principle, the resolution process could also *span* and *move* over time and space. Leaving the full discussion to Subsection III-B and III-C, intuitively, moving back in time means to ask the LPaaS service the solution of a given goal at a certain moment in the past, whereas moving forward in time either means to ask to "predict" the solutions that could become available in the future, or to wait until they become provable. Analogously, moving in space in either direction means to consider a different region of validity associated to the logic theory, providing solutions that are situated (hold true) in a given region of space—and only provable therein.

B. The Temporal Dimension

Being situated in time means that the *temporal context* when the LP service is executed may affect its properties, computations, and interactions with clients. Since reconstructing a *global* notion of time in pervasive systems is either unfeasible or non-trivial, each LP service should be supposed to operate on its own local time—that is, computing deadlines, leasing times, and the like according to its *local perception* of time. Also, it is worth emphasising that the notion of time assumed in LPaaS is not bound to a specific definition of time, but is related to each step of the computation in the resolution process—which may be unrelated to physical time.

Time can affects both computations, leading to *time-dependant computations*, and the *temporal validity* of logic theories, thus of all the individual facts and clauses therein, whose validity can be collectively bound to a given time horizon—i.e., being true up to a certain instant in time, and no longer since then. Computation requests may then arbitrarily restrict the *temporal scope* of the expected solutions, specifying the temporal bounds of the logical facts and clauses to consider as valid while proving a goal.

Subsection IV-A discusses the specific LPaaS API dedicated to managing temporal aspects, as reported by TABLE I.

C. The Spatial Dimension

Being 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. Intentionally, we do not focus on the structure of space, such as whether there is a global coordinate system shared by all LPaaS instances, or each LPaaS server has its own local one, and which metrics of distance to adopt, etc.: it only matters that the LPaaS service has a notion of locality which situates it in space. The region of validity is then to be considered *local* to the service, as represented by the server container. For instance, in the following we consider the notion of *neighbourhood* – intended as the collection of the LPaaS services sufficiently close to each other – without specifying how such a notion is technically computed and maintained.

Analogously to Subsection III-B, the notion of space and its properties, such as topology, may be exploited to design *space-dependant computations*, and affect LPaaS properties such as logic theories, individual facts, and clauses—whose validity can be collectively bound to a given region in space. Requests

may then arbitrarily restrict the *spatial scope* of the expected solutions—that is, explicitly specify the spatial bounds to be considered as valid while proving a given goal.

In this perspective, LPaaS is capable of interacting with its surroundings to contact the other LPaaS services in the neighbourhood, that can represent the same knowledge but with a different local truth—depending on their location.

Subsection IV-B discusses the specific LPaaS API dedicated to managing these aspects, as reported by TABLE I.

IV. Spatio-Temporal Situatedness in LPaaS

LPaaS provides an application-level API to exploit the built-in situatedness of both the service and clients as discussed in Section III, enabled and supported by the service container.

A. The Temporal Dimension

Of all the issues described in Subsection III-B, some affect the resolution process, other pertain only to the way clients interact with the service.

Client-Server Interaction: Temporal situatedness is basically implied by distribution per se: moving information in a network takes time, thus aspects such as expiration of requests, obsolescence of logic theories, and timeliness of replies need to be taken into account when designing a distributed LP service. To this end, LPaaS introduces a family of `solve` predicates with a specific `within(`*`Time`*`)` argument (intended as server-side local time) for the resolution, thus preventing the server from being indefinitely busy: if the resolution process does not complete within the given time, the request is cancelled, and a failure response is returned to the client.

The client could also ask for a *stream* of solutions, as it is likely the case in IoT scenarios when exploiting sensors or monitoring processes: this is why `solve` can also take an `every(`*`Time`*`)` argument, meaning that each new solution should be returned not faster than every *`Time`* milliseconds.

Note that, w.r.t. Fig. 1, here we are not "moving the dot" along the time dimension: rather, we are simply letting clients configure their interactions with the LPaaS service along the temporal dimension—namely, *when* they want a solution.

Resolution process: To capture the time-bounded validity of theories, each axiom in the LPaaS knowledge base is decorated with a time interval. When a new resolution process starts, the logic theory used for the resolution includes only the axioms valid at the `Timestamp` provided in the client query.

This timestamp can span both in the *past* and in the *future*. The first means to ask for solutions that were valid in the past, while the second opens intriguing possibilities about "reasoning on the future". An appealing perspective could be to postpone the resolution process until future information becomes available, or to ask for some sort of prediction to some intelligent "oracle". This may imply the ability to track the history of modifications to the logic theory, in order to guess its future state. This fascinating perspective would make the LPaaS service ultimately behave as a *prediction model* trained on the evolution of the logic theory.

Here, w.r.t. Fig. 1, we actually are moving the dot along the time dimension, affecting the resolution process, thus its outcomes. Furthermore, the query to the server could specify not a timestamp, that is an instant in time, but rather a *temporal interval*. With respect to Fig. 1, each instant in the interval would be considered by the resolution process, adding the corresponding solutions to the solution list. This possibility is left as a future work and not included in the current version of LPaaS API—TABLE I.

B. The Spatial Dimension

Yet again, some of the issues described in Subsection III-C affect the resolution process, while some other pertain only to the way clients interact with the service.

Client-Server Interaction: Since the service container is physically located somewhere, the LPaaS server inherently has a notion of its own location: so, specifying its *surroundings* amounts basically at specifying the "width" of such a region, according to some (custom) metric. Alternatively, we might be interested in specifying a region from another, different position—i.e., the client's own one.

The key point is that LPaaS can explore its surroundings to discover other LPaaS instances, representing different local knowledge, and forward to them the query looking for expanding the solutions it found by itself. The primitive `solveNeighborhood/2`:

```
solveNeighborhood(-SList, region(?Pos, +Distance))
```

associates the inference process to a region centred in *`Pos`* – if omitted, the server position is considered – and distance specified by *`Distance`*; the resulting list of solutions is returned in *`SList`*.

Intentionally, the precise syntax of the region terms (*`Pos`* and *`Distance`*) is left unspecified, so that it can be general enough to cover the widest range of possible interpretations: *`Distance`*, in particular, could be a physical distance in meters, leading to a circular region, or the maximum number of hops in a network, leading to a region of virtually any shape, etc. Indeed, the specific language to be used for the space description is an open research aspect, hence outside the scope of this paper.

Similarly to the time case, the client could also open a *stream of solutions across space*: this could be interpreted, for instance, as widening/narrowing the region to be considered at each cycle of the resolution process (Fig. 2). Syntax

```
solveNeighborhood(-SList,region(?Pos,+Distance,+Span))
```

fits the purpose: the extra *`Span`* argument specifies how much to widen/narrow the considered range at each cycle. As above, the precise syntax of *`Span`* is left intentionally unspecified. Last but not least, in dynamic KBs time and space could intervene together, leading to a family of `solveNeighborhood` primitives (TABLE I).

Along this line, fascinating possibilities come from considering *moving regions* as streams of solutions bound to spatial constraints, possibly evolving over time. This implies that time and space are considered together, as inseparable

dimensions—in fact, moving in space takes time, and the notion of stream implies some notion of time. (This is not part of the current LPaaS API in TABLE I.)

Resolution process: Distribution *per se* constitutes a premise to spatial situatedness: each LP instance runs on a different device, on a different network host, accessing the different computational and network resources that are *locally* available. Moreover, since LP services encapsulate their logic theory, the locally-gathered knowledge affects the result.

C. Infrastructural Aspects: Mobility

Besides the kind of mobility shown in Fig. 1, where the LP engine conceptually moves in time and space, another kind of mobility stems from use cases in mobile computing and pervasive systems.

Generally speaking, mobile computing deals with systems in which either software or devices need to move to accomplish their tasks, whereas pervasive computing deals with application scenarios in which computing devices are densely scattered in the physical world—and, possibly, moving. Starting from this consideration, LPaaS aims at dealing with both physical (mobile hosts) and logical (code migration) mobility, so as to support distributed location-aware computation.

As a first step towards a full-fledged perception of the time/space dimensions, *mobility* captures the idea that both the LPaaS server and clients could move over the network: time and space are perceived to maintain the client/server connections, i.e. somehow to "reroute" the server solutions to the clients despite their movement, but have no impact on the resolution process itself—that is, on the provided solutions.

Accordingly, mobile clients may request LP services from different locations at each request, possibly while moving, which means that the LP service should be able to identify and track clients to reply to the correct network address. The issue must be resolved at the infrastructural level, via some middleware layer that possibly supports some notion of communication session enabling users to consistently interact with the LPaaS service regardless of their mobility—there including possible disconnections, reconnections, or IP changes.

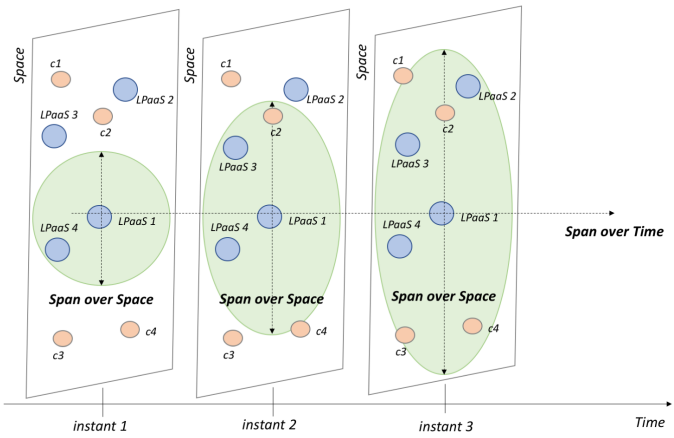

Fig. 2. The `solveNeighborhood` primitive spanning over time and space.

V. Case Study

As a scenario to conceptually validate the LPaaS approach to spatial awareness here proposed, let us consider the case of a user looking for a parking spot in proximity of a metro station in London (Fig. 3).

The standard, non-spatial query (Fig. 3, *(a)*) would be just to ask the nearest LPaaS metro service if there is a free parking slot at the region covered by the logic theory of that station. In the case of a negative answer, one could just proceed to another metro service and re-ask the same query there.

Spatial queries, instead, make it possible to expand the query to a region of space. Since the domain refers to metro stations, it seems reasonable to assume as "distance" the number of stations between the LPaaS server station and the parking station, possibly restricted to a single metro line (i.e. black line only). For instance, `distance(1,_)` includes the stations adjacent to the server on any line, while `distance(1,black)` includes the adjacent stations on the black line only.

In Fig. 3, *(b)* we assume that the user is close to the *Elephant and Castle* station. In the spatial solve:

```
solveAllNeighborhood ((station_with_parking(X),
    n_free_place(X,N)), S, region(_, distance(1,_)).
```

goal

```
station_with_parking(X), n_free_places(X,N)
```

looks for possible parkings in a region defined as at most one station away (`region(_, distance(1,_))`), retrieving the number `N` of free places. The service replies with a parking near the `X='London Bridge'` station, with `N=5` free places, reachable in one step on the `black` line.

If the distance is restricted to a given line, i.e. brown as in Fig. 3 *(c)*, the answer is different—in this case, no solution.

Conversely, if the region is expanded to two stations of distance (Fig. 3 *(d)*), three parking spots will be found:

```
X/'London Bridge', N/5, distance(1,black)
X/'Canada Water', N/2, distance(2,'black - grey')
X/'Charing Cross', N/0, distance(2,brown)
```

the latter, unfortunately, with no free places.

More complex scenarios could be envisioned—for instance, a user could ask the service the number of expected free places, possibly based on some prediction algorithm or analysing the history of the knowledge in each parking.

VI. Conclusion

Pervasive and situated systems of any sort are increasingly demanding intelligence to be scattered throughout the computational devices populating the physical environment—as clearly demonstrated by IoT scenarios like smart homes, personal healthcare assistants, energy grids, etc.

The LPaaS approach aims at fitting such a challenging context by introducing a standard interface that is general enough to account for both stateful and stateless services, with both static and dynamic knowledge bases, in a configurable and customisable way. The distinguishing feature of LPaaS is to support the spatio-temporal awareness, thus allowing for the distribution of *situated intelligence* where and when needed.

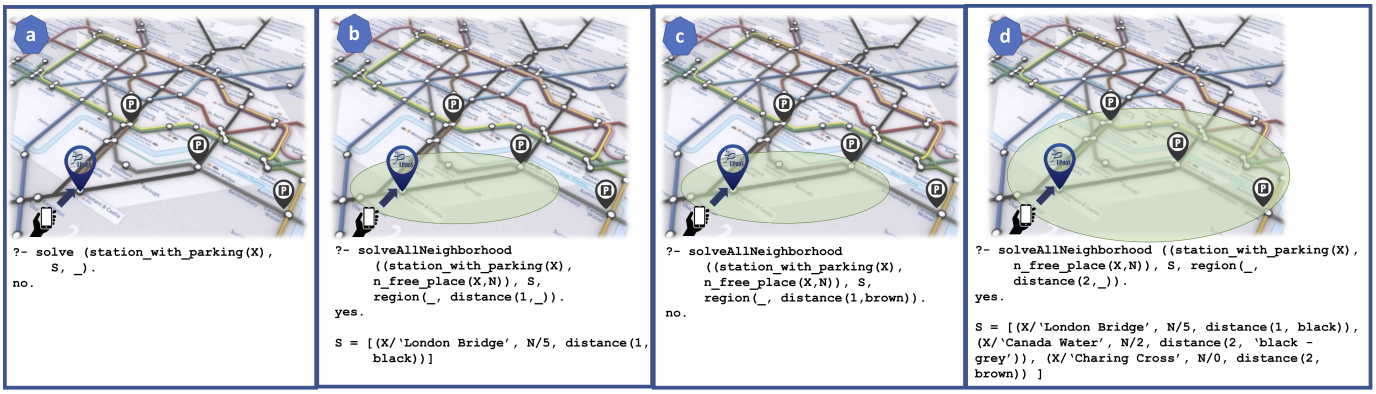

Fig. 3. Examples of LPaaS queries exploiting spatial situatedness.

The paper provides an interpretation of what it means to let LP span across space and time, and discusses the LPaaS model and architecture extensions towards spatio-temporal situatedness, by identifying a suitably-expressive set of spatio-temporal primitives. A real application scenario is discussed, highlighting the potential of such an approach, and showing how taking the space around either the client or the server into account opens the chance to opportunistically federate LP engines by need as a form of dynamic service composition.

Of course, this is not the end of the story: many improvements can be devised in several directions, starting from building a specialised logic-oriented middleware, dealing with heterogeneous platforms, as well as with distribution, life-cycle, interoperation, and coordination of multiple, situated Prolog engines is a goal for our future research, aimed at exploring the full potential of logic-based technologies in the context of IoT scenarios and applications.

References

[1] J. Beal, D. Pianini, and M. Viroli, "Aggregate programming for the Internet of Things," *IEEE Computer*, vol. 48, no. 9, pp. 22–30, Sep. 2015. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=7274429

[2] M. Hazas, J. Scott, and J. Krumm, "Location-aware computing comes of age," *Computer*, vol. 37, no. 2, pp. 95–97, Feb. 2004. [Online]. Available: http://ieeexplore.ieee.org/document/1266301/

[3] T. E. Starner, "Wearable computers: no longer science fiction," *IEEE Pervasive Computing*, vol. 1, no. 1, pp. 86–88, Jan. 2002. [Online]. Available: http://ieeexplore.ieee.org/document/993148/

[4] R. Scoble and S. Israel, *The Age of Context*. Patrick Brewster Press, Apr. 2014.

[5] F. Giunchiglia, "Contextual reasoning," *Epistemologia, special issue on I Linguaggi e le Macchine*, vol. 16, pp. 345–364, 1993.

[6] I. Uddin, A. Rakib, H. M. U. Haque, and P. C. Vinh, "Modeling and reasoning about preference-based context-aware agents over heterogeneous knowledge sources," *Mobile Networks and Applications*, vol. 23, no. 1, pp. 13–26, 2018.

[7] N. Y. Asabere, "Towards a viewpoint of context-aware recommender systems (cars) and services," *International Journal of Computer Science and Telecommunications*, vol. 4, no. 1, pp. 10–29, 2013.

[8] J. Brownlee, *Clever algorithms: nature-inspired programming recipes*. Jason Brownlee, 2011.

[9] A. D. Palù and P. Torroni, "25 years of applications of logic programming in Italy," in *A 25-year Perspective on Logic Programming*, A. Dovier and E. Pontelli, Eds. Springer, 2010, pp. 300–328. [Online]. Available: http://link.springer.com/10.1007/978-3-642-14309-0_14

[10] M. Martelli, "Constraint logic programming: Theory and applications," in *1985-1995: Ten years of Logic Programming in Italy*, M. Sessa, Ed., 1995, pp. 137–166.

[11] Y. Leng and L. Zhao, "Novel design of intelligent internet-of-vehicles management system based on cloud-computing and internet-of-things," in *Electronic and Mechanical Engineering and Information Technology (EMEIT), 2011 International Conference on*, vol. 6. IEEE, 2011, pp. 3190–3193.

[12] Y. Chen and H. Hu, "Internet of intelligent things and robot as a service," *Simulation Modelling Practice and Theory*, vol. 34, pp. 159–171, 2013.

[13] R. Calegari, E. Denti, S. Mariani, and A. Omicini, "Logic programming as a service in multi-agent systems for the Internet of Things," *International Journal of Grid and Utility Computing*, In press.

[14] ——, "Logic Programming as a Service (LPaaS): Intelligence for the IoT," in *2017 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, May 2017, pp. 72–77. [Online]. Available: http://ieeexplore.ieee.org/document/8000070/

[15] T. Erl, *Service-Oriented Architecture: Concepts, Technology, and Design*. Upper Saddle River, NJ, USA: Prentice Hall / Pearson Education International, 2005. [Online]. Available: http://www.pearson.com/us/higher-education/program/Erl-Service-Oriented-Architecture-Concepts-Technology-and-Design/PGM137253.html

[16] J. A. Robinson, "A machine-oriented logic based on the resolution principle," *Journal of the ACM*, vol. 12, no. 1, pp. 23–41, Jan. 1965. [Online]. Available: http://dl.acm.org/citation.cfm?id=321253

[17] R. Calegari, G. Ciatto, S. Mariani, E. Denti, and A. Omicini, "Micro-intelligence for the IoT: SE challenges and practice in LPaaS," in *2018 IEEE International Conference on Cloud Engineering (IC2E 208)*. IEEE Computer Society, 17–20 Apr. 2018, pp. 292–297.

TABLE I
LPaaS Client Interface.

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()`
`solveNeighborhood(+Goal, -Solution, region(?P, +Space), ?TimeS)`		`solveNeighborhood(-Solution, region(?P, +Space), ?TimeS)`
`solveNNeighborhood(+Goal, +N, -SList, region(?P, +Space), ?TimeS)`		`solveNNeighborhood(+N, -SList, region(?P, +Space), ?TimeS)`
`solveAllNeighborhood(+Goal, -SList, region(?P, +Space), ?TimeS)`		`solveAllNeighborhood(-SList, region(?P, +Space), ?TimeS)`
`solveNeighborhood(+Goal, -Solution, region(?P, +Space, +Span), ?TimeS)`		`solveNeighborhood(-Solution, region(?P, +Space, +Span), ?TimeS)`
`solveNNeighborhood(+Goal, +N, -SList, region(?P, +Space, +Span), ?TimeS)`		`solveNNeighborhood(+N, -SList, region(?P, +Space, +Span), ?TimeS)`
`solveAllNeighborhood(+Goal, -SList, region(?P, +Space, +Span), ?TimeS)`		`solveAllNeighborhood(-SList, region(?P, +Space, +Span), ?TimeS)`
	`reset()`	
	`close()`	

The table reports only methods operating on a dynamic KB that take an additional `Timestamp` argument, expressing the required time validity; static KB methods are analogous without the `Timestamp` parameter.