Towards Logic Programming as a Service: Experiments in tuProlog

Roberta Calegari, Enrico Denti, Stefano Mariani, Andrea Omicini
Dipartimento di Informatica, Scienza e Ingegneria (DISI)
ALMA MATER STUDIORUM–Università di Bologna
Email: {roberta.calegari, enrico.denti, s.mariani, andrea.omicini}@unibo.it

***Abstract*—In this paper we explore the perspective of *Logic Programming as a Service* (LPaaS), with a broad notion of "service" going beyond the mere handling of the logic engine lifecycle, knowledge base management, reasoning queries execution, etc. In particular, we present tuProlog *as-a-service*, a Prolog engine based on the tuProlog core made available as an encapsulated service to effectively support the spreading of intelligence in pervasive systems—mainly, Internet-of-Things (IoT) application scenarios.**

So, after recalling the main features of tuProlog technology, we discuss the design and implementation of tuProlog as-a-service, focussing in particular on the iOS platform because of the many supported smart devices (phones, watches, etc.), the URL-based communication support among apps, and the multi-language resulting scenarios.

I. INTRODUCTION

Today applications are more and more *pervasive* and *intelligent*, calling for *situated intelligence*—light-weight, effective intelligence chunks placed where and when needed to locally tackle the specific reasoning needs in complex distributed systems. This is particularly true in the fast-growing field of the Internet-of-Things (IoT), where connectivity and interoperability are just the basic steps towards higher-level, customised, variously-situated services.

The complexity of IoT system engineering calls for suitable, easily deployable infrastructures, meant to make the designers' and developers' task easier by providing commonly-required services to applications. Such infrastructures should *(i)* be both statically and dynamically (easily) configurable, so as to match the application needs; *(ii)* govern components and applications interaction; *(iii)* encapsulate intelligence in suitable forms for applications' exploitation.

In this scenario, where software engineering, programming languages, and distributed artificial intelligence meet, logic-based languages have the potential to play a prominent role both as *intelligence providers* and *technology integrators*.

For the former, typical LP features – such as programs as logic theories, computation as deduction, and programming with relations and inference – make logic languages a natural choice for building intelligent components. In the context of IoT, this also implies that logic-based technologies should be implemented taking strongly into account specific engineering criteria such as deployability, scalability, and interoperability.

For the latter, logic languages already proved to be effective as both communication and coordination languages [1], whereas, more generally, declarative models and technologies are known to impact on the modelling of complex, heterogeneous systems as multi-agent systems (MAS) [2]—also suggesting that their actual vocation should rightfully include supporting scalable, configurable, intelligent infrastructures for Internet-based applications.

The tuProlog [3] engine, deployed as a Java JAR or Microsoft .NET DLL, is inherently easily deployable and exploitable by other applications as a *library service*—that is, from a software engineering standpoint, a suitably encapsulated set of related functionalities. However, this might not be enough in complex IoT scenarios, where the mobility/cloud ecosystem aims at delivering infrastructure, platforms, and software *as a service* – according to a more Service Oriented Architecture (SOA) interpretation of the term "service" –, enabling people to benefit from ubiquitous information access. Emphasis is more and more on *on-demand* applications, where the enabling infrastructure – servers, storage, networks, and client devices – moves towards cloud computing. Service-oriented computing also promotes the idea of assembling application components into a network of services that can be loosely coupled to create flexible, dynamic business processes and agile applications spanning organisations and computing platforms.

In the remainder of the paper, after recalling the main features of the tuProlog technology, we outline the general architecture for *Logic Programming as a Service* (LPaaS) – without limiting ourselves to the case where "as a service" means "as a Web/Cloud Service" – aimed at fully managing the Prolog engine lifecycle (either dedicated or not to a given client application), the knowledge base management, and the query execution. Then, we focus on the tuProlog case, taking iOS as our experimental platform – where the actual instantiation of the as-a-service paradigm is much closer to a SOA-/Cloud-oriented interpretation – and discuss some simple application examples.

II. tuPROLOG IN A NUTSHELL

tuProlog [3] is an open-source, light-weight Prolog framework for distributed applications and infrastructures, released under the LGPL license [4]. Unlike most Prolog programming environments – such as [5], [6], which are typically very efficient but also monolithic and thought to operate as stand-alone systems –, tuProlog is intentionally designed to be *minimal*, *dynamically configurable*, *easily deployable*, *interoperable*, but above all *multi-paradigm* – promoting seamless integration of the logic/declarative paradigm, on the one side, with the object-oriented imperative paradigm, on the other

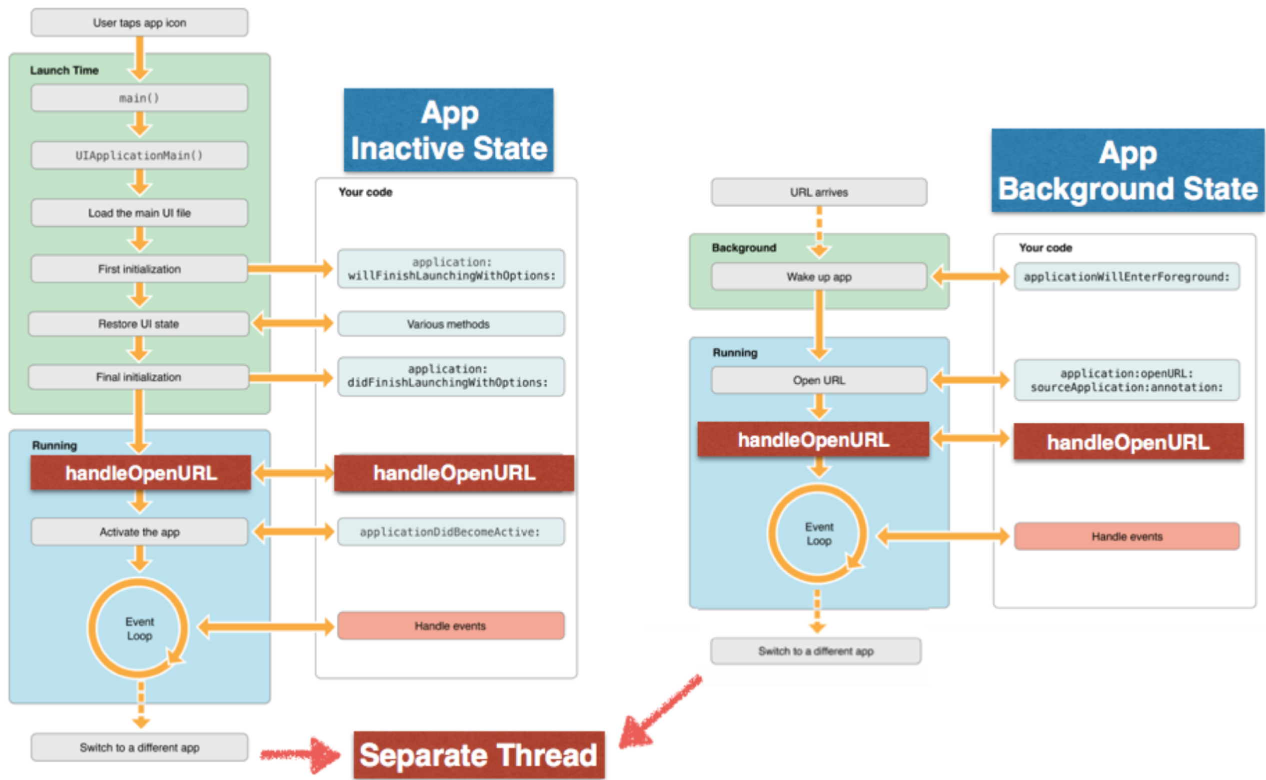

Fig. 1. Overview of the 2PaaS architecture

– and *multi-language*—enabling Prolog to inter-operate with Java (and viceversa) easily and directly, with no need for special pre-declarations, static stub files, etc. [7]. Over the years, tuProlog has grown to become also *multi-platform*, from Microsoft .NET to Android and Apple iOS [8]. So, besides Java, the object-oriented side now covers potentially any language on the .NET platform; other languages, like the newborn Apple Swift [9], are also being included.

Minimality means that the tuProlog core contains only the Prolog engine essentials – roughly speaking, the reasoning engine and the required basic mechanisms – whereas any other feature is implemented via *libraries*: users can configure their system by dynamically loading/unloading libraries at any time.

Each library provides a package of related predicates, functors, and operators; new libraries can also be easily developed, either in Prolog or in Java/.NET language (or a hybrid mix of the two) as a suitable Java/.NET class, extending the tuProlog `Library` class and following simple naming conventions. Library-defined predicates, functors and operators are indistinguishable from tuProlog built-ins—actually, most of the perceived built-ins are actually provided by some pre-loaded library.

Easy deployability means that, generally speaking, installing is just a matter of making the JAR archive/.NET assembly available in the desired location: the only exception is when the hosting platform imposes its install procedure, as in the case of Android/iOS app (for which going through the Android/iOS installer is mandatory).

In addition, tuProlog is compatible with major interoperability standards such as TCP/IP, RMI, and CORBA, and is exploited as the enabling technology inside the TuCSoN coordination infrastructure [10], [11], which provides logic-based, programmable tuple spaces – *tuple centres* [12] – as coordination media for distributed processes and agents.

III. Logic Programming as a Service

Coherently with the view outlined above, this paper looks forward to the idea of providing *Logic Programming as a Service* (LPaaS), conceptually situated inside the Software as a Service layer of, e.g., a typical Cloud-based environment. Nevertheless, the first step to achieve such a goal is to carefully design the architecture of a logic programming engine according to the general software engineering principles of modularity, encapsulation, and reusability. The architecture would provide the conceptual foundation for the "as a service" paradigm in the context of heterogeneous computational environments—e.g., as a RESTful Web Service, as a Cloud-hosted app, etc.

In the LPaaS context, according to the broad acceptation of term "service" described in Section I, the basic functionality to be provided is the ability to reason over logic theories, which translates into the ability to submit queries and retrieve results; related functionalities include the creation and configuration of the reasoner (the logic engine) with the proper knowledge base.

The interface is then supposed to define the API to provide such functionalities, namely:

- *create engine*: to instantiate a dedicated engine;
- *reset engine*: to reset the engine to the initial state and possibly the empty knowledge base;

Fig. 2. The derivative application scenario: the symbolic derivative query (left) and result (right) on iPhone®and iWatch®.

- *set theory*: to load the knowledge base into the engine;
- *query*: to submit queries and retrieve (possibly multiple) results.

Further degrees of freedom include whether multiple engines could be allowed, whether a given engine should/should not be reserved to a given client application, etc.

If the Prolog language is adopted, such functionalities have to be tailored to the Prolog-specific behaviour, assumptions, and syntax, and APIs have to be mapped onto suitable Prolog predicates. In the next section we specialise our approach to the specific case of the tuProlog system, which provides not only a light-weight Prolog engine particularly featured for this kind of applications, but also a multi-paradigm and multi-language working environment, thus paving the way towards further forms of interaction and expressiveness.

Among the different platforms supported by tuProlog, in this paper we take the iOS platform as our reference *(a)* because of its adoption in pervasive contexts, *(b)* for the availability of several sorts of smart devices (phones, watches, etc.) that constitute a challenging testbed, and *(c)* for its architectural similarity with more specific SOA/Cloud-oriented notions of "as a service".

IV. tuProlog as a Service on iOS

As a challenging application context for LPaaS, here we focus on the development of the tuProlog service for iOS, for both Objective-C and Swift applications. The service is embedded in a tuProlog app, acting as the *service provider* for all other applications running on the mobile device. Its API obviously adheres to the above-presented API, tailored to the Prolog syntax plus some convenience mechanisms—namely:

- *create engine*: for convenience, this functionality is just embedded in the first call to the service;
- *reset engine*: mapped onto the `reset` primitive;
- *set theory*: mapped onto the `theory` primitive;
- *query*: mapped onto a set of primitives, namely `query` to issue the query, `result` and `solution` to retrieve the query result, and `nextSol` to explore further solutions.

A nice peculiarity of the iOS platform, which is one of the main reasons to choose it in our experiments, is the built-in *URL scheme functionality*, which provides a simple and effective way for applications' inter-communication via a user-defined, URL-based protocol. In short, all is needed to communicate with an app is to create an appropriately-formatted URL and ask the system to open it; on the opposite side, the custom scheme needs to be declared and properly implemented—see Fig. 1.

In the tuProlog case, two custom URL schemes need to be defined—one for incoming requests, to be processed by the tuProlog Mobile App and encoding the above API; and another for the query results, to be processed by the client app.

The tuProlog app implements the following URL scheme:

```
tuPrologMobile://?src=srcURL&
                      command=argument&
                      dst=dstURL
```

where:

- *`srcURL`* is the client URL scheme: as a result, a new Prolog engine is created in tuProlog Mobile and associated with that client;

- *command* is one of `theory`, `query`, `nextSol`, `reset`, with *argument* respectively being *theory* (the Prolog theory to be set in the engine, or the link where to donwload it[1]), *query* (the text of the query), `nextSol`, `reset`;
- *dstURL* is the URL scheme of the client to which the result must be sent, possibly different from *srcURL*; if `null`, the result is shown in tuProlog app console.

On the other hand, the query result returned by the tuProlog service is encoded by the following URL scheme:

```
dstURL://command=commandresult
```

where

- *dstURL* is the URL scheme of the client receiving the result;
- *command* is one of the below four (`result`, `solution`, `nextSol`, `reset`), with the respective *commandresult*.

The specific syntax of *commandresult* varies with *command*, namely:

```
dstURL://?result=result&
                 engine=engineVersion&
                 engineAge=engineAge

dstURL://?solution=solution&
                   engine=engineVersion&
                   engineAge=engineAge

dstURL://?nextSol=nextSolution&
                  engine=engineVersion

dstURL://?reset=OK
```

where

- *result* is either `Yes` or `No` for success or failure, respectively;
- *engineVersion* is the version of the tuProlog system;
- *engineAge* can be `new` or `still` for a dedicated or cached engine, respectively;
- *solution* is the solution of the query;
- *nextSolution* is the next solution of the query;
- *error* contains potential errors.

Such a URL scheme is a first step towards a more SOA/Cloud-oriented interpretation of the LPaaS paradigm, with respect to the more general notion envisioned for tuProlog—where, essentially, "as a service" is interpreted according to the general software engineering principles of modularity, encapsulation, and reusability. In fact, making LPaaS available on a Web Server, a Cloud infrastructure, or other communication protocols adopted in IoT scenarios is just a matter of designing and deploying a suitable wrapper API, translating e.g. RESTful requests to the appropriate library method calls.

[1]To be supported in the next release.

Fig. 3. The multi-language toy example: Swift app using Java entities

V. Application Scenarios

Fig. 2 shows one first, typical application scenario: the client is a Swift/Objective-C application that provides a GUI for the computation of the symbolic derivative, while the symbolic computation is delegated to the tuProlog service. The Apple Watch extension (WatchKit) is also supported, so as to replicate the client GUI on the the iWatch®(which, in principle, could also play a more active role).

From the user's side, the first step is to create a dedicated Prolog engine, loaded with the proper logic theory:

```
tuPrologMobile://?src=tuPrologMobileClient&
  theory=dExpr(T,DT) :- dTerm(T,DT).
  dExpr(E+T,[DE+DT]) :- dExpr(E,DE), dTerm(T,DT).
  dExpr(E-T,[DE-DT]) :- dExpr(E,DE), dTerm(T,DT).
  dTerm(F,DF) :- dFactor(F,DF).
  dTerm(T*F,[[DT*F]+[T*DF]]) :- dTerm(T,DT), dFactor(F,DF).
  dTerm(T/F,[[F*DT]-[T*DF]]/[F*F]) :- dTerm(T,DT),
  dFactor(F,DF). dFactor(x,1).
  dFactor(N,0) :- number(N).
  dFactor([E],DE) :- dExpr(E,DE).
  dFactor(-E,-DE) :- dExpr(E,DE).
  dFactor(sin(E), [cos(E)*DE] ) :- dExpr(E,DE).
  dFactor(cos(E), [-sin(E)*DE] ) :- dExpr(E,DE).
```

The engine can now be queried via the top-level predicate `dExpr(+function, ?derivative)`. For instance, the request to derive the $\cos x \cdot \sin x$ function translates into the URL:

```
tuPrologMobile://?src=myTuPrologMobileClient&
  query=dExpr(cos(x)*sin(x), D).&dst=myTuPrologMobileClient
```

whose answer, $-\sin^2 x + \cos^2 x$, is returned (in a non-simplified form) as the URL:

```
tuPrologMobileClient://?solution=yes.
  D / ['+'(['*'(['*'('-'(sin(x)),1)],sin(x))],
  ['*'(cos(x),['*'(cos(x),1)])])]
  dExpr('*'(cos(x),sin(x)),
  ['+'(['*'(['*'('-'(sin(x)),1)],sin(x))],
```

```
['*'(cos(x),['*'(cos(x),1)])])])
&errors=&engine=2p Mobile - 3.0 beta
&engineAge=still
```

This URL encodes both the variable binding and the whole solution: in particular, the Prolog variable `D`, which contains the computed derivative, results

```
D / ['+'(['*'(['*'('-'(sin(x)),1)], sin(x))],
         ['*'(cos(x), ['*'(cos(x),1)])]) ]
```

Technically, the client app, written in Objective-C, triggers the iOS service based on the provided URL, while on the other side the server app, written in Java, parses the command, creates the Prolog engine, performs the required activity and returns the result.

More complex scenarios in a multi-language and multi-platform perspective could exploit tuProlog multi-paradigm programming support (`OOLibrary`) in a broader sense, with Swift or Objective-C apps interacting with Java entities. Fig. 3, for instance, shows a toy Swift app which creates Java objects, calls some methods, and elaborates the (Java) result back in Swift.

More precisely, as detailed in the following code, a Java string with the "`member(X, [a,d]).`" text is first bound to the Prolog variable `Q`, then a Java `Prolog` engine is bound to the Prolog variable `P` and used to solve the query `Q`, binding the result to the Prolog variable `S`; this result is finally converted back to a Java string via `toString`, binding the final result (`yes` in this case) to the Prolog variable `SOL`.

```
tuprologmobile://?src=null&query=
  new_object('java.lang.String', ['member(X, [a,d]).'], Q),
  new_object('alice.tuprolog.Prolog', [], P),
  P  <- solve(Q) returns S,
  S  toString returns SOL&dst=null
```

As shown in Fig. 3, the result is exploited by the client Swift application for its own purposes–here, trivially, just to print it in its console.

In a wider perspective, this approach fits particularly well those pervasive application scenarios where intelligence needs to be spread onto a broad set of heterogeneous devices, yet the choice of where logic computation should actually take place depends on many conditions—such as the available computational power, the available bandwidth, the instability of the connection, the need for situatedness.

In the IoT context, for instance, the Home Manager project [13] aims at supporting the construction of Socio-Technical Smart Spaces (STSS), namely in the case of a *smart home* and the surrounding environment where users live, according to the Butlers architectural vision [14]. There, a variety of smart devices, appliances, sensors, etc. need to be properly integrated and coordinated so as to provide advanced services to the home users, immersed in the smart space. To this end, the house is seen as a multi-agent system (MAS), coordinated via the TuCSoN middleware [11], [10]: smart appliances participate to the agent society by means of an agent, which embeds the device intelligence, while the social intelligence (with related policies, global goals, etc.) is implemented on top of TuCSoN tuple centres [12].

While Home Manager devices are supposed to be equipped with enough computational power to support their participation to the MAS and to perform the required reasoning, yet, in a more realistic scenario, some supposedly-smart devices could be not powerful enough to actually hold the computation locally—or, it could be impractical to do so because of the complexity of the required configuration. On the other hand, when enough computational power is available on the devices, the need for situatedness (along with the software engineering principles of locality and encapsulation) could instead suggest to compute directly on the Home Manager devices by querying situated Prolog engines reasoning on locally-available data.

In the overall, Home Manager – and, more generally, the smart home scenario – clearly shows how LPaaS could be an effective way to tackle multiple different aspects – from deployment to maintenance, cost issues, hardware requirements, etc. – while efficiently spreading intelligence where and when needed in pervasive contexts.

VI. Related Work

Many Prolog systems offer some form of support for the HTTP protocol and for the exploitation of Prolog as a service. The (Semantic) Web is one of the most promising application areas for SWI-Prolog [15]. Prolog handles the semantic web Resource Description Framework (RDF) model naturally, where RDF provides a stable model for knowledge representation with shared semantics. The PiLLoW library (Programming in Logic Languages on the Web) [16], developed in Ciao Prolog and available for Ciao [17], SWI-Prolog, SICStus [5] Prolog, and YAP, is one of the most widely known examples: a comparison between PiLLoW and SWI-Prolog for HTML documents handling can be found in [18].

Logic programming has also been used for the composition of Semantic Web Services: in [19], for instance, the GOLOG language [20] is extended for this purpose through the provision of high-level generic procedures and customising constraints. Finally, ProWeb [21] is an ALP-Prolog library aimed at embedded HTTP servers for controlling appliances: there, the notion of Request Processing Modules (RPM) supports different protocols, including HTTP, to support remote access.

While PiLLoW is an add-on library, aimed at directly handling HTTP, CGI and other internet protocols, tuProlog does not provide such services directly: rather, its modular and Java-based architecture makes it easy both to embed it in other applications, and, more generally, exploit it as a service through its API. Other features, if necessary, could be developed in the form of suitable tuProlog libraries and loaded by need—for instance, as in the case of the RDF library [22].

VII. 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. To meet such a requirement, light-weight logic programming engines are a crucial need, aimed at providing the reasoning services on-demand to the most heterogeneous client applications. In its turn, this requires the logic engine to be modular, multi-platform, and multi-language—as tuProlog is.

Then, making available such a logic programming service according to the different application needs, onto heterogeneous infrastructures, and across different interaction paradigms is a matter of designing suitable wrapper interfaces around the tuProlog service API—as we have done with the iOS platform. Accordingly, future work is devoted to further extend the reach of the LPaaS paradigm considering both traditional SOA infrastructures – e.g., tuProlog as a RESTful web service – and pervasive deployment scenarios from the IoT landscape—e.g., making tuProlog available over Bluetooth Low Energy connections.

Also, building a specialised tuProlog-oriented middleware, dealing with heterogeneous platforms, as well as with distribution, life-cycle, interoperation, and coordination of multiple, situated Prolog engines – possibly based onto the existing TuCSoN middleware – 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.

Acknowledgements

Authors would like to thank Dipl. Eng. Alberto Sita for his contribution to this project and his work to the new prototype of the iOS tuProlog app. Also, we would like to thank Asia Mariani for her help in fixing the last version of the paper.

References

[1] E. Denti and A. Omicini, "Engineering multi-agent systems in LuCe," in *ICLP'99 International Workshop on Multi-Agent Systems in Logic Programming (MAS'99)*, S. Rochefort, F. Sadri, and F. Toni, Eds., Las Cruces, NM, USA, 30 Nov. 1999.

[2] A. Omicini and F. Zambonelli, "MAS as complex systems: A view on the role of declarative approaches," in *Declarative Agent Languages and Technologies*, ser. Lecture Notes in Computer Science, J. A. Leite, A. Omicini, L. Sterling, and P. Torroni, Eds. Springer, May 2004, vol. 2990, pp. 1–17, 1st International Workshop (DALT 2003), Melbourne, Australia, 15 Jul. 2003. Revised Selected and Invited Papers. [Online]. Available: http://link.springer.com/10.1007/978-3-540-25932-9_1

[3] E. Denti, A. Omicini, and A. Ricci, "tuProlog: A light-weight Prolog for Internet applications and infrastructures," in *Practical Aspects of Declarative Languages*, ser. Lecture Notes in Computer Science, I. Ramakrishnan, Ed. Springer, 2001, vol. 1990, pp. 184–198, 3rd International Symposium (PADL 2001), Las Vegas, NV, USA, 11–12 Mar. 2001. Proceedings. [Online]. Available: http://link.springer.com/10.1007/3-540-45241-9_13

[4] tuProlog, "Home page," http://tuprolog.unibo.it, ALMA MATER STUDIORUM–Università di Bologna.

[5] SICStus Prolog, "Home page," http://sicstus.sics.se, SICS Swedish ICT.

[6] GNU Prolog, "Home page," http://www.gprolog.org.

[7] E. Denti, A. Omicini, and A. Ricci, "Multi-paradigm Java-Prolog integration in tuProlog," *Science of Computer Programming*, vol. 57, no. 2, pp. 217–250, Aug. 2005. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642305000158

[8] E. Denti, A. Omicini, and R. Calegari, "tuProlog: Making Prolog ubiquitous," *ALP Newsletter*, Oct. 2013. [Online]. Available: http://www.cs.nmsu.edu/ALP/2013/10/tuprolog-making-prolog-ubiquitous/

[9] Apple Inc., *The Swift Programming Language (Swift 2.2 Edition)*, Jun. 2014.

[10] A. Omicini and F. Zambonelli, "Coordination for Internet application development," *Autonomous Agents and Multi-Agent Systems*, vol. 2, no. 3, pp. 251–269, Sep. 1999, Special Issue: Coordination Mechanisms for Web Agents. [Online]. Available: http://link.springer.com/10.1023/A:1010060322135

[11] TuCSoN, "Home page," http://tucson.unibo.it, ALMA MATER STUDIORUM–Università di Bologna.

[12] A. Omicini and E. Denti, "From tuple spaces to tuple centres," *Science of Computer Programming*, vol. 41, no. 3, pp. 277–294, Nov. 2001. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642301000119

[13] Home Manager, "Home Page," https://apice.unibo.it/xwiki/bin/view/Products/HomeManager, ALMA MATER STUDIORUM–Università di Bologna.

[14] E. Denti, "Novel pervasive scenarios for home management: the butlers architecture," *SpringerPlus*, vol. 3, no. 52, pp. 1–30, January 2014. [Online]. Available: http://www.springerplus.com/content/3/1/52/abstract

[15] SWI-Prolog, "Home page," http://www.swi-prolog.org.

[16] D. Cabeza and M. V. Hermenegildo, "Distributed WWW programming using (Ciao-) Prolog and the PilloW library," *Theory and Practice of Logic Programming*, vol. 1, no. 3, pp. 251–282, May 2001. [Online]. Available: http://journals.cambridge.org/action/displayAbstract?aid=77729

[17] Ciao, "Home page," http://ciao-lang.org.

[18] J. Wielemaker, Z. Huang, and L. Van Der Meij, "SWI-Prolog and the Web," *Theory and Practice of Logic Programming*, vol. 8, no. 3, pp. 363–392, May 2008. [Online]. Available: http://journals.cambridge.org/article_S1471068407003237

[19] S. Mcilraith and T. C. Son, "Adapting GOLOG for composition of Semantic Web Services," in *8th International Conference on Principles and Knowledge Representation and Reasoning (KR-02)*, D. Fensel, F. Giunchiglia, D. L. McGuinness, and M.-A. Williams, Eds. Morgan Kaufmann, 22–25 Apr. 2002, pp. 482–493.

[20] H. J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. B. Scherl, "GOLOG: A logic programming language for dynamic domains," *The Journal of Logic Programming*, vol. 31, no. 1-3, pp. 59–83, 1997, special Issue "Reasoning about Action and Change". [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743106696001215

[21] M. Bathelt, U. Gall, B. Hindel, and C. Kurzke, "Accessing embedded systems via WWW: The ProWeb toolset," *Computer Networks and ISDN Systems*, vol. 29, no. 8-13, pp. 1065–1073, Sep. 1997, papers from the 6th International World Wide Web Conference. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S016975529700024X

[22] tuProlog, "RDF library," http://apice.unibo.it/xwiki/bin/view/Tuprolog/Libraries, ALMA MATER STUDIORUM–Università di Bologna.