An Application of Declarative Languages in Distributed Architectures: ASP and DALI Microservices

Stefania Costantini, Giovanni De Gasperis, Lorenzo De Lauretis *

Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica Università degli Studi di L'Aquila, L'Aquila (Italy)

Received 15 November 2020 | Accepted 13 December 2020 | Published 10 February 2021

Abstract

In this paper we introduce an approach to the possible adoption of Answer Set Programming (ASP) for the definition of microservices, which are a successful abstraction for designing distributed applications as suites of independently deployable interacting components. Such ASP-based components might be employed in distributed architectures related to Cloud Computing or to the Internet of Things (IoT), where the ASP microservices might be usefully coordinated with intelligent logic-based agents. We develop a case study where we consider ASP microservices in synergy with agents defined in DALI, a well-known logic-based agent-oriented programming language developed by our research group.

Keywords

AI For Ubiquitous Computing, Answer Set Programming, Intelligent Software Agents, Knowledge Based Systems, Microservices.

DOI: 10.9781/ijimai.2021.02.001

I. Introduction

The remarkable success of Answer Set Programming (ASP) in a wide variety of applications calls for the definition of specific software engineering principles. ASP is a successfully logic programming paradigm (cf. [1] and the references therein) stemming from the Answer Set (or "Stable Model") semantics of Gelfond and Lifschitz [2], [3], and based on the programming methodology proposed by Marek, Truszczyński and Lifschitz [4], [5]. ASP is put into practice by means of effective inference engines, called solvers[1]. ASP has been widely applied in many fields, e.g., to information integration, constraint satisfaction, routing, planning, diagnosis, configuration, computer-aided verification, biology/biomedicine, knowledge management, and many others.

In this paper we discuss the possibility of exploiting ASP to define components for distributed systems, to be deployed over different nodes of a network. In this perspective, the connections between components and the ways of exchanging information should be clearly specified. Our approach is inspired by the microser-vices architectural abstraction, which can be described as a particular way of designing distributed software applications as suites of independently deployable in-teracting services (cf. for instance the survey [6], and https://martinfowler.com/articles/microservices.html#CharacteristicsOfAMicroserviceArchitecture).

A microservice is indeed a component, as it is a unit of software that is independently replaceable and modifiable: in fact, it intended as a self-contained piece of business functionality with clear interfaces that can be accessed by the "external world". This kind of architectural abstraction enables distribution, as each microservice is meant to be executed as an independent process, and heterogeneity, as it allows different services to be written in different programming languages. Microservices are a suitable architectural abstraction for the Internet of Things (IoT): a microservice may incapsulate a physical object, where service inputs and/or outputs can possibly be linked to sensors/actuators. Microservices are by their very nature heterogeneous, so open issues are: how microservices communicate with each other (synchronous, asynchronous, which is the message format, etc.); and, the protocols used for the communication.

Microservices in real distributed software architectures and in cloud computing are usually deployed via lightweight containers. In standard terminology borrowed from software engineering, a container is a standard unit of software that packages up code and all its dependencies; so, the application runs quickly and reliably and can be seamlessly transferred from one computing environment to another. A widely used tool to create containers is Docker, available in the form of an open source Docker Engine[2]. A Docker container image consists in a lightweight, standalone, executable package of software that includes all elements needed to run an application: code, run-time support, system tools, system libraries and settings.

Along this line, we propose ASP microservices that might be blended into heterogeneous systems, and even into Multi-Agent-System (MAS) since each such component may be seen as a reactive agent. They could in perspective be employed in cloud computing and in IoT, including robotic applications. In this paper we discuss how these components, that we call μASP-Services (μASPSv's), can be

[1] Many performant ASP solvers are available as open-source tools, a list is reported at https://en.wikipedia.org/wiki/Answer_ set_programming.

* Corresponding author.

E-mail addresses: stefania.costantini@univaq.it (Stefania Costantini), giovanni.degasperis@univaq.it (Giovanni De Gasperis), lorenzo.delauretis@graduate.univaq.it (Lorenzo De Lauretis).

[2] See https://www.docker.com

specified, how their interfaces to the "external world" can be defined, and how they should procedurally behave. In fact, a μASPSv is meant to be based upon a 'core' ASP program whose activities, however, should be triggered by external stimula/requests coming from some source, and whose results should be returned to the requesters.

In our view, the 'core' ASP program should be included into a container, that can be possibly realized via the Docker technology, which should also include: an interface, to provide the Âμ ASPSv with inputs, and to select and deliver the outputs; solving capabilities to compute the answer sets. So, a docker deployment for a Âμ ASPSv should include the so For ASP, standalone versions of the most important solvers are nowadays available. New solutions have been recently introduced [7], that allow for incremental solving of an ASP program under atoms/rules addition/deletion, and so might be used to provide a μASPSv with new inputs and cancel old ones. Thus, a docker deployment for a μASPSv should include the source program, its 'execution shell', and the solver.

A small specimen of the proposed approach is represented in the following example, which is meant to be (a fragment of) the code of a controller component/agent, acting in the IoT. This piece of code might be in fact the ASP 'core' of a μASPSv. *test_ok* is the input coming from a sensor, with value 'true' if the controlled device is working properly, (otherwise the value is set to false).

device_ok ← t*est_ok*.

device_fault ← not *test_ok*.

wait ← not *wait*, not *sensor_input*.

In this simple example, inconsistency (due to the odd cycle over *wait*) is to be interpreted as a 'no-operation' controller state, where the component is waiting for sensor's outcome. It can be assumed that the sensor provides results at a certain frequency. The outcome, i.e., *device_ok* or *device_fault*, is to be delivered to whatever other components would ask for it.

In order to work in a standalone way within a distributed system, an interface (or 'shell') will manage the 'core' ASP program, and in particular will perform the following functions. First, manage the inputs and outputs of the μASPSv: i.e., be able to detect input arrival and to dispatch the outputs according to the request coming from the μASPSv's external environment. In the above example, inputs can be: (1) queries over the device state for which an answer has to be delivered, and (2) sensor outcomes, which are to be considered as particular inputs which activate the module. In the general case, upon the arrival of inputs, the shell will: (i) add the inputs to the ASP program as facts; (ii) evaluate the answer sets of the resulting ASP program; (iii) according to previously-received requests, extract (from the answer sets) the answers and and deliver them to the external environment. Notice that the shell, after delivering the outputs, will remove (some or all of) the last-added program facts so as to bring back the controller to the 'no-operation' state. In a 'stateless component', all inputs will be removed, while some of the inputs can be left if instead the component is meant to have a state; the shell functioning is enabled (or at least greatly simplified) by the new advanced solving capabilities provided in particular by the clingo ASP solver [7].

There are however complex devices in the Internet of Things that should be managed in a coordinated way. Take for instance a car, where modern cars include several control devices for the various parts. Each such device will be managed by a microservice, where such microservices should produce coordinated behavior. It is thus a reasonable choice to define these components as agents. In this way, the overall control over the complex device will be managed by a Multi-Agent-System (MAS), which is by definition capable of integrated behavior. Several approaches to logic-based agent-oriented languages exist (cf., e.g., [8]–[10]). We may notice that such kind of agents can be the natural complement to ASP microservices. In general terms, one might want to adopt ASP microservices whenever there is the need to cope with uncertainty, or the need to manage possible alternative scenarios. When instead immediate direct reactive/proactive behavior is required, logical agent may represent a suitable tool. Among the different existing logical-based agent frameworks, to develop our case study we choose the DALI logic-based agent-oriented language and framework (introduced in Section V), which has been developed by our research group, and that (as illustrated later) we have already used in synergy with ASP modules in past work.

In this paper we introduce a formal definition of ASP microservices and we outline a possible logic-based semantics of an overall heterogeneous distributed system encompassing such modules, and other logical components/agents. The paper is structured as follows. In Section II we introduce basic concepts about microservices. In Section III we recall (for the sake of completeness) the Answer Set Programming paradigm, and in Section IV we briefly survey and discuss existing approaches to modularity in ASP. We introduce our contribution in Sections VII and IX, i.e.: (1) how to define and implement μASPSv's so as to be able to get inputs and extract answers, and how the inner ASP program might be structured; (2) how to provide a formal semantics to a generic microservice architecture possibly encompassing μASPSv's. In Section VIII we discuss a small case study, developing a specific μASPSv which implements an intelligent agent managing a road intersection (i.e., a "virtual traffic light"), where cars are modeled as DALI logical agents. Finally, in Section X we conclude. This paper is in our view an evolution of the work in [11], in the sense that there, as illustrated in Section VI, ASP modules were invoked as auxiliary modules by agents in a DALI multi-agent system. Here, we make it possible for an ASP program to act as an independent component, that is able to interact with other components, among which agents.

II. Background: Microservices

In order to better understand Microservices, let us first introduce the concept of "Service". A Service, as a software component, is a mechanism to enable access to one or more software capabilities [12]. It provides other applications with stable, reusable software functionalities at an application-oriented, business-related level of granularity using certain standards [13]. Service-Oriented Architecture (SOA) is a software architectural style that uses services as the main building component [12]. Key features of SOA are heterogeneity, standardization and "evolvability" of services.

Microservices can be seen as a technique for developing software applications that, inheriting all the principles and concepts from the SOA style, permits to structure a service-based application as a collection of very small loosely coupled software services [14].

A MicroServices Architecture (MSA) is an evolution of the SOA architecture, making the communication lighter and the software parts (Microservices) smaller. As empasized in [15], it can be seen as a new paradigm for programming applications by means of the composition of small services, each one running its own processes and communicating via light-weight mechanisms. Key features of MSA are bounded scope, flexibility and modularity [15]. I.e., there is a clear definition of the data a microservice service is responsible for and is "bound to". So, a microservice owns this data and is responsible for its integrity and mutability.

The work in [16] shows that a distributed MSA can easily fit into an IoT system. In particular, the set of microservices can be seen as a Multi-Agent-System, cooperating to realize all system functionalities.

At the current day, microservices are still a new and emerging paradigm, having building standards not perfectly defined and communication protocols that are not well specified: in fact, following one of the definitions of microservices [14], [15], they are small loosely

coupled software services that communicate, possibly exploiting service discovery to find the route of communication between any two of them. In our work, we are proposing a new approach, that is μ ASPSv's, which are based upon an inner ASP program.

III. Background: Answer Set Semantics (AS) and Answer Set Programming (ASP)

The following introduction consists of standard material taken (literally for what concerns long-established scientific terminology and definitions) from [1], [17]–[19]. "Answer Set Programming" (ASP) (cf. [1] and the references therein) is a successful programming paradigm based on the Answer Set Semantics. In ASP, one can see an answer set program (for short, just "program") as a set of statements that specify a problem, where each answer set represents a solution compatible with this specification. Whenever a program has no answer sets (no solution could be found), it is said to be inconsistent, otherwise it is said to be consistent.

Syntactically, an ASP program Π is a collection of rules of the form

$H \leftarrow A_1, ..., A_m, not\ A_{m+1}, ..., not\ A_{m+n}.$

where H is an atom, $m, n \geq 0$, and each A_i, $i \leq m+n$, is an atom. Atoms and their negations are called literals. Symbol ← is often indicated as :- in practical programming. The left-hand side and the right-hand side of the clause are called head and body, respectively. A rule with empty body is called a fact. A rule with empty head is a constraint, where a constraint of the form '$\leftarrow L_1,..., L_n$.' states that literals $L_1,..., L_n$ cannot be simultaneously true in any answer set. Constraints are often rephrased as '$f \leftarrow not\ f, L_1,..., L_n$.' where f is a fresh atom. To avoid the contradiction over f, some of the Li's must be false thus forcing f to be false, and this, if achieved, fulfills the constraint.

Actually, an ASP rule can have a more general form including a disjunction of literals in the head, and "classical negation" of atoms [3]; various useful programming constructs have been introduced over time; for simplicity, we consider the basic form, i.e., "normal logic programs". The interested reader can refer, e.g., to [20] for a complete up-to-date discussion about ASP syntax and practical use.

The answer set (or "stable model") semantics (AS) [2] can be defined in several ways (cf., e.g., [21], though more recently several other definitions have appeared in the literature). However, answer sets of a program Π are found among the supported minimal classical models of the program (interpreted as a first-order theory in the obvious the model (directly or indirectly) by its own negation. This is why it can be the case that no answer set exists: take, e.g, simple ASP program $p \leftarrow not\ p$ which is equivalent to first-order theory $p \vee p$ with unique minimal model $\{p\}$ which is not an answer set as p is supported, in the model, by its own negation. As it is well-known, AS extends the three-valued Well-Founded semantics [22] for normal logic programs, where every program Π has a well-founded model $wfm(\Pi) = \langle T, F \rangle$ where T is the set of true atoms, F is the set of false atoms, and the remaining atoms (implicitly) form the set $U = Undef(\Pi)$ of the undefined atoms. For every answer set M it holds that $T \subseteq M$, so finding the answer sets accounts to suitably assigning truth values to the undefined atoms.

The ASP approach to problem-solving consists basically in the following: (i) encoding of the given problem via an ASP program; (ii) computing the "answer sets" of the ground program via an ASP solver (a list of available solvers can be found at https://en.wikipedia.org/wiki/ Answer_set_programming), where, as a preliminary step, solvers perform the "grounding" of the program, by substituting all variables with the constants occurring in the program; (iii) extracting the problem solutions by examining such answer sets; in fact, answer set contents can be in general reformulated in order to present the solution in terms of the given problem.

A top-down query answering device which is prolog-style, i.e., does not compute answer sets in advance to extract the query answers, has been defined in [23] for RAS, where RAS is a variation of AS where every program admits answer sets[3]. RAS and AS coincide however over a wide class of programs: some sufficient conditions that identify classes of programs where the two semantics coincide are reported in [26]. Queries that have been introduced are, first of all, "? A" asking whether A is true w.r.t. some answer set of given program Π. Other queries are the following: query "? not A" asks whether A is false w.r.t. some answer set of Π, and therefore it succeeds if not A is true in some of them (this implements the operator not introduced in [27]); query "? not not A" asks whether not A is false in some answer set, and therefore it succeeds if A is true in some of them, which corresponds to query "? MA", M standing for 'possibility' in the modal logic sense; query "? not not A" asks whether it is not true that A is false w.r.t. some answer set of Π, i.e., that A is true in all of them, which corresponds to "? KA", K standing for 'knowledge' in the modal logic sense; query "? not notnot A" asks whether A is false in every answer set, meaning Knot A, i.e., not MA (a new operator NOT is a shorthand for not notnot A).

IV. Background: Modularity in ASP

Existing approaches to modularization of ASP programs have been extensively reviewed in [18], to which the reader may refer for a complete account. Reporting faithfully from there, such approaches can be divided into two lines: "programming-in-the-large", where programs are understood as combinations of separate and independent components, combined by means of compositional operators; "programming-in-the-small", in which logic programming is enriched with new logical connectives for managing subprograms.

Considering the programming-in-the-small vision: in [28], program modules are viewed as generalized quantifiers; [29] proposes templates for defining subprograms; [30] developed a declarative language for modular ASP, which allows a programmer to describe a state how one ASP module can import processed answer sets from another ASP module. The work in [31] explores how to divide an ASP program into components according to its structure in terms of cycles.

Lifschitz and Turner's "splitting set theorem" (cf. [32]), or variants of it, is underlying many programming-in-the- large approaches. The basic idea is that a program can be divided into two parts: a "bottom" part and a "top" part, such that the former does not refer to predicates defined in the latter. Computation of the answer sets of a program can be simplified when the program is split into such parts.

[33] defines the notion of a "DLP-function" which is basically a module for which a well-defined input/output interface is provided; a suitable compositional semantics for modules is introduced. [34] provides a simple and intuitive notion of a logic programming module that interacts through an input/output interface. This is achieved by accommodating modules as proposed by [35] to the context of Answer Set Programming. Full compatibility of the module system with the stable model semantics is achieved by allowing positive recursion to occur inside modules only.

[36] focuses on modular non-monotonic logic programs (MLP) under the answer set semantics, where modules may provide input to other modules. Mutually recursive module calls are allowed.

[37] defines modules in terms of macros that can be called from a program. [38] provides modules specification with information hiding, where modules exchange information with a global state.

[3] To the best of our knowledge, the only alternative query-answering device for ASP that does not compute the answer sets in advance has been introduced in [24], [25], though under some syntactic-semantic limitations.

In [39] a technique is proposed to allow an answer set program to access the brave or cautious consequences of another answer set program. [40] proposes "modular logic programs" as a modular version of ASP. This work consider programs as modules and define modular programs as sets of modules. The authors introduce "input answer sets", which is the key semantic object for communication between modules.

[41] proposes to adopt ASP modules in order to simulate (within reasonable complexity) possibility and necessity operators. Such operators (given the underlying modules) are meant to be usable in ASP programs, but possibly also programs written under other programming paradigms.

It can be seen that none of the above approach tackles modularization in view of using ASP modules as standalone components in distributed systems. Therefore, our approach is a novelty in the landscape of the current literature.

V. Background: Logical Agents and DALI

The material exposed in this section, which reports about our previous work concerning logical agents so as to provide the notions needed in the subsequent sections, is largely taken (in some parts literally, to be faithful to well- established terminology) from [42]–[52] and from the DALI web site https://github.com/AAAI-DISIM-UnivAQ/DALI.

The original perspective on agents in Artificial Intelligence was focused on the agents' reasoning process, thus identifying "intelligence" as rationality, thus neglecting the interactions of the agents with the environment and with other agents. This perspective has been heavily criticized for instance in [53], [54], that adopts in an extreme way the opposite point of view, arguing that "intelligent" behavior results solely from the ability of an agent to react appropriately to changes in its environment.

Reasoning about beliefs, but also about what an agent means and chooses to do, is the basis of the seminal approach of the BDI (Belief, Desires, Intention) logic for modelling agents by [55], that resulted in the definition of the AgentSpeak agent-oriented logic programming language [56]. At the same time, in the approach of [57], agents were theories (logic programs), each one with its name, and they were able to communicate with each other via two communication primitives (tell/told). A view of logical agents, able to be both rational and reactive, i.e., capable not only to reason and to communicate, but also to provide timely response to external events, has been introduced in [58], [59].

After those seminal approaches, both the notion of agency and its interpretation in computational logic have greatly evolved. Many computational-logic-based agent- oriented languages and frameworks to specify agents and Multi-Agent Systems (MAS) have in fact been defined over time (for a survey of these languages and architectures the reader may refer, among many, to [8]–[10]). Their added value with respect to non-logical approaches is to provide clean semantics, readability and verifiability, as well as transparency and explainability 'by design' (or almost), as logical rules can easily be transposed into natural-language explanations.

DALI [42], [43], [60] is an Agent-Oriented Logic Programming language, where the autonomous behaviour of a DALI agent is triggered by several kinds of events: external events, internal, present and past events.

External events are syntactically indicated by the postfix E. Reaction to each such event is defined by a reactive rule, where the special token :>. The agent remembers to have reacted by converting an external event into a past event (postfix P). An event perceived but not yet reacted to is called "present event" and is indicated by the postfix N. It is often useful for an agent to reason about present events, that make the agent aware of what is happening in its external environment.

In DALI, actions (indicated with postfix A) may have or not preconditions: in the former case, the actions are defined by actions rules, in the latter case they are just action atoms. The new token :< characterizes an action rule that specifies an action's preconditions. Similarly to events, actions are recorded as past actions.

Internal events is the device which makes a DALI agent proactive. An internal event is syntactically indicated by the postfix I, and its description is composed of two rules. The first one contains the conditions (knowledge, past events, procedures, etc.) that must be true so that the reaction (in the second rule) may happen. Thus, a DALI agent is able to react to its own conclusions. Internal events are automatically attempted with a default frequency, customizable by means of user directives.

The DALI communication architecture [44] implements the DALI/FIPA protocol, which consists of the main FIPA primitives[4] , plus few new primitives which are peculiar to DALI. Notice that, DALI has been made compatible with the Docker technology (cf. [61] for details). So, a DALI agent can be deployed within a container.

The semantics of DALI is based upon the declarative semantic framework introduced in [45], aimed at encompassing approaches to evolving logical agents, by understanding changes determined by external events and by the agent's own activities as the result of the application of program- transformation functions.

We abstractly formalise an agent as the tuple $Ag = \langle P_{Ag}, E, I, A\rangle$ where Ag is the agent name and P_{Ag} is the "agent program" according to the specific language adopted. E is the set of the external events, i.e, events that the agent is capable to perceive and recognize: let $E = \{E_1, \ldots, E_n\}$ for some n. I is the set of internal events (distinguished internal conclusions, that may include agent's desires and intentions): let $I = \{I_1, \ldots, I_m\}$ for some m. A is the set of actions that the agent can possibly perform: let $A = \{A_1, \ldots, A_k\}$ for some k. Let $ev = (E \cup I \cup A)$.

In the DALI syntax, used below for the examples, atoms indicated with a postfix correspond to events of various kinds. In particular, if p is an atom, pE is an external event, pA is an action and pI an internal event.

According to this semantic account, one will have an initial program $\Pi 0$ obtained by the program P_{Ag} provided by a programmer. According to events that happen, agent's activities and internal reasoning, and actions which are performed, Π_0 will "evolve" through corresponding program-transformation steps (each one transforming Π_i into Π_{i+1}, cf. [45]), and thus gives rise to a Program Evolution Sequence $PE = [\Pi_0, \ldots, \Pi_n, \ldots]$. The program evolution sequence will imply a corresponding Semantic Evolution Sequence $ME = [M_0, \ldots, M_n, \ldots]$ where M_i is the semantic account of Π_i.

Different languages and different formalisms in which an agent can possibly be expressed will influence the following key points: (i) when a transition from Π_i to Π_{i+1} takes place, i.e., which are the external and internal factors that determine a change in the agent; (ii) which kind of transformations are performed; (iii) which semantic approach is adopted, i.e., how M_i is obtained from Π_i.

The semantic account includes an Initialization step, where the program P_{Ag} written by the programmer is transformed into a corresponding program Π_0 by means of some sort of knowledge compilation. In DALI for instance, the initialization step extracts the

[4] FIPA is a widely used standardized ACL (Agent Communication Language), cf. http://www.fipa.org/specs/fipa00037/SC00037J.html for language specification, syntax and semantics.

list of internal and external events, and the control directives that are associated to the program (e.g., for defining priorities among events and frequencies for checking the occurrence of events). In general in fact, Π_0 can be simply a program (logical theory) or can have additional control information associated to it.

Agents usually record events that happened and actions that they performed. Notice that an agent can describe the state of the world only in terms of its perceptions, where more recent remembrances define the agent's approximation of the current state of affairs. We thus define set $\mathscr{P}$ of current (i.e., most recent) past events, and a set PNV where we store all previous ones (under certain conditions). We define the 'history' H of an agent as the tuple $\langle \mathscr{P}, PNV \rangle$, dynamically augmented with new events that happen. In DALI, a past event in P is in the form $pP : T_i$, where p is an atom corresponding to an event, postfix P stands for 'past' and T_i is a time-stamp indicating when the event has been perceived. In [62] we have defined Past Constraints, which allow one to define when and upon which conditions (apart from arrival of more recent versions) past events should be moved into PNV, and later on possibly removed.

Definition 1 (Evolutionary semantics). Let Ag be an agent. The evolutionary semantics ε^{Ag} of Ag is a tuple $\langle H, PE, ME \rangle$, where H is the history of Ag, and PE and ME are its program and semantic evolution sequence.

DALI has been fully implemented, and a programming environment has been devised. The DALI programming environment [60] is freely available, and at the current stage of development offers a multi-platform folder environment, built upon Sicstus Prolog [63] programs, shells scripts, Python scripts to integrate external applications, a JSON/HTML5/jQuery web interface to integrate into DALI applications, with a Python/Twisted/Flask web server capable to interact with A DALI MAS at the backend. We have recently devised a cloud DALI implementation, reported in [64], [65]. As shown in [64], the preexisting DALI framework has been extended to "DALI 2.0" by using open sources packages, protocols and web-based technologies. DALI agents can thus be developed to act as high level cognitive robotic controllers, and can be automatically integrated with conventional embedded controllers. The web compatibility of the framework allows real-time monitors and graphical visualizers of the underline MAS activity to be specified, for checking the interaction between an agent and some external device, that can possibly be a robotic subsystem. The cloud package ServerDALI allows a DALI MAS to be integrated into any practical environment. In [65] illustrate the recent "Koiné DALI" framework, where a Koiné DALI MAS can cooperate without problems with other MASs, programmed in other languages (logical or non-logical), and with object-oriented applications. In summary, the enhanced DALI can be used for multi-MAS applications and hybrid multi-agents and object-oriented applications, and can be easily integrated into preexisting applications.

The DALI framework has been experimented, e.g., in applications for: unattended hardware testing of hardware-software platforms in telecommunication industry; user monitoring and training; emergencies management (such as first aid triage assignment); security or automation contexts; home automation and processes control. More generally, DALI has proved to be useful in every situation that is characterised by asynchronous events sources that require reasoning over a dynamic data collection: either simple events, and/or events that are correlated to other ones even in complex patterns. In fact, in order to be able to perform Complex Event Processing, i.e., to actively monitor event data so as to make automated decisions and take time-critical actions, DALI has been empowered with CEP capabilities [66], of which the implementation at this day is partial, but is being actively developed: since the 2018 release, DALI supports the double concurring events occurrence in a predefined time window, so that reaction rules can be defined where two events from different asynchronous sources happen to fall in the same time interval. An architecture encompassing DALI agents and called F&K (Friendly-and-Kind) system [67] has been proposed for (though not restricted to) applications the e-Health domain. We have since long equipped DALI with a plugin for invoking ASP solvers and thus executing ASP modules in the so called ASP_DALI event, postfix P stands for 'past' and T_i is a time-stamp extension available at our github organization repositories[5]. An ongoing experimentation is about emotion recognition in the context of cognitive robotics [68], were real time analysis of the non verbal communication interaction between a human and the anthropomorphic NAO robot is performed by an extended DALI, consisting in an ASP_- DALI and QuLog/Teleor [69] multi-agent system. In this experimental setup, several sub-symbolic perception systems generate real-time fluents about the emotional state of the human while interacting with the robot, and the MAS in background determines the best emotional state according to a predefined model in a timely manner, so as to suggest the most appropriate behaviour to the robot.

VI. DALI and ASP in Synergy: Past Work

The work presented in [11] studied the application of DALI and ASP to the problem of dynamic goal decomposition and planning in scenarios characterised by a strong inter-dependency between action and context, for instance those related to rescue intervention in a territory upon occurrence of some kind of catastrophic event. The paper in particular proposed an architecture that integrates DALI MASs (DALI Multi-Agent Systems) and ASP modules for reaching goals in a flexible and timely way.

The effectiveness of this solution was demonstrated by means of a case-study where DALI agents cooperate in order to explore an unknown territory. The solution is based upon a MAS instead of a monolithic software solution because it is important that each software component, implemented as an agent, can partially retain its autonomy during asynchronous event processing. In fact, in this way each agent can be enriched with high-level reasoning/control behaviours that can coexists with the planning/executing activity. The MAS solution also permits to distribute the computational effort and increases overall robustness.

The DALI MAS is intended to fulfill the so-called bounded rationality principle, by which a plan for reaching a goal has to be devised and executed in a timely manner before a ultimate T_{max} deadline. There is a second deadline $T_{PlanMax} < T_{Max}$ by which a plan has to be computed and selected, so that the remaining time is sufficient for plan execution.

In the context of microservices we might improve this solution by defining a specific agent role called "micro- meta-planner" that shall supervise the task allocation over ASP and DALI agents, and which is responsible of the real-time compliance of the overall system. For example, in those situation were the ASP module could not deliver answer sets in polynomial time, the micro-planner shall take over either by providing a fail-safe plan, or by providing a set of short plans' definitions aimed to obtain better working conditions for the ASP solver and its grounding subsystem, such as the GRINGO grounder [70].

Thus, given the input set $T_{PlanMax}$, T_{Max}, G, N, where G is the goal, and N is the instance size of the problem to be solved (if applicable), the MAS operates via the following steps.

i) Decompose the overall goal into suitable sub-goals;

[5] https://github.com/AAAI-DISIM-UNIVAQ

ii) For each sub-goal, generate (via an ASP module) a plan within the $T_{PlanMax}$ deadline;

iii) Execute the plan within the T_{Max} deadline; in case of failure (insufficient time), maximize the length of the partially executed plan;

iv) In case of a change of conditions in the environment, re- plan, possibly limiting this activity to specific sub-goals resulting from the partitioning.

Sub-goals can be determined by any kind of goal partitioning algorithm. In the disaster management case study, it was obtained simply by sub-dividing the main geographical area into slightly overlapping sub-territories.

The planner agent equipped with an ASP module may find more than one plan for each (sub-)goal; so, metrics can be applied by which a plan could be preferred to another one.

VII. μ ASPSv's: Specification and Implementation Guidelines

The present work can be seen as an evolution of the work in [11], in the sense that we make it possible for an ASP program to act as an independent component, instead of being invoked as an auxiliary module by an agent.

In this section we provide in fact an abstract definition of a μASPSv, and some more specific indication of how such a component might be enacted and inserted into a distributed system, and how the inner ASP program might be structured.

Definition 2. Let Π be an ASP program, and let $U = Undef(\Pi)$. A μASPSv based upon Π, denoted as μASPSv(Π), has the following specification:

- Inner ASP program Π;
- Activation signal A (optional), with $A \in Undef(\Pi)$;
- Stop signal S (optional), with $S \in Undef(\Pi)$;
- Input set $\{I_1, ..., I_k\} \subseteq Undef(\Pi)$;
- Output set $\{O_1, ..., O_h\} \subseteq Heads(\Pi)$.
- Query result set $\{Q_1 = v_1, ..., Q_r = v_r\}$ where $\{Q_1, ..., Q_r\}$ are queries[6], formulated over atoms occurring in $Heads(\Pi)$ and the v_is can have value "true" or "false".

The elements listed above have the following meaning.

Whenever the activation signal is expected, if A is not true in Π, then μASPSv(Π) is in a state of no-operation.

Whenever the stop signal is expected, if S becomes true in Π, then μASPSv(Π) will go back into a state of no-operation.

The input set is a set of atoms that, when some of them are added to Π, contribute to answer sets computation. Each of such atom corresponds to an input/request received from the μASPSv's surrounding environment.

The output set is a set of atoms extracted from the answer sets of Π plus the current input set. Each of these atoms corresponds to an output/answer to be delivered into the μASPSv's surrounding environment.

The query result set is a set of truth values elicited from the answer sets of Π. Each of these values corresponds to result of a query, to be delivered into the μASPSv's surrounding environment.

Notice that, we admit as inputs atoms included in $Undef(\Pi)$, i.e., atoms that have truth value "undefined" in the well-founded model. This means that external inputs are intended to activate behaviors in program Π, without however threatening is basic functioning, represented by the atoms which are true or false in the well-founded model.

In order to make it possible for μASPSv(Π) to operate dynamically, thus receiving inputs and delivering outputs and answers, a suitable shell program must be defined, in any programming language able to be interfaced with an answer set solver. Below we provide a schematic essential definition of such a shell program, to be used as a guideline for actual definition and implementation. The shell program will rely upon an input-output table, where each potential and actual input and potential and actual output will be annotated, together with the list of external components sending inputs, and the list of external components to which outputs are to be delivered.

Definition 3. The shell responsible to manage an ASP microservice μASPSv(Π) can be specified by the following pseudo-code.

begin

1. while not activation then no-operation endwhile;
2. if activation then add atom A to Π as a fact to bring μASPSv(Π) into operation;
3. while not stop do at frequency f
   (a) detect and annotate actual inputs
   $\{I_{j1}, ..., I_{jr}\} \subseteq \{I_1, ..., I_k\}$;
   (b) add $\{I_{j1}, ..., I_{jr}\}$ to Π as facts;
   (c) obtain the answer sets $\{S_1, ..., S_n\}$ of (the augmented) Π;
   (d) elicit outputs $\{O_1, ..., O_v\} \subseteq \{O_1, ..., O_h\}$;
   (e) extract query results $\{Q_1, ..., Q_t\} \subseteq \{Q_1, ..., Q_r\}$;
   (f) deliver outputs and query results according to requests;
   (g) remove $\{I_{v1}, ..., I_{vs}\} \subseteq \{I_{j1}, ..., I_{jr}\}$ from Π
   (h) and remove relative annotations;
   endwhile;
4. add atom S to Π as a fact and remove atom A, to bring Π into no-operation.

end.

This shell program is able to activate and stop a μASPSv, and to execute, until possibly a stop signal arrives, a loop where: the inputs are received from the external environment and delivered to Π; and, outputs and query results are extracted from the answer sets of Π (given the inputs) and delivered to the external environment. Precisely, each input will arrive from some external component, and each output will have to be delivered to some other (or to the same) component. At the end of each cycle some or all of the inputs will be removed from Π and the relative annotations will be eliminated; removing all the inputs determines a stateless component, while omitting to remove some of the inputs, forever of for some time interval, accounts to defining a stateful component. Input detection will occur at a certain frequency, suitable for each particular kind of component, environment, and application domain. Some of the inputs may come from sensors (and therefore they do not require any answer) and some of the outputs may go to actuators. This is also annotated in the input-output table. The parts concerning the activation and stopping of the µASPSv (first and second line after the begin, and last line before the end) will be omitted if the component is running forever rather being first activated and then stopped.

Notice that the above definitions can find easier practical application thanks to the advanced features of modern solvers such as clingo [7], that provides "multi-shot"solving features, coping with grounding and solving in continuously changing logic programs. In particular, "multi-shot"solving allows a given ASP program to evolve

[6] c.f. previous section for possible queries.

during the reasoning process, because data or constraints are added, deleted, or replaced. This is exactly what is needed in order to send to a μASPSv the activation and stop signals, and to cancel old inputs and add new ones.

Many practical aspects remain however to be defined in order to obtain an implementation. For instance, if a μASPSv is to be situated within a multi-agent system, input-output-query exchange might happen by means of the above-mentioned FIPA ACL. The shell program can be made FIPA-compliant (i.e., able to exchange and understand FIPA messages) either by developing suitable code, or, better, by importing a suitable library such as, e.g., the freely available JADE library[7]. The JADE library is an advance middleware that offers many functionalities to "agentify" imperative or object-oriented or other kinds of programs. In fact it provides: the agent abstraction (i.e., a given program, when running, is seen by the external environment as an agent); the ability of peer to peer inter-agent FIPA asynchronous message-passing; a yellow pages service supporting subscription of agents and a discovery mechanism, and many other facilities to support the development of distributed systems.

So for instance, an input can be sent to a μASPSv via a FIPA "request" message with the input as argument, to be interpreted on the μASPSv's side as a request to reply with a "confirm" message, containing the corresponding output. A query can be sent to the μASPSv via a FIPA "query-if" message whose answer will be again a "confirm", conveying the truth value of the query. Notice that, to avoid ambiguities, the FIPA syntax provides the facility to identify each message via a certain arbitrary identifier, so that the answer message can indicate that it is 'in-reply-to' to that identifier.

The JADE yellow pages services might be exploited by μASPSv's which would want to register as agents with a name and a role, and then communicate with each other in an asynchronous way. Or, since most MASs offer such a mediator service, μASPSv's might enroll in any agent community. Finally, they might communicate peer-to-peer with other agents that they are aware of, or that they locate via the mediator.

Let us now consider how to structure the 'core' program Π, on which a microservice μASPSv(Π) is based. First, activation and stopping of a module can be simply obtained by a couple of constraints, that make the program inconsistent (in no-operation state) if either activation A has not arrived, or stopping signal S has been issued:

$: - not\ A.$ % *module activation*

$: - S.$ % *module stop*

Then, when the module has been activated, upon arrival of new inputs, the inner program Π will in general 'produce' (admit) answer sets. If the answer set is unique then the outputs can be univocally identified. Otherwise the shell, in the 'elicit outputs' part, will have to adopt some kind of policy (e.g., preferences, utilities, costs or other) to select which answer set to consider. The queries, being by definition specified upon the whole set of answer sets, will always return an univocal result. In case, given the present input, Π should be inconsistent, then the output will consist in a failure signal (e.g., in the FIFA ACL, there is the "failure" primitive to be used in such cases).

VIII. Case Study

The case study that we propose here is inspired to issues raised by applications related to autonomous vehicles. Presently, machine learning mechanism have been defined to allow autonomous cars to comply with traffic lights by detecting their color, so as to pass with green and stop with red similarly to traditional cars. Such mechanisms must be trained, are prone to errors, and are potentially subject to adversarial machine learning.

In our view, physical traffic lights might in perspective disappear, to be substituted by monitoring agents that would receive requests to pass from cars and consequently issue authorisations. This either in routes dedicated to autonomous vehicles, or in the (very reasonable) hypothesis to equip also 'traditional' cars with a device to interact with the monitoring agents.

Below we propose the sample design of the inner program concerning a μASPSv which implements the monitoring agent of a road intersection, taking the place of a physical traffic light. In the example, the traffic light agent is called *tl* and, for the sake of simplicity, behaves like a 'real' traffic light but just takes the colors green (*g* for short) and red (*r* for short). In fact, the yellow is no longer necessary as we assume that the involved cars (each one equipped with its own driver agent) will obey the directives. We have two lanes, one going north-south (*ns* for short) and the other one east-west (*ew* for short), crossing at the traffic light. If the traffic light is green in one direction it must be red in the other one, and vice versa. The traffic light is activated by a signal $active(t1)$, and never stopped unless there is a fault, detected by the module itself by means of a sensor. A fault is supposed to have occurred whenever $fault\_tl$ is true, i.e., it has been returned by the sensor.

$tln(t1).$ % *Traffic – Light Identifier*

$active(t1).$

$: - not\ active(t1).$ % *Sensor Check activation*

$: - lane(L), fault\_tl(t1, L, T).$ % *Sensor Check Possible Fault*

Each car, say here $c1$, $c2$, $c3$, $c4$ and $c5$[8], wants to go, but it is allowed to proceed only if it gets the green traffic light. Otherwise, it remains dummy. We assume that all cars behave in the same way. Each one issues a request of format $car(C)$, $want\_go(C, t1, L, T)$ where L is the lane, with possible values *ns* for north-south and *ew* for east-west; T is the time of the request. Requests by various cars may for example give rise to the addition of the following facts to the μASPSv's program.

%*INPUT* : *CARS*

$car(c1).$

$car(c2).$

$car(c3).$

$car(c4).$

$car(c5).$

%*INPUT* : *REQUESTS*

$want\_go(c1, t1, ns, 2).$

$want\_go(c2, t1, ns, 2).$

$want\_go(c3, t1, ew, 2).$

$want\_go(c4, t1, ns, 4).$

$want\_go(c5, t1, ew, 4).$

The following facts and rules define the lanes, and specify that this monitoring agent has a lookahead of five time instants: after that, it will have to be re-run.

$lane(ns).$

$lane(ew).$

$time(1..5).$

$next(Y, X) :- time(X), time(Y), Y = X + 1.$

[7] https://jade.tilab.com where references to several related publications can also be found

[8] The specification of which cars come and go in the traffic light surroundings can be within the module's inputs, and so the car list will be updated by the shell.

The rules below define the color that the traffic light takes (in a very standard way) as transitions from green to red and vice versa, where the initial color is green. In reality, such a monitoring agent can employ a much more sophisticated protocol such as for instance the Contract Net Protocol (CNP). If adopting CNP, the agent might grant priority to particular kinds of vehicles, e.g., police cars, ambulances, cars transporting a disabled person, etc. More generally, any policy to grant passage according to criteria could be implemented.

```
tl(r, TL, L1, T1) :-
        time(T), lane(L1), lane(L2), tln(TL), L1! = L2,
        next(T1, T), tl(g, TL, L1, T), tl(r, TL, L2, T).
tl(g, TL, L1, T1) :-
        time(T), lane(L1), lane(L2), tln(TL), L1! = L2,
        next(T1, T), tl(r, TL, L1, T), tl(g, TL, L2, T).
tl(g, TL, ns, 1) :- tln(TL).
tl(r, TL, ew, T) :-tln(TL), time(T), tl(g, TL, ns, T).
```

In our case the implemented protocol is fair, as cars that cannot go now because it is red on their lane will be deferred to the next time instant (by delaying their request), when the color will be green (output in format *go*(*Car, t1, Lane, Time*)).

```
go(C, TL, L, T) :-
        time(T), car(C), tln(TL), lane(L),
want_go(C, TL, L, T), tlpg, TL, L, T).
wait(C, TL, L, T) :-
        time(T), car(C), tln(TL), lane(L),
want_go(C, TL, L, T), tl(r, TL, L, T).
want_go(C, TL, L, T1):- car(C), tln(TL), lane(L),
wait(C, TL, L, T), next(T1, T).
:- time(T), car(C), tln(TL), lane(L),
   go(C, TL, L, T), tl(r, TL, L, T).
```

Clearly, this program can be 'cloned' (mutatis mutandis) to manage any number of traffic lights. For the reader's convenience, this program is standalone and can be run exactly as it is to check its results.

We now provide a definition of a car in DALI. Or rather, we define an agent capable to manage the situation where the car has to pass an intersection controlled by a μASPSv such as the one defined above. This agent will presumably be a component of an overall multi-agent system managing the many appliances included in most recent cars.

The agent will receive data about its present position from an infrastructure (which the road system may be equipped with at low cost), that will periodically broadcast the information, that will be received by cars. Then, the car will sense the presence of a crossing (with its associated traffic-light component) from a signal broadcasted up to a certain distance, that will communicate the identifier *tl* of that traffic light. The car will annotate the present position's external events as past events (a reaction that does nothing has exactly the purpose of annotating), where the most recent past event will be taken by default in consideration during subsequent operation, to extract position parameters. An external event signalling the presence of a crossing will determine a reaction where the agent issues a request to pass to *tl*. The request will be issued by sending a message whose performative will be the FIPA primitive *request*. The message will include the agent's name (available in the predefined special variable *Me*) and the present time, obtained by the system's primitive *time* (*T*). The predefined predicate *messageA* (...) is processed by the DALI communication architecture, which will fill the remaining unspecified parameters expected by the FIPA syntax with default values, and will actually send out a correct FIPA message. The agent becomes aware of being enabled to pass when, via the *enabled_passI* first rule (where posfix *I* indicates an internal event), that will be attempted automatically at a certain frequency, it will detect the arrival of a message containing the FIPA primitive *accept_proposal*. This primitive signals that the traffic light accepts the request, and thus grants the permission, in this case unconditionally: the list which occurs as second parameter (here empty) might in general indicate conditions to be fulfilled. So, success of the internal event via the first rule determines a reaction (second rule), which consists in the action *passA* that will be physically enacted by the car.

```
present_positionE(Road, Direction) :> true.
crossingE(TL) :> request_to_pass(TL).
request_to_pass(TL) :-time(T),
        present_positionP(_, Direction),
        messageA(TL,
            request(want_go(Me, TL, Direction, Time), Me)).
enabled_passI :-messageA(TL,
            accept_proposal(want_go(_, _, _), [], Me)).
enable_passI :> passA.
```

To make the two components interact it is not needed to import the whole FIPA protocol. For this simple case, the traffic light μASPSv's shell may extract the request from the input message, and "package" the permission to pass (when granted) into the required syntax before sending it back to the agent. An underlying (though minimal) middleware must be implemented, so that each component (many cars and traffic lights might in fact be present) can send/receive input/outputs to the others. Notice that, as said before, DALI has been integrated with the Docker technology, that may help to get this part "for free" or almost.

IX. Overall System's Semantics

The semantics of a single μASPSv is fully specified by: (i) the answer sets of the inner ASP program; (ii) the policy employed in its shell to select one single answer set; (iii) the set of queries that the shell possibly performs over the entire set of answer sets, whose meaning is formally specified in [26], [71]. We aim however to provide a semantics for the overall distributed system composed of heterogeneous microservices (where some of them can be agents), in order to provide a firm ground and a guideline for implementation.

To do so, we resort to Multi-Context Systems (MCSs), that are a well-established paradigm in Artificial Intelligence and Knowledge Representation, aimed to model information exchange among heterogeneous sources [72]–[74]. However, with some abuse of notation (and some slight loss of generality) we adapt and readjust the definitions to fit into our framework. To represent the heterogeneity of sources, each component in a Multi-context system, called 'context', is supposed to be based on its own logic, defined in a very general way [73]. In particular, a logic is defined by the following features.

- A set *F* of possible formulas (or *KB*-elements) under some signature.
- A set *KB* of knowledge bases built out of elements of *F*. in our framework, *KB* can also be a program in some programming language.
- A function *ACC*, where *ACC*(*kb, s*) means that *s* is an acceptable set of consequences of knowledge base $kb \in KB$, i.e., $s \subseteq Cn$, where *Cn* is the set of all possible consequences that can be drawn from *kb*. We assume here that *ACC* produces a unique set of consequences. In case of a program written in a non-logical programming language, such set can be the set of legal outputs given some input, that will be a subset of all possible outputs *Cn*; for logical

components, it will be (one of) the *kb* model(s). For instance, as we have seen the shell of a μASPSv will produce as consequences the elements occurring in the answer set selected according to some policy, along with query results.

A (Managed) multi-context system (MCS)

$$M = \{C_1, ..., C_r\}$$

is a set of $r = |M|$ contexts, each of them of the form $C_i = \langle c_i, L_i, kb_i, br_i \rangle$, where:

- c_i is the context name (unique for each context; if a specific name is omitted, index i can act as a name). In [75] a context's name can be a term called "context designator", denoting the kind of context (for instance, mycardiologist(c), customercare(c), helpdesk(h), etc.).
- L_i is a logic.
- $kb_i \in KB$ is a knowledge base.
- br_i is the set of bridge rules this context is equipped with.

Contexts in an MCS are meant to be heterogeneous distributed components, that exchange data. In fact, bridge rules are the key construct of MCSs, as it describes in a uniform way the communication/data exchange patterns between contexts. Each bridge rule $\rho \in br_i$ has the form

$$op_i(s) \leftarrow (c_1 : p_1), ..., (c_j : p_j) \quad (1)$$

where the left-hand side s is called the head, and the right-hand side is called the body, and the comma stands for conjunction. The meaning is that, each data item p_i is supposed to come from context c_i. Whenever all the $c_1, ..., c_j$ have delivered their data item to the destination context *ci*, the rule becomes applicable[9]. In case context designators are employed, prior to checking a bridge rule for applicability, such terms must have been substituted by actual context names from which to acquire the data. For μASPSv's, this task will be performed by the shell, that must then be endowed with a list of contexts of eachtype. When the rule is actually applied (where, in our approach, application is optional and must be explicitly triggered in the destination context's code), its conclusion s, once elaborated by operator op_i, will be added to c_i's knowledge base. Operator op_i can perform any elaboration on the "raw" input s, such as format conversion, filtering, elaboration via ontologies, etc. Its operation is specified via a management function mng_i, which is thus crucial for knowledge incorporation from external sources. For simplicity, here we assume mng_i to be monotonic (i.e., to produce from s one or more data items) . Therefore, we can extend the previous definition of a context as

$Ci = \langle c_i, L_i, kb_i, br_i, mng_i \rangle$.

Notice that, in [66], [76], [77], the MCS approach has been extended so that a context can possibly be a logic- based agent, and extensions to bridge-rules format have been introduced for data and ontologies exchange in this new setting.

A data state (or belief state) $\vec{S}$ of an MCS M is a tuple $\vec{S} = (S_1, ..., S_r)$ such that for $1 \leq i \leq r$, $S_i \subseteq Cn_i$. A data state can be seen as a view of the distributed system by an external "observer". $app(\vec{S})$ is the set composed of the head of those bridge rules which are applicable in $\vec{S}$. This means, in logical terms, that their body is true w.r.t. $\vec{S}$. In practical terms, we may say that a bridge rule ρ associated to context *ci* is applicable in $\vec{S}$ if all the data mentioned in the body of the bridge rule can be delivered to the destination context. This is the case whenever they are available in the contexts of origin, i.e., they occur in the present respective data state items in $\vec{S}$. In the original formulation of MCS, all applicable bridge rules are automatically applied, and their results, after the elaboration by the management function, are added to the destination context's knowledge base, that therefore grows via bridge-rule application. Starting from a certain specific data state, some bridge rules will be applicable and therefore they will be applied. This will enhance the knowledge base in some of the contexts, thus determining (in these contexts) a new set of acceptable consequence, and therefore a new overall data state. In the new state other bridge rules will be applicable, and so on, until a "stable" state, called Equilibrium, will be reached. Technically, $\vec{S}$ is an equilibrium for an mMCS M iff, for $1 \leq i \leq |M|$,

$$S_i = ACC_i(mng_i(app(\vec{S}), kb_i))$$

which states that each element of the equilibrium is an acceptable set of consequences after the application of every applicable bridge rule, whose result has been incorporated into the context's knowledge base via the management function.

In [75] it is proved that, in the kind of MCS that we have just described, an equilibrium will be reached in a finite number of steps. Notice however that this definition assumes the system to be isolated from any outside influence, and that an equilibrium, one reached, will last forever. Instead, in real systems there will be interactions with an external environment, and so equilibria may change over time. Moreover, each context is not necessarily a passive receiver of data sent by others.

To take these aspects into account, [75] proposes some extensions to the original formulation, among which the following, that are relevant in the present setting.

- It is noticed that contexts' knowledge bases can evolve in time, not only due to bridge-rule application. In fact, contexts receive sensor inputs (passively or in consequence to active observation), or can be affected by user's modification (e.g., a context may encompass a relational database that can be modified by users). So, each context c_i will have an associated Update Operator $\mathcal{U}_i$ (that can actually consist in a tuple of operators, each one performing a different kind of update). Updates and bridge rules both affect contexts' knowledge base over time. So (assuming an underlying discrete model of time) we will be able to consider, when necessary, $c_i[T]$ meaning context c_i at time T, with its knowledge base $kb_i[T]$; consequently we will have an evolution over time of contexts. Therefore, we will have a definition (not reported here) of Timed Equilibria. Notice only that a timed equilibrium can be reached at time $T+1$ only if the actual elapsed time between T and $T+1$ is sufficient for the system to "stabilize" by means of bridge-rules application on the updated knowledge bases.
- Mandatory bridge-rule application (as it is in the original MCS definition) constitutes a limitation: in fact, contexts would be forced to accept inputs unconditionally, and this may be often inappropriate. Consider for instance a context representing a family doctor: the context may accept non-urgent patient's requests for appointments or consultation only within a certain time windows. So, [75] introduces conditional bridge-rule application, formalized via a timed triggering function, tr_i, which specifies which applicable bridge rules are triggered (i.e., they are practically applicable) at time T. It does so either based on certain pre-defined conditions, or by performing some reasoning over the present knowledge base contents. Therefore, the implementation of $tr_i[T]$ my require an auxiliary piece of program, that in a μASPSv's shell will presumably be a logic program.

So, considering contexts which are μASPSv's, in order to fit in the vision of the overall system as an MCS, their shell must be empowered as follows.

[9] In the original formulation of bridge-rule syntax, there can be additional literals not $(c_1 : p_{j+1})$, ..., not $(c_j : p_n)$ in the body, meaning that in order for the bridge rule to be applicable, the $p_{j+1} ... p_n$ must be false in the relative contexts. We disregard this part, as non-logical components cannot use logical negation. There is no loss of generality however, as each of the $p_1, ..., p_j$ can state a negative fact.

- Include the bridge rules associated to a μASPSv, and the definition/implementation of the triggering function.
- Include a facility to resolve the context designators, so as to check for applicability a triggered bridge rule after substituting context designators occurring therein with actual contexts' names.
- Include the definition of the specific management function, so as to be able to apply it on bridge-rules' results.

In the case study of previous section, each traffic light should be equipped with a bridge rule that, by means of the instantiation of a suitable context designator (say, anycar(c)) collects the cars' requests. Symmetrically, cars should be equipped with a bridge rule to collect the permission to go by the traffic light (the nearest one, whose identifier should replace a context designator of the form, e.g., a_traffic_light(t)). The triggering function may allow cars to enable reception of traffic-light communications only when needed.

Context designators are therefore useful to write general bridge rules to be then customized to the particular situation at hand. They also allow to devise a system where components do not know or are aware of each other in advance, and where components can possibly join/leave the system at any time. A suitable middleware should be realized to allow component's shells to instantiate bridge rules. In our case study, that concerns an infrastructure for car traffic, both cars and traffic lights might for instance broadcast their name and geo-localization. In this way, cars might locate the traffic light of interest, and traffic lights might become aware of nearby cars that might send them a request.

X. Concluding Remarks

We have proposed a methodology for developing microservices in Answer Set Programming, by means of the creation of a particular kind of components, that can be activated/stopped, can receive external requests and can deliver answers. We have provided a definition of μASPSv's and explained how they might be implemented, and we have outlined a programming methodology. We have shown by means of a case study how such components can be defined, and how they might interact with other heterogeneous components, e.g., DALI logical agents.

We have also outlined a possible uniform semantics to specify an heterogeneous system in which μASPSv's could be situated, also in synergy with logic-based autonomous agents. This is an absolute novelty for microservices in general, as no attempt has ever been made to provide such a uniform model for an overall system. The proposed semantics can constitute the ground for principled implementations. Overall, this work can be considered as a creative combination of existing technologies, in view of entirely new application domains of answer set programming and logic programming in general.

Important application fields for μASPSv's are Cloud computing and IoT. We consider particularly important the various kinds of robotic applications and the underlying infrastructural aspects (as shown in the case study related to autonomous vehicles), and human-robot interaction. Promising future applications might concern personalised assistance in healthcare, where heterogeneous components might include: μASPSv's that manage sensors such as wearable devices to monitor the patient's conditions; personal assistant (possibly robotic) agents; and components representing the available appliances for patient's management and vital support, and knowledge sources that provide criteria for, e.g., evaluation of medical checks, dosage of drugs, and medical diagnosis.

Future work includes: develop a real implementation; refine the programming methodology; provide a user friendly graphical interface, and perform experiments in realistic environments. We plan to carry out an effective integration of μASPSv's and DALI multi-agent systems, and extend it to heterogeneous systems, possibly including also QuLog/Teleor and AgentSpeak agents. We will then perform experiments in the various domains where DALI is being applied, including robotics. We have in mind applications concerning cognitive robotic architectures, comprising hybrid multi-agent systems with object detectors as perception layer, and DALI-ASP as reasoning layer.

Acknowledgements

We thank the anonymous reviewers for their interest in the DALI language, and for insightful suggestions about modern ASP grounders/solvers. This work has been realized and funded over the years only by internal basic research activities of the authors, and by consulting work concerning the exploitation of DALI tools in real world applications, covered by no-disclosure agreements.

References

[1] G. Brewka, T. Eiter, M. T. (eds.), "Answer set programming: Special issue," AI Magazine, vol. 37, no. 3, 2016.

[2] M. Gelfond, V. Lifschitz, "The stable model semantics for logic programming.," in Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, August 15-19, 1988 (2 Volumes), vol. 88, 1988, pp. 1070–1080, MIT Press.

[3] M. Gelfond, V. Lifschitz, "Classical negation in logic programs and disjunctive databases," New generation computing, Springer, vol. 9, no. 3-4, pp. 365–385, 1991, doi: 10.1007/BF03037169.

[4] V. W. Marek, M. Truszczyński, "Stable models and an alternative logic programming paradigm," in The Logic Programming Paradigm, Springer, 1999, pp. 375–398, doi: 10.1007/978-3-642-60085-2_17.

[5] V. Lifschitz, "Answer set planning," in Logic Programming: The 1999 International Conference, Las Cruces, New Mexico, USA, November 29 - December 4, 1999, 1999, pp. 23–37, MIT Press.

[6] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, S. Tilkov, "Microservices: The journey so far and challenges ahead," IEEE Software, vol. 35, no. 3, pp. 24–35, 2018, doi: 10.1109/MS.2018.2141039.

[7] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, "Multi-shot ASP solving with clingo," Theory Pract. Log. Program., vol. 19, no. 1, pp. 27–82, 2019, doi: 10.1017/S1471068418000054.

[8] R. H. Bordini, L. Braubach, M. Dastani, A. E. F. Seghrouchni, J. J. Gómez-Sanz, J. Leite, G. M. P. O'Hare, A. Pokahr, A. Ricci, "A survey of programming languages and platforms for multi-agent systems," Informatica (Slovenia), vol. 30, no. 1, pp. 33–44, 2006.

[9] A. Garro, M. Mühlhäuser, A. Tundis, M. Baldoni, C. Baroglio, F. Bergenti, P. Torroni, "Intelligent agents: Multi-agent systems," in Encyclopedia of Bioinformatics and Computational Biology - Volume 1, S. Ranganathan, M. Gribskov, K. Nakai, C. Schönbach Eds., Elsevier, 2019, pp. 315-320, doi: 10.1016/b978-0-12-809633-8.20328-2.

[10] R. Calegari, G. Ciatto, V. Mascardi, A. Omicini, "Logic-based technologies for multi-agent systems: a systematic literature review," Auton. Agents Multi Agent Syst., vol. 35, no. 1, p. 1, 2021, doi: 10.1007/s10458-020-09478-3.

[11] S. Costantini, G. De Gasperis, "Dynamic goal decomposition and planning in MAS for highly changing environments," in Proceedings of the 33rd Italian Conference on Computational Logic, Bolzano, Italy, September 20-22, 2018, vol. 2214 of CEUR Workshop Proceedings, 2018, pp. 40–54, CEUR-WS.org.

[12] D. Ameller, X. Burgués, O. Collell, D. Costal, X. Franch, M. P. Papazoglou, "Development of service-oriented architectures using model-driven development: A mapping study," Information and Software Technology, vol. 62, pp. 42 – 66, 2015, doi: https://doi.org/10.1016/j.infsof.2015.02.006.

[13] C. Legner, R. Heutschi, "Soa adoption in practice-findings from early soa implementations," 2007, Association for Information Systems.

[14] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, L. Safina, "Microservices: yesterday, today, and tomorrow," in Present and ulterior software engineering, Springer, 2017, pp. 195–216, doi: 10.1007/978-3-319-67425-4_12.

[15] L. De Lauretis, "From monolithic architecture to microservices

architecture," in 2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), 2019, pp. 93–96, IEEE.

[16] P. Krivic, P. Skocir, M. Kusek, G. Jezic, "Microservices as agents in iot systems," in KES International Symposium on Agent and Multi-Agent Systems: Technologies and Applications, 2017, pp. 22–31, Springer.

[17] S. Costantini, A. Formisano, "Answer set programming with resources," Journal of Logic and Computation, vol. 20, no. 2, pp. 533–571, 2010, doi: 10.1093/logcom/exp071.

[18] A. Dyoub, S. Costantini, G. De Gasperis, "Answer set programming and agents," Knowledge Eng. Review, vol. 33, p. e19, 2018, doi: 10.1017/S0269888918000164.

[19] S. Costantini, "About epistemic negation and world views in epistemic logic programs," Theory Pract. Log. Program., vol. 19, no. 5-6, pp. 790–807, 2019.

[20] W. Faber, "An introduction to answer set programming and some of its extensions," in Reasoning Web. Declarative Artificial Intelligence - 16th International Summer School 2020, Oslo, Norway, June 24-26, 2020, Tutorial Lectures, vol. 12258 of Lecture Notes in Computer Science, 2020, pp. 149–185, Springer.

[21] V. Lifschitz, "Twelve definitions of a stable model," in Proc. of the 24th Intl. Conf. on Logic Programming, vol. 5366 of LNCS, 2008, pp. 37–51, Springer.

[22] A. Van Gelder, K. A. Ross, J. S. Schlipf, "The well-founded semantics for general logic programs," Journal of the ACM, vol. 38, no. 3, pp. 620–650, 1991, doi: 10.1145/116825.116838.

[23] S. Costantini, A. Formisano, "Negation as a resource: A novel view on answer set semantics," in Logic Programming and Nonmonotonic Reasoning, 12th Intl. Conf., LPNMR 2013, vol. 8148 of Lecture Notes in Computer Science, 2013, pp. 257–263, Springer.

[24] K. Marple, G. Gupta, "Dynamic consistency checking in goal-directed answer set programming," TPLP, vol. 14, no. 4-5, pp. 415–427, 2014, doi: 10.1017/S1471068414000118.

[25] G. Gupta, E. Salazar, K. Marple, Z. Chen, F. Shakerin, "A case for query-driven predicate answer set programming," in ARCADE 2017, 1st International Workshop on Automated Reasoning: Challenges, Applications, Directions, Exemplary Achievements, Gothenburg, Sweden, 6th August 2017, vol. 51 of EPiC Series in Computing, 2017, pp. 64–68, EasyChair.

[26] S. Costantini, F. A. Lisi, R. Olivieri, "Digforasp: A european cooperation network for logic-based AI in digital forensics," in Proceedings of the 34th Italian Conference on Computational Logic, Trieste, Italy, June 19-21, 2019, vol. 2396 of CEUR Workshop Proceedings, 2019, pp. 138–146, CEUR-WS.org.

[27] Y. Shen, T. Eiter, "Evaluating epistemic negation in answer set programming," Artificial Intelligence, vol. 237, pp. 115–135, 2016, doi: 10.1016/j.artint.2016.04.004.

[28] T. Eiter, G. Gottlob, H. Veith, "Modular logic programming and generalized quantifiers," in Logic Programming and Nonmonotonic Reasoning, Springer, 1997, pp. 289–308, doi: 10.1007/3-540-63255-7_22.

[29] G. Ianni, G. Ielpa, A. Pietramala, M. C. Santoro, F. Calimeri, "Enhancing answer set programming with templates.," in 10th International Workshop on Non-Monotonic Reasoning (NMR 2004), Whistler, Canada, June 6-8, 2004, Proceedings, 2004, pp. 233–239.

[30] L. Tari, C. Baral, S. Anwar, "A language for modular answer set programming: Application to ACC tournament scheduling.," in Answer Set Programming, Advances in Theory and Implementation, Proceedings of the 3rd Intl. ASP'05 Workshop, Bath, UK, September 27-29, 2005, vol. 142 of CEUR Workshop Proceedings, 2005, CEUR-WS.org.

[31] S. Costantini, "On the existence of stable models of non-stratified logic programs," Theory and Practice of Logic Programming, vol. 6, no. 1-2, 2006, doi: 10.1017/S1471068405002589.

[32] V. Lifschitz, H. Turner, "Splitting a logic program.," in Logic Programming, Proceedings of the Eleventh International Conference on Logic Programming, Santa Marherita Ligure, Italy, June 13-18, 1994, vol. 94, 1994, pp. 23–37, MIT Press.

[33] T. Janhunen, E. Oikarinen, H. Tompits, S. Woltran, "Modularity aspects of disjunctive stable models," Journal of Artificial Intelligence Research, pp. 813–857, 2009, doi: 10.1613/jair.2810.

[34] E. Oikarinen, Modularity in answer set programs. PhD dissertation, Helsinki University of Technology, Finland, 2008.

[35] H. Gaifman, E. Shapiro, "Fully abstract compositional semantics for logic programs," in Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 1989, pp. 134–142, ACM.

[36] M. Dao-Tran, T. Eiter, M. Fink, T. Krennwallner, "Modular nonmonotonic logic programming revisited," in Logic Programming, Springer, 2009, pp. 145–159, doi: 10.1007/978-3-642-02846-5_16.

[37] C. Baral, J. Dzifcak, H. Takahashi, "Macros, macro calls and use of ensembles in modular answer set programming," in Logic Programming, Springer, 2006, pp. 376–390, doi: 10.1007/11799573_28.

[38] M. Balduccini, "Modules and signature declarations for a-prolog: Progress report," Workshop on Software Engineering for Answer Set Programming (SEA'07), pp. 41–55, 2007.

[39] W. Faber, S. Woltran, "Manifold answer-set programs for meta-reasoning," in Logic Programming and Non-monotonic Reasoning, Springer, 2009, pp. 115–128, doi: 10.1007/978-3-642-04238-6_12.

[40] Y. Lierler, M. Truszczyński, "Modular answer set solving," Late-Breaking Developments in the Field of Artificial Intelligence, Bellevue, Washington, USA, July 14-18, AAAI, vol. WS-13-17, 2013.

[41] S. Costantini, "Answer set modules for logical agents," in Datalog Reloaded - First Intl. Workshop, Datalog 2010, Revised Selected Papers, O. de Moor, G. Gottlob, T. Furche, A. J. Sellers Eds., no. 6702 in Lecture Notes in Computer Science, Springer, 2011, pp. 37–58, doi: 10.1007/978-3-642-24206-9_3.

[42] S. Costantini, A. Tocchio, "A logic programming language for multi-agent systems," in Logics in Artificial Intelligence, Proc. of the 8th Europ. Conf.,JELIA 2002, LNAI 2424, 2002, pp. 1–13, Springer-Verlag, Berlin.

[43] S. Costantini, A. Tocchio, "The DALI logic programming agent-oriented language," in Logics in Artificial Intelligence, 9th European Conference, JELIA 2004, Proceedings, vol. 3229 of Lecture Notes in Computer Science, 2004, pp. 685–688, Springer.

[44] S. Costantini, A. Tocchio, A. Verticchio, "Communication and trust in the DALI logic programming agentoriented language," Intelligenza Artificiale, vol. 2, no. 1, pp. 39–46, 2005. Journal of the Italian Association AI*IA.

[45] S. Costantini, A. Tocchio, "About declarative semantics of logic-based agent languages," in Declarative Agent Languages and Technologies III, Third International Workshop, DALT 2005, Selected and Revised Papers, vol. 3904 of Lecture Notes in Computer Science, M. Baldoni, U. Endriss, A. Omicini, P. Torroni Eds., Springer, 2005, pp. 106–123, doi: 10.1007/11691792_7.

[46] S. Costantini, P. Dell'Acqua, L. M. Pereira, "A multilayer framework for evolving and learning agents," in Proceedings of Metareasoning: Thinking about thinking workshop at AAAI 2008, Chicago, USA, 2008.

[47] S. Costantini, A. Tocchio, "DALI: An architecture for intelligent logical agents," in Proceedings of the Int. Workshop on Architectures for Intelligent Theory-Based Agents (AITA08), AAAI Spring Symposium Series, 2008.

[48] S. Costantini, "Self-checking logical agents," in Proceedings of the Eighth Latin American Workshop on Logic, Languages, Algorithms and New Methods of Reasoning LA-NMR 2012, vol. 911 of CEUR Workshop Proceedings, 2012, pp. 3–30, CEUR-WS.org. Invited Paper, Extended Abstract in Proceedings of AAMAS 2013, 12th Intl. Conf. on Autonomous Agents and Multi-Agent Systems.

[49] S. Costantini, A. D'Andrea, G. De Gasperis, N. Florio, Tocchio, "DALI logical agents into play," in Proceedings of the AI*IA Workshop "Popularize Artificial Intelligence" (PAI-2012), 2012.

[50] S. Costantini, G. D. Gasperis, "Complex reactivity with preferences in rule-based agents," in Rules on the Web: Research and Applications - 6th International Symposium, RuleML 2012, Montpellier, France, August 27-29, 2012. Proceedings, vol. 7438 of Lecture Notes in Computer Science, 2012, pp. 167–181, Springer.

[51] S. Costantini, G. De Gasperis, G. Nazzicone, "Exploration of unknown territory via DALI agents and ASP modules," in Distributed Computing and Artificial Intelligence, 12th International Conference, DCAI 2015. Proceedings, vol. 373 of Advances in Intelligent Systems and Computing, 2015, pp. 285–292, Springer.

[52] S. Costantini, G. De Gasperis, G. Nazzicone, "DALI for cognitive robotics: Principles and prototype implementation," in Practical Aspects of Declarative Languages - 19th International Symposium, PADL 2017,

Proceedings, vol. 10137 of Lecture Notes in Computer Science, 2017, pp. 152–162, Springer.

[53] R. A. Brooks, "Intelligence without reason," in Proceedings of the 12th International Joint Conference on Artificial Intelligence. Sydney, Australia, August 24-30, 1991, 1991, pp. 569–595, Morgan Kaufmann.

[54] R. A. Brooks, "Intelligence without representation," Artif. Intell., vol. 47, no. 1-3, pp. 139–159, 1991.

[55] A. S. Rao, M. Georgeff, "Modeling rational agents within a BDI-architecture," in Proc. of the Second Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'91), 1991, pp. 473–484, Morgan Kaufmann.

[56] A. S. Rao, "AgentSpeak (L): BDI agents speak out in a logical computable language," in Agents Breaking Away, 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, Eindhoven, The Netherlands, January 22-25, 1996, Proceedings, Springer, 1996, pp. 42–55, doi: 10.1007/BFb0031845.

[57] S. Costantini, P. Dell'Acqua, G. A. Lanzarone, "Reflective agents in metalogic programming," in Meta-Programming in Logic, 3rd International Workshop, META-92, Proceedings, vol. 649 of Lecture Notes in Computer Science, 1992, pp. 135–147, Springer.

[58] R. A. Kowalski, F. Sadri, "Towards a unified agent architecture that combines rationality with reactivity," in Logic in Databases, International Workshop LID'96, Proceedings, vol. 1154 of Lecture Notes in Computer Science, 1996, pp. 137–149, Springer.

[59] P. Dell'Acqua, F. Sadri, F. Toni, "Combining introspection and communication with rationality and reactivity in agents," in Logics in Artificial Intelligence, European Workshop, JELIA '98, Dagstuhl, Germany, October 12-15, 1998, Proceedings, vol. 1489 of Lecture Notes in Computer Science, 1998, pp. 17–32, Springer.

[60] G. De Gasperis, S. Costantini, G. Nazzicone, "Dali multi agent systems framework, doi 10.5281/zenodo.11042." DALI GitHub Software Repository, July 2014. Accessed: December 2020, DALI: http:// github.com/AAAI-DISIM-UnivAQ/DALI.

[61] S. Costantini, G. De Gasperis, V. Pitoni, A. Salutari, "DALI: A multi agent system framework for the web, cognitive robotic and complex event processing," in Joint Proceedings of the 18th Italian Conference on Theoretical Computer Science and the 32nd Italian Conference on Computational Logic co-located with the 2017 IEEE International Workshop on Measurements and Networking (2017 IEEE M&N), vol. 1949 of CEUR Workshop Proceedings, 2017, pp. 286–300, CEUR-WS.org.

[62] S. Costantini, "Defining and maintaining agent's experience in logical agents," in Proc. of the Seventh Latin American Workshop on Non-Monotonic Reasoning LANMR 2011, vol. 804, 2011, pp. 151–165. (also in the Informal Proc. of the LPMAS "Logic Programming for Multi-Agent Systems" Workshop at ICLP 2011).

[63] M. Carlsson, P. Mildner, "Sicstus prolog–the first 25 years," arXiv preprint arXiv:1011.5640, 2010.

[64] S. Costantini, G. De Gasperis, G. Nazzicone, "DALI for cognitive robotics: Principles and prototype implementation," in Practical Aspects of Declarative Languages - 19th International Symposium, Proceedings, vol. 10137 of Lecture Notes in Computer Science, 2017, pp. 152–162, Springer.

[65] S. Costantini, G. De Gasperis, V. Pitoni, A. Salutari, "Dali: A multi agent system framework for the web, cognitive robotic and complex event processing," in Proceedings of the 32nd Italian Conference on Computational Logic, vol. 1949 of CEUR Workshop Proceedings, 2017, pp. 286–300, CEUR-WS.org. http://ceurws.org/Vol-1949/CILCpaper05.pdf.

[66] S. Costantini, "ACE: a flexible environment for complex event processing in logical agents," in Engineering Multi-Agent Systems, Third International Workshop, EMAS 2015, Revised Selected Papers, vol. 9318 of Lecture Notes in Computer Science, 2015, pp. 70–91, Springer.

[67] F. Aielli, D. Ancona, P. Caianiello, S. Costantini, G. De Gasperis, A. D. Marco, A. Ferrando, V. Mascardi, "FRIENDLY & KIND with your health: Human-friendly knowledge-intensive dynamic systems for the e-health domain," in Highlights of Practical Applications of Scalable Multi-Agent Systems. The PAAMS Collection International Workshops of PAAMS 2016, Proceedings, vol. 616 of Communications in Computer and Information Science, 2016, pp. 15–26, Springer.

[68] S. Costantini, G. De Gasperis, P. Migliarini, "Multiagent system engineering for emphatic human-robot interaction," in 2019 IEEE Second International Conference on Artificial Intelligence and Knowledge Engineering (AIKE), 2019, pp. 36–42, IEEE.

[69] K. L. Clark, P. J. Robinson, "Concurrent task programming of robotic agents in teleor.," in RuleML+ RR (Supplement), 2017.

[70] M. Gebser, T. Schaub, S. Thiele, "Gringo: A new grounder for answer set programming," in International Conference on Logic Programming and Non- monotonic Reasoning, 2007, pp. 266–271, Springer.

[71] S. Costantini, A. Formisano, "Query answering in resource-based answer set semantics," Theory and Practice of Logic Programming, vol. 16, no. 5-6, pp. 619–635, 2016, doi: 10.1017/S1471068416000478.

[72] G. Brewka, T. Eiter, "Equilibria in heterogeneous nonmonotonic multi-context systems," in Proc. of the 22nd AAAI Conf. on Artificial Intelligence, 2007, pp. 385–390, AAAI Press.

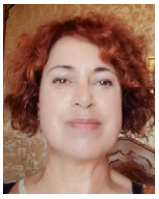
[73] G. Brewka, T. Eiter, M. Fink, "Nonmonotonic multicontext systems: A flexible approach for integrating heterogeneous knowledge sources," in Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning - Essays Dedicated to Michael Gelfond on the Occasion of His 65th Birthday, vol. 6565 of Lecture Notes in Computer Science, 2011, pp. 233–258, Springer.

[74] G. Brewka, S. Ellmauthaler, J. Pührer, "Multi-context systems for reactive reasoning in dynamic environments," in ECAI 2014, Proc. of the 21st European Conf. on Artificial Intelligence, 2014, pp. 159–164, IJ- CAI/AAAI.

[75] P. Cabalar, S. Costantini, G. De Gasperis, A. Formisano, "Multi-context systems in dynamic environments," Ann. Math. Artif. Intell., vol. 86, no. 1-3, pp. 87–120, 2019, doi: 10.1007/s10472-019-09622-0.

[76] S. Costantini, G. De Gasperis, "Exchanging data and ontological definitions in multi-agent-contexts systems," in Proceedings of the RuleML 2015 Challenge, the Special Track on Rule-based Recommender Systems for the Web of Data, the Special Industry Track and the RuleML 2015 Doctoral Consortium hosted by the 9th International Web Rule Symposium (RuleML 2015), Berlin, Germany, August 2-5, 2015, vol. 1417 of CEUR Workshop Proceedings, 2015, CEUR-WS.org.

[77] S. Costantini, "Knowledge acquisition via non-monotonic reasoning in distributed heterogeneous environments," in 13th Int. Conf. on Logic Programming and Nonmonotonic Reasoning LPNMR 2013. Proc., vol. 9345 of Lecture Notes in Computer Science, 2015, pp. 228–241, Springer.

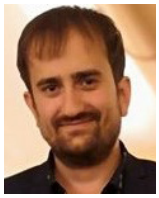


Stefania Costantini

Stefania Costantini (Female, married, two children, Daniele and Alice, born in 1993 and 1996) graduated with honors at the University of Pisa, Italy, in 1983. She worked at Italtel SIT (a telecommunications company) in Milan, Italy. In 1986 she became research associate and then (in 1990) Assistant Professor of Computer Science at the University of Milan, Italy. In 2001 she became Associate Professor at the University of L'Aquila (Italy) where, since 2005, she is Full Professor in Computer Science at the Department of Computer Science and Engineering and Mathematics (DISIM). She is the Head of the research group AAAI@AQ (Autonomous Agents and Artificial Intelligence at the University of L'Aquila). She has more than 150 publications. Her research interests are in (theory and practice of) Artificial Intelligence and Computational Logic, including Intelligent Software Agents and Multi-Agent Systems, Answer Set Programming, Non-Monotonic Reasoning, Knowledge Representation, Cognitive Robotics. She invented, defined and coordinated the first implementation of the DALI agent- oriented logic programming language. She served in the Program Committee of the main Conferences of her fields of interest, and she is a member of the Editorial Board of the journal Theory and Practice of Logic Programming (Cambridge). She is currently the President of the Italian Association of Computational Logic (GULP), and Member of the Board of the Italian Association for Artificial Intelligence (AIxIA).

Giovanni De Gasepris

Master Degree in Electronics Enginering cum Laude (1991) and Ph.D. in Electronics Engineering (1995) from the University of L'Aquila, Italy. He has been a Post Doctoral Fellow at University of Texas, M.D. Anderson Cancer Center in Houston, TX, USA (1995-1998). Technologist and freelance computer consultant in Italy (1998-2006). Contract lecturer of Computer Engineering at University of L'Aquila, Italy (2000-2006). Since 2007 he is an Assistant Professor at Department of Information Engineering and Computer Science and Mathematics, at the University of L'Aquila, Italy. He is teaching since 2015 the course of "Intelligent Systems and Robotics Laboratory" at the Master Degree in Computer Science, and the course "Virtual Reality and Archeomatics" at the Master Degree in Cultural Heritage at the Human Science Department. His current research interests are: Cognitive Robotics, Internet of Things, Natural Language Processing, Emotion Recognition, Virtual Reality. He is Core Developer and Coordinator for the development of many open source software packages through the research group github organization (https://github.com/AAAI-DISIM-UNIVAQ), among which the DALI logic multi agent system framework and its ASP_DALI extension. He is member of the Italian Association for Artificial Intelligence (AIxIA) since 2009.



Lorenzo De Lauretis

born in L'Aquila in 1991, graduated in Computer Science at Università degli studi dell'Aquila in 2016. He got his Master Degree cum Laude in Computer Science in 2018 at Università degli studi dell'Aquila. He became a PhD student in October 2018.