Defining Routines in DaaS: Visual Logic Programming for Mesh Network Policies - Preliminary Report

Lorenzo Grillo[1], Pierpaolo Sestito[2]

University of Calabria, Department of Mathematics and Computer Science

Abstract

Configuring event-driven behaviors in heterogeneous Internet of Things (IoT) mesh networks typically demands low-level programming expertise, creating a significant knowledge-acquisition bottleneck for end-users. This paper addresses the problem within the Device as a Service (DaaS) paradigm, an overlay mesh architecture in which devices communicating over diverse physical media, such as Ethernet, Wi-Fi, and Bluetooth Low Energy, are abstracted into a single logical network. To make this complex infrastructure accessible to non-expert users, we introduce a Visual Logic Programming interface based on a drag-and-drop paradigm: users assemble network policies from Trigger, Condition, and Action building blocks, which are automatically compiled into Answer Set Programming (ASP) rules. The choice of ASP as the target formalism is deliberate: its support for integrity constraints and default reasoning allows the system to detect conflicting policies, a capability that simple procedural if-then engines lack. The compiled rules are intended to be evaluated in real time by a Clingo-based routine engine on DaaS nodes with sufficient computational resources, enabling routing logic to be updated without interrupting device operations.

Keywords

Answer Set Programming, Mesh Network, Logic Programming, Visual Logic Programming,

1. Introduction

The current networking landscape is populated with multiple heterogeneous technologies, ranging from energy-efficient short-range systems [1, 2] to secure long-distance interconnections [3]. Integrating these technologies into a unified environment requires not only robust communication abstractions, but also accessible tools for configuring their behavior. This configuration task represents a classic instance of the *knowledge acquisition bottleneck* [4]: translating domain knowledge into formal, executable specifications typically demands low-level programming expertise that most network operators and IoT practitioners do not possess.

To address the communication layer, we build on the Device as a Service (DaaS) paradigm, which deploys an *overlay mesh network*, effectively creating a 'network within the network'. While DaaS autonomously manages core operations such as heterogeneous channel selection (i.e., dynamically switching between different communication technologies) and distributed time synchronization, *configuring custom event-driven behaviors* on DaaS nodes (i.e., devices) remains inaccessible to non-expert users. This paper addresses that gap with three contributions:

1. A **Visual Logic Programming interface** based on a drag-and-drop paradigm, allowing users to define network policies from Trigger, Condition, and Action building blocks without writing code;
2. A **formal compilation pipeline** τ that translates visual routines into Answer Set Programming (ASP) rules, evaluated in real time by a Clingo-based routine engine within the node's control plane on DaaS nodes with sufficient computational resources[1];

Joint Proceedings of the Workshops and Doctoral Consortium of the 42nd International Conference on Logic Programming (ICLP 2026), co-located with the Federated Logic Conference (FLoC 2026), July 18–19 and 24–25, 2026, Lisbon, Portugal

lorenzo.grillo@unical.it (L. Grillo); pierpaolo.sestito@unical.it (P. Sestito)

0009-0002-6099-3895 (L. Grillo); 0009-0004-7685-6152 (P. Sestito)

© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

[1]The *control plane* of a DaaS node denotes the local control layer that exposes the node's operational capabilities to the routine system. These capabilities include the available communication interfaces, the supported actions, and the runtime facts that can be observed by routines

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

published 2026-11-03

CEUR-WS.org/Vol-4311/EULP_regular_01.pdf

3. A **conflict detection and resolution** mechanism that leverages ASP integrity constraints and default reasoning to safely handle contradictory policies, a capability that simple if-then engines cannot provide.

The remainder of this paper is structured as follows: Section 2 provides background on ASP and overlay *mesh networks*. Section 3 introduces the DaaS architecture. Section 4 presents our core contribution. Section 6 concludes and outlines future work.

2. Background

2.1. Logic Programming and Answer Set Programming

Logic Programming (LP) [5] is a declarative paradigm that shifts the focus from detailing *how* a computation proceeds to specifying *what* the problem constraints are. Within this domain, Answer Set Programming (ASP; [6, 7]) has emerged as a powerful framework for knowledge representation and complex reasoning.

Formally, let $\mathcal{LP}$ denote the set of logic programs over predicates in $\mathcal{P}$. We specifically consider normal logic programs under the stable model semantics. A *normal rule* is an expression *"head if body"* of the form

$$a \leftarrow b_1, \ldots, b_m,\ not\ c_1, \ldots, not\ c_n,$$

where a, b_i, and c_j are atoms. An *integrity constraint* is a rule with an empty head, written as

$$\bot \leftarrow b_1, \ldots, b_m,\ not\ c_1, \ldots, not\ c_n,$$

which serves to discard any model that satisfies its body. A *normal program* Π is a finite set of normal rules and integrity constraints over predicates in $\mathcal{P}$. Programs are interpreted under their standard ground instantiation with respect to the fixed finite domain of constants. Given a ground program Π and a set of ground atoms I, the *Gelfond–Lifschitz reduct* Π^I is obtained by removing all rules whose body contains a negative literal $not\ c$ with $c \in I$, and by deleting all remaining negative literals from the bodies of the remaining rules. A set of atoms I is a *stable model* (or *answer set*) of Π if I is the least model (i.e., the smallest set of atoms satisfying the positive reduct) of the reduct Π^I.

For the DaaS architecture, leveraging ASP and modern solvers like Clingo provides two critical advantages. First, its declarative nature aligns naturally with the mental model of end-users, allowing them to intuitively define event-driven [8] network behaviors without procedural coding. Second, the strict separation between the logic (the rules) and the execution control (the solver) enables the dynamic hot-swapping of policies (i.e., updating a device's ruleset in real time without interrupting system operations). Consequently, device behaviours can be modified on the fly.

2.2. Node Routines

A *node routine* is a set of rules that governs the behavior of a device. These rules are defined using a constrained vocabulary, which specifies the available actions A, triggers T, and their permitted data types. For instance, a rule contained within a routine might be formulated as follows:

```
drop_packet :- packet(_, _, Size, _), Size > 1024.
```

where `packet(_, _, Size, _)` is the runtime fact associated with an incoming packet, `Size > 1024` is the condition, and `drop_packet` is the action produced when the condition holds.

2.3. Overlay Mesh Networks

An *overlay mesh network* combines a resilient, distributed mesh topology [9] with a logical communication layer built over an existing physical *underlay* [10]. This architecture establishes a virtual mesh

across a potentially heterogeneous physical infrastructure, connecting nodes via logical links that may not coincide with the physical ones [11, 12].

This approach is particularly relevant for IoT deployments, where devices may communicate through different underlay technologies: wired standards such as Ethernet (IEEE 802.3), wireless protocols such as Wi-Fi (IEEE 802.11), or energy-constrained short-range media such as Bluetooth Low Energy (BLE), each with distinct bandwidth, latency, and power characteristics [13, 14].

3. DaaS Architecture

DaaS provides the networking substrate on which the routines introduced in this paper are executed. In this work, DaaS is considered as an overlay mesh architecture for heterogeneous IoT networks. The overlay can be viewed as a graph whose vertices are DaaS nodes and whose edges are logical communication links established over the available physical or wireless underlay technologies.

Operationally, mesh communication is based on local forwarding decisions. A node is not required to have a direct physical connection to the final destination of a message. Instead, a packet may traverse a multi-hop path: at each hop, the current node selects one of its logical neighbors as the next hop and forwards the packet through one of its available communication interfaces. The end-to-end path therefore emerges from the composition of node-local decisions.

This view is central to the routines considered in this paper. A routine does not program the whole network globally; it controls the local behavior of a DaaS node. Examples include forwarding a packet to a selected neighbor, dropping a packet, disabling a communication link, or changing a local queue parameter. These node-local decisions are the target of the visual Trigger–Condition–Action language introduced in Section 4.

The *control plane* of a DaaS node denotes the local control layer that exposes the node's operational capabilities to the routine system. These capabilities include the communication interfaces actually available on the node, the actions that the node can execute, and the runtime facts that can be observed by routines. For instance, if a node does not provide a BLE interface, BLE-related actions are not exposed in the visual programming interface and cannot be used in compiled routines. The control plane is distinct from the data plane, which performs the actual packet transmission and forwarding.

3.1. Heterogeneous Underlay Routing

DaaS supports heterogeneous communication through *Protocol Channeling*. Each communication technology, such as Ethernet, Wi-Fi, or Bluetooth Low Energy, is managed by a dedicated driver that exposes a common communication abstraction to the overlay. As a result, the routine layer can reason over logical links and actions without depending on protocol-specific implementation details.

Figure 1 illustrates this mechanism. Different underlay links may rely on different technologies, while the DaaS overlay selects a logical multi-hop path from source to destination. A packet may therefore be forwarded through different communication media at different hops while still being handled as part of a single logical DaaS communication path.

For the purposes of this paper, the relevant point is that forwarding decisions remain local to each node. When a packet is processed, the node evaluates its current routine set against local runtime facts, such as packet attributes, link availability, node identifier, or synchronized time. The resulting action atoms are then mapped to concrete DaaS operations by the node runtime.

3.2. Synchronized Runtime Facts

DaaS also provides a synchronized logical notion of time through dATS [15]. The details of the synchronization protocol are outside the scope of this paper. Here, dATS is relevant only because synchronized time can be exposed to routines as a runtime fact.

In particular, the node runtime provides facts of the form `time(T)`, where `T` denotes the current logical network time. Therefore, a visual condition such as "before 08:00" can be compiled into the

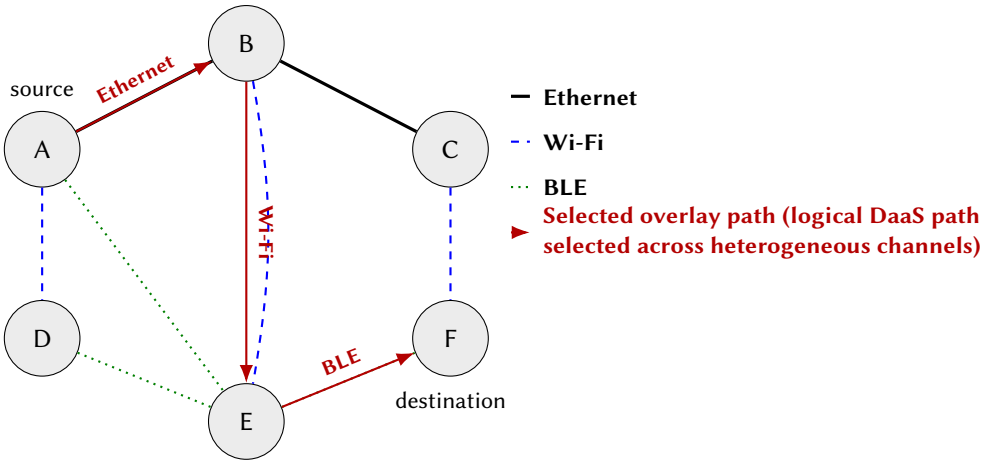


Figure 1: *Protocol Channeling* at the DaaS overlay level. Nodes expose different communication interfaces, while the overlay selects a logical multi-hop path across heterogeneous underlay channels such as Ethernet, Wi-Fi, and BLE.

ASP body `time(T), T < 8`. This allows time-dependent routines to be evaluated against a coherent network time rather than against unrelated local hardware clocks.

4. Visual Logic Programming for DaaS Routines

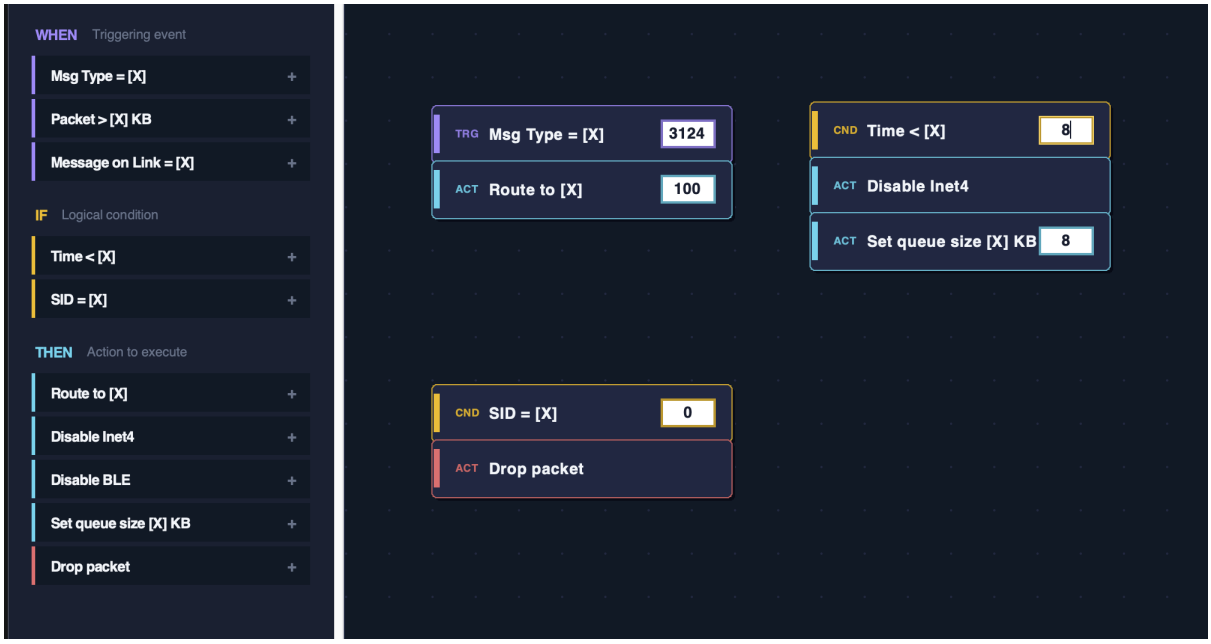


Figure 2: The DaaS Routine Visual Programming interface. The left-hand side contains the available building blocks, organized as triggers, conditions, and actions, which can be dragged and dropped to define a behaviour/routine. The depicted routines illustrate three scenarios: (top-left) routing packets of type 3124 to remote node 100; (bottom-left) dropping packets with invalid SID = 0; and (right) activating low-power mode before 08:00.

The visual interface shown in Figure 2 employs a drag-and-drop paradigm where users assemble policies from three building blocks: *Triggers* ($\mathcal{T}$), *Conditions* ($\mathcal{C}$), and *Actions* ($\mathcal{A}$). To provide a concrete understanding of these elements within the current DaaS prototype, we briefly enumerate their implementations. The set of triggers $\mathcal{T}$ includes system-level events related to incoming data, such

as `msg_type`, `pkt_size`, and `link` (the type of communication used, i.e. BLE). The set of conditions $\mathcal{C}$ comprises contextual prerequisites and logical checks, such as `time` constraints. Finally, the set of actions $\mathcal{A}$ contains consequent execution commands such as `route`, `drop_packet`, `disable_link` (e.g. disabling BLE or IPv4), and `set_queue_size` for the buffer of the incoming packet. Formally, a routine is a tuple $R = (t, C, A)$ where $t \in \mathcal{T}$ represents the trigger, $C \subseteq \mathcal{C}$ is a set of conditions, and $A \subseteq \mathcal{A}$ is a non-empty set of actions.

4.1. Compilation to ASP

Every routine is automatically compiled into ASP rules via a translation function τ. Each trigger is mapped to a runtime fact pattern, each condition becomes part of the rule body, and each action becomes a rule head. Formally, for a routine $R = (t, C, A)$, where $C = \{c_1, \ldots, c_n\}$, the translation produces one ASP rule for each action $a_i \in A$:

$$\tau(R) = \left\{ \; a_i \leftarrow \mathit{trigger}(t, v), c_1(V_1), V_1 \; op_1 \; \theta_1, \ldots, c_n(V_n), V_n \; op_n \; \theta_n \;\; \middle| \;\; a_i \in A \; \right\}.$$

Here, v denotes the runtime value associated with the triggering event, such as a message type, packet size, source identifier, or communication link. Each c_k denotes the predicate associated with the k-th visual condition, while V_k is the variable bound by that predicate at runtime. The operator op_k is the comparison operator selected in the visual block, such as $=, \neq, <, \leq, >, \geq$, and θ_k is the constant value or threshold provided by the user. For example, the visual condition "before 08:00" is translated into the ASP body literals `time(T), T < 8`, where c_k is `time`, V_k is `T`, op_k is $<$, and θ_k is 8.

4.2. Routines Loading

Routines are loaded through the DaaS API by providing the path of a compiled routine file. The runtime safely updates the active set of policies during execution, without interrupting device operations.

Each compiled routine can be represented as a JSON object containing a rule name and the corresponding generated ASP rule. Listing 1 shows the compiled representation of the time-dependent rule used in the conflict scenario.

```
{
 "rule_name": "Disable Wi-Fi before 08:00",
 "rule": "disable_link(wifi) :- time(T), T < 8."
}
```

Listing 1: JSON representation of a compiled routine

4.3. Conflict Detection and Resolution

A key advantage of targeting ASP, rather than procedural if-then rules, is the ability to detect and resolve policy conflicts via integrity constraints and default reasoning. In the current preliminary design, consistency is checked during routine evaluation against the current runtime facts; load-time validation of all possible conflicts is left for future work.

4.4. Conflicting Policies

As a preliminary design hypothesis, the current DaaS model can naturally support conflict detection and resolution through ASP integrity constraints and default reasoning. Consider two coexisting routines: (A) "when packet size exceeds 64 KB, route to node 100 via Wi-Fi" and (B) "before 08:00, disable Wi-Fi to save power." At 06:00, a large packet arrives and both routines may generate potentially contradictory actions.

Listing 2 reports the intermediate ASP encoding of the conflicting policies. This encoding is automatically generated from the user's visual design 2, prior to being serialized into the JSON structure shown in Listing 1.

```
1 route(3124, 100, wifi) :- packet(_, 3124, Size, _), Size > 64, not
     disable_link(wifi).
2 disable_link(wifi) :- time(T), T < 8.
3 :- route(_, _, Link), disable_link(Link).
4 route(3124, 100, ble) :- packet(_, 3124, Size, _), Size > 64, disable_link(wifi).
```

Listing 2: Conflict detection and fallback encoding

In this preliminary formulation, the default literal `not disable_link(wifi)` prevents the Wi-Fi route from being selected when Wi-Fi is disabled. The integrity constraint prevents answer sets in which packets are routed through disabled technologies, while the fallback rule, defined by the user, illustrates how an alternative channel can be selected declaratively when available.

On the node side, the DaaS library handles both the injection of runtime facts and the interpretation of solver results. Triggers are activated reactively: each event, such as a packet arrival or a link change, is translated into corresponding ASP facts, while condition-related facts, such as `time/1` and `sid/1`, are periodically refreshed by the node control loop.

4.5. Algorithm Execution

Algorithm 1 summarizes this envisioned execution cycle. It serves as an abstract runtime model: the compiled ASP program Π is stored on the node. Dynamic runtime facts are generated based on the current state of the node and the triggering event and passed to the embedded Clingo solver. This fact-generation step, denoted as `TranslateToASP(E)`, is distinct from τ: τ compiles user routines into static ASP rules, whereas `TranslateToASP(E)` converts a runtime event into temporary ASP facts. Finally, the resulting action atoms are mapped back to concrete DaaS operations. To clarify the execution frequency, condition facts are refreshed periodically, while the Clingo solver is invoked in an event-driven manner, specifically once per trigger event (e.g., upon the arrival of a packet).

If the solver produces no answer set, the routine evaluation is treated as inconsistent for the current event. In this case, the node executes no action for that event. This fail-safe behavior prevents contradictory user-defined routines from being translated into unsafe network operations.

The execution cycle separates three responsibilities. First, the node refreshes condition facts that describe its current state, such as the synchronized logical time or node identifier. Second, intercepted events are translated into trigger facts and combined with the compiled ASP program. Third, the answer set returned by the solver is scanned for action atoms, which are interpreted by the DaaS runtime as concrete operations on the device or communication layer. This separation keeps the user-defined routine declarative, while the runtime remains responsible for sensing, solving, and executing the resulting actions.

5. Related Work

Logic programming has been explored as a reasoning layer for IoT and cyber-physical systems. Calegari et al. propose Logic Programming as a Service as a way to provide intelligence for IoT systems and multi-agent IoT environments [16, 17]. Other works use logic programming to model access-control policies in IoT [18], to support risk prediction and prevention in sensor-rich independent-living environments [19], and to reason about trustworthiness in cyber-physical systems by combining ontology-based representations with ASP [20]. Semantic and ontology-based approaches have also been investigated in smart environments, for instance by mapping sensor data into ontological representations and querying them with SPARQL to extract contextual activity patterns [21]. These works show

Algorithm 1: NODE ROUTINE EXECUTION($\Pi, \Delta t$)

Input : ASP Logic Program Π, condition refresh frequency Δt
Output: Physical state changes $\mathcal{A}_{exec}$

$\mathcal{S}$:= Clingo.Embed(Π) // Embed solver with the logic program
while *node is active* **do**
  S_{node} := GetCurrentState()
  $\mathcal{F}_{cond} := \{\texttt{time}(T), \texttt{sid}(ID) \mid T, ID \in S_{node}\}$
  if intercept *event* E **then**
    $\mathcal{F}_{trig}$:= TranslateToASP(E)
    result := $\mathcal{S}.solve(\mathcal{F}_{cond} \cup \mathcal{F}_{trig})$
    if *result is UNSAT* **then**
      continue // No action is executed for this event
    $\mathcal{AS}$:= *result.answerSet*()
    foreach *action atom* $a \in \mathcal{AS}$ **do**
      switch a **do**
        case *route(Pkt, Dest, Link)* **do**
          Forward($Pkt, Dest, Link$)
        case *disable_link(Link)* **do**
          PowerOff($Link$)
        case *set_queue_size(Size)* **do**
          SetQueueSize($Size$)
        case *drop_packet* **do**
          Discard()
  wait Δt

that declarative reasoning can support policy-oriented and context-aware decisions in connected environments. Our contribution is complementary: we focus on end-user routine definition for DaaS mesh-network policies and use ASP as the target formalism for validation and conflict detection.

ASP and related logic-programming techniques have also been applied to configuration, synthesis, and distributed reasoning problems. Yonga et al. use ASP to encode architecture synthesis for distributed smart-camera networks [22], while Le et al. study the use of logic programming for distributed constraint optimization problems [23]. More recently, Azzolini et al. combine ASP and Prolog-based continuous reasoning for adaptive container-image distribution in the cloud–edge continuum [24]. These works are relevant because they use declarative reasoning to select, validate, or adapt configurations in distributed settings. In contrast, our work targets a visual Trigger–Condition–Action language for DaaS nodes, where the generated ASP rules are used to validate and execute local network policies such as routing, packet dropping, link disabling, and queue reconfiguration.

Embedded and edge intelligence research shows that moving computation closer to IoT devices can reduce latency and bandwidth usage, but requires careful consideration of real-time, power, computation, memory, and communication constraints [25]. This is particularly relevant for DaaS nodes with heterogeneous hardware capabilities. In this preliminary work, we do not assume that every constrained IoT device can host a full ASP runtime; rather, Clingo-based routine evaluation is intended for DaaS nodes with sufficient computational resources, while benchmarking solver execution on highly constrained devices remains future work.

Visual programming has been widely investigated as a way to make IoT systems more accessible to users who are not expected to write low-level code. Ray surveys visual programming languages for IoT, highlighting how graphical abstractions can simplify the development of applications over

heterogeneous devices [26]. Silva et al. review visual programming approaches for distributed computation in IoT, emphasizing the need to coordinate computations across multiple devices and execution contexts [27]. Other works propose concrete visual environments for IoT and related platforms, such as visual programming for autonomous decentralized systems [28], IoT and robotics education platforms [29], distributed IoT-centric complex event processing [30], and SmartThings automation [31]. Differently from these approaches, our work uses a DaaS-specific visual interface not only to simplify programming, but also to compile user-defined routines into ASP rules that can be checked against logical constraints.

6. Conclusion

This paper presented a Visual Logic Programming interface for the DaaS paradigm, addressing the knowledge-acquisition bottleneck that prevents non-expert users from configuring event-driven behaviors in heterogeneous IoT mesh networks. The three contributions are as follows. First, we defined a formal Trigger-Condition-Action paradigm in which network policies are expressed as tuples $R = (t, C, A)$ and assembled through a graphical drag-and-drop interface, requiring no programming expertise. Second, we formalized a compilation pipeline τ that translates visual routines into Answer Set Programming rules, intended to be evaluated in real time by a Clingo-based routine engine on DaaS nodes with sufficient computational resources, without interrupting device operations. Third, we showed how ASP's integrity constraints and default reasoning support conflict detection and resolution between coexisting policies, a capability that simple procedural if-then engines cannot provide.

The current implementation is a preliminary design: the conflict detection mechanism is formalized but not yet validated on real hardware, and no user study has been conducted to measure cognitive load or error rates for non-expert users. Future work will address both dimensions. On the systems side, we plan to benchmark the embedded Clingo solver on resource-constrained devices (ESP32, low-power ARM) to quantify grounding and solving latency as a function of program size. On the human-computer interaction side, we will conduct user studies following established evaluation frameworks for end-user programming to assess the accessibility of the visual interface.

Acknowledgments

This work was supported by the Italian Ministry of Health (MSAL) under POS projects CAL.HUB.RIA (CUP H53C22000800006) and RADIOAMICA (CUP H53C22000650006); by the Italian Ministry of Enterprises and Made in Italy under project STROKE 5.0 (CUP B29J23000430005); under PN RIC project ASVIN "Assistente Virtuale Intelligente di Negozio" (CUP B29J24000200005); by Regione Calabria under project NextGenGuides "Innovazione Tecnologica per le Guide Turistiche del Futuro tramite l'ausilio di tecnologie innovative AI e sistemi di geolocalizzazione avanzata" (CUP J39I24002040005); under project MOZART "Modelli e tecniche di mOdernizzazione di applicaZioni e data silos a supporto di clienti esterni, field force e impiegati di backoffice basati su AI GeneRativa e apprendimento auTomatico" (CUP J49I24001740005); by the LAIA lab (part of the SILA labs).

Declaration on Generative AI

During the preparation of this work, the author(s) used Google Gemini and GPT-5 in order to: Grammar and spelling check. After using these tool(s)/service(s), the author(s) reviewed and edited the content as needed and take(s) full responsibility for the publication's content.

References

[1] G. Koulouras, S. Katsoulis, F. Zantalis, Evolution of bluetooth technology: Ble in the iot ecosystem, Sensors 25 (2025) 996.

[2] M.-S. Pan, Y.-C. Tseng, Zigbee and their applications, in: Sensor networks and configuration: Fundamentals, standards, platforms, and applications, Springer, 2007, pp. 349–368.

[3] S. Devalal, A. Karthikeyan, Lora technology-an overview, in: 2018 second international conference on electronics, communication and aerospace technology (ICECA), IEEE, 2018, pp. 284–290.

[4] M. Gelfond, V. Lifschitz, Logic programs with classical negation, in: D. Warren, P. Szeredi (Eds.), Logic Programming: Proc. of the Seventh International Conference, 1990, pp. 579–597.

[5] M. Gelfond, V. Lifschitz, The stable model semantics for logic programming. icslp, 1988, 1988.

[6] V. Marek, M. Truszczyński, Stable models and an alternative logic programming paradigm, in: The Logic Programming Paradigm: a 25-year Perspective, 1999, pp. 375–398. doi:10.1007/978-3-642-60085-2_17.

[7] I. Niemelä, Logic programming with stable model semantics as a constraint programming paradigm, 1999. doi:10.1023/A:1018930122475.

[8] P. Kolios, C. Panayiotou, G. Ellinas, M. Polycarpou, Data-driven event triggering for iot applications, IEEE Internet of Things Journal 3 (2016) 1146–1158. doi:10.1109/JIOT.2016.2564428.

[9] I. F. Akyildiz, X. Wang, A survey on wireless mesh networks, IEEE Communications Magazine 43 (2005) S23–S30. doi:10.1109/MCOM.2005.1509968.

[10] S. Tarkoma, Overlay Networks: Toward Information Networking., Auerbach Publications, 2010.

[11] Y. Owada, G. Sato, K. Temma, T. Kuri, M. Inoue, T. Nagano, An implementation of layer 2 overlay mesh network and edge computing platform for iot, in: 2019 Twelfth International Conference on Mobile Computing and Ubiquitous Network (ICMU), IEEE, 2019, pp. 1–6.

[12] T. Qiu, N. Chen, K. Li, M. Atiquzzaman, W. Zhao, How can heterogeneous internet of things build our future: A survey, IEEE Communications Surveys & Tutorials 20 (2018) 2011–2027.

[13] E. Avelar, L. Marques, D. dos Passos, R. Macedo, K. Dias, J. Rezende, Interoperability issues on heterogeneous wireless communication for smart cities, Computer Communications 58 (2015) 4–15. doi:10.1016/j.comcom.2014.10.005.

[14] S. P. Selvaraju, A. Balador, H. Fotouhi, M. Vahabi, M. Björkman, Network management in heterogeneous iot networks, IEEE Wireless Communications 28 (2021) 64–71. doi:10.1109/MWC.001.2000242.

[15] E. Garone, A. Gasparri, F. Lamonaca, Clock synchronization for wireless sensor network with communication delay, in: 2013 American Control Conference, 2013, pp. 771–776. doi:10.1109/ACC.2013.6579929.

[16] R. Calegari, E. Denti, S. Mariani, A. Omicini, Logic programming as a service (lpaas): Intelligence for the iot, in: 2017 IEEE 14th International Conference on Networking, Sensing and Control (ICNSC), IEEE, 2017, pp. 72–77.

[17] R. Calegari, E. Denti, S. Mariani, A. Omicini, Logic programming as a service in multi-agent systems for the internet of things, International Journal of Grid and Utility Computing 10 (2019) 344–360.

[18] I. Bohé, M. Willocx, J. Lapon, V. Naessens, A logic programming approach to incorporate access control in the internet of things, in: IFIP International Internet of Things Conference, Springer, 2022, pp. 106–124.

[19] A. MILEO, D. MERICO, R. BISIANI, Reasoning support for risk prediction and prevention in independent living, Theory and Practice of Logic Programming 11 (2011) 361–395. doi:10.1017/S147106841000058X.

[20] T. H. Nguyen, T. C. Son, M. Bundas, M. Balduccini, K. C. Garwood, E. R. Griffor, Reasoning about trustworthiness in cyber-physical systems using ontology-based representation and asp, in: International Conference on Principles and Practice of Multi-Agent Systems, Springer, 2020, pp. 51–67.

[21] N. Agti, L. Sabri, O. Kazar, A framework for an ontological querying-based cognitive perspective

for activity recognition., International Journal of Computers and Their Applications 32 (2025) 112–123.
[22] F. Yonga, M. Mefenza, C. Bobda, Asp-based encoding model of architecture synthesis for smart cameras in distributed networks, ACM Transactions on Design Automation of Electronic Systems (TODAES) 20 (2015) 1–28.
[23] T. Le, T. C. Son, E. Pontelli, W. Yeoh, Solving distributed constraint optimization problems using logic programming, Theory and Practice of Logic Programming 17 (2017) 634–683.
[24] D. Azzolini, S. Forti, A. Ielo, Continuous reasoning for adaptive container image distribution in the cloud–edge continuum, Cluster Computing 28 (2025) 461.
[25] L.-M. Ang, K. P. Seng, M. Wachowicz, Embedded intelligence and the data-driven future of application-specific internet of things for smart environments, International Journal of Distributed Sensor Networks 18 (2022) 15501329221102371.
[26] P. P. Ray, A survey on visual programming languages in internet of things, Scientific Programming 2017 (2017) 1231430.
[27] M. Silva, J. P. Dias, A. Restivo, H. S. Ferreira, A review on visual programming for distributed computation in iot, in: International Conference on Computational Science, Springer, 2021, pp. 443–457.
[28] Y. Chen, Analyzing and visual programming internet of things and autonomous decentralized systems, 2016.
[29] G. De Luca, Z. Li, S. Mian, Y. Chen, Visual programming language environment for different iot and robotics platforms in computer science education, CAAI Transactions on Intelligence Technology 3 (2018) 119–130.
[30] M. O. Gökalp, A. Koçyiğit, P. E. Eren, A visual programming framework for distributed internet of things centric complex event processing, Computers & Electrical Engineering 74 (2019) 581–604.
[31] N. Bak, B.-M. Chang, K. Choi, Smart block: A visual programming environment for smartthings, in: 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC), volume 2, IEEE, 2018, pp. 32–37.