Beyond Physical Mashups: Autonomous Systems for the Web of Things

Andrei Ciortea
Siemens Corporate Technology
Berkeley, CA 94704, USA
andrei.ciortea@siemens.com

Olivier Boissier
Univ. Lyon, MINES Saint-Étienne
CNRS Lab Hubert Curien UMR 5516
Saint-Étienne, France F-42023
olivier.boissier@emse.fr

Alessandro Ricci
University of Bologna
Cesena, Italy
a.ricci@unibo.it

ABSTRACT

By abstracting devices to Web resources, the Web of Things (WoT) fosters innovation and rapid prototyping in the Internet of Things (IoT): it enables developers to use standard Web technologies for creating mashups of Web services that perceive and act on the physical world (a.k.a. *physical mashups*). In recent years, however, it has become apparent that current programming paradigms for Web development have important shortcomings when it comes to engineering IoT systems: static Web mashups cannot adapt to dynamic IoT environments, and manually mashing-up the IoT does not scale. To address these limitations, WoT researchers started to look for means to engineer WoT systems that are more autonomous in pursuit of their design objectives. The engineering of autonomous systems has already been explored to a large extent in the scientific literature on artificial intelligence. In this position paper, we distill that large body of research into a coherent set of abstractions for engineering autonomous WoT systems.

KEYWORDS

Autonomous agents; multi-agent systems; Web of Things

ACM Reference format:
Andrei Ciortea, Olivier Boissier, and Alessandro Ricci. 2017. Beyond Physical Mashups: Autonomous Systems for the Web of Things. In *Proceedings of Eighth International Workshop on the Web of Things, Linz, Austria, October 22, 2017 (WoT '17),* 5 pages.
DOI: 10.1145/3199919.3199924

1 INTRODUCTION

The Web of Things (WoT) [11] has significantly lowered the entry-barrier for developers and users of Internet of Things (IoT) systems. Two paradigms that were very successful early-on in showcasing the benefits of Web-enabled devices were inspired by *dataflow programming* and *rule-based systems*. In the former approach, developers use a visual programming language and a WoT mashup editor to create directed graphs of devices and digital services[1] [1, 10]. In the latter approach, developers and/or end-users define event-condition-action rules, where an event triggers the execution of one or more actions.[2,3] Both approaches ease the development of *reactive* WoT systems. On the upside, these systems are very responsive to sensory input, but their main drawback is that action is *tightly coupled* to perception, and thus these systems cannot adapt their behavior to changes in the environment or to new user requirements at runtime.

To avoid the static chaining of actions used in reactive WoT systems, more recent approaches turned to automated planning [12]: given a design goal or a user-specified goal, it is left to the WoT system to figure out how to achieve that goal using a reasoner and semantic descriptions of services discovered at runtime. Such *proactive* WoT systems are more adaptable (the chain of actions is created at runtime), but they are also less responsive: automated planning is computationally costly, and environmental changes are taken into account only before the planning phase (the inferred plan can become invalid during execution).

Note that many of the underlying research questions the WoT community is now confronted with, such as *how to balance reactive and proactive behavior in autonomous systems*, *how to enable proactive behavior in resource-constrained systems*, or *how to engineer large-scale autonomous systems*, have been explored to a large extent in the scientific literature on artificial intelligence and, in particular, *autonomous agents and multi-agent systems (AAMAS)* [17].

Our claim is that AAMAS research already provides models and technologies that can be applied to design and develop complex autonomous systems for the WoT. In fact, in previous publications, we have already demonstrated the successful transfer of multi-agent technology to the development of WoT systems [5], and we have already looked at some of the challenges of bringing autonomous agents to WoT environments [7].

In this position paper, we define a conceptual alignment between AAMAS and WoT research, and propose a coherent set of abstractions for engineering autonomous WoT systems.

2 AGENTS FOR THE WOT

Agent-oriented programming was first articulated as a paradigm in [16], but it is a subfield of artificial intelligence whose origins can be traced back to Georgeff and Lansky's work on reactive planning at the Stanford Research Institute in the mid 1980's [9]. Existing models and languages for programming autonomous agents and multi-agent systems are thus the result of over 30 years of cumulative ongoing research.

[1]http://nodered.org, accessed: 02.10.2017.

[2]http://www.ifttt.com, accessed: 02.10.2017.
[3]http://www.zapier.com, accessed: 02.10.2017.

In this section, we distill that research into *three main design choices* that we believe can ease the engineering of autonomous WoT systems. We discuss and motivate each choice in what follows.

2.1 Cognitive Agents

The first proposed design choice is to decouple perception from action through *deliberation*: the WoT system is endowed with the ability to reflect on its own internal state and the state of its environment in order to *decide* on a course of action. Deliberation allows the system to pursue goals much like classical planners, but also enables it to react to important events or (if necessary) to switch its focus altogether and adopt new goals. We refer to autonomous deliberative systems as *cognitive agents*.

In what follows, we refine the abstract concept of a cognitive agent down to the implementation level. We present the *Belief-Desire-Intention (BDI)* architecture [14], which is the mainstream architecture for cognitive agents in the AAMAS community, and discuss programming languages for BDI agents.

2.1.1 Representing the internal state. To ease the engineering of autonomous WoT systems, it is necessary to provide developers with a level of abstraction that facilitates designing, programming, but also inspecting and debugging autonomous behavior. The BDI architecture [14] provides a formal "human-oriented" level of abstraction for representing the internal state of an agent in terms of *mental attitudes*:

- *beliefs*: information the agent holds about the world; beliefs are not necessarily true, they may be out of date or inaccurate;
- *desires*: states of affairs the agent wishes to bring to the world (i.e., the agent's goals);
- *intentions*: the states of affairs the agent has decided to work towards (i.e., goals the agent is committed to achieve).

Developers can then program agents to *deliberate* about their own beliefs, desires, and intentions, and modify their mental attitudes as needed. For instance, an agent can decide to suspend the execution of an intention in order to react to an important event, or it can even drop an intention if it becomes unachievable. The agent remains reactive in pursuit of its goals, and can even pursue new goals if the situation warrants.

2.1.2 Deliberation and means-end reasoning. BDI agents deliberate on their internal state and the state of their environment through *reasoning cycles*. A simplified view of a BDI agent's typical reasoning cycle is depicted in Figure 1. The BDI architecture consists of data structures for handling the agent's beliefs, desires, and intentions, and a queue for handling external and internal events [14]. Examples of *external events* include sensory input, or messages received from other agents. Examples of *internal events* include beliefs generated by the agent itself (i.e., mental notes), or sub-goals generated in pursuit of a given goal. In every cycle, an interpreter processes one or more events, and updates the agent's beliefs, desires, and intentions. Three important (and customizable) pieces of the BDI architecture are the functions used to update the agent's beliefs and desires, and to select its intentions. For further details on the inner workings of BDI agents, we refer interested readers to [3, 14].

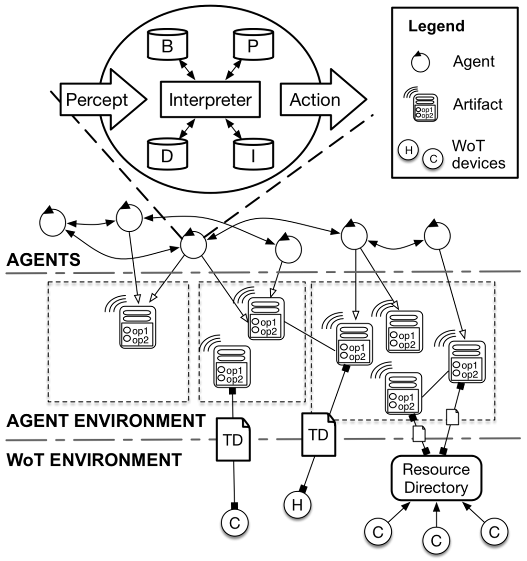

Figure 1: Systems of cognitive agents for the WoT.

The BDI architecture is an abstract model that focuses on deliberation without constraining how agents achieve their goals. In most implementations, however, agents achieve their goals using libraries of *plans* defined by developers (cf. Figure 1).[4] Similar to the other key data structures in the BDI architecture, agents can update their libraries of plans at runtime (e.g., by exchanging plans with one another). This is thus an effective, formal, and modular approach to engineer adaptable agents whose behavior can evolve *at runtime*. Nevertheless, this implementation choice implies a limitation: unless the agent is able to synthesize its own plans (e.g., through automated planning), its behavior is bounded by the set of pre-programmed plans made available at runtime. This pragmatic approach stems from the original motivation behind reactive planning [9, 14]: to enable goal-directed behavior in resource-bounded agents and under real-time constraints. This motivation remains highly relevant for developers of WoT systems.

2.1.3 Programming languages for BDI agents. The typical *program* of a BDI agent is composed of the agent's initial beliefs, goals, and plans. Multiple languages and frameworks are available for programming BDI agents. One of the most prominent agent programming languages is AgentSpeak(L) [13], and its more recent extended version known as Jason [3].

In what follows, we illustrate agent programming in Jason for a healthcare application.[5] Our objective is to implement a WoT application that reminds an elderly patient to take her medication at certain hours. The patient signals she took her medication by pressing a button on a remote control, and the application sends visual reminders through a lamp.

We implemented the above application using a Philips Hue light bulb and an Arduino-based circuit with an infrared receiver. An

[4]In a sense, we can conceive of *plans* as beliefs about means to achieve goals [14].
[5]We will not explore the features of the Jason language in detail, but we refer interested readers to [3] for further information.

extract from the agent program is shown in Listing 1. The hours for pill administration are provided in the agent's initial set of beliefs, but they could also be provided by end-users at runtime. The agent has a single initial `!start` goal, which performs all required initializations and kicks off patient monitoring by creating the `!monitor` sub-goal (cf. Listing 1). The agent achieves its goals by executing plans written in Jason, where a plan has the form:

```
triggering_event : application_context <- plan_body .
```

Listing 1: Jason agent that monitors a patient's medication.

```
/* Initial beliefs */
pills_time(11, 00).
pills_time(23, 00).
// (...)

/* Initial goals */
// Initializes the agent and creates the "!monitor" sub-goal
!start.

/* Plans */
// (...)
+!monitor: .time(Hour, Min, _) & pills_time(Hour, Min)
    & (not took_pills(_, Hour, Min))
    & (not take_pills_notification(_, Hour, Min)) <-
  // Send visual reminder via Philips Hue artifact
  turnLightOn(0.409, 0.518)[artifact_name("bulb")];
  // Create a mental note for this notification
  jia.currentTimeMillis(Id);
  +take_pills_notification(Id, Hour, Min);
  // Resume monitoring
  !monitor.

+!monitor : true <-
  // Suspend intention for 60 seconds
  .wait(60000);
  // Resume monitoring
  !monitor.
```

For instance, as shown in Listing 1, if it is time to take the medication, the agent does not believe the patient had already taken her medication, and no visual reminder has yet been sent, the agent notifies the patient it is time to take her medication by turning on a lamp. A new `!monitor` sub-goal is then created to continue monitoring the patient.

2.2 Agents in WoT Environments

The second proposed design choice is to separate concerns of autonomous deliberative behavior from concerns of supporting and enacting that behavior. That is to say, we distinguish between a *cognitive agent* and the *environment* in which it is situated, and consider both as first-class abstractions in WoT systems. This separation of concerns simplifies agent development, and increases the reusability of system components and the evolvability of the overall system. This design choice draws from a line of research on engineering agent environments [15].

2.2.1 Agents and artifacts. We model an agent's *environment* as a dynamic set of *artifacts*, where an artifact is a computational object that exposes:

- *observable properties*: state variables that can be perceived by the agent;
- *observable events*: non-persistent, fire-and-forget signals that carry information and can be perceived by the agent;
- *operations*: environment actions provided to the agent; operations can change the values of observable properties or they can trigger events.

Artifacts therefore provide agents with a *generic*, *uniform interface* defined in terms of properties, events, and operations. The set of all interactions an agent can have with its environment is determined by the artifacts available at runtime. Agents use artifacts in pursuit of their goals, and they can create or destroy artifacts at runtime.

The *artifact* abstraction, as presented above, was introduced by *Agents & Artifacts (A&A)* [15], a well-known meta-model for multi-agent systems inspired by activity theory. We refer interested readers to [15] for further details on the A&A meta-model.

In our previous example in Listing 1, the agent interacts with a Philips Hue light bulb via an artifact implemented using CArtAgO, a framework for A&A [15]. The agent logic is thus insulated from low-level details of accessing the Philips Hue HTTP API, and thus the agent program is simplified. Moreover, the artifact can be shared with other agents situated in the same environment, or the artifact class can be reused in other applications.

The practicality of modeling devices as artifacts is obvious, but agents can also benefit from purely digital artifacts. For instance, we can further simplify the agent program in Listing 1 by creating a *clock* artifact that emits a signal every minute. The agent can then simply react to new clock signals, which yields the more concise implementation shown in Listing 2. The clock artifact can also be reused in other applications.

Listing 2: Monitoring agent using a clock artifact.

```
/* Plans */
// (...)
+clock_time(Hour, Min) : pills_time(Hour, Min)
    & (not took_pills(_, Hour, Min))
    & (not take_pills_notification(_, Hour, Min)) <-
  // Send visual reminder via Philips Hue artifact
  turnLightOn(0.409, 0.518)[artifact_name("bulb")];
  // Create a mental note for this notification
  jia.currentTimeMillis(Id);
  +take_pills_notification(Id, Hour, Min).
```

2.2.2 Artifacts for the WoT. The light bulb artifact in our application is already Web-enabled, but it is tightly coupled to the Philips Hue HTTP API. However, it is worth to note the similarity between the artifact model defined by A&A and the *WoT Thing Description (TD)* currently being standardized in the W3C WoT WG[6]. Both models define three types of interaction patterns, namely *properties*, *events*, and *operations* (or *actions*), with the WoT TD model being slightly more generic: a TD can expose writable properties, whereas artifact properties are read-only. Applying the WoT TD to decouple *artifacts* from *devices* is thus straightforward. Furthermore, agent environments could then benefit from any discovery mechanisms for WoT TDs (see Section 2.3.1).

Note that the WoT TD could also be applied as a wrapper for an agent-based system. That could be the case, for instance, when a BDI interpreter is embedded on a WoT device.

[6]https://w3c.github.io/wot-thing-description/, accessed: 02.10.2017.

2.3 Agent Discovery and Interaction

The third proposed design choice is to model non-trivial autonomous WoT systems as decentralized systems composed of multiple cognitive agents (rather than monolithic systems). This design choice decentralizes control, which can increase system robustness, and enhances modularity, which increases the reusability and maintainability of agents and the evolvability of the overall system. Decentralization and modularity also enable the development of *open* systems in which agents discover and interact with one another at runtime. We discuss discovery and interaction in open autonomous WoT systems in what follows.

2.3.1 Discovery. In open WoT systems, a practical resource discovery mechanism is the use of *directories* that allow devices to register and advertise descriptions of resources.[7] Similar directory-based mechanisms have been proposed in the AAMAS community for the discovery of agents (and services in their environment)[8], and can be easily implemented on top of directories of Web resources (cf. Figure 1).

Going further, we recently proposed to use *HATEOAS* [8] to engineer *hypermedia-driven agent environments* [6], which could be layered on top of WoT environments in order to enhance discoverability (e.g., see [7]). Discovery mechanisms based on crawling, similar to the ones used by Web search engines, could then help populate and maintain directories of agents and artifacts for open WoT systems.

2.3.2 Interaction. In open systems, little assumptions can be made about the design and implementation of components. It naturally follows that the engineering of such systems must focus to a large extent on component interactions. For instance, to support an open Web, REST focuses on the semantics of component interactions by defining a uniform interface between components [8].

There are two main schools of thought for defining the meaning of interactions in open multi-agent systems. Early research on agent communication and interaction was strongly influenced by *speech act theory*, which treats language as action: the meaning of a *speech act*, such as communicating a piece of information or delegating a goal, is defined in terms of its intended effect on the hearers' mental attitudes (i.e., their beliefs, desires, intentions). Cognitive agents running on various WoT devices, for instance, can then interact with one another by exchanging messages expressed in a formal, high-level communication language with well-defined semantics (see [5] for an example WoT application). Sequences of such messages (a.k.a. *agent interaction protocols*) can be standardized in order to support loosely coupled interactions among agents in open WoT systems.[9]

Agent communication languages based on mental attitudes are simple and intuitive for developers. Their main drawback, however, is that interactions are subjective and can be ambiguous (to interpret messages, agents make assumptions about the mental attitudes of others). This approach is thus best suited for systems in which it can be safely assumed that agents are sincere and cooperative. In open systems, however, this is usually not the case. An alternative approach is to define the meaning of agent interactions in terms of *commitments* agents make to one another. Commitments are unambiguous and objective (they are defined externally to the agents). This approach imposes less constraints on the design and implementation of agents, and facilitates the monitoring and testing for compliance of agent behavior. For further information on this line of research, we refer interested readers to [4].

[7]https://tools.ietf.org/html/draft-ietf-core-resource-directory-11, accessed: 02.10.2017.
[8]http://www.fipa.org/specs/fipa00001/SC00001L.html, accessed: 02.10.2017.
[9]Note that in the early 2000s, the Foundation for Intelligent Physical Agents (FIPA) released a set of standards for communication and interaction in multi-agent systems (http://www.fipa.org/repository/standardspecs.html, accessed: 02.10.2017).

3 THE ROAD AHEAD

As the WoT community turns towards autonomous systems, we believe the alignment between AAMAS and WoT research becomes more and more relevant. In this position paper, we presented a coherent set of abstractions that ease the design, programming, inspection, and debugging of autonomous systems for the WoT. The proposed abstractions draw from many years of research on AAMAS and are grounded in formal, mainstream models. Moreover, the tooling required to use these abstractions for prototyping autonomous WoT systems is already available [2, 5]. While there is still a big gap between research and development on the one hand and large-scale deployment and usage on the other, the biggest hurdle we see to bringing autonomous systems to the WoT is mostly an engineering task: to provide WoT developers with mature, Web-compliant, and industry-grade multi-agent technology.

From a research perspective, the conceptual alignment presented in this paper enables the transfer to autonomous WoT systems of a large body of work on advanced research topics in AAMAS, such as agent coordination, regulating autonomous behavior, reinforcement learning, or distributed decision making. We refer interested readers to [17] for a textbook introduction to these topics and others.

REFERENCES

[1] Michael Blackstock and Rodger Lea. 2012. IoT mashups with the WoTKit. In *Internet of Things (IOT), 2012 3rd International Conference on the.* IEEE, 159–166.
[2] Olivier Boissier, Rafael H Bordini, Jomi F Hübner, Alessandro Ricci, and Andrea Santi. 2013. Multi-agent oriented programming with JaCaMo. *Science of Computer Programming* 78, 6 (2013), 747–761.
[3] Rafael H Bordini, Jomi Fred Hübner, and Michael Wooldridge. 2007. *Programming multi-agent systems in AgentSpeak using Jason.* Vol. 8. John Wiley & Sons.
[4] Amit K Chopra and Munindar P Singh. 2016. From social machines to social protocols: Software engineering foundations for sociotechnical systems. In *Proceedings of the 25th International Conference on World Wide Web.* International World Wide Web Conferences Steering Committee, 903–914.
[5] Andrei Ciortea, Olivier Boissier, Antoine Zimmermann, and Adina Magda Florea. 2016. Responsive Decentralized Composition of Service Mashups for the Internet of Things. In *Proceedings of the 6th International Conference on the Internet of Things (IoT).* ACM.
[6] Andrei Ciortea, Olivier Boissier, Antoine Zimmermann, and Adina Magda Florea. 2017. Give Agents Some REST: A Resource-oriented Abstraction Layer for Internet-scale Agent Environments. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems (AAMAS).* International Foundation for Autonomous Agents and Multiagent Systems, 1502–1504.
[7] Andrei Ciortea, Antoine Zimmermann, Olivier Boissier, and Adina Magda Florea. 2016. Hypermedia-driven Socio-technical Networks for Goal-driven Discovery in the Web of Things. In *Proceedings of the 7th International Workshop on the Web of Things (WoT).* ACM.
[8] Roy Thomas Fielding. 2000. *Architectural styles and the design of network-based software architectures.* Ph.D. Dissertation. University of California, Irvine.
[9] Michael P Georgeff and Amy L Lansky. 1987. Reactive reasoning and planning.. In *AAAI*, Vol. 87. 677–682.
[10] Dominique Guinard, Vlad Trifa, Thomas Pham, and Olivier Liechti. 2009. Towards physical mashups in the web of things. In *Networked Sensing Systems (INSS), 2009 Sixth International Conference on.* IEEE, 1–4.
[11] Dominique Guinard, Vlad Trifa, and Erik Wilde. 2010. A resource oriented architecture for the Web of Things. In *Internet of Things (IOT), 2010.* IEEE, 1–8.
[12] Matthias Kovatsch, Yassin N Hassan, and Simon Mayer. 2015. Practical semantics for the Internet of Things: Physical states, device mashups, and open questions.

In *Internet of Things (IOT), 2015 5th International Conference on the*. IEEE, 54–61.
[13] Anand S Rao. 1996. AgentSpeak (L): BDI agents speak out in a logical computable language. In *European Workshop on Modelling Autonomous Agents in a Multi-Agent World*. Springer, 42–55.
[14] Anand S Rao, Michael P Georgeff, et al. 1995. BDI Agents: From Theory to Practice.. In *ICMAS*, Vol. 95. 312–319.
[15] Alessandro Ricci, Michele Piunti, and Mirko Viroli. 2011. Environment programming in multi-agent systems: an artifact-based perspective. *Autonomous Agents and Multi-Agent Systems* 23, 2 (2011), 158–192.
[16] Yoav Shoham. 1993. Agent-oriented programming. *Artificial intelligence* 60, 1 (1993), 51–92.
[17] Gerhard Weiss. 2000. *Multiagent systems: a modern approach to distributed artificial intelligence*. MIT press.