A model for trustworthy orchestration in the Internet of Things

Michele Bottone, Giuseppe Primiero, Franco Raimondi
Department of Computer Science
Middlesex University
London, UK
Email: {m.bottone,g.primiero,f.raimondi}@mdx.ac.uk

Vincenzo De Florio
Evolution, Complexity and COgnition Group
The Global Brain Institute
Vrije Universiteit Brussel, BE
Email: vincenzo.deflorio@gmail.com

***Abstract*—Embedded systems such as Cyber-Physical Systems (CPS) are typically designed as a network of multiple interacting elements with physical input (or sensors) and output (or actuators). One aspect of interest of open systems is fidelity, or the compliance between physical figures of interest and their internal representation. High fidelity is defined as a stable mapping between actions in the physical domain and intended or expected values in the system domain and deviations from fidelity are quantifiable over time by some appropriate informative variable. In this paper, we provide a model for designing such systems based on a framework for trustworthiness monitoring and we provide a Jason implementation to evaluate the feasibility of our approach. In particular, we build a bridge between a standard publish/subscribe framework for CPS called MQTT and Jason to enable automatic reasoning about trustworthiness.**

I. Introduction

In the Internet of Things (IoT) paradigm, applications are typically built by *orchestrating* many connected components (the *things*) to achieve a certain goal. For instance, an intrusion detection system can be built by connecting sensors for movement with actuators for alarms; or a monitoring system can coordinate temperature and humidity sensors with an air-conditioning system as actuator. In this kind of scenarios, it is common to adopt a publish/subscribe infrastructure in which sensor publish their data, and actuators subscribe to certain topics from the former. MQTT is a standard platform for M2M/IoT connectivity [6]. It is designed to be lightweight and aimed at connections with remote locations, with small code footprint and network bandwidth requirements.

An aspect distinct from simple monitoring of events is the *evaluation* of the *fidelity* of the system of sensors and actuators, especially when these include several types of monitors and users. This links the IoT paradigm to another crucial aspect of smart environment design, namely their *trustworthiness*. In the context of connected environments is not only advantageous to be able to collect as much information as possible to optimize functionalities. It also becomes essential to overview standards of efficiency and security, to be able to identify malfunctioning and detect possible system intrusions that might become safety critical.

In this paper we offer such a combined analysis. We present a model to characterise the trustworthiness of a fully functional IoT monitoring system by building upon the work presented in [1] for "standard", software-only systems. In particular, we show how to monitor individual sensor values in terms of fidelity functions that trace three distinct behavioural patterns *dynamically at run-time* and use the MQTT publish/subscribe model to define controllers that connect the sensors to an actuator. We further define a higher-order function to assess the overall fidelity of the system and induce a trustworthiness evaluation to indicate the running conditions of the system, according to a scale that includes stability and safety conditions. Our contribution is not limited to a theoretical model: we evaluate our approach using a Jason implementation, a Java-based multi-agent systems interpreter for the BDI architecture, developed as an extension of AgentSpeak [5].

The rest of this paper is structured as follows. In Section II we illustrate the idea of a controller for multi-agents system in cyber-physical domains. In Section III we briefly introduce the Jason framework and its syntax and semantics. In Section IV we describe the encoding of fidelity patterns from Section III in Jason. We offer concluding remarks and further research directions in Section V.

II. Janus

A notion of fidelity for systems involving software interacting with human users is introduced in [1]. It relies on the idea that compliant systems manifest a correspondence between domains of actions, but that such a property might not always be perfect. If one is able to monitor and measure the drifting from an ideal total compliance, then one is in a position to assess the level of trustworthiness of said system. In turn, one can evaluate which safety-security actions are required to maintain the system working.

Let us consider an open system $nOPS$, interacting with n environments, each based on sensors and actuators. $nOPS$ will receive information from the sensors (e.g. temperature and humidity) and send commands to the actuators. Data from the sensors and activities performed by the actuators are raw fact $[r]_i, 1 \leq i \leq n$, accompanied by appropriate binary operations $[\dot{+}]_i$ to form algebraic structures. The internal representations of $[r]_i$ by $nOPS$ are expressed by $[q]_i, 1 \leq i \leq n$ internally in its monitoring system, together with operations $[\oplus]_i$ corresponding to $[\dot{+}]_i$. The latter is a somewhat faithful representation of the dynamic variation of the corresponding

class of facts. The mapping of raw facts to their representation in $nOPS$ is given by reflective maps Φ_i, bijective functions expressing *perfect fidelity* iff the association of $([r]_i, [\dot{+}]_i)$ to $([q]_i, [\oplus]_i)$ is an isomorphism. The shifting from perfect fidelity in the system representation is called *drifting* and it is expressed by the association of an error component Δ_i over time t to – what could be called – a function ϕ_i of *imperfect fidelity*:

$$\phi_i : [r]_i \to [q]_i,$$
$$\forall + \in [\dot{+}]_i, \forall \cdot \in [\oplus]_i : \phi_i(r_1 + r_2) = \phi_i(r_1) \cdot \phi_i(r_2) \cdot \Delta_i(t).$$

The monitoring of raw facts and their representations can be performed through a client for reading values of associated, asynchronously and continuously updated variables for each of the involved services. In [1], such a client is called *Janus*. The role of *Janus* is to retrieve periodically the value of each such variable and store it in the associated memory cells. Examples include the value of CPU usage percentage, the state of some executable like MPlayer and the facts associated with a UI. The values retrieved are compared with "reference behaviours", representing the expected behavioural patterns of a trustworthy system. Ideally, such reference value would be those associated with a perfect fidelity function. The comparison may be used to detect gradual or sudden behavioural driftings: the former can be associated to progressive deterioration of the system, the latter to instantaneous system breakdown or takeover.

In the present work, we extend the model of the *Janus* to a M2M/IoT paradigm. We extend the *Janus* client to retrieve values from sensors and interact with actuators. The state of each component as derived from the activities of the publish/subscribe infrastructure can be compared to a set of behavioural patterns, each specifically designed to reflect a fidelity level. The *Janus* client is then able to derive an overall evaluation on the system by composing such individual fidelity patterns. Trustworthiness is identified as a second order property of the system, induced cumulatively from such patterns, and used to assess malfunctioning and required intervention on the IoT platform. Each component c in the platform is assigned a fidelity function: $\phi_c, c \in \{s, a\}$, respectively for a sensor and an actuator, such that $\phi_c : [r]_c \to [q]_c$ expresses the value obtained by mapping the input value from the component's observable behaviour to the preselected expected fidelity pattern associated to that component. Fidelity is then approximated as the inversely proportional function of the drifting from appropriate mappings ϕ_c. For a platform with two sensors s_i, s_j and two actuators a_i, a_j,

$$\phi_s = 1/f(\Delta(t)_{s_i}, \Delta(t)_{s_j})$$
$$\phi_a = 1/f(\Delta(t)_{a_i}, \Delta(t)_{a_j})$$

and some function f which can be weighted according to domain specific parameters. Each ϕ_c is evaluated as displaying an high, medium or low fidelity level when compared to the expected behaviour. The value $\Phi^{nOPS}(t) = \{\phi_s, \phi_a\}$ is the global fidelity value for the system parametrised over time. We consider four levels of analysis of fidelity:

1) a *trustworthy system* identifies high levels of $\Phi^{nOPS}(t)$, inducing optimal, sustainable working conditions;
2) an *unstable system* identifies high-to-medium ϕ_s and low ϕ_a levels, meaning that monitoring is well-functioning but interaction with the environment through actuators might be poor, inducing reconfigurable working conditions;
3) an *unsafe system* identifies high-to-medium ϕ_a and low ϕ_s levels, meaning that monitoring is poor, despite the fact that interaction with the environment through actuators might be efficient, inducing unsecure working conditions;
4) an *untrustworthy system* identifies low-levels of $\Phi^{nOPS}(t)$, inducing inadvisable or below-safety working conditions.

III. JASON

In this section we introduce Jason, a belief-desire-intention (BDI) framework for multi-agent systems, which we use as the underlying platform for *Janus*. BDI architectures [3] are among the most popular models of software agency. They are loosely based on the concept of a *reasoning cycle*: the agent has *beliefs*, based on what it perceives and communicates with other agents; beliefs can produce *desires*, intended as states of the world that the agent wants to achieve; the agent deliberates on its desires and decides to commit to some; desires to which the agent is committed become *intentions*, to satisfy which the agent executes plans that lead to action. The behaviour of the agent (i.e., its actions) is thus explained or caused by what it intends (i.e., the desires it decided to pursue). Ideally, within the BDI architecture, agents should react to changes in its environment as soon as possible while keeping its proactive (i.e., desires-oriented) behaviour.

AgentSpeak(L) [2] is an abstract declarative programming language for implementing BDI agents with Prolog-like instructions, which can be extended to fit specific needs. Its syntax defines agent programs as a set of logical beliefs, rules and plans. Formally, the syntax of an agent program is defined in the following way. For $\mathcal{S}$ a finite set of symbols including predicates, actions, and constants, and $\mathcal{V}$ a set of variables one can define vectors of terms in the first-order logic:

+ If b is a predicate symbol and $\mathbf{t}$ a term, we define $b(\mathbf{t})$ to be a belief atom.
+ if $b_A(\mathbf{t})$ and $b_B(\mathbf{t})$ are belief atoms, where A and B can be conjunctions, disjunctions or negations of belief literals, then the rule $b_A(\mathbf{t}) :- b_B(\mathbf{t})$ describes how the latter is inferred from the former.
+ If $g(\mathbf{t})$ is a belief atom, then $!g(\mathbf{t})$ and $?g(\mathbf{t})$ are goals, $!g(\mathbf{t})$ denoting an achievement goal and $?g(\mathbf{t})$ a test goal.
+ If $p(\mathbf{t})$ is a belief atom or goal, then $+p(\mathbf{t})$ and $-p(\mathbf{t})$ are triggering events with $+$ and $-$ denoting respectively the addition and deletion of a belief to be held or goal to be achieved.
+ If a is an action symbol and $\mathbf{t}$ a term, then $a(\mathbf{t})$ is an action.

+ If e is a triggering event, $c_1, \ldots, c_m$ are beliefs and $q_1, \ldots, q_n$ are goals or actions, the rule $e : c_1, \ldots, c_m \leftarrow q_1, \ldots, q_n$ defines a plan, with $c_1, \ldots, c_m$ its context and $q_1, \ldots, q_n$ its body.

Jason [4] extends the AgentSpeak syntax into a flexible, extensible Java-based, open-source development environment and interpreter, which is easily customisable. Jason programming revolves around plans, which are the closest thing there is to a function or method in a declarative language. Actions in the body of an expression are executed in sequence as a consequence of the triggering of the plan, which can consist of belief addition and removal, requests to achieve and unachieve (sub)goals, or built-in or user-defined internal actions that change the environment or the agent's mental state over time. In Jason, ground literals are also extended by strong negation, annotations, and message passing. We refer to [5] for additional details.

IV. The Orchestrating System

In this section we describe how the fidelity patterns are encoded in Jason. We set up a multi-agent system (MAS) where each component is a separate agent, introducing a special agent – the controller or *Janus* – that oversees and assesses the overall fidelity of the system. We further define a higher-order function to assess the overall fidelity of the system and induce a trustworthiness evaluation to indicate the system's running conditions, according to the scale including stability and safety conditions introduced in Section II. We release all the software open source, with the Jason and Java files available at https://bitbucket.org/mdxmase/janus-jason/.

A. Environment and Communications Protocols

The default Jason environment comes with built-in classes that make use of the `.send` and `.broadcast` internal actions for single and multiple inter-agent communication, respectively. In addition to this infrastructure, we built a custom environment `MQTTEnvironment` to bridge between MQTT messages and Jason. The Java class `MQTTEnvironment.java` does two things: it implements the `MQTTCallback` interface for MQTT connection/disconnection and it sets up the topics in the publish/subscribe format so that when a message is published, it adds the relevant percept to the belief base together with the value of the sensor.

The following code snippet shows how the environment can make a connection to an MQTT broker and how messages published under a certain topic can be translated into Jason beliefs in the method `messageArrived` by means of the internal action `addPercept`

```
/* Called before the MAS execution with the args
     informed in .mas2j */
@Override
public void init(String[] args) {
   // [...]
   try {
      client = new MqttClient("tcp://localhost:1883",
           "JasonMQTTEnvironment");
      client.connect();
      // [...]
      client.setCallback(this);
      client.subscribe("MQTTJason/#"); // # is the
           wildcard for multiple topics
      // [...]
   }

public void messageArrived(String topic, MqttMessage
     message) throws Exception {
  // Assuming a default format for publish messages:
  // MQTTJason/sensorid/temperature [for temperature]
  // MQTTJason/sensorid/humidity [for humidity]
  String[] components = topic.split("/");
  if ( components[2].equals("temperature") ) {
    addPercept(Literal.parseLiteral("temp_sensor(" +
                components[1] +
                "," +
                new String(message.getPayload()) +
                ")"));
      // [...]
      }
}
```

B. System Configuration

In Jason, it is possible to create a pre-specified number of identical types of agents using the same code, by using a multiplicity symbol #. The code snippet below creates a system running on a single processor with four kind of agents: the `controller`, `temp_sensor` (four instances), `humi_sensor` (eight instances), and one type of actuator, `airconditioner`.

```
MAS janus_test {
  infrastructure: Centralised
  environment: MQTTEnvironment
  agents:
    controller;
    temp_sensor #4;
    humi_sensor #8;
  airconditioner;
  classpath:
    "/Path/paho.client.mqttv3-1.0.2.jar";
  aslSourcePath: "src/asl";
}
```

1) The individual components: **Humidity and temperature sensors**. Both agents `humi_sensor` and `temp_sensor` implement similar behaviours: they broadcast the values for humidity and temperature respectively with a certain frequency. This is done either by the built-in `broadcast(tell, humi_sensor(Me,H))` internal action, or by bridging a value for the real world. In our case, we have used two DHT22 temperature and humidity sensors connected to two Raspberry Pi (cf. https://www.adafruit.com/products/385).

Air Conditioner. We use an Air Conditioner as the model of an agent that has to go through a predefined sequence of states. At the start it is in a waiting state. If it receives a message `turnon` from the controller, it will go through states `set_up`, `cooling`, `cooldown`, and `waiting` again. The `turnon` message is only accepted if it is in state `waiting`. When there is a change of state, a message is broadcast so that it can be monitored by the controller.

Controller. The most complex part of the application logic resides in this agent. The controller continuously monitors the sensor values and does pattern matching of the individual components' behaviour, and prints the current state of the

system according to a fidelity function. In particular, we implement the three possible patterns for fidelity:

1) component values remain in a range,
2) values do not oscillate more frequently than X,
3) component goes through a sequence of states.

In the language of Section II, Pattern 1 and Pattern 2 express sensor malfunction and thus ϕ_s, while Pattern 3 denotes ϕ_a, i.e. drifting of the actuators value from the fidelity pattern.[1] Intuitively, the latter can be thought of as more akin to an "erroneous system deployment", since actuator `airconditioner` is the terminal interfacing the system's users.

Each pattern is implemented as a rule associated to each component. These individual rules are then used in the context of `controller`'s plans to raise a warning. For the temperature sensors, the first rule checks the Fréchet distance between the current value and a *baseline* value, defined as an affine periodic function on the hour of the day, so that the baseline ranges between 15 (midnight) and 25 (noon), with a valid range of baseline ± 5 degrees. For the humidity sensors, Pattern 2 is defined as too many requests in a given time window – for example a limit of 3 messages in 10 seconds in a given time window. Finally, for the `airconditioner` agent, the rule checks whether it has gone through the correct sequence of states, thus something will be wrong if (a) there was no previous message, thus we are at the start, but we don't receive message "waiting"; or (b) there was an illegal transition, encoded by listing all the possible transitions in sequence.

2) System fidelity: In the setting of [1], fidelity driftings are calculated based on system processes with variables of a given type that are associated with some shared volatile memory segments and read/write access rights. Since Jason runs on top of the Java virtual machine, we keep track of pattern violations directly in the `controller` agent code. Fidelity is then approximated for each pattern type as the inversely proportional function of the drifting in the reporting of raw facts, enabling a continuous cycle of monitoring, apperception, control and action.

In our Jason evaluation, fidelity corresponding to each pattern is defined as a rule keeping track of violations and the dynamical evaluation of trustworthiness of the systems happens in the main `controller` loop at run-time as relevant plans for each outcome are selected if certain patterns in the fidelity evaluation match as true. Each plan also defines an action that is executed in response to each evaluation (for example, disabling a sensor or stopping the entire system).

In a system based on mechanical rules, the trustworthiness assessment produces four outcomes:

1) *Trustworthy*, with high levels of fidelity on all patterns, inducing optimal, sustainable working conditions;
2) *Unstable*, with high-to-medium values on Pattern 3, and low values on at least one of Pattern 1 or Pattern 2, inducing reconfigurable working conditions;
3) *Unsafe*, with high-to-medium values on both Pattern 1 and Pattern 2, and low levels on Pattern 3, inducing alarm-rising working conditions;
4) *Untrustworthy*, with low-levels of fidelity on all patterns, inducing inadvisable or below-safety working conditions.

We tested the performance of the Janus client by instantiating up to several hundred temperature and humidity sensors. Running times were acceptable up to a system size of 500 components, when the overhead of the Java virtual machine in Jason becomes too much to handle for quad-core Retina MacBook Pro 2014. Given that real-life home intelligent systems tend to be in the dozens of components and large systems – such as those of a skyscraper – are composed of thousands of sensors and actuators, initial evaluation has been deemed encouraging.

V. Conclusion

We have presented an extension of the Janus model of trustworthiness for cyber-physical systems based on computing fidelity functions over a MQTT messaging protocol. This implementation has the advantage of fast deployment and can be used to operationally control a sizeable number of individual sensors and actuators. The Janus client scales well in simulations to several hundreds of components, before becoming too unwieldy to handle for modern laptops. Future work will investigate the performance of the system in a typical external environment compared to a simulated environment.

[1] In [1], where cyber-physical systems with users are under consideration, Pattern 1 and Pattern 2 are used to express machine fidelity, Pattern 3 denotes user fidelity.

References

[1] V. De Florio and G. Primiero, A Framework for Trustworthiness Assessment based on Fidelity in Cyber and Physical Domains, in Elhadi M. Shakshuki (ed.) *Proceedings of the 6th International Conference on Ambient Systems, Networks and Technologies (ANT 2015), the 5th International Conference on Sustainable Energy Information Technology (SEIT-2015)*, London, UK, June 2-5, 2015, pp. 996–1003, 2015.

[2] A.S. Rao, AgentSpeak(L): BDI agents speak out in a logical computable language, in *Proceedings of the 7th European workshop on modelling autonomous agents in a multi-agent world*, 1996.

[3] M.E. Bratmann, *Intention, Plans, and Practical Reason*, Cambridge University Press, 1999.

[4] J.F. Hübner and R.H. Bordini, *Jason*, http://jason.sourceforge.net/wp/.

[5] J.F. Hübner, R.H. Bordini and M. Wooldridge, *Programming Multi-agent Systems in AgentSpeak Using Jason*, John Wiley & Sons, 2007.

[6] A. Stanford-Clark and A. Nipper, *MQ Telemetry Transport*, http://mqtt.org, 1999.

[7] Eclipse Paho, http://www.eclipse.org/paho/.