Hybrid Controller Synthesis for the IoT

Arthur Gatouillat, Youakim Badr, Bertrand Massot

Hybrid controller synthesis for the IoT

Arthur Gatouillat
Univ Lyon, INSA Lyon
LIRIS, UMR5205
arthur.gatouillat@insa-lyon.fr

Youakim Badr
Univ Lyon, INSA Lyon
LIRIS, UMR5205
youakim.badr@insa-lyon.fr

Bertrand Massot
Univ Lyon, INSA Lyon
INL, UMR5270
bertrand.massot@insa-lyon.fr

ABSTRACT

The Internet-of-Things designates the interconnection of a variety of communication-enabled physical objects, and IoT-based systems and devices must operate with a deterministic behavior and respect user-defined system goals in any situation. We thus defined hybrid controller synthesis for decentralized and critical IoT-based systems relying on a set of rules to handle situations with asynchronous and synchronous event processing. This framework defines a declarative rule-driven governance mechanism of locally synchronous sub-systems enabling the hybrid control of IoT systems with formal guarantees of the satisfaction of system-wide QoS requirements. In order to prove the practicality of our framework, then applied if to a critical medical Internet-of-Things use case, demonstrating its usability for critical IoT applications.

CCS Concepts

• Computer systems organization→Dependable and fault-tolerant systems and networks • Computer systems organization→Embedded and cyber-physical systems.

Keywords

Adaptive IoT; Hybrid controller synthesis; Rule-based control

1. INTRODUCTION

The Internet of Things (IoT) paradigm designates the interconnection of a variety of communication-enabled physical objects (e.g. sensors, actuators, robots, wearable devices, etc.) integrated into wide-scale systems. In many IoT-based systems for critical applications (e.g. healthcare, traffic control, building automation, etc.), connected objects have very limited hardware and their usages require a continuous control in an always evolving physical world. More particularly, IoT-based systems and devices must operate with a deterministic behavior and respect user-defined system goals in almost any situation (e.g. device failure, loss of data packages, low power consumption, etc.).

In response to such requirements, self-adaption software frameworks were developed [10, 17, 18]. Notably, self-adaptive software systems (SAS), which designates the study of adaptation of centralized or distributed applications in response to changes in digital environments. These changes are mainly due to human interventions and systems must maintain an appropriate quality-of-service and safe behavior. Such systems are typically based on closed feedback loops to adjust their behaviors to either internal changes (such as changes in software architectures and available services), or external changes (such as changes in user loads and contextual information). In order to enable self-diagnostics capabilities for adaptive systems, monitors are implemented to sense internal and external contextual information that can be used to trigger self-adaptation strategies. These strategies aim at the guarantee of expected functional and non-functional requirements.From the IoT perspective, self-adaptation is a salient property of connected devices. It allows smart objects to be configured and adapted to extreme conditions while preserving the target system requirements in terms of automation, security and safety goals. Self-adaptation mechanisms driven by adaptation goals dynamically modify smart objects behavior. Discrete controllers for IoT-based applications have also been proposed to ensure that they evolve following predefined state transition automata [20, 21]. Nevertheless, they rely on the use of synchronous programming languages and event processing, and assume that controlled systems should satisfy the synchrony hypothesis, by which all required events should simultaneously be available to trigger a transition from one state to another. As result, the computing time to react in response to events should be negligible in comparison with the rate of events generated by the system itself [9]. Otherwise the reactive system will fail to timely respond to changes. While this hypothesis holds for small-size IoT-based systems, it becomes invalid for complex systems (i.e., systems of systems) because of the large number of generated events and the difficulty of their synchronization. This mandates the investigation of hybrid discrete controllers and adaptation strategies to handle synchronous and asynchronous event processing and to ensure secure behavior based on state transitions, in order to control large-scale IoT-based systems.

Yet another important issue in self-adaptive systems is the specification of the monitoring logic in adaptation strategies. A monitoring and adaptation logic can be expressed with either imperative programming or imperative programming approaches. The imperative programming approach, implemented in languages such as Java, C, Perl and many others specialized languages (i.e. LNT [1] and BZR [6]), defines the control of the sequence flow of instructions to be executed. However, a purely manual imperative approach for IoT-based control is not appropriate. Indeed, IoT-based systems are highly distributed, heterogeneous and might account hundreds or thousands of devices. The description of such systems in purely imperative languages will lead to a massive and difficultly maintainable codebase, resulting in the need of investigating a declarative and decentralized approach to specify the monitoring logic in self-adaptation strategies.

The main advantage of a declarative approach relies on its ability to not specify directly the sequence flow of instructions to be executed by the system in response to changes. SQL queries, functional languages, business rules and production rules are few examples of declarative programming. Particularly, rule-based controls have recently gained interest for home-automation and IoT environments with a special focus on monitoring and adaptation strategies expressed in terms of *IF-Condition-Then-Action* rules [2, 3, 19]. Conditions are logic expressions over events generated by the IoT systems and/or contextual information whereas actions are operations that must be trigger in order to self-adapt the IoT system. Rules-driven controllers in large scale and critical IoT-based systems lacks formal verification mechanisms that can avoid conflicts, dead locks and inconsistent situations. As a matter of fact, the translation of a declarative based logic into an imperative logic is necessary to cope with both the verification capabilities of imperative approaches and the expressiveness and modularity of declarative approaches.

In this paper, we propose a hybrid controller synthesis for decentralized and critical IoT-based systems. Our hybrid controller relies on a set of rules to handle situations with asynchronous and synchronous event processing. In the synchronous scheme, all

The definitive Version of Record was published in: Arthur Gatouillat, Youakim Badr, and Bertrand Massot. 2018. Hybrid controller synthesis for the IoT. In Proceedings of the 33rd Annual ACM Symposium on Applied Computing (SAC '18). ACM, New York, NY, USA, 783-790, https://doi.org/10.1145/3167132.3167219

events must simultaneously be available in a near real-time manner in order to check whether a rule's condition holds and then triggers its corresponding action. In the asynchronous scheme, events are queued upon their arrival. Once all events are available, the controller checks whether any rule's condition holds. Our hybrid controller synthesis emphasizes on non-functional properties to express its self-adaptation behaviors. Monitors on quality of service (QoS) of non-functional properties generate streams of events. In order to validate our controller, we develop an e-health continuous monitoring use-case, where IoT-based systems are used to remotely monitor risk patients. We also implement a declarative rule-driven governance mechanism of locally synchronous sub-systems enabling the hybrid control of smart homes and smart objects to guarantee the satisfaction of QoS requirements specified in service level agreements (SLA). SLAs specify end-user requirements in terms of functional and non-functional properties, such as safety, health awareness and resource awareness. By ensuring separation of concerns for adaption objectives, context monitoring and adaptation strategies, our system is able to handle changing user requirements and to redeploy the appropriate controllers if necessary.

The remaining paper is organized as follows: section 2 describes related works on self-adaptation in software systems, classical control and home-automation. Section 3 briefly introduces the e-Health use-case applied to our self-adaptation system, focusing on the safety property in the context of healthcare. Section 4 introduces the notion of layered SLAs, the global QoS ontology and our rule grammar. The implementation of our hybrid controller and its experiments are described in section 5. Eventually, research perspectives and conclusions about our work are given in section 6.

2. RELATED WORKS

The work described in this paper is at the intersection of three fields of study: self-adaptation in software systems, classical control and home-automation. In fact, software adaptation contributions study the integration of techniques enabling better software reaction to a changing digital environment. In most contributions, variations of monitor analyzer planner executor and knowledge feedback (MAPE-K) loops as detailed in [11]. In this feedback loop, monitors (i.e., sensors) are used to trigger system adaptation deployed using executors (i.e. actuators) using analyzers and planners provided with shared knowledge about the system. Because of its genericity, MAPE-K feedback loops can be adapted to deal with various self-adaptation concerns. For instance, the DYNAMICO adaptation framework [15–17] introduces a self-adaptation framework based on three distinct but communicating MAPE-K loops, each of the loop being used to control a specific aspect of software adaptation (i.e., adaptation of the monitoring infrastructure, adaptation of the control objectives and finally system adaptation). Formal adaptation frameworks based on MAPE-K loops have also been proposed in [10, 18], where adaptation strategies are modeled as plan automata. However, for both these contributions, adaptation strategies must be specified manually by the end-user, and such approach lacks expressivity and is thus difficult to apply to wide scale systems, where global adaptation strategies can be very complex. Moreover, typical DYNAMICO implementations (i.e. SMARTERCONTEXT monitoring infrastructure with the QoS-CARE/FRASCATI middleware [15]) are not relevant to distributed smart objects with limited resources.

Automated controller synthesis was studied in the control community, more particularly under the field of discrete controller synthesis (DCS). In such approach, controllers are synthesized automatically from a labeled transition system description of the functional elements of the system to be controlled and a set of control objectives (also called a control *contract*) usually specified as rules [4–6, 20, 21]. The DCS community relies on the use of synchronous languages (e.g. SIGNAL [13] or Heptagon/BZR [6, 7]) to specify target systems and control objectives. Synchronous languages enable the specification of the components of the system as concurrent labeled transition systems. Labelled transitions systems model functional and non-functional behavior using two sets, one representing the states of the system and the other representing the transitions between the states. Transitions are associated with variables over functional or non-functional properties, which are categorized as either controllable or non-controllable in the discrete controller synthesis community. Controllable variables can be triggered externally by the controller in order to verify control objectives, while non-controllable transitions can only be triggered internally and the triggering of transitions associated with non-controllable variables cannot be forced. Such techniques have been successfully used to achieve functional control of smart houses in [20]. However, the study was limited to only a few sensors and actuators, are the scalability issue was not explored.

Globally asynchronous locally synchronous systems are a category of systems which exhibit a global asynchronous behavior with local subsystems adopting a synchronous behavior [14]. Considering the IoT still is mainly built around networks of gateways controlling smaller networks of devices, this is model of computation is a good abstraction for such systems. Indeed, because the number of event in a gateway-controlled sub-network is limited because of the smaller number of devices connected to a single gateway, the synchrony hypothesis is verified. However, when a global view of the system is adopted, where numerous gateways are interconnected and communicate, the high number of event generated mandated an asynchronous approach. While this model of computation is typically used to describe very low-level systems [14], SystemJ, a higher-level system specification language, was developed [12]. Such language was used along with data-compression to specify IoT-based systems [8]. However, this language does not propose automated controller generation, which is a key aspect of controller design for the IoT. Indeed, the changing nature of IoT systems when sensors and actuators can be added or removed to the network at any moments mandates the presence of automation tools. This dynamic nature of IoT systems also calls for great maintainability, and is penalized by using centralized languages such as SystemJ.

Rule-based control strategies was widely studied by the home automation community. This field of study focuses on improving quality of life by instrumenting houses with a wide variety of sensors, actuators or gateways, in order to enable better monitoring and control of houses occupants on their environment. The ultimate goals of this community are broad, but they can be summarized as the enabling ambient intelligence to achieve better home lifecycle, and perform self-adaptation to address a variety of concerns such as energy efficiency, safety, security, comfort or remote patient monitoring [3]. More particularly, rule-based monitoring infrastructure were used to enable remote elderly adults monitoring and assistance [19] or to provide assisted decision-making in medical situations [2]. Unfortunately, such solutions typically lack any formal analysis or guarantees of non-functional properties, and potential devices' failure are not considered, which limits their use for critical applications.

Our contribution is at the center of the contributions described in this section. By adopting a hybrid approach by using asynchronous rules as a driver of discrete controller synthesis of synchronous subsystems, and by adapting software adaptation tools enabling the management of changing monitoring infrastructure and control objectives, our approach is a comprehensive answer to the challenges of wide IoT-based systems control.

3. MOTIVATION CASE STUDY

As a motivation case study, we consider the remote monitoring of a set of patients at risk for cardiac malfunction. To successfully achieve this goal, patients are equipped with a variety of body-wearable biomedical sensors that continuously monitor a wide range of biomedical signals (e.g. cardiac and respiratory activity, physical activity, electrodermal activity). Such physiological sensors can be used to detect suspicious health events that can trigger medical response if deemed necessary. Such body- wearable sensors are battery operated, and feature limited processing and storage capabilities because of the energy consumption constraints brought associated with battery operation. Additionally, the living environment is also continuously monitored, using both battery operated and continuously powered sensors. As a result, the overall system is built around several instrument houses occupied by several instrumented patients. Consequently, our adaptation framework must be scalable and modular, and adding a patient and a house in our framework must be a transparent operation.

As in most IoT-based systems, devices used to monitor patients present with strong constraints in terms of resources and communication capabilities: the computing abilities of monitoring devices are very limited (CPU frequency up to a few hundreds of megahertz), as well as storage (up to a few megabytes) and volatile memory (up to a few hundreds of kilobytes). Strong resources constraints, especially in the case of battery operated devices, has implications on the communication protocols used by these wireless objects. Indeed, in order to maintain a good battery life, the wireless communication protocols used in such objects must be lightweight, both in terms of physical characteristics and software requirements, in order to avoid excessive communication overhead.

Considering the adaptation requirements of this medical IoT-based system, this case-study is of peculiar interest. Indeed, the adaptation goal is to guarantee robust and continuous monitoring of the patients, and it is achieved considering the qualitative safety quality of service property. To satisfy this goal, the adaptation strategy considers three quality of service factors: the *resource-awareness* factor in which adaptive behavior is triggered using devices resources monitoring, the *resilience* factor, which is verified through the substitution of failed objects with sub-optimal but functional alternatives, and the *healthcare awareness* factor (i.e., the definition of patient specific monitoring threshold used to trigger medical or technical intervention).

These adaptation goals mandate the implementation of safety-enabled smart homes for each patient. In each of these smart-homes, a flock of resources-aware sensors are deployed and used to satisfy self-adaptation requirements in terms of resource consumption, resilience and external assistance. The adaptation strategy is thus based on the behavioral modification of smart-sensors by remotely modifying their configuration parameters based on a set of control objectives, specified as a set of rules, or the triggering of external medical response if monitored health-parameters exceed specified thresholds. In the following sections, we limit ourselves with a few patients, equipped with identical biomedical and environmental sensors:

- A battery-operated and multi-function *heart sensor*, including *heart rate* (HR), *heart rate variability* (HRV) and *respiratory measurement* (RR). The sensor exposes streaming services to acquire these measurements. It also monitors its battery level and can determine if it is unattached. The sensor's low-battery failsoft mode can be internally and remotely triggered to extend the battery-life. In the low-battery failsoft mode, the respiration measurements are stopped, as well as the computation of the HRV parameters. In this mode, the sensor will not be able to determine its attachment status, and the HR measurements are not streamed in real-time but they are rather sent every five minutes as an average value.
- A battery operated *electrodermal activity* (EDA) sensor, which exposes a single streaming measurement service. Similarly to the cardiac and respiratory activity sensor, it is equipped with self-battery monitoring capability and a failsoft mode that can be internally or externally triggered.
- Line powered ambient sensors such as *position sensor* (PO) and *occupancy* [YB1]*sensor* (CO). The position sensor streams the coordinates of the monitored patient within the space whereas the occupancy sensor detects the presence of the patient in their living environment. These sensors can be remotely activated and turned-off. Since they are line powered, they do not require self-battery monitoring.

These ambient and critical medical sensors are embedded in the houses and are worn by the monitored patients. Fixed and mobile gateways, such as fixed Raspberry Pis or mobile smartphones, are wirelessly connected to the sensors using low-energy protocols (i.e., Bluetooth). They also are connected to service-oriented analytical and medical framework through an Internet-related protocol (i.e., HTTP RESTful API). The self-adaptation framework is implemented in the service-oriented analytical framework and the gateways. The global control architecture is described in the next section.

4. HYBRID CONTROLLER SYNTHESIS

4.1 Global Framework Description

In order to enable self-adaptation of decentralized and critical IoT-based systems, we introduce a hybrid self-adaptation framework as illustrated in Figure 1. The framework seeks to enable declarative rule-driven governance mechanism not only with respect to changes in the ambient environment, but also changes in control objectives or in the monitoring infrastructure. The framework extends the DYNAMICO reference architecture [17] to the realm of the IoT, taking into considering sensors' resources awareness and decentralized nature of IoT-based systems. Indeed, DYNAMICO aims at designing and implementing self-adaptive software, where control objectives, adaptation strategies and the monitoring infrastructure are considered as three interacting but distinct feedback control loops. By ensuring separation of concerns for adaption objectives, context monitoring and adaptation strategies, DYNAMICO architecture and its MAPE-K control loops are able to handle changes in user requirements and to adjust itself accordingly.

The hybrid self-adaptation framework includes several components, namely the asynchronous rule engine, asynchronous controllers, synchronous monitors, and synchronous subsystems each of which comprises battery powered physical devices, line powered physical devices and gateways. These components interact through three closed loops; The higher-level loop is the *control objectives feedback loop*, which dictates the reaction of the system to changing control objectives (i.e. in our case, changing control in the SLA). *The lower-level loop* is the monitoring feedback loop, and it enables the IoT-based system with adaptation capabilities with respect to changing monitoring infrastructure. This feedback loop also infers context variables to be measured from the contracted QoS requirements as specified in the service level objectives of the SLAs, and adapts or redeploys relevant monitors with respect to updated QoS obligations. The last *feedback control loop* describes target system regulation strategies to preserve the contracted QoS.

In the IoT context, the monitoring feedback loop and the adaptation feedback loop are implemented in gateways, measuring and controlling a set of connected sensors and actuators. The control objectives adaptation feedback loop, because of its higher-level nature, is implemented in centralized servers, controlling distributed synchronous controllers.

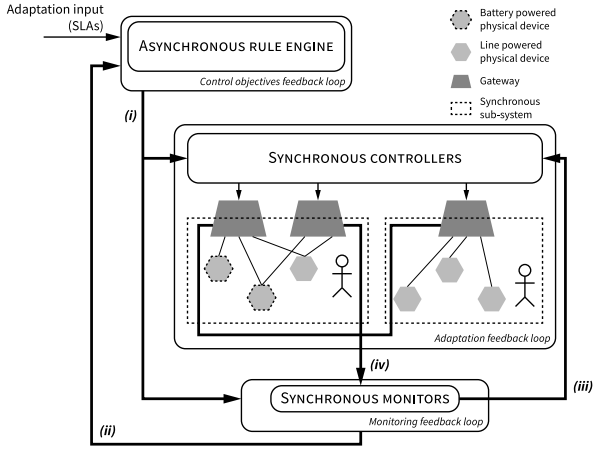

Figure 1. Hybrid self-adaptation framework

As displayed in Figure 1, the three feedback control loops are connected through four distinct interactions. The first interaction (*i*) holds between the control objectives feedback loop and the adaptation and monitoring feedback loops and describes the deployment of new control and monitoring strategies that are triggered by changes in control objectives (QoS monitoring).

For example, the resource-aware adaptation in our use-case deals with the sensors' battery levels. Accordingly, the monitoring and control infrastructure will measure and adapt taking into account battery level variables. These variables are thus used to trigger adaptations within the adaptation feedback loop. Battery level monitors must consequently be implemented in the monitoring feedback loop.

The second interaction (*ii*) describes the communication between the monitoring feedback loop and the control objectives feedback loop. In an asynchronous hybrid context, this interaction is triggered when the monitoring feedback loop detects the necessity of a change in the control objectives. For instance, if a battery-operated sensor becomes unresponsive because of an emptied battery, an eventual control strategy should be applied to infer the health status of the monitored patient from environmental sensors.

The third interaction (*iii*) holds between the monitoring feedback loop and the adaptation feedback loop and is used when abnormal monitoring events occur without mandating changes in the control objectives. This interaction typically performs predictive adaptation, where preemptive adaptation actions are taken in order to avoid later adaptation of critical situations. For instance, in our medical IoT use-case, this interaction is active when a monitor detects a higher than usual long-term battery drain[YB2]. In order to keep the system in a quasi-optimal non-functional state, the triggering of the low-battery failsoft mode occurs at a higher battery percentage, in order to prevent the sensor undergo critical battery failure.

The last interaction (*iv*) takes place between the adaptation feedback loop and the monitoring feedback loop. It represents streams of captured events from the internal context of the monitoring feedback loop. It also verifies the monitoring system consistency after an adaptation occurred. For example, it checks if sensors subsided to a failed sensor are in a functional state, to guarantee constant QoS across the whole adaptation process.

The articulation of these components through feedback loops is straightforward: the asynchronous rule-engine triggers both adaptive behavior or controller resynthesis if a control objective, and thus a control rule, changes. The newly synthesized controller is then deployed to the appropriate gateways at runtime. The system thus self-adapts without any execution interruption.

As described in Figure 2, the controller synthesis self-adaptation process relies on three ontologies, namely the SLA ontology, the failure ontology and the expert knowledge ontology. In this figure, the *objectives analyzer*, *objectives controller* and *adaptation analyzer* denotes elements of the objectives MAPE-K feedback loop and the adaptation MAPE-K feedback loop. These elements are embedded in the global MAPE-K loops described in Figure 1, and can be seen as standard adaptation-enabling elements.

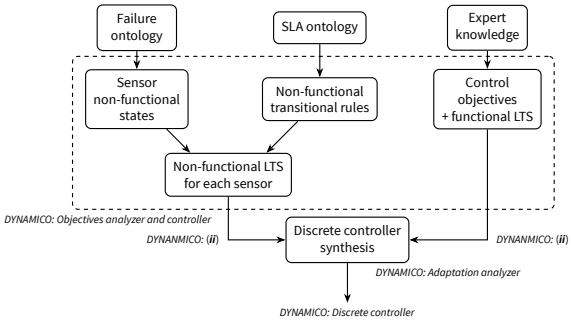

Figure 2. Controller synthesis process

The integration of the DYNAMICO-based self-adaptation reference architecture with an asynchronous rule-based specification of control objectives and synchronous discrete controller synthesis enables safe control of critical IoT-systems. In the following sub-sections, we introduce every component from a top-down perspective using layered SLAs. We also introduce the QoS ontology and the cardiac sensor failure ontology and demonstrate how the discrete synchronous specification can be formally validated with respect to the contracted QoS requirements.

The hybrid nature of our self-adaptation framework comes into play when considering the global system: the asynchronous rule-engine, because of the mass of system-generated event, are queued upon arrival and are processed using a first-in-first-out strategy. Once the buffered events are processed asynchronously by the rule engine, synchronous action can be sent to the controlled sub-systems. The subsystem monitoring occurs synchronously, because of the small size of the considered sub-systems.

4.2 Multi-Level SLA Adaptation

The complexity and the distributed nature of IoT systems mandate a hierarchical separation of SLAs to accurately represent functional and non-functional guarantees at different granularity levels. Since our use-case describes human centric IoT applications, an SLA is required to capture expected level of services by patients and medical staffs. We describe how the SLAs are scattered throughout the IoT-based system.

System-level SLAs designate contracts between end-users and services providers at the system level. Indeed, end-users do not need finer granularity to specify their requirements at sensors and actuators levels. Instead, they express system-level objectives and specify global functional and non-functional requirements. System-level SLA are then rewritten into fine-grained SLAs at the device level for further analysis.

Device-level SLAs represent guarantees provided by manufacturers about their devices' functional and non-functional

properties. These SLAs are closely related to physical and operational device characteristics.

It is worth noting the difference between smart devices and simpler devices when considering device SLAs. Indeed, smart devices exhibit capabilities to interact with their environments and adjust their configuration parameters. As a result, smart devices can be reconfigured even if their SLAs change over time. However, SLAs of simple devices remain static and their SLAs can only be slightly modified. Simple devices are thus black-boxes designed by manufacturers to have predefined functional and non-functional properties that cannot be reconfigured over time. The finer granularity of device-level SLAs enables optimization and reasoning at the system scale by providing precise system descriptions.

Human-level SLAs specify personal characteristics differentiating users (i.e., patients) in human-centric IoT systems. The presence of human in the control loop justifies the accurate description of the system properties (i.e., biological properties) being controlled or monitored. These system properties can greatly vary from one individual to another in terms of various pathologies that can impact physiological parameters.

Expressions in each of these SLAs can be mapped to QoS factors, as described in the ontology in Figure 3. For instance, the *resource awareness* QoS factor is typically a device-level SLA because of resources variability between devices (e.g. continuously powered sensors do not have the need for a low battery SLO obligations, while battery operated sensors do). The resilience QoS factor is typically a system level SLA, where resiliency is specified at the system level. For example, if a sensor is failing, it is then subsided by other sensors in order to compensate for the loss of information caused by sensor malfunction. However, the health-awareness QoS factor is a human-level SLA. In our use-case, the cardiac activity is monitored in order to detect and prevent cardiac malfunctions. However, cardiac malfunction is associated with different diseases producing different effects on heart activities. In order to accurately detect a specific heart malfunction, a corresponding QoS factor must thus be adapted for each monitored patient, leading to the establishment of a human-level SLA.

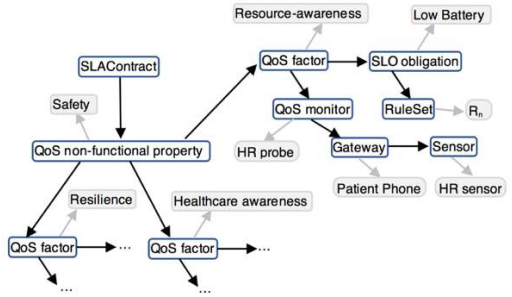

Figure 3. QoS ontology

From the self-adaptation perspective, we use the SLA as a system input. Rules and requirements specified in SLAs, especially, the set of rules provided as service level objectives (SLO) will be used to generate controllers, guaranteeing that the system behaves according to the SLA and adapts itself with respect to environmental changes.

In order to express system-wide requirements to be included in SLAs, we propose a rule-based language to specify control objectives. In the following sub-section, we present the rule grammar and it semantic. We then explain the generation of discrete synchronous controllers and their coordination with asynchronous controllers.

4.3 Modeling Rule-Based Control Objectives

Rules to specify control objectives follow the Event-Condition-Action (ECA) pattern. They are defined as a set of asynchronous rules each of which is activated in response to the evaluation of a condition (or a set of conditions) by executing the corresponding action. Rules describe adaptation strategies based on events generated by sensors and captured by monitors, related to QoS factors and predefined SLOs. Because rules describe adaptation strategies with respect to device-related monitored variable, they can be considered as control input and objectives. A rule has the following syntax:

```
Rule name
      ON event
      IF conjunctions of condition are found to be true
     DO actions are executed
End
```

The basic structure is a list of conditions and actions. A condition denotes a constraint or a filter, acting on data and events in a specific domain of interest. Data and events are generated by sensors, actuators or object instances (i.e., complex data structures). Once the condition holds, its corresponding action is executed, taking the matching data or events as parameters.

Action refers to the execution of device services, taking as parameters events and data specified in the control strategy. The example below illustrates a rule. Its syntax follows the rule language grammar as illustrated in Figure 4.

```
SENSOR ecgSensor TYPE ECGSensorType
SENSOR posSensor TYPE PositionSensorType
Rule "Sensor-Low Battery"
ON batteryLevelLow
  IF
       $e: ECGSensorType(batteryLevel < 20%)
       $s: posSensor(batteryLevel < 10%)
  DO
       $e.setFailSoftMode();
End
```

The first line declares an instance of a device, called `ecgSensor`, of type `ECGSensorType`. Similar to objects and classes in the object-oriented paradigm, the device type is a common data structure of similar devices. Each device is described by a set attribute value pairs. Attributes may hold information about devices such as characteristics, contextual information, configuration parameters, and their sensing data from the physical environment.

As illustrated in the example, the rule starts with the keyword `Rule` followed by a string, denoting the rule's name. The line in the left-hand side of the rule is the conjunction of logical predicates, each of which is written on a separate line. The predicate can be applied on individual device instances or on all instances of a given device type, defined with the keyword `TYPE` in the rue-based language (see Figure 4). The predicate works as a function with a condition (called also *filter condition*) as its input parameter. The filter condition is a logical expression on device attributes. The logical operator "`And`" between predicates is explicitly omitted. In the before mentioned example, there are two predicates:

- The *`ECGsensorType(filter-condition)`* predicate applies the *filter_condition* on all device instances, having the ECG sensor as their type. The filter condition is not more than a logical expression on ECG senor attributes. All ECG sensors that have their *`batteryLevel`* *attributes less than 20%* are selected, making the rule's condition a non-empty set of device instances (i.e., true). As a result, the corresponding rule's action is thus executed. In the first rule, the service, `setFailSoftMode()`, for example, is executed to set the

sensor performance into the failsoft mode. The predicate here is applied on all instances of a given device type.

- The `posSensor`*(filter-condition)* predicate applies the *filter_condition,* `batteryLevel` < 20%, on the `posSensor` instance of the `posSensor`'s type. The predicate here is applied on an individual device instance.

In the first rule, the *$* prefix is called the *bind operator*, which binds a variable to a device type (i.e., `$s: posSensor()` binds the `s` variable to the `posSensor` instance) or to a device instance (i.e., `$e: posSensor()` binds the `e` variable to the `posSensor` instance).

In order to trigger the rule, we need simultaneously all available device instances of `ECGSensorType` and the `posSensor` instance. Between `ECGsensorType()` and `posSensor()` predicates, an implicit AND operator is used to create the conjunction of predicates. Therefore, the rule's condition is activated when there is at least one ECGSensor instance with an attribute `batteryLevel` less than 20% AND the `posSensor` sensor `batteryLevel` attribute that is less than 10%. The rule's action is then triggered to adapt the `ECGsensor` sensors in response to low level of battery power.

In sum, predicates on device types are particularly useful to specify adaptation strategies at the system level while the adaptation must only be triggered in relevant situations. Predicates on device instances allow a fined grained control of adaptation at the device level.

Global variables

In order to interact with contextual data that is not in device attributes, we introduce the keyword `GLOBAL` to declare a variable and bind it to the environment surrounding devices. Global variables can refer to external services, cached data in memory or parameter values for setting up the rule engine at runtime. For example, the following statement declares the global variable BobHome of type Home, which is declare as an Object.

```
GLOBAL BobHome Home;
```

In our context, global variable can be used to save configured device states so that, if a controller resynthesis occurs, the newly synthesized controller can be deployed and starts its execution with the right sensor state.

Rule based language

```
<ECA-system> ::= <variables> <objectives> <adaptation> <devices>
<variables> ::= <variable> | <variable> <variables>
<variable> ::= GLOBAL <variable_name> | GLOBAL <object_name>

<objectives> ::= <objective> | <objective> <objectives>
<objective> ::= QUALITY <quality_name> ON <SLA_expression>

<rules> ::= <rule> | <rule> <rules>
<rule> ::= RULE <rule_name> ON < quality_name> IF < predicates>
                            DO <adaptations>
<adaptations> ::= <action_name> | <action_name> <adaptations>
                 |<action_name> ALTERNATE(<action_name>)

<predicates> ::= <predicate> | < predicate > [AND] < predicates >]
<predicate> ::= <device_Type>(<filter>)
               |<sensor_name>(<filter>)
               |<actuator_name>(<filter>)
               |<event_name>(<filter>)

<filter> ::= <sensor_name>.<attribute_name> <operator> <value>
           | <actuator_name>.<attribute_name> <operator> <value>
           | <deviceType>.<attribute_name> <operator> <value>

<action_name> ::= <sensor_name>.<service_name>(<parameters>)
                | <actuator_name>.<service_name>(<parameters>)
                | <gateway_name>.<service_name>(<parameters>)

<devices> ::= <device> | <device> <devices>
<device> ::= <sensor> | <actuator> | <gateway>
<sensor> ::=  SENSOR <sensor_name> TYPE <object_name> <events>
<actuator> ::= ACTUATOR <actuator_name>] TYPE <object_name> <events>
<gateway>::= GATEWAY <gateway _name>] TYPE <object_name> <events>

<events> ::= <event> | <event>,<events>
<event> ::= Event <event_name> <attributes>
<object> ::= OBJECT <object_name> ATTRIBUTES <attributes>
                                  [SERVICES <services> END]
<attributes> ::= <attribute>| <attribute>  <attributes>]
<attribute> ::= VAR <attribute_name> [= <attribute_value>]

<services> ::= <service> | <service> <services>
<service::=SERVICE <service_name>(<parameters>) [RETURN(<parameters>) END]

<operator>::= > | < | >=|<=| == | != | in
<parameters>::=<parameter>|<parameter>, <parameters>
<parameter>::= <object_name>.<attribute_name>
<SLA_expression> ::= <object_name>.<attribute_name> <operator> <value>
```

Figure 4. The Grammar of the Rule-based Language

In Figure 4, we present the rule based language grammar. It allows declaring system objectives, adaptation strategies, devices (sensors, actuators and gateways) each of which is defined as a generic object, containing a data structure (a set of attribute value pairs), services and generated events. The language is not limited to rule description but it also allows the complete description of the feedback loops and the feedback loop interactions as defined in previous section. Indeed, if we consider the resource-aware battery-derived adaptation, the set of rules describing the adaptation can be written as follows:

```
//Control objective
QUALITY LowBattery ON BatterySaving

//Control rule1
ON LowBattery IF ECGSensorType(BatteryLevel < 20%) DO
TRY ECGSensorType.setMode(LowBatteryAdaptation)
```

The control objective describes the objectives feedback loop, where the reference control objectives are provided after the *IF* statement and the *QUALITY* statement is used to feed the monitoring feedback loop and the adaptation feedback loop.

The control rule describes the interaction between the adaptation feedback loop and the monitoring feedback loop. The statement after the *IF* describes the monitor that must be implemented in the control feedback loop, and the quality is the reference context input of this feedback loop. The statement after the *DO* describes the adaptation mechanism that occur in the adaptation feedback loop, and more specifically in the adaptation feedback loop controller. In the context of discrete controller syntheses, the rule-based language defined in this section will be used to provide the control objectives to the controller synthesizer. This rule based definition of the objectives allows for greater expressiveness, which allow external users to easily specify their desired control objectives.

The main advantages of using rules in the context of IoT does not only come from a small group of rules, but from a large, ever-changing group of rules that define the behavior of a complex system that requires studious development to maintain it operational when using imperative programming languages.

The rule-based language also enables the formulation of control objectives and strategies with respect to desired service-level objectives. Objectives are later used to synthesize synchronous controllers and deploy them in the gateways in order to control a specific sub-set of devices.

4.4 Synchronous Sub-Systems Modelization

The prime modeling framework for discrete controller synthesis relies on labeled transition systems (LTS) to model to-be-controlled sub-systems. Discrete controller synthesis typically uses synchronous programming languages embedded with control contract specifications to build correct-by-construction controllers. The strength of discrete controller synthesis stems from the produced code that is assumed to be correct. In fact, the code generation will fail if inconsistencies are detected, either in the control rules or in the models of the controlled sub-systems. The discrete controller synthesis thus makes possible to avoid further formal analysis, and thus saves development time.

Commonly speaking, labeled transition systems are defined as a tuple $(S, L, \rightarrow, s_{in})$, with S a set of states, L a set of transition labels, $\rightarrow \subseteq S \times L \times S$ a transition relation between states, and s_{in} an initial state. The set of transition labels is defined as $L = (events, actions, \backslash)$, where $\backslash \subseteq events \times actions$.

In our context, we use two LTS to represent a sensor: a functional LTS, describing the relationship between the different functional states, and a non-functional LTS which describes the relationship between the objects non-functional states. The two LTS are synchronized using the following syntax: the statement "$e \setminus a$" can be interpreted as the control of event e by service a when event e during the firing of the transition.

As specified earlier, variables in the context of discrete controller synthesis are divided into two distinct sets: the controllable variables and the non-controllable variables. In our context, we chose to model controllability using the '$' character. Particular attention to this variable separation problems must be paid when modeling the various devices included into the adaptation framework, because the correctness of the synthesized controller directly depends on what is defined as controllable and non-controllable.

Figure 5 introduces the LTS model of the cardiac and reparatory sensor used in our case-study. The model captures both the functional and non-functional evolution of the sensor. In this example, local and remote service calls, modeled under the form "*$e.service_call()*," are considered to be controllable. However, model inputs such as battery level (abbreviated as '*batt*' in Figure 5), or the unattached flag (abbreviated as '*unatt*,' and is true when the sensor auto-detects that it is unattached from the monitored patient) are defined as non-controllable. This is because these variables are related to the physical domain, and the physical world typically behave unpredictably. As a rule of thumb, all the external and physical monitored variables should be considered as uncontrollable because of the unpredictable nature of the physical world.

On the left side of the model, we introduce the model inputs, which are used in contract-based discrete controller synthesis as variables to be monitored. In our context, every model input must thus be assigned to a dedicated monitor in the monitoring feedback loop.

The model outputs are presented on the right side (see Figure 5). For conciseness purposes, outputs are abbreviated as functional and non-functional states, meaning that all states of the model are exposed to the controller synthesizer in terms of a set of mutually exclusive Boolean flags. Output are set to be true when the sensor is in the corresponding functional or non-functional state. Such outputs are used in control contracts, which are specified as first order logic rules in most of the synchronous languages enabled with discrete controller synthesis capabilities.

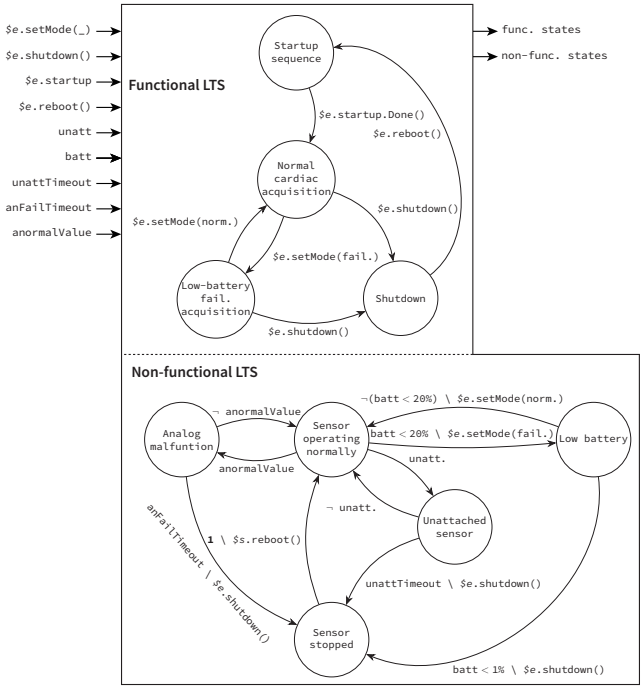

Figure 5. Cardiac sensor LTS model

In order to assist end-users with the specification of devices in IoT-based systems, we propose a failure ontology detailing non-functional failure states than can typically occur in sensors. As illustrated in Figure 6, the ontology is divided into four main symptoms of failure.

The first symptom of failure is the detection of abnormal values being measured or streamed to the gateway. Such abnormal values can either be caused by an analog malfunction of the sensor, as the measurements are converted into numerical values using analog-to-digital converters. Such incorrect values can also be caused by a failing digital section. Indeed, if signal processing is embedded into the sensor, a signal processor malfunction could cause incorrect calculations, resulting in incorrect values. These are the two failure symptoms. However, because of the critical nature of a cardiac malfunction event, it was included into this failure symptom section, to illustrate the fact that controllers need to distinguish between critical incorrect (i.e. resulting from heart malfunction) values and expected incorrect values (i.e. resulting from sensor malfunction). The definition of what is considered incorrect values is defined by expert knowledge about the monitored biomedical process. Incorrect values are typically specified using thresholds.

The second symptom of failure is sensor unattachment. Indeed, because we consider wearable biomedical sensors, which users typically wear on their body using wet electrodes that are subject to decay over time, sensors can become unattached, resulting in incorrect biomedical process measurement. However, sensors can be equipped with self-diagnostics capabilities and be able to detect unattachment. This event however mandates external intervention,

as sensors must be reattached either by the monitored patient or an external medical worker.

A low battery is the third symptom of failure, and two situations can result from such event. Either the sensor is equipped with a battery-saving failsoft mode (where the sensor typically streams less precise values, or where the stream occurs at a lower rate), or the battery is drained until total battery failure, where the sensor is stopped internally in order to protect the battery. This last case mandates external intervention, so it must be delayed as late as possible. Because of the loss of data quality causes, the battery failsoft mode must not occur too early during the battery discharge process, and a compromise must be found between lower data quality and battery life. Such compromise can be determined using externally provided expert knowledge.

Finally, the last symptom of failure is the interruption of the communication between the sensor and the gateway. This failure can have three causes. The first possible cause is that the sensor is out of range. Such failure can easily occur, especially if fixed gateways are used. Indeed, because low-power wireless communication protocols feature limited range (usually up to a few tens of meters), if a fixed gateway is used, the monitored patient can easily get out of the communication range. The next source of failure is a radio-dedicated component malfunction. As radio communication of connected objects is typically implemented onto specific radio integrated circuits, a malfunction of such chip can cause a loss of connectivity. The last source of failure is a gateway malfunction, which can also cause loss of connectivity between sensors and the gateway.

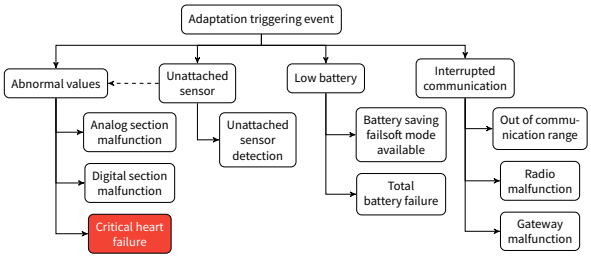

Figure 6. Cardiac sensor failure ontology

Using this ontology, models can be developed accounting for all the identified failure symptoms, and adaptations strategies can be derived for all the failure causes. This enables comprehensive adaptation process with guarantees that is accounts for all identified failure sources, thus providing robust global system behavior. Implementation of all the elements (from the rule engine to the discrete controller synthesis) of our hybrid self-adaptation framework for the IoT is detailed in the following section.

5. IMPLEMENTATION

In order to validate our hybrid self-adaptation framework and self-adaptation strategies, we developed a prototype using existing languages, controller synthesis and rule engines. The asynchronous rule engine is implemented using the asynchronous capabilities of the Drools rule engine [22]. This rule engine was developed as a business rule engine with a web-based control interface, along with an Eclipse plugin for further development. Rule evaluation is based on the Rete algorithm, and is distributed with an open-source license. We have developed a domain specific language compiler based on our language using Xtext, which provides us with a fully-featured and statically-typed programming language. The compiler outcome produces rules as expected by the Drools rule engine. By such, Drools support enable our self-adaptation rule-based language to be easily managed and monitor a massive and potentially changing set of rules. Such characteristic is suitable for scalable IoT purposes. The only limitation being the resources available for rule evaluation. Since this tool is implemented on external servers, the resources available are virtually unlimited when compared with devices' resources.

The discrete controller synthesis is implemented using the Heptagon/BZR synchronous language for controller synthesis [6]. In this language, objects are modeled using a textual representation of labelled transition systems. The discrete controller synthesis is realized with respect to control contracts specified by using a simple grammar. Three contract keywords are defined: *with*, *assume* and *enforce*. The keyword *with* is used to specify the set of controllable variables that can be used by the controller for self-adaptive behavior, while the keyword assume describes a set of initial *assumption* to assist the controller synthesizer and to avoid certain locking behaviors and the keyword *enforce* is used to provide the controller synthesizer with a set of rules that the global synchronous system must observe. Such rules are provided as first order logic statements, and such statement is determined using a first logic translation of the business process rules specified in Drools. It is worth noting that it is not necessary to translate all rules specified in Drools, but only lower level rules that are relevant to a specific monitoring context. These rules are enabled in gateways to support adaptive behaviors.

6. CONCLUSION

In this paper, we present a hybrid controller synthesis framework for critical IoT systems. Our system presents self-adaptive behavior with respect to changing control objectives, evolving monitoring infrastructure and dynamic internal and external context by adopting separation of concerns and defining three distinct but communicating control loops: the objective feedback control loop, the monitoring feedback control loop and eventually the adaptation feedback control loops. This framework is equipped with an asynchronous rule engine and synchronous discrete controller synthesis capabilities in order to provide a hybrid self-adaptation framework for the IoT. The presence of discrete controller synthesis enables automatic controller generation from formally correct synchronous programming languages, providing functional and non-functional guarantees for critical IoT-based systems.

7. REFERENCES

[1] Abid, R. et al. 2017. Asynchronous synthesis techniques for coordinating autonomic managers in the cloud. *Science of Computer Programming*. 146, (Oct. 2017), 87–103.

[2] Augusto, J.C. et al. 2007. Enhanced healthcare provision through assisted decision-making in a smart home environment. *2nd Workshop on Artificial Inteligence Techniques for Ambient Inteligence* (2007).

[3] Bonino, D. and Corno, F. 2010. Rule-based intelligence for domotic environments. *Automation in Construction*. 19, 2 (Mar. 2010), 183–196.

[4] Cano, J. et al. A case study in safe design of ECA rules for IoT.

[5] Cano, J. et al. 2014. Coordination of ECA Rules by Verification and Control. *Coordination Models and Languages*. E. Kühn and R. Pugliese, eds. Springer Berlin Heidelberg. 33–48.

[6] Delaval, G. et al. 2010. Contracts for modular discrete controller synthesis. *ACM Sigplan Notices* (2010), 57–66.

[7] Delaval, G. et al. 2013. Integrating discrete controller synthesis into a reactive programming language compiler. *Discrete Event Dynamic Systems*. 23, 4 (Dec. 2013), 385–418.

[8] Eliasson, J. et al. 2015. Towards industrial Internet of Things: An efficient and interoperable communication framework. (Mar. 2015), 2198–2204.

[9] Gamatié, A. 2010. Synchronous Programming: Overview. *Designing Embedded Systems with the SIGNAL Programming Language*. Springer New York. 21–39.
[10] Iftikhar, M.U. and Weyns, D. 2014. Activforms: Active formal models for self-adaptation. *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems* (2014), 125–134.
[11] Kephart, J.O. and Chess, D.M. 2003. The vision of autonomic computing. *Computer*. 36, 1 (Jan. 2003), 41–50.
[12] Malik, A. et al. 2010. SystemJ: A GALS language for system level design. *Computer Languages, Systems & Structures*. 36, 4 (Dec. 2010), 317–344.
[13] Marchand, H. et al. 2000. Synthesis of Discrete-Event Controllers based on the Signal Environment. *Discrete Event Dynamic Systems*. 10, 4 (2000), 325–346.
[14] Muttersbach, J. et al. 2000. Practical design of globally-asynchronous locally-synchronous systems. (2000), 52–59.
[15] Tamura, G. et al. 2013. Improving context-awareness in self-adaptation using the DYNAMICO reference model. *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems* (2013), 153–162.
[16] Villegas, N.M. et al. 2011. A framework for evaluating quality-driven self-adaptive software systems. *Proceedings of the 6th international symposium on Software engineering for adaptive and self-managing systems* (2011), 80–89.
[17] Villegas, N.M. et al. 2013. DYNAMICO: A reference model for governing control objectives and context relevance in self-adaptive software systems. *Software Engineering for Self-Adaptive Systems II*. Springer. 265–293.
[18] Weyns, D. et al. 2010. FORMS: a formal reference model for self-adaptation. (2010), 205.
[19] Yuan, B. and Herbert, J. 2014. Context-aware hybrid reasoning framework for pervasive healthcare. *Personal and Ubiquitous Computing*. 18, 4 (Apr. 2014), 865–881.
[20] Zhao, M. et al. 2014. Discrete Control for Smart Environments Through a Generic Finite-State-Models-Based Infrastructure. *Ambient Intelligence*. E. Aarts et al., eds. Springer International Publishing. 174–190.
[21] Zhao, M. et al. 2013. Discrete Control for the Internet of Things and Smart Environments. *8th International Workshop on Feedback Computing, San Jose, CA, USA, June 25, 2013* (2013).
[22] https://www.drools.org/. Accessed on Sept. 10th 2017.