Models and Analysis for User-Driven Reconfiguration of Rule-Based IoT Applications

Francisco Durán[a], Ajay Krishna[b], Michel Le Pallec[c], Radu Mateescu[b], Gwen Salaün[d]

[a] *ITIS Software, University of Málaga, Málaga, Spain*
[b] *Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP[1], LIG, 38000 Grenoble, France*
[c] *Nokia Bell Labs, Nozay, France*
[d] *Univ. Grenoble Alpes, CNRS, Grenoble INP[1], Inria, LIG, 38000 Grenoble, France*

Abstract

Introduction. The Internet of Things consists of devices and software interacting altogether in order to build powerful and added-value services. One of the main challenges in this context is to support end users with simple, user-friendly, and automated techniques to design such applications. IFTTT-style rules are a popular way to build IoT applications as it addresses this challenge. *Problem statement.* Given the dynamicity of IoT applications, these techniques should also consider that these applications are in most cases not built once and for all. They can evolve over time and objects may be added or removed for several reasons (replacement, loss of connectivity, upgrade, failure, etc.). There is a need for techniques and tools supporting the reconfiguration of rule-based IoT applications to ensure certain correctness properties during this update tasks. *Methodology.* In this paper, we propose new techniques for supporting the reconfiguration of running IoT applications, represented as a set of coordinated rules acting on devices. These techniques compare two versions of an application (before and after reconfiguration) to check if several functional and quantitative properties are satisfied. This information can be used by the user to decide whether the actual deployment of the new application should be triggered or not. *Contributions and results.* The analysis techniques have been implemented using encodings into formal specification languages and verification is carried out using corresponding analysis frameworks. All these techniques for designing new applications, analyzing the aforementioned reconfiguration properties, and deploying the new applications have been integrated into the

[1]Institute of Engineering Univ. Grenoble Alpes

WebThings platform and applied on real-world examples for validation of the approach.

Key words: IoT, Reconfiguration, Verification, Web of Things.

1. Introduction

The Internet of Things (IoT) is a network of physical devices and software entities that interact together to fulfil an overall objective of providing added-value services. Consumers build IoT applications by choosing a set of candidate objects (devices) and specifying the interactions between these objects. These applications however are not meant to be built once and for all. Over time, the consumer needs may change, and also the objects undergo wear and tear. Therefore, the applications would need to be redesigned and redeployed. The process of redesign (reconfiguration) may involve addition or removal of objects to modify the services, replacement of worn out objects, and specifying new or redefining existing interactions between the objects.

Frameworks such as IFTTT [1], Zapier, and WebThings [2], simplify the process of building IoT applications by using a simple yet powerful programming paradigm of "*IF event(s) THEN action(s)*" rules, also known as Event-Condition-Action (ECA) rules. With these rules, if an event, or set of events, is raised, then an action, or set of actions, is triggered. *E.g.*, the rule *IF motion-on THEN light-on*, specifies that if presence is detected by a motion sensor object, then the connected light in the room is turned on. One can specify a bunch of rules involving a set of objects, and these rules will execute the actions whenever the corresponding events in the objects are triggered. Although the IFTTT-style rules simplify the design of applications, these frameworks provide limited support for reconfiguration of applications. Reconfiguration is typically a non-automated process led by the users and it might result in erroneous applications due to complexity of rules, concurrent nature of the application, or simply a mistake from the user. There are no efficient mechanisms to check whether the reconfiguration is satisfactory or whether the redesign leads to inconsistent states in objects. Updating an IoT application should be carried out with specific care because this may induce additional costs or hazardous situations. As an example, in a crop irrigation application, a negligent update of the application (e.g., rebooting the application after reconfiguration to start from the beginning) may result in the delivery of additional pesticides to the crops since the information on the amount of pesticide already delivered is not taken into account. This situation implies additional costs and possible contamination of the crops. Reconfiguration could be dangerous too in the context of a smart home for the elderly and domiciliary care. Simply unexpectedly switching off lights or heaters, or closing doors may put lives at risk.

In this work, we propose new techniques for supporting the reconfiguration of rule-based IoT applications. Our proposals encompass different aspects of reconfiguration. First, we consider a composition language that allows writing of more advanced IoT applications by providing basic constructs for the composition of rules, such as the sequential execution of rules, the choice between several rules, the concurrent execution of several rules, or the repetition of rules. These advanced application are inherently more complex, so to help users avoid reconfiguration errors, we propose various property-based verification. Finally, we enable the process of application redesign to deployment to be as automated as possible by relying on an execution platform, such as WebThings, to deploy and effectively run the application.

The objective of the property-based verification is to analyse whether the proposed reconfiguration preserves the consistency of the application, i.e., the application can resume after reconfiguration from where it was before interruption. It also proposes techniques to quantitatively compare the current application and reconfigured application in terms of operating costs. More precisely, an IoT application is described in this work using a set of objects and a rule-based composition expression that specifies how the objects interact together. Given a current and a new IoT application as well as a global state of the current application, the reconfiguration property (called *seamless* reconfiguration) determines if the given global state is reachable for objects remaining in the new application. If this is the case, it means that replacing the current application by the new application can be achieved seamlessly from the user's perspective. We also define two additional properties called, *conservative* reconfiguration and *impactful* reconfiguration, to check whether all former behaviours can still be executed in the new application, and whether all new behaviours can be executed after reconfiguration, respectively. These three properties focus on the reconfiguration of the application given a global state. Complementary to these properties, functional properties of interest can be verified on the application. These properties are analysed for all possible executions of the application independently of any global state. Our approach also allows the verification of quantitative properties (probability of occurrence of a specific event, or the cost of the application), to help users in deciding whether to deploy the new application.

As far as the implementation of the proposals is concerned, we have extended the Mozart [3] tool, which was built on top of the WebThings [2] platform to support the design and deployment of IoT applications described using compositions of rules. More precisely, this Mozart extension consists of three new components. At the design level, a new UI allows the user to specify the new application. Once the new application is specified, a verification component is called to check that reconfiguration properties are satisfied. To check the functional properties, we provide an encoding into rewriting logic, and specifically into its implementation in the Maude framework [4], and we rely on Maude tools to generate and traverse all possible executions of both applications. To analyse quantitative properties, we provide an encoding into the LNT [5] specification language and we rely on the CADP [6] analysis tools for probabilistic model checking and cost analysis. Finally, deployment of the new configuration is achieved preserving the consistency of the remaining objects. Note that the only step of our approach requiring human intervention is the design of the new version of the application. The two other steps (verification and deployment) are fully automated by several tools we implemented and validated on several examples.

The main contributions of this paper are summarised as follows:

- a behavioural model for IoT applications based on an IoT standard (Web of Things);
- a set of functional and quantitative properties for supporting the reconfiguration of IoT applications, including properties related to execution time and cost of the reconfiguration process;
- automated analysis techniques for verifying these properties on IoT applications, including encodings into rewriting logic and into the LNT process algebraic specification language, which are used to verify qualitative and quantitative properties using, respectively, the Maude system and the CADP toolbox;
- a tool is provided, with a friendly GUI and connections to Maude and the CADP verification toolbox in order to support the verification of both functional and quantitative properties;
- the approach has been applied to several case studies for validation purposes; and
- the paper also presents our tool support based on the WebThings platform, and validation of the approach on real-world use-cases in the context of smart homes.

The rest of this paper is organised as follows. Section 2 introduces the model of objects and the rule-based composition language. Section 3 defines properties of interest for reconfiguration of ruled-based IoT applications. Section 4 presents the encoding of IoT applications into rewriting logic and the verification of properties using Maude. Moreover, it covers the encoding of the applications as a process algebra specification for performing quantitative analysis. Section 5 describes the extensions of the WebThings platform to support reconfiguration and also covers the evaluation of the proposals. Section 6 surveys related work and Section 7 concludes the paper.

2. IoT Models

An IoT application consists of a set of IoT objects or *things* interacting all together to fulfil a certain overall goal. To illustrate the type of situations we consider in this paper, and to illustrate the different concepts, let us consider the following running example.

Example (A Smart Home). Our smart home is ready for a hot summer. While the resident is away, the windows are closed and appliances are off. When the resident enters the home (door is open), a light in the passageway is turned on. If the temperature is greater than 27 degrees Celsius, then the light turns red indicating a hot summer day. When the resident moves to the living room (as detected by the sofa), the window blinds are opened. If it is a pleasant day, then the windows are opened, and when it is too hot, a fan in the room is turned on.

2.1. Behavioural Models

Given the heterogeneity of devices and platforms existing in the IoT ecosystem, there is a need for a standard description format for objects. Proposals such as Constrained RESTful Environments [7], OpenWeave [8], and the Thing Description in the Web of Things (WoT) framework [9] are available, but there is no widely accepted standard yet. Therefore, although inspired by the Web of Things description model [9], in this work, we prefer to rely on an abstract model for describing objects. In this abstract model, an IoT object is defined by a set of properties — a property is a pair (*identifier*, *value*) — and by a behavioural model. For the sake of simplicity, an IoT object is represented only by its behavioural model in the rest of this paper.

To precisely modelling the ordering of events/actions in an object, behavioural descriptions are represented as Labelled Transition Systems (LTSs). We use a question mark (?) or an exclamation mark (!) to indicate that the object is receiving or emitting a value from/to its environment, respectively.

```
R1: IF door(open,true) THEN light(on,true)
R2: IF thermo(temp-warm,true) THEN light(colour,red)
R3: IF sofa(presence,true) THEN window(on,true)
R4: IF thermo(temp-warm,true) THEN fan(on,true) ∧ window(open,true)
R5: IF thermo(temp-warm,false) THEN window(open,true)
R6: IF door(open,false) THEN light(on,false) ∧ fan(on,false) ∧
    window(open,false)
```

Listing 1: Smart home rules

Definition 1 (IoT Object)**.** *An IoT object O is modelled as a Labelled Transition System $LTS = (S, A, T, s^0)$, where S is a set of states, A is a set of events/actions associated with transitions, $T \subseteq S \times A \times D \times S$ is the transition relation where $D = \{!, ?\}$, and $s^0 \in S$ is the initial state. A transition $(s_1, e, d, s_2) \in T$ (also noted $s_1 \xrightarrow{ed} s_2$) indicates that the system can move from state s_1 to state s_2 by performing an event/action named e in a certain direction (! for sending, ? for receiving).*

Then, an IoT application is described by a set of objects and a composition expression, which acts like an orchestrator indicating how the involved objects interact together. Before defining an IoT application, let us focus on the description of behaviours and on the composition language. We use a simple rule-based composition language for this purpose. This language assumes "*if event(s) then action(s)*" rules as basic elements. A rule is triggered when one or several events are issued by specific objects and, as a reaction, one or several actions are issued to other objects defined as target. Each event or action is accompanied by its object identifier.

Definition 2 (Rule)**.** *Given objects $O_1, \ldots, O_n$, with $O_i = (S_i, A_i, T_i, s_i^0)$, for $i = 1, \ldots, n$, a rule R is defined as "**IF** EVT **THEN** ACT" with*
EVT ::= event (Oid) | EVT$_1$ and EVT$_2$ | EVT$_1$ or EVT$_2$,
ACT ::= action (Oid) | ACT$_1$ and ACT$_2$,
where the terminal symbols are $event, action \in \bigcup_{i=1}^{n} A_i$, and Oid $\in \{1, \ldots, n\}$ is an object identifier.

Example (Rules). Listing 1 shows the rules involved in our smart home application example.

Rules can be composed to build more complex expressions, using basic operators such as sequence, choice, concurrent execution or repetition of rules.

Definition 3 (Composition Language)**.** *A composition C is an expression built over a set of rules R using the following operators:*

$C \ ::= \ R \ \mid \ C_1 \ ; \ C_2 \ \mid \ C_1 \ + \ C_2 \ \mid \ C_1 \,||\, C_2 \ \mid \ C_1^k$

where C_1 ; C_2 represents a composition expression followed by another one, $C_1 + C_2$ represents a choice between two composition expressions, $C_1 \,||\, C_2$ represents the concurrent execution of two composition expressions, and C_1^k represents the execution k times of a composition expression (if $k = *$, C_1 executes a finite number of times).

Example (Rule Composition). Although the rules in Listing 1 are the key elements of the behaviour of the smart home, the scenario described was not expecting their arbitrary use. When the resident open the door a light is turned on. If the temperature is higher than a given value, the light turns red. Then, when the sofa detects the presence of the resident, depending on how warm the temperature is, the window blinds are opened and the windows are opened or the fan is turned on. Finally, when the resident leaves the house, everything is turned off. This scenario can be expressed by composing the rules as follows: $R1\,;R2\,;(R3\,||\,(R4+R5))\,;R6$. This expression indicates that rule $R2$ is executed after $R1$, followed by $R3$ together with either $R4$ or $R5$, and finally the rule $R6$ will be executed.

Note that the composition language in Definition 3 is a regular language. Thus, any composition expression C written with this language can be transformed into an LTS with rules as labels. A sequence (C_1 ; C_2) transforms to a sequence of several transitions. It allows one to define an order for the execution of rules. A choice ($C_1 + C_2$) is encoded as a single state with two outgoing transitions. It enables one to execute a rule from a set of rules in a group. The concurrent execution ($C_1 \,||\, C_2$) is flattened to build all possible interleavings using Milner's expansion law [10]. A bounded loop (C_1^k) is encoded by repeating k times the same behaviour in sequence. A finite iteration (C_1^*) is described using a transition coming back to the state encoding the beginning of the loop. In the rest of this paper, we use interchangeably the terms composition expression and composition LTS.

Definition 4 (IoT Application)**.** *Given a set of objects $\{O_1, \ldots, O_n\}$, a set of rules R, and a composition expression C over R, an IoT application is defined as $(\{O_1, \ldots, O_n\}, C)$.*

2.2. Execution Semantics

Let us now explain how an IoT application, consisting of a set of objects and a composition expression, evolves. The communication model being asynchronous, each object is equipped with an input message buffer (FIFO). The composition expression and all objects start their execution from their initial states. Then, an application can evolve in two cases: execution of a rule or buffer consumption. In the first case, let us assume a basic rule with one event and one action. If the event appearing in the left part of the rule has been issued, the rule can be triggered and the action appearing in the right part of the rule is pushed to the corresponding object's buffer. The event can occur as a result of changes in the physical environment (e.g., change in temperature) or by interacting directly with the objects (e.g., a user toggling a switch). In the second case, one object can individually consume from its input buffer if there is something in its buffer and the object can consume according to its LTS model. In both cases, the global state of the application changes. A global state consists of the current state of all objects involved in the application (including their buffers) and the current state of the composition expression/LTS.

Definition 5 (One-step Execution Semantics)**.** *Given an IoT application* $(\{O_1, \ldots, O_n\}, C)$ *defined by a set of objects* $O_i = (S_i, A_i, T_i, s_i^0)$, $i = 1, \ldots, n$, *(with* B_i *the input buffer for object* O_i*) and by a composition LTS* $C = (S, A, T, s^0)$, *and given its current global state* $(((s_1, B_1), \ldots, (s_n, B_n)), s)$, *the application can evolve using the following two rules:*

- *(rule execution)*
  $(((s_1, B_1), \ldots, (s_j, B_j), \ldots, (s_k, B_k), \ldots, (s_n, B_n)), s)$
  $\xrightarrow{m!} (((s_1, B_1), \ldots, (s'_j, B_j), \ldots, (s_k, B_k m'), \ldots, (s_n, B_n)), s')$
  if $\exists j, k \in \{1 \ldots n\}$, $j \neq k$, $s_j \xrightarrow{m!} s'_j \in T_j$, *and* $s \xrightarrow{if\ m(j)\ then\ m'(k)} s' \in T$.
- *(buffer consumption)*
  $(((s_1, B_1), \ldots, (s_j, mB_j), \ldots, (s_n, B_n)), s)$
  $\xrightarrow{m?} (((s_1, B_1), \ldots, (s'_j, B_j), \ldots, (s_n, B_n)), s)$
  if $\exists j \in \{1, \ldots, n\}$, $s_j \xrightarrow{m?} s'_j \in T_j$.

At a given global state, several executions may be possible due to the use of choice and interleaving operators in the composition expression. By applying the one-step execution semantics whenever it is possible, we can cover all executions of the IoT application and thus give an LTS-based semantics to such applications. In Definition 5, we use a basic rule for simplicity. However, it can easily be extended to the general case as presented in Definition 2.

Some of the properties in the rest of the paper rely on execution traces, which are used to guide the execution of IoT applications. A trace is a

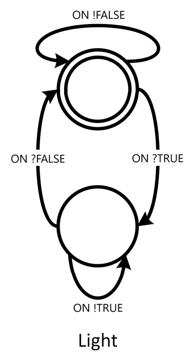

Figure 1: LTS of a light.

sequence of couples *(object identifier, action)*, where an action is either an occurrence of the event associated with a rule or a buffer consumption as shown in the former definition.

Definition 6 (Trace)**.** *Given an application* $\mathcal{A} = ((O_i \dots O_n), (S, A, T, s^0))$, *a trace* t *is an ordered sequence of pairs* $< (oid_1, a_1), \dots, (oid_m, a_m) >$, *where, for each pair,* oid *is the identifier of an object* O_i, *and* a *is either an event* $m!$ *that triggers a rule or an action* $m?$ *consumed from a buffer.*

Assuming that the LTSs of the objects and of the composition expression are deterministic, the execution is deterministic, and it is defined by the actions appearing in that trace. In our work, we also use the notion of filtered trace, which consists of selected pairs from t which belong to the remaining objects in a reconfigured application.

Definition 7 (Filtered Trace)**.** *Given an application* $\mathcal{A}_{current}$ *consisting of objects* $O_{current}$, *its reconfigured version* $\mathcal{A}_{new}$ *consisting of objects* O_{new}, *and a trace* t *corresponding to the sequence of actions executed in* $\mathcal{A}_{current}$, *the filtered trace* t_f *consists of pairs from* t *appearing in the same order, such that for each pair* $t_{elem} = (oid, a)$ *of* t, t_{elem} *also belongs to* t_f *iff* oid *is the identifier of an object* $O_i \in O_{current} \cap O_{new}$.

Example (Smart Home Rules). Figure 1 shows the LTS corresponding to a light. The light has a property *on*, which indicates whether the light is turned on or not. The concentric circle in the LTS denotes the initial state where the light is in the turned off state as seen by the *ON !FALSE* transition label. The object transitions from this initial state to the second state, upon receiving the *ON ?TRUE* action.

Example (Reconfiguration of Smart Home Rules). The set of rules represents the initial application. Suppose, after a while, the summer gets extremely hot due to a heatwave. Therefore, the resident prefers to install an air conditioner (AC) as the heat is unbearable even after turning on the fan. Now, she needs to reconfigure the application such that the AC is turned on instead of the fan, and she should ensure that the windows are not open when the AC is running. This can be done by updating the rules $R4$ and $R6$. First, the fan needs to be replaced by an air conditioning system in $R4$, then the window needs to be closed in $R4$ when the AC is turned on to maintain the cooling efficiency and save energy. Finally, $R6$ needs to be updated to turn off the AC, when the user exits the house.

This reconfiguration requires time and monetary investment. So, the user can take advantage of the reconfiguration properties (described in Section 3) to compare the behaviour of the current application with the new application and also compare the two applications quantitatively in terms of operating costs. It is worth noting that these checks are done at design-time, so that the user can make a better informed decision on whether to go ahead with the proposed changes.

3. Reconfiguration Properties

In this section, we present several properties that should be ensured before replacing the current application by a new one. First, we present three properties that take as input two applications, the current and the new one, as well as the global state of the current application before its reconfiguration. These properties are called *seamless*, *conservative*, and *impactful* reconfiguration, and assess the impact of replacing the current application by the new one in its current global state. In Section 3.3, we describe the quantitative properties that can be analysed by considering all possible executions of an application. The details on how reconfiguration takes place in practice (deployment) are described in Section 5.

3.1. Seamless Reconfiguration

To check the *seamless* reconfiguration property, we need the following inputs: the current and a new application defined by their respective set of objects and composition expressions, and the current global state of the current application (i.e., the application before reconfiguration). The seamless reconfiguration property checks whether this state can be reached again in the new application. This is important because this means that the deployment of the new application is possible without starting again this application from the beginning, thus seamlessly replacing the current application by the new application from the perspective of the user.

When reconfiguring an application, one can remove objects, add new objects, and change the composition expression. In this work, we consider that an application can be seamlessly reconfigured if all the remaining objects (i.e., common to the current and new applications) can reach again the state where they were before initiating the reconfiguration, according to the new composition expression. We focus on remaining objects because the states of removed objects do not need to be restored and new objects can start their behaviour from any state (they do not have an execution history). Since each remaining object must reach again the state where it was before reconfiguration, we also need the trace executed by the current application from its initial state up to the current global state. This trace is useful to check whether there is one execution of the new composition expression where all remaining objects can reach their former states repeating the same behaviour. This trace is obtained by instrumenting the IoT platform (WebThings in this work) to capture all the events/actions issued by the objects involved in the application.

When simulating the new application execution, guided by a trace which was executed on the current application, the remaining objects have to repeat the same actions. As far as the new objects are concerned, they evolve with respect to the new composition expression whose evolution is guided by the trace. The states in which the new objects would have to start on when deploying the new application are obtained by executing the trace on the new application. It is worth noting that when carrying out the deployment, it is not required to re-execute the actions on real objects whenever seamless reconfiguration is satisfied.

Definition 8 (Seamless Reconfiguration)**.** *Given two applications $\mathcal{A}_{current} = (O_{current}, C_{current})$ and $\mathcal{A}_{new} = (O_{new}, C_{new})$, each defined by a set of objects and a composition expression, given the current global state $(((s_1, B_1), \ldots, (s_n, B_n)), s)$ of application $\mathcal{A}_{current}$ and the trace t to reach that state consisting of a sequence of tuples (object identifier, action), the seamless reconfiguration property is satisfied if, when executing application A_{new} guided by the trace t, each remaining object $O_i \in O_{current} \cap O_{new}$ starting from s_i^0 can execute the actions in t and reach its current state s_i.*

Example (Seamless reconfiguration of our smart home). Let us consider now a subset of the rules for the smart home in Listing 1. Assume that the user begins automating the home with a small set of objects and rules. Initially, she wants to have a light to turn on when she enters a room and off when she exits it. The automation requires two motion sensors to detect her entry and exit, and a light to turn on and off upon detection of motion. The rules $R1$ and $R2$ associated with the original composition and the composition expression $R1\,;R2$ are shown in Figure 2 (left).

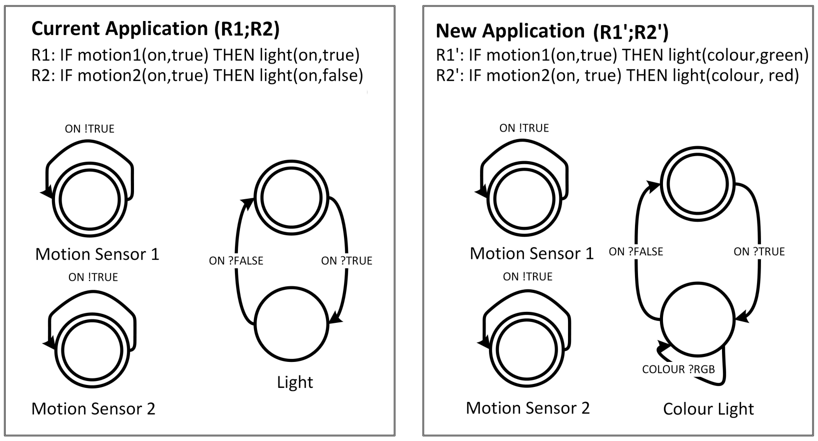
Current Application (R1;R2)
R1: IF motion1(on,true) THEN light(on,true)
R2: IF motion2(on,true) THEN light(on,false)
ON !TRUE
Motion Sensor 1
ON !TRUE
Motion Sensor 2
ON ?FALSE
ON ?TRUE
Light
New Application (R1';R2')
R1': IF motion1(on,true) THEN light(colour,green)
R2': IF motion2(on, true) THEN light(colour, red)
ON !TRUE
Motion Sensor 1
ON !TRUE
Motion Sensor 2
ON ?FALSE
ON ?TRUE
COLOUR ?RGB
Colour Light

Figure 2: Light being replaced by a colour light in a reconfiguration

After a while, the user prefers to replace the light with a colour light and update the rules to change colours upon detection of motion. Assume a global state where the rule $R1$ is executed, i.e., the *motion1* is on and *light* is turned on. The remaining objects in the new application are the motion sensors. Since they have a single state that is reachable, the reconfiguration is seamless, i.e., the state of each remaining object is maintained on deployment of the new application.

3.2. Conservative and Impactful Reconfigurations

The seamless reconfiguration definition indicates whether the remaining objects can reach again their former states in the new application. We can go further than this initial check by comparing more precisely both applications in terms of preserved behaviours and new behaviours. Therefore, we propose a couple of additional properties that could be helpful in order to better characterise the intended reconfiguration before applying it in practice.

A reconfiguration is called *conservative* if the seamless property is preserved and if, from the global state in which the reconfiguration is applied, all behaviours that could be executed in the current application (objects and composition expression) are still executable in the new application. This means that everything that was possible before is still possible in the new application from that state (each trace that can be executed in the current application is still executable in the new one). This check is useful when one wants an application to provide more services or features still preserving exactly what was possible before. Note that the global state of the current application cannot be used as a starting point in the new application because some objects may have been removed or added. To obtain the "equivalent" global state in the new application, we use the trace executed by the current application in the new application: remaining objects replay the same events/actions and new objects evolve following the new composition expression. In this way, we are able to compute a global state in the new application (the one computed to check that the seamless property is satisfied), and we use that state as starting point for checking conservative reconfiguration.

Definition 9 (Conservative Reconfiguration)**.** *Given two applications* $\mathcal{A}_{current} = (O_{current}, C_{current})$ *and* $\mathcal{A}_{new} = (O_{new}, C_{new})$*, given the current global state* $(((s_1, B_1), \ldots, (s_n, B_n)), s)$ *of application* $\mathcal{A}_{current}$ *and the trace* t *that was executed to reach that global state, the conservative reconfiguration property is satisfied if the seamless reconfiguration property is satisfied and if each trace* t' *that can be executed in* $\mathcal{A}_{current}$ *from* $(((s_1, B_1), \ldots, (s_n, B_n)), s)$ *can also be executed (after filtering it on the remaining objects* $O_{current} \cap O_{new}$*) in* $\mathcal{A}_{new}$ *from* $(((s'_1, B'_1), \ldots, (s'_m, B'_m)), s')$ *where the global state* $(((s'_1, B'_1), \ldots, (s'_m, B'_m)), s')$ *is obtained by executing* $\mathcal{A}_{new}$ *guided by* t*.*

A reconfiguration is called *impactful* if the seamless property is preserved and if the whole behaviour of each new object can be entirely executed in the new application. To check that, we first execute the former trace to obtain the global state in the new application. Then, we compute all behaviours that are reachable from that global state according to the new composition expression. The entire behaviour of each new object must be covered to say that the reconfiguration is impactful. This property allows one to verify whether the newly introduced behaviours are fully utilised in the new application.

Definition 10 (Impactful Reconfiguration)**.** *Given two applications* $\mathcal{A}_{current} = (O_{current}, C_{current})$ *and* $\mathcal{A}_{new} = (O_{new}, C_{new})$*, given the current global state* $(((s_1, B_1), \ldots, (s_n, B_n)), s)$ *of application* $\mathcal{A}_{current}$ *and the trace* t *that was executed to reach that state, the impactful reconfiguration* *property is satisfied if the seamless reconfiguration property is satisfied and if each new object $O_i \in O_{new} \backslash O_{current}$ has its entire behaviour appearing in $\{t\} \cup Tr$ (i.e., for each O_i, for each $s_1 \xrightarrow{ed} s_2 \in T_i$, e appears at least once in $\{t\} \cup Tr$), where $(((s'_1, B'_1), \ldots, (s'_m, B'_m)), s')$ is the global state obtained by executing $\mathcal{A}_{new}$ guided by t and Tr is the set of all traces that can be executed in A_{new} from $(((s'_1, B'_1), \ldots, (s'_m, B'_m)), s')$.*

Note that conservative and impactful reconfiguration properties are independent from each other. If all the objects and the composition expression are in their respective initial states, the conservative property is not systematically preserved, but the impactful property may not be preserved either because the new composition expression may prevent some new behaviour from being executed.

Example (Conservative and Impactful Reconfiguration). Let us consider the example in Figure 2 again. The reconfiguration of the current application involves light and motion sensors. Assume the global state corresponds to the execution of rule $R1$ (i.e., *motion1* is *on* and *light* is turned *on*), then the reconfiguration is conservative because it satisfies the seamless property and the replaced light still contains the behaviour that was possible previously. Thus, all behaviours that could be executed in the current application are still executable in the new application. The reconfiguration is impactful as well because it is seamless, the newly introduced behaviour of colour is utilised in the new application, and it is reachable from the current global state.

Beyond the reconfiguration properties introduced previously in this section, it is also possible to check classic safety and liveness properties on the new application (only) using model checking techniques. In this case, this additional verification analyses all the possible executions of the new application independently of the global state. Deadlock freeness for instance can be checked on the new application. This property is generic in the sense that it does not depend on the application. In contrast, other properties may depend on the application. For instance, if we go back to our example in Listing 1, we could verify that the AC is eventually turned off once the resident is away.

3.3. Quantitative Properties

Here we describe the quantitative properties that help the user to compare the current application with the new application in terms of operating costs. The quantitative properties are analysed on the overall behaviour of the application, which is captured in terms of an LTS. It is to be noted that the overall LTS is obtained as a result of composing the objects and composition expression behaviours described in Section 2. In order to perform quantitative analysis, we enrich the LTS defined in Definition 5 with probability information and transform it into a probabilistic transition system (PTS).

Definition 11 (Probabilistic Transition System)**.** *A Probabilistic Transition System (PTS) is a transition system where the labels contain both action and probabilistic information. More precisely, a PTS is defined as a tuple* (S, A, T, s^0, P)*, where* S *is a set of states,* A *is a finite set of events/actions,* s^0 *is the initial state, and* $T \subseteq S \times A \times D \times S$ *is a transition relation with a probability label* $P : T \to [0, 1]$*. A transition* $(s_1, e, d, s_2) \in T$ *(also noted* $s_1 \xrightarrow{ed} s_2$*) indicates that the system can move from state* s_1 *to state* s_2 *by performing an event/action named* e *with a probability* $P(s_1, e, d, s_2)$ *and for every* $s \in S$ *the sum of probabilities* $\Sigma_{s \xrightarrow{ed} s'} P(s, e,\ d, s') = 1$*.*

Execution Probability. Probability of execution indicates how often action(s) in rule(s) would be executed for a given probability of occurrence of event(s) involved in the rule. This probability of occurrence of an event can be estimated by the knowledge of the environment. An event typically occurs as a result of a change in the environment. For instance, when the room gets warm, the indoor thermometer captures the change in temperature and translates it to an event. So, one can look at the weather patterns and estimate the probability of temperature in a day being above certain threshold. By computing the probability of executing an action, users can modify their rules to optimise the number of times a certain action is being executed in an application.

Example (Fan-On Probability). Consider the rules in Listing 1, in this application, one can compute the probability of turning on the fan for a given day, considering all the rules in the application. If the weather forecast indicates 80% chance of being a warm day, then the transition corresponding to the temperature being warm in the PTS will be assigned a probability of 0.8. Taking this information into account, the probability of turning on the fan (in the application as a whole) will be computed.

Execution Cost. Cost of execution is another measure that is relevant when a reconfiguration needs to be applied. It allows users to compare the costs of running the current and new application. One can assign costs to each event and action in the rules (events typically involve sensing and therefore have negligible costs). Using these cost information, the cost of each finite trace of the application can be computed. From the computed costs, the traces associated with the minimum and maximum cost can be identified. The users can compare the minimum and maximum costs associated with the current and the new application to make a decision on whether to deploy the new application.

Combined Measure. In the above mentioned computation of costs, we consider only the cost of execution. However, this may not give a complete picture of operational costs of running an application. In order to overcome this, we can combine the probability of execution and costs to better reflect the actual operating costs. Once the cost of executing a trace is found, we compute the probability of execution of that trace. This measure can be compared across the current and newly reconfigured application.

Example (Cost Given a Probability). Turning on the air conditioner in the smart home with the rules in Listing 1 is a very costly operation, so the trace associated with maximum costs will always involve the rules having an air conditioning system. However, if we consider this application in a cold weather, the probability of the temperature going above the warm threshold is close to zero. Therefore, in practice, the trace containing the air conditioner is not going to be expensive to run.

3.3.1. Quantitative Analysis of Smart Home Rules

Now let us briefly illustrate the quantitative analysis on a concrete example described in Listing 1. The probability of execution metric is computed by assigning probabilities to events (environment actions).

Figure 3 shows the probability of turning on the fan for different values environment being warm. Upon assigning probabilities ranging from 0.01 to 0.9 to the environment action (warm), one can understand how the probability of turning on the fan varies with respect to varying temperature. It can be seen that the probability of turning on the fan is a bit lesser than 0.5 when there is a very low (0.01) probability of environment being warm. Ideally, this should have been 0.5 as the rule corresponding to the fan action is under a choice operator. However, once we account for other possible environment actions (e.g., user turning off the fan), the probability of turning on the fan dips below the expected number of 0.5. Further, as the probability of temperature being warm increases, the probability of turning on the fan becomes closer to 1. It is worth noting that since the rules in the new application are composed exactly like in the current application,

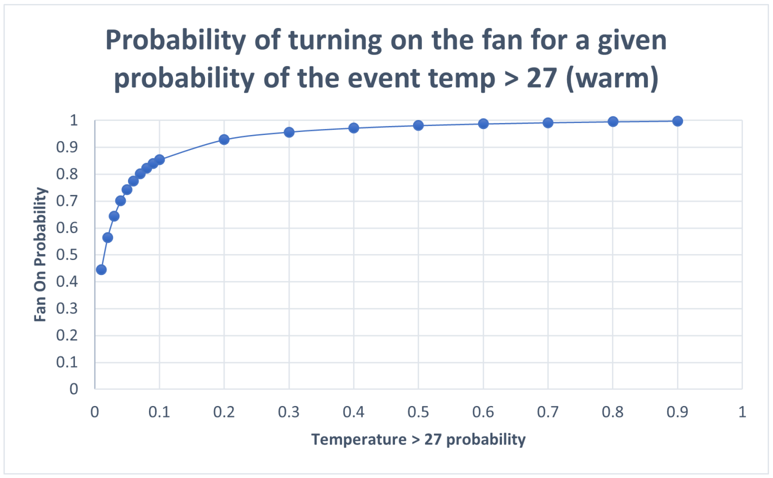

Figure 3: Event-Action probability

the probability of turning on the AC remains identical to the probability of turning on the fan.

Since the probabilities of turning on the AC and fan are similar, it is worth comparing the costs of the applications. For the sake of simplicity, let us assign a logical cost of 1 to all events (as events typically involve sensing which is not an expensive operation). We further assign specific costs to actions based on their energy consumption. Environment actions are associated with zero costs as they are not part of the application. The quantitative analysis for cost of execution identifies that the trace with maximum cost in the current application involves fan and in the new application, it is the AC that contributes to the maximum cost. The maximum costs in the current and new application are 84 and 106 units. Further, the analysis finds that the probability of execution of the maximum cost trace is 0.45, when we considered all remaining events and actions to be equiprobable. This indicates that the probability of turning on the AC in the new setup is non-negligible and it will lead to increased operating costs.

4. Encodings and Verification

This section shows how the different properties presented in the former section can be automatically checked via (i) an encoding into rewriting logic and the use of Maude's verification tools [4] for functional properties (seamless, conservative, impactful), and (ii) an encoding into the LNT [5] specification language and the use of CADP [6] analysis tools for quantitative properties.

We have chosen to build two different encodings as they serve different purposes. The Maude encoding is more suitable for analysis of history-based reconfiguration properties, because these properties aim at tracking state changes with respect to the original configuration and term rewriting especially using equational logic, is an efficient way to perform these kinds of analysis. The LNT encoding combined with the quantitative analysis available in the CADP toolbox simplifies the verification of properties based on probabilities and costs as these analyses are more reliant on actions rather than states.

4.1. Verification of Reconfiguration Properties using Maude

Maude is a high-level language and a high-performance system that supports membership equational logic, rewriting logic specification, and programming of systems. Rewriting logic [11] is a logic of change that can naturally deal with state and with highly nondeterministic concurrent computations. Rewriting logic is parameterised by an equational logic and Maude integrates an equational style of functional programming with rewriting logic computation. In the Maude implementation of rewriting logic, the equational logic is membership equational logic [12]. Membership equational logic is a Horn logic whose atomic sentences are equalities $t = t'$ and membership assertions of the form $t : S$, stating that a term t has sort S. Such a logic extends order-sorted equational logic, and supports sorts, subsort relations, subsort polymorphic overloading of operators, and the definition of partial functions with equationally defined domains. Further details can be found in [4].

The implementation in Maude of an IoT application consists of four steps, which aim at specifying successively IoT objects or devices, rules, composition expressions, and applications. As stated in Section 2, an object (Listing 2) is described by an LTS consisting of an initial state and a set of transitions (the alphabet and the set of states can be deduced from the set of transitions). A rule is defined as a single event or a set of events (and/or) in the left part, and as a single action or a set of actions in the right part. A composition (Listing 2) can make use of all the operators introduced in Section 2 (sequence, choice, parallel, iteration). Finally, an application consists of a set of objects and a composition expression.

Once the applications are described in the Maude specification language, we can rely on the Maude framework for the verification of reconfiguration properties. The seamless reconfiguration property is encoded as an operation

```
fmod LTS is
   pr STATE .
   pr SET{Transition} .

   sort LTS .
   op model : State Set{Transition} -> LTS .
endfm

fmod DEVICE is
   pr LTS .

   sort Device .
   op dev : Id LTS -> Device .
endfm

fmod COMPOSITION is
   pr RULE .
   pr INT .

   sort Composition .
   subsort Rule < Composition .
   op seq : Composition Composition -> Composition [assoc id: none] .
   op ch : Composition Composition -> Composition [comm] .
   op par : Composition Composition -> Composition [comm] .
   op iter : Composition Int -> Composition .
   op none : -> Composition .
endfm
```

Listing 2: Definition of the composition language

in Maude which takes as input the current application, the new application, the global state reached by the current application, and the trace executed by the current application to reach that state. It returns a Boolean response indicating whether the current global state for the remaining objects is reachable by executing that trace. New objects can be involved in order to reach that state, but whatever state they reach, it does not impact the seamless reconfiguration property.

This property is checked by first executing the trace in the new application and returning the reached global state. This state is unique because the LTS models of the new application are deterministic and the execution is guided by the given trace. Then, we check whether both global states coincide for the set of remaining objects. Note that the Maude specification precisely encodes the execution semantics of the models described in Section 2. In particular, each object is equipped with an input buffer for modelling the communication model used in our IoT application model. Listing 3 shows a few operations used for computing the seamless reconfiguration property. The first operation (checkSeamlessReconfiguration) takes as

```
op checkSeamlessReconfiguration : Application Application
     Set{Tuple{Id, State}} List{Tuple{Id, Label}} -> Bool .
op checkSeamlessReconfigurationAux : Application Application
     Set{Tuple{Id, State}} List{Tuple{Id, Label}} Set{Id} -> Bool .

---- filters the trace to keep labels belonging to remaining objects
eq checkSeamlessReconfiguration(App1, App2, GS, Tr)
  = checkSeamlessReconfigurationAux(App1, App2, GS,
      filterTrace(Tr, computeCommonObjects(App1, App2)),
      computeCommonObjects(App1, App2)) .

---- runs the trace in the new application until it is possible
eq checkSeamlessReconfigurationAux(App1, App2, GS, Tr, Ids)
  = compareGS(App1, App2, GS,
      getGlobalState(runTrace(App2, Tr, Ids)))
    and
    getBoolRes(runTrace(App2, Tr, Ids)) .
```
Listing 3: Specification of the seamless reconfiguration property

input all required elements (two applications, one trace and one global state) and filters out all events/actions in the trace that do not belong to the set of remaining objects. The second operation (checkSeamlessReconfigurationAux) takes as input two applications, one global state, the trace (filtered to keep only actions executed by the remaining objects) and the set of remaining objects. This operation calls the auxiliary function runTrace to execute the second application guiding this execution by the trace. If an object is not available anymore (removed in the second application), any object can be run instead (yet according to the new composition expression). The operation compareGS checks that the remaining objects have reached the same state in both global states. The operation runTrace also returns a Boolean value indicating whether the whole trace was executed for the remaining objects.

As far as the conservative property is concerned, we first check that seamless reconfiguration is preserved. Then, we start from the computed global state in the new application. We execute all possible behaviours in the new application, and we check that there is a match in the current application for each possible trace. We stop when we have traversed all behaviours and they all match, or when there is a mismatch. Listing 4 shows how we compare traces executed in both applications. We first run the trace given as input in both applications and get as results the state of all buffers in both applications as well as the global state of the second application. Then, operation compareFutureTraces executes all possible traces in the current application, and checks whether the new application can do the same.

```
op checkConservativeReconfiguration : Application Application
     Set{Tuple{Id,State}} List{Tuple{Id,Label}} -> Bool .

eq checkConservativeReconfiguration(
     app(Dev1,Comp1), app(Dev2,Comp2), GS1, Tr)
 = compareFutureTraces(
     Dev1, Comp1, GS1,
     getBuffers(runTrace(app(Dev1, Comp1), Tr, keepAllIds(Dev1))),
     Dev2, Comp2,
     getGlobalState(
       runTrace(
         app(Dev2, Comp2),
         filterTrace(Tr,
           computeCommonObjects(app(Dev1, Comp1), app(Dev2, Comp2))),
         computeCommonObjects(app(Dev1, Comp1), app(Dev2, Comp2)))),
     getBuffers(
       runTrace(
         app(Dev2, Comp2),
         filterTrace(Tr,
           computeCommonObjects(app(Dev1, Comp1), app(Dev2, Comp2))),
         computeCommonObjects(app(Dev1, Comp1), app(Dev2, Comp2))))) .
```
Listing 4: Specification of the conservative property

The impactful property checks that all new behaviours can be executed in the new application. First, seamless reconfiguration is verified and we start from the global state returned by this initial check. Second, we focus only on the second application and we compute and store all observable events for each device (input and output) from that state following the new composition expression. Finally, we check that all events have been traversed, for new devices only, from their respective initial states.

The analysis of classic safety and liveness properties is achieved by using the object-oriented and rule-based capabilities of Maude. Given an application (a set of objects and a composition expression), we define a class called Simulation with four attributes: the current global state, the current trace, the current state of the composition expression, and a set of buffers (one input buffer per object). Then, we define six rules corresponding to all possible evolutions of our system. There are five rules corresponding to the evolution of the composition expression (one rule for sequence, choice, interleaving, and two rules for iteration), and one rule corresponding to the consumption by one object from its buffer. We illustrate with this later rule (Listing 5). This rule shows how one object can consume from its buffer (adequate current state and available message in buffer), and how the global state, the buffer for that object, and the current trace are updated.

The readers can find online [13] several sample executions illustrating

```
rl [consumeFromBuffer] :
  < AId : IoTApp |
      devices :
        (dev(O1, model(S, (S1 Pr1 - M ? V -> S2 Pr2, Transitions))),
         Devs),
      Atts >
  < SId : Simulation |
      gstate : (ids(O1, S1), GS),
      trace : Tr,
      buffers : (buf(O1, ((M ! V) LL)), Bfs),
      Atts1 >
  =>
  < AId : IoTApp |
      devices :
        (dev(O1, model(S, (S1 Pr1 - M ? V -> S2 Pr2, Transitions))),
         Devs),
      Atts >
  < SId : Simulation |
      gstate : (ids(O1, S2), GS),
      trace : (Tr (idl(O1, M ? V))),
      buffers : (buf(O1, LL), Bfs),
      Atts1 > .
```
Listing 5: Specification of the buffer consumption rule

the use of the specification for verifying properties.

4.2. Verification of Quantitative Properties using CADP

LNT is a value-passing process algebraic specification language with imperative programming style designed for modelling concurrent systems. As such, it provides a convenient way to formally specify the composition of rules, since objects can be represented naturally as communicating processes and rule composition expressions can be described compositionally as combinations of LNT operators. LNT is equipped with a formal operational semantics, which enables to translate an LNT description into an LTS using the compilers of the CADP toolbox [6]. The LNT operators used in the encoding are: **select** (choice), **par** (interleaving or parallel composition), **;** (sequential composition), **loop** (repetition), and **if** (conditional). Behaviours are encoded in a **process** and they are parameterized by gates (communication endpoints) and data variables. Behaviours communicate via rendezvous on gates by specifying the emission (**!**) and reception (**?**) of data values.

Each object is encoded as a process in LNT. The properties associated with the object are encoded using local variables. Property values are updated when synchronizations occur on gates *action* (changes triggered by the execution of rules) and *envaction* (changes triggered by the external environment). Upon a change in value of one of its properties, an object emits the new value of the property on gate *event*, as mentioned in Section 2. Synchronizations on gate *envaction* account for change in object behaviour caused by the external environment. For example, a user (environment) can turn off the light even when there is no rule manipulating the light. It is to be noted that LNT provides higher level abstractions for defining behaviour. Therefore, the properties are encoded as variables, but the resulting LTS generated by the LNT compilers consists of expanded range of values similar to the encoding in Maude.

Listing 6 shows the outline of a generic object process. First, as mentioned in Section 2, an object can emit events upon change in property values. This emission of event is encoded in line 5, where we can see the emission (!) of the current value of property p_1. Second, actions are added to the action queue on receiving an action request (line 8). The property *p1* is received (?) and appended to the *actionQ* of the object. These action requests need to be consumed at some point and the consumption of an action request is represented by the *delete* operation, which removes the request from the *actionQ* as shown in line 11. It is to be noted that a successful execution of an action results in a change of state in the object and therefore, it is followed by the emission of an event notifying the change. *envaction* is similar to a regular action, except that it has an immediate effect, without going through the action queue. Finally, *done* is an auxiliary operation to track one complete execution of the application scenario. By default, a composition is designed to run infinitely (unbounded) and presence of *done* label helps to track the completion of one execution of the composition expression. All these operations are modelled as a choice using the **select** operator. The entire process is enclosed in a **loop** to reproduce the reactive behaviour of the objects.

An ECA rule of the format **IF** EVT **THEN** ACT is specified using the sequence (**;**) operator of LNT. Each rule is transformed into a process of the form $EVT; ACT$, where the **select** operator is used to encode the disjunction of events in EVT and the **par** operator is used to encode the conjunction of events in EVT and actions in ACT. The composition of rules is also encoded in a similar manner.

Finally, the MAIN process which describes the interaction between devices, environment, and a GLOBALLISTENER process, which keeps track of the events emitted by the objects, is specified. Listing 7 shows a generic outline of the MAIN process.

Readers interested in the concrete models of applications may refer to

```
process OBJECT [event,action,envaction:any] is
  var actionQ:QUEUE, p1:bool, p2:nat, p3:...
    loop
      select
        event(!o1,?any RuleId,!p1,?any bool) [] event(...) []...
      []
        action(!o1,?any RuleId,?p1Val);
        actionQ := append(p1Val,actionQ) []...
      []
        p1:= get_p1_value_fromQ(actionQ);
        actionQ := delete_p1_value_fromQ(p1, actionQ) [] ..
      []
        envaction(!o1,?p1)
      []
        done
      end select
    end loop
  end var
end process
```

Listing 6: Outline of the LNT process of an IoT object

the examples online [13].

4.2.1. Quantitative Analysis with MCL

Model Checking Language (MCL) [14, 15] is a branching-time temporal logic that can be used for expressing properties that are interpreted on LTSs. In this work, we use MCL specifically for quantitative analysis as it is a data-handling temporal language equipped with the Evaluator model checker [14] of CADP, and LNT is the input language of CADP toolbox. By encoding the application in LNT, we automatically derive the LTS representing its behaviour (denoted by LTS_C) using the CADP compilers. Properties are expressed in terms of the events and actions labelling the transitions of LTS_C. Event labels have the form EVENT !R !O !k, !v !B, where R is the rule identifier, O is the object identifier, and key-value pair of the object property is denoted by k and v, respectively. The Boolean clause B serves to track whether the rule R is active in the composition when the event is detected. Actions invoked by the rules are represented by labels ACTION !R !O !k !v and actions invoked by the environment are represented by labels ENVACTION !O !k !v.

Probability of Execution. The probability of execution associated with an ENVACTION is indicated by suffixing it with the probability value, i.e., ENVACTION !O !k !v **; prob P**. This is done through renaming directives which append probabilities to environment actions in LTS_C to generate a PTS.

```
process MAIN [...] is
    par event, action, done in
      par
        COMPOSITION [event, action, done] || GLOBALLISTENER[event]
      end par
    ||
      par envaction in
        par done in
          OBJECT1 [event, action, envaction, done]
        ||
          OBJECT2 [event, action, envaction, done] || ...
        end par
      ||
        ENVIRONMENT [envaction, done]
      end par
    end par
end process
```

Listing 7: Outline of the Main process

```
  prob
   (not DONE)*.{ACTION !R !O !k !v} is >=? 0
  end prob
```

Listing 8: MCL expression for computing probabilities

The Evaluator tool supports on-the-fly verification of probabilistic properties on PTS.

The computation takes into account the probabilities associated with events and actions whenever they are specified. Otherwise, all the outgoing transitions (events or actions) from a state are considered equiprobable, with the sum of their probabilities equal to one. Typically, the probability of occurrence of environment actions will be provided by the users and the probability of remaining actions will be computed by the Evaluator tool. Listing 8 shows the MCL specification for computing the probability of executing an action triggered by rule R and updating property k with value v in object O. prob is the keyword denoting a probabilistic property. As the applications can run forever, we need to limit probability evaluation to one execution, otherwise, in the long run all the probabilities of executing the actions will converge to one. Therefore, "not DONE" label is used to specify one execution of the application and {ACTION ...} represents the action whose probability needs to be computed.

Cost of Execution. The cost of execution represents the costs associated with executing a trace in an application. The events and actions can be associated with certain cost values and these costs are aggregated to compute

```
< loop (c:nat := 0) : (r:nat) in
   { EVENT ... !true ?c2:nat}.continue (c + c2)
  |
   { ACTION ... ?c2:nat}.continue (c + c2)
  |
   not ({EVENT ...} or {ACTION ...} or DONE).continue (c)
  |
   DONE.exit(c)
 end loop > (r <= MIN_COST)
```

Listing 9: MCL expression for computing cost

the overall cost of executing a trace. This cost information can be provided by the users. Again, the cost of an action is indicated by suffixing it with the cost value in LTS_C, i.e., ACTION !O !k !v **!C**. Here we represent costs in an abstract manner as natural numbers, which could denote costs such as electricity consumption, battery usage, etc. Specifically, users can compute the traces having the maximum and minimum cost in an application.

The computation of cost involves a sequence of operations on LTS_C. First, all the events and actions in LTS_C are assigned default costs by renaming suffixing them with default costs. Then, the default costs are overwritten with the costs provided by the user. The events and actions for which the user does not indicate the costs, will maintain the default costs (which could be configured to any value). Listing 9 describes the MCL specification for computing the minimum cost. The "<...>" modality expresses the existence of a trace satisfying the **loop** generalised regular formula. The cost variable c is updated whenever an EVENT or an ACTION is matched, any other label is ignored (i.e., cost is not updated). The cost suffix in event and action labels is captured in the variable $c2$. Once again, DONE is used to limit the check to one execution of the application. The variable MIN_COST can be any arbitrary natural number. Evaluator returns either true or false indicating whether there exists a trace with cost less than the MIN_COST. Depending on the result, we increase or decrease the MIN_COST value dichotomously to converge at the trace corresponding to the minimum cost in the application. This approach can be used for computing maximum cost by changing the MCL expression to check for r >= MAX_COST.

Combined Measure. The combined measure takes into account both costs and probabilities. This is expressed in MCL by specifying the cost expression described in Listing 9 and combining it with **prob** keyword to compute the probability of execution of the trace which satisfies the cost property.

5. Reconfiguration in Action with WebThings

This section covers the integration of the proposals in a tool based on WebThings. We introduce the WebThings framework and then we present the UI, verification, and deployment extensions for supporting reconfiguration.

5.1. WebThings

The IoT ecosystem is a diverse field consisting of various manufacturers with different underlying technologies and standards. Web of Things (WoT) [9] is one of the standardisation efforts to simplify the design of IoT applications. It is based on the architectural styles and principles of the web, which is prevalent and thereby eliminates the need to learn various disparate technologies to build the applications. In WoT, objects are identified via a URI and each object has an associated Thing Description (TD), described in machine-readable JSON-TD. A TD of an object describes its behaviour, the operations it supports, i.e., the interfaces to monitor or alter its state, security configuration, and protocol bindings. The WoT standardisation is led by W3C and at the time of writing, the specifications are available as a recommendation.

WebThings [2] is a platform for monitoring and controlling devices over the web. It is based on WebThings TD, a specification complementary to the W3C's work on abstract data model. In our work, we use the WebThings specification as it provides a concrete implementation which can be extended quickly and efficiently. The Things UI component in WebThings allows users to build IoT automation in the form of "*If event(s) then action(s)*" Event-Condition-Action (ECA) rules. It also provides web APIs for monitoring and controlling IoT objects. Many of the popular objects are already supported by the platform and more objects are being added.

Mozart [3] is a tool built on top of the WebThings platform to support the design and deployment of complex applications. In addition to individual ECA rules, it allows users to compose these rules using the composition language described in Section 2. Composition of rules enables building of more expressive application scenarios. Further, the tool allows verification of these applications for correctness at design time before proceeding with the deployment. In this work, Mozart was extended to support end-to-end (i.e., design to deployment) reconfiguration. The next subsection covers the reconfiguration support in detail.

5.2. Reconfiguration Support

R-Mozart is the extension of the Mozart tool to support the reconfiguration of applications. The extensions were made at three different levels. First, a new set of interfaces were built that enabled users to redesign the deployed application. These interfaces are connected to the verification component, which transforms the compositions into formal specifications to perform verification of properties and quantitative analysis. Finally, a new set of APIs were developed to deploy the reconfigured applications.

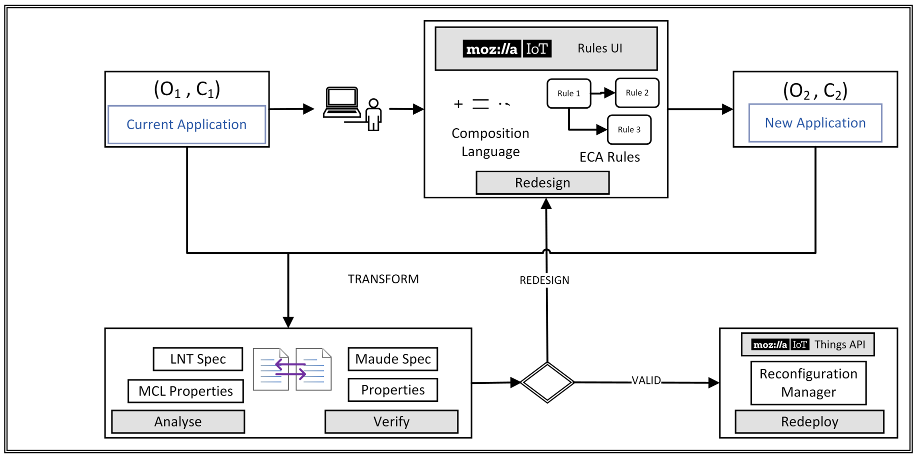

Figure 4: Reconfiguration workflow

Now, let us take a look at how reconfiguration works in practice. The reconfiguration workflow is shown in Figure 4. We assume that a first version of an IoT application has been designed and deployed by a user. Then, once a reconfiguration of the running application is envisaged, the user moves the concerned application to the newly designed reconfiguration UI. It allows her not only to modify the rules, which may involve adding, removing, or changing objects, but also to change the way in which the rules are composed in an application. Once the redesign is finalised, the user can compare the new application with the current application to check the impact of reconfiguration. The interface provides options to perform two kinds of analysis: i) verification of reconfiguration properties, and ii) quantitative analysis. The first analysis involves one or several of the properties presented in Section 4. In order to perform quantitative analysis, the user may indicate the costs and probabilities of actions/events by filling the form in the UI. A response is provided to the user indicating the results of the verification and quantitative analysis. The response of the verification is a boolean value indicating whether the properties are satisfied. The result of quantitative analysis is shown as a table listing the events and actions along with their probabilities and costs. If the response is not satisfactory, she can revise the design or keep the application running as it is. Otherwise, the user can proceed to the next step where the reconfiguration manager handles the deployment.

The manager performs two tasks: i) undeploy the removed objects, while preserving the state of remaining objects, and ii) deploy the new objects, compute the state of the new objects, and run the reconfigured application. The current states of all objects are stored in a database along with the execution history of the application. During the reconfiguration, new rules are created or the existing ones are updated. New rules are created using the Rules UI and when they are enabled, event listeners associated to the events in these rules are created. Similarly, when the rules are disabled, their associated event listeners are removed. Here, we mention rules and not individual objects because adding or removing objects is a modification to a rule, as objects are a part of an event or an action. In other words, adding an object means including the object in a rule, from the available pool of objects and removing an object implies it is no longer used in a rule. Now these rules need to be deployed for the application to run. Remaining objects maintain their previous states. As for newly added objects, we simulate the execution trace of the current application on the new composition expression. As a result, we obtain the states from where the new objects have to start when deploying the new application. Newly added rules are initialised in disabled state. As a last step, we use the execution history of the current application to compute the state from where the new composition expression should start, which allows us to determine the set of rules to be enabled. From here, the MOZART execution engine takes care of running the application. It follows the composition expression semantics by enabling or disabling relevant subsets of rules as the execution of the expression progresses.

It is worth noting that the reconfiguration process can be initiated at any moment since there is no way to stop the remaining objects from being active. All the events issued during this process are stored. Once the new application is deployed, only those related to the remaining objects are executed, the other ones are discarded. As for the trace corresponding to the history of execution, it is pruned by removing all actions corresponding to removed objects. Sometimes execution history can be quite long, but since objects in IoT typically exhibit cyclic behaviour, we do not have to consider the complete execution trace but only the part of the trace originating from the last appearance of the initial state. Note that the trace is reset to null, only when the whole application is re-initialised by the user. In that case, all objects and the composition expression start from their initial states.

As far as implementation is concerned, a simplified view of the components and technologies used in the tool is shown in Figure 5.

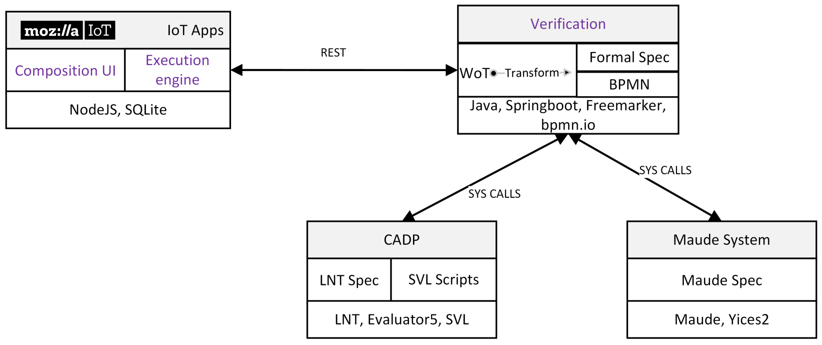

Figure 5: R-Mozart technology stack

WebThings Gateway is built on Node.js. Our extension takes advantage of the existing packages available in the platform to implement the execution engine. Rules, the composition of rules, the state of objects, and execution traces are stored in a file-based SQLite database. BPMN visualisation is handled using bpmn.io JavaScript library. The backend transformation and verification component is implemented as a Spring Boot application hosted on an embedded Tomcat server. The transformation from JSON to LNT and Maude specification is handled using Freemarker templating engine. Communication with the CADP verification toolbox and Maude system is done via system calls. Current states of the associated objects and the execution state of the application are stored in an SQLite database. States of the individual objects are collected using the monitoring APIs provided by the WebThings API. The state of the composition expression is updated by manipulating the event listeners. During deployment, new objects are moved to appropriate states using the control APIs that allow to set object state (e.g., switch on the lamp or change its colour to red).

Some screenshots from the tool for reconfiguration are shown in Figure 6. The bottom left screenshot shows the available rules and composition operators on the left frame, along with the current application on the canvas. The screenshot on the top left shows an ECA rule. The screenshot on the background shows the reconfigured application, along with the options for comparison and deployment which can be seen on the bottom right of the image. The current state of the objects can be monitored using the Thing

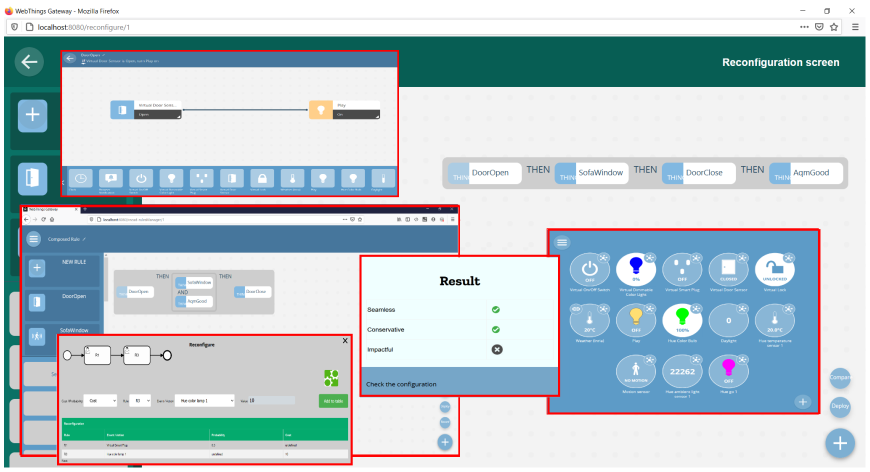

Figure 6: Screenshots of the reconfiguration tool

UI as shown in the screenshot on the bottom right. Finally, the central screenshot shows the response from the property analysis and the capture on the bottom left (foreground) the table listing the event/action probability and costs. It is worth noting that the tool abstracts the complexities of formal analysis away from the users and thus keeps the training required to learn the tool to a minimum.

5.3. *Evaluation*

We evaluated our proposal in a real-world setup by deploying a set of applications and reconfiguring them to perform property analysis and quantitative analysis. In this section, we briefly comment on the application in terms of performance, usability, and deployment.

Experimental Setup. We hosted the tool on a local machine (PC) and on a Raspberry Pi 3, connected to a private wireless network. Further, we added a set of connected devices which included Philips Hue lights, Hue motion sensors, Hue Play lights, connected thermometer, and connected speakers. WebThings allows one to create virtual objects that mimic the behaviour of IoT devices and we created a set of virtual objects (e.g., door sensor, thermostat, smart plugs) as a substitute for real devices in certain IoT applications. Since the devices were on the same network, they were easily discovered and added to the monitoring interface of the WebThings UI. Using these devices, we built the set of rules and deployed them. The rules were executed by modifying the physical environment (e.g., triggering

App 1	App 2	Trace	Seamless			Conservative			Impactful		
Objs.	/ Rls.	Steps	Result	Rews	Time	Result	Rews	Time	Result	Rews	Time
3 / 2	7 / 11	2	true	119	0 ms	false	172	0 ms	false	186	0 ms
5 / 2	10 / 11	2	true	136	0 ms	false	192	0 ms	false	278	0 ms
8 / 7	15 / 20	2	true	179	0 ms	false	236	0 ms	false	497	2 ms

Table 1: Outputs for some of the simulations carried out.

motion) or changing the states of the devices using the WebThings APIs (e.g., increase or decrease relative humidity values in a virtual thermostat). Later, these applications were redesigned, and reconfiguration analysis was performed on them. The validated applications were deployed through the newly developed reconfiguration manager.

Performance. The response times for obtaining the results of verification were measured to quantify the performance of the application. Let us first focus on the time it takes to compute reconfiguration properties using Maude. Table 1 shows some results for several experiments we carried out. The first three columns show the complexity of the applications — number of objects and number of rules — and the number of steps in the execution traces. The rest of the columns show the results for the different types of checks presented in the previous sections, namely, seamless, conservative, and impactful reconfiguration. For each of them, we provide the result of the check, the number of rewrites and the execution time. This table shows that Maude-based analysis takes a few milliseconds to verify the reconfiguration properties for applications including up to 15 objects and 20 rules. This efficient performance can be largely attributed to the fact that the analysis is based on the execution trace, and even for large applications, the analysis involves only a small subset of the states (not need to traverse the whole behaviour).

The time required to perform quantitative analysis is significantly higher as it needs to explore the complete behaviour of the application to compute the quantitative measures. It takes about a minute to analyse an application with 4 rules and 6 objects. Compositional verification techniques can be used to achieve faster results.

Usability. In order to test the usability of the application, we identified two participants from the target user group. *User 1* with programming experience and *User 2* without programming experience. However, both users were familiar with smart home automation and had the experience of using connected devices in their homes. The users were briefly trained (15 minutes) to use the R-Mozart tool. Then they were provided with the initial application in deployed state with the rules in Listing 1. The users were then provided with the textual description of the planned reconfiguration. It took less than 2 minutes for *User 1* to reconfigure application by adding the objects and modifying the rules. *User 2* needed additional 2 minutes to complete the same exercise. At the end of the exercise, a Single Ease Question (SEQ) was asked, which resulted in a rating of 6 by both the users. During the experiments we did not focus more on usability of the tool as the extended interface is similar to the design interface of MOZART. The end-user usability of MOZART was found to be satisfactory [3, 16].

Deployment. The deployment of the application typically took only a few milliseconds because the devices are already on the network and the updates occur only at the level of rules. Deployment involves changing the state of the objects and updating the state of rules through REST API calls.

6. Related Work

Dynamic reconfiguration is an omnipresent problem in computer science and it has been studied in several areas. It was one of the main problems in software architectures, where several formal frameworks such as Darwin [17] or Wright [18] were proposed in order to specify dynamic reconfiguration of component-based systems whose architectures can evolve at runtime (by adding or removing components and connections). These techniques aim at helping users to formally design dynamic applications. In [19, 20], Kramer et al. show how to formally describe behavioural models of components using the FSP specification language and analyse these models using the Labelled Transition System Analyser (LTSA), which allows the verification of temporal properties on the component architecture.

These works mainly focus on modelling component-based dynamic architectures. In terms of verification, LTSA for instance was used to analyze these architectures, but was not targetting the verification of the reconfiguration process. In this paper, we focus on the design of IoT applications but our main goal is the automated analysis of the impact of reconfiguration on the application from a consistency, correctness, and quantitative perspective.

Seamless reconfiguration is not a new notion and was used in several works focusing on dynamic reconfiguration [21, 22]. The work by Vogel et al. [22] presents a flexible approach to seamless reconfiguration of EJB-based enterprise applications. This work provides generic and reusable procedures for automatically supporting reconfiguration tasks. The role of the administrator is reduced to selecting an appropriate strategy and creating a reconfiguration plan that configures a generic procedure for a concrete reconfiguration.

The main difference of our approach compared to these works is that we present seamless reconfiguration in the context of IoT applications, and we go beyond this notion by proposing additional properties such as impactful and conservative reconfiguration. Moreover, we go beyond functional properties by also analyzing the application from a quantitative perspective by using complementary verification techniques, namely probabilistic model checking.

As far as reconfiguration of component assemblies is concerned, in [23, 24], Boyer et al. present a reconfiguration protocol applying changes to a set of connected components for transforming a current assembly to a target one given as input. Reconfiguration steps aim at (dis)connecting ports and changing component states. The protocol is robust in the sense that all the steps of this protocol preserve a number of architectural invariants. For designing this reconfiguration protocol, the authors used value-passing process algebra and model checking techniques for detecting and correcting behavioural issues that showed up during the protocol design [23]. They also proved the protocol correctness by using theorem proving techniques [24].

Similarly to this work, we propose reconfiguration mechanisms with some formal guarantees. This work differs with ours in terms of the type of targeted application, component-based versus ruled-based IoT applications. In addition, the verification techniques are used with a different goal: they use it to verify the protocol they propose for applying reconfiguration, whereas we propose analysis techniques to check that the reconfiguration proposed by the user respect some specific functional and quantitative guarantees.

In the cloud computing area, in [25], Fischer et al. present a system that manages application stack configuration. It provides techniques to configure services across machines according to their dependencies, to deploy components, and to manage the life cycle of installed resources. In [26], Durán and Salaün present a reconfiguration protocol for dynamically updating a cloud application consisting of components deployed on virtual machines. The reconfiguration tasks consist of addition / removal of virtual machines and components hosted on these virtual machines. The protocol is robust because it preserves the application consistency and respects important architectural invariants related to software dependencies. It is reliable because it supports failures of virtual machines.

In these two works, the configuration and reconfiguration protocols propose some guarantees ensured by the management process, e.g., the reconfiguration protocol preserves some pre-defined architectural properties. In our approach, our goal is different because our analysis techniques are helpful to validate the reconfiguration plan proposed by the user.

The approach presented by Seeger et al. in [27] propose to extend semantic application descriptions (called *recipes*) with constraints to enable dynamic and automatic reconfiguration of IoT applications. Using recipes, dynamic choreographies can be created that self-adapt to changing device states without human intervention. In [28], Koziolek et al. introduce the OpenPnP reference architecture, which allows a significant reduction of configuration and integration efforts during industrial plant commissioning. The OpenPnP architecture reduces configuration and installation time by up to 90 percent, while scaling to IIoT systems with many nodes. OpenPnP also provides concepts for replacing malfunctioning devices.

These works mostly focus on providing algorithmic solution to effectively perform deployment and reconfiguration of (I)IoT applications. Even though we also provide similar techniques (illustrated using the WebThings platform), our main goal was to propose analysis techniques for reasoning on the behaviour of the application before deciding actual reconfiguration.

In [29], Martino et al. present a review of the most common architectural solutions available today to shape an IoT system, ranging from already standardized architecture to commercial ones. Elements from such architectures are compared, analysed and mapped one against the other to determine a stable reference for security and interoperability analysis. According to the reference architectures, a set of real-case scenarios are introduced and describe the most common configurations of IoT devices. According to these scenarios, security and interoperability challenges are identified and current solutions to these challenges are presented.

The goal of our work is not to make any contribution at the definition or standardization level for IoT architectural solutions. We assume a given description of IoT applications and focus on the problem of its reconfiguration, which should be handled with care in order to preserve important functional and quantitative properties.

As far as cost analysis is concerned, in [30], Huang et al. present a theoretical approach of performance modelling and analysis for IoT services. An atomic service is formulated by a queueing model, and quantitative analysis is conducted for different task arrival distributions. Then, a hierarchical services computing system that provides IoT services in edge computing paradigm can be modelled by a queueing network, and the methodology of its performance analysis for resource management and task scheduling. This work is expected to provide system designers and managers with a predictive approach for performance evaluation without implementing the services and systems, which can be helpful for their design and optimisation with high efficiency and little cost.

This work does not target the optimization and cost analysis in the context of the reconfiguration of the IoT application, which is one of our goals here. One of the perspectives of this work is to use runtime monitoring of applications to automatically derive probabilities related to the applications.

7. Conclusion

In this paper, we have focused on IoT applications consisting of devices interacting as described in a composition expression of ECA rules. These applications are not built once and for all any more. The goal is to give the possibility to change these applications (addition or removal of objects, update of the composition expression) and to provide formal guarantees during the reconfiguration process. We have defined several properties that characterise the consistency and correctness of the application to be reconfigured. We have also proposed verification techniques that allow one to analyse not only the update of an application with respect to a certain global state of the application, but also to analyse all possible executions of the new application to check whether it preserves certain functional properties. Quantitative properties based on probabilities and costs can also be analysed for comparing, e.g., the cost of both applications before deciding to replace the current one by the new one. All these checks are fully automated using two encodings into rewriting logic and process algebra, respectively, and simulation and model checking tools (Maude and CADP). These verification results as well as additional components for supporting the design and deployment of the new application were integrated into the WebThings platform as an extension of the MOZART tool. This allowed us to apply it to several smart home applications for validation purposes. As far as future work is concerned, we plan to monitor the application during runtime to derive the probabilities of occurrence of events. This would allow us to build a more realistic quantitative model of the application.

Acknowledgements. F. Durán has been partially supported by projects PGC2018-094905-B-100 and UMA18-FEDERJA-180.

References

[1] S. Ovadia, Automate the internet with If This Then That (IFTTT), Behavioral & social sciences librarian 33 (4) (2014) 208–211.

[2] Mozilla, WebThings, https://iot.mozilla.org/ (2020).

[3] A. Krishna, M. Le Pallec, A. Martinez, R. Mateescu, G. Salaün, MOZART: Design and Deployment of Advanced IoT Applications, in: Proc. of WWW'20, ACM, 2020.

[4] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. L. Talcott (Eds.), All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic, Vol. 4350 of LNCS, Springer, 2007.

[5] D. Champelovier, X. Clerc, H. Garavel, Y. Guerte, F. Lang, C. McKinty, V. Powazny, W. Serwe, G. Smeding, Reference manual of the LNT to LOTOS translator.

[6] H. Garavel, F. Lang, R. Mateescu, W. Serwe, CADP 2011: a toolbox for the construction and analysis of distributed processes, Int. J. Softw. Tools Technol. Transf. 15 (2) (2013) 89–107. doi:10.1007/s10009-012-0244-z.

[7] IETF, Constrained RESTful Environments, https://datatracker.ietf.org/wg/core/charter/ (2019).

[8] Nest Labs, A Secure and Reliable Communications Backbone for the Connected Home, https://openweave.io/ (2019).

[9] W3C, Web of Things at W3C, https://www.w3.org/WoT/ (2019).

[10] R. Milner, Communication and Concurrency, Prentice Hall, 1989.

[11] J. Meseguer, Conditional Rewriting Logic as a Unified Model of Concurrency, Theoretical Computer Science 96 (1) (1992) 73–155.

[12] A. Bouhoula, J.-P. Jouannaud, J. Meseguer, Specification and Proof in Membership Equational Logic, Theoretical Computer Science 236 (1) (2000) 35–132.

[13] F. Durán, A. Krishna, M. Le Pallec, R. Mateescu, G. Salaün, Models and Analysis for User-Driven Reconfiguration of Rule-Based IoT Applications: Online Code,

`https://github.com/ajaykrishna/mozart/tree/mozart` (March 2022).

[14] R. Mateescu, J. I. Requeno, On-the-fly model checking for extended action-based probabilistic operators, International Journal on Software Tools for Technology Transfer 20 (5) (2018) 563–587.

[15] R. Mateescu, D. Thivolle, A model checking language for concurrent value-passing systems, in: International Symposium on Formal Methods, Springer, 2008, pp. 148–164.

[16] F. Durán, A. Krishna, M. Le Pallec, R. Mateescu, G. Salaün, R-MOZART: A reconfiguration tool for webthings applications, in: 43rd IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2021, Madrid, Spain, May 25-28, 2021, IEEE, 2021, pp. 41–44.

[17] J. Magee, J. Kramer, Dynamic Structure in Software Architectures, in: Proc. of SIGSOFT FSE'96, 1996, pp. 3–14.

[18] R. Allen, R. Douence, D. Garlan, Specifying and Analyzing Dynamic Software Architectures, in: Proc. of FASE'98, Vol. 1382 of LNCS, Springer, 1998, pp. 21–37.

[19] J. Kramer, J. Magee, Analysing Dynamic Change in Distributed Software Architectures, IEE Proceedings - Software 145 (5) (1998) 146–154.

[20] J. Magee, J. Kramer, D. Giannakopoulou, Behaviour Analysis of Software Architectures, in: Proc. of WICSA'99, Vol. 140 of IFIP Conference Proceedings, Kluwer, 1999, pp. 35–50.

[21] L. Rosa, L. E. T. Rodrigues, A. Lopes, A Framework to Support Multiple Reconfiguration Strategies, in: Proc. of Autonomics 2007, Vol. 302 of ACM International Conference Proceeding Series, ACM, 2007, p. 15.

[22] T. Vogel, J. Bruhn, G. Wirtz, Autonomous Reconfiguration Procedures for EJB-based Enterprise Applications, in: Proc. of SEKE'2008, Knowledge Systems Institute Graduate School, 2008, pp. 48–53.

[23] F. Boyer, O. Gruber, G. Salaün, Specifying and Verifying the Synergy Reconfiguration Protocol with LOTOS NT and CADP, in: Proc. of FM'11, Vol. 6664 of LNCS, Springer, 2011, pp. 103–117.

[24] F. Boyer, O. Gruber, D. Pous, Robust Reconfigurations of Component Assemblies, in: Proc. of ICSE'13, IEEE/ACM, 2013, pp. 13–22.

[25] J. Fischer, R. Majumdar, S. Esmaeilsabzali, Engage: A Deployment Management System, in: Proc. of PLDI'12, ACM, 2012, pp. 263–274.

[26] F. Durán, G. Salaün, Robust and Reliable Reconfiguration of Cloud Applications, J. Syst. Softw. 122 (2016) 524–537.

[27] J. Seeger, R. A. Deshmukh, V. Sarafov, A. Bröring, Dynamic IoT Choreographies, IEEE Pervasive Computing 18 (1) (2019) 19–27.

[28] H. Koziolek, A. Burger, M. Platenius-Mohr, J. Rückert, G. Stomberg, OpenPnP: A Plug-and-produce Architecture for the Industrial Internet of Things, in: Proc. of ICSE(SEIP)'19, IEEE / ACM, 2019, pp. 131–140.

[29] B. D. Martino, M. Rak, M. Ficco, A. Esposito, S. A. Maisto, S. Nacchia, Internet of things reference architectures, security and interoperability: A survey, Internet Things 1-2 (2018) 99–112.

[30] J. Huang, S. Li, Y. Chen, J. Chen, Performance modelling and analysis for IoT services, Int. J. Web Grid Serv. 14 (2) (2018) 146–169.