Design and Deployment of Expressive and Correct Web of Things Applications

AJAY KRISHNA, Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, France
MICHEL LE PALLEC, Nokia Bell Labs, Paris Saclay, France
RADU MATEESCU, Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, France
GWEN SALAÜN, Univ. Grenoble Alpes, CNRS, Grenoble INP, Inria, LIG, France

Consumer Internet of Things (IoT) applications are largely built through end-user programming in the form of event-action rules. Although, end-user tools help simplify the building of IoT applications to a large extent, there are still challenges in developing expressive applications in a simple yet correct fashion. In this context, we propose a formal development framework based on the Web of Things specification. An application is defined using a composition language which allows users to compose the basic event-action rules to express complex scenarios. It is transformed into a formal specification which serves as the input for formal analysis, where the application is checked for functional and quantitative properties at design time using model checking techniques. Once the application is validated, it can be deployed and the rules are executed following the composition language semantics. We have implemented these proposals in a tool built on top of Mozilla WebThings platform. The steps from design to deployment were validated on real-world applications.

CCS Concepts: • **Information systems** → **Web applications**; • **Software and its engineering** → *Formal software verification.*

Additional Key Words and Phrases: IoT, behavioural modelling, composition, formal verification, web of things

ACM Reference Format:
Ajay Krishna, Michel Le Pallec, Radu Mateescu, and Gwen Salaün. 2021. Design and Deployment of Expressive and Correct Web of Things Applications. 1, 1 (December 2021), 30 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The Internet of Things (IoT) refers to everything connected via the internet. It encompasses everyday objects with internet connectivity, which enables them to communicate with other objects. Broadly, an IoT application involves physical devices and software objects deployed as lightweight nodes that communicate with each other to perform utilitarian tasks. The IoT ecosystem is rapidly growing towards playing a central role in many application areas like healthcare, transportation, agriculture, manufacturing, smart homes, and smart cities. It is expected that nearly 125 billion connected IoT devices are to be deployed by 2030 [43]. This explosive growth can be explained by understanding how IoT delivers value to our daily lives.

This rapid growth can be seen in the consumer market as well. Home users are buying more and more IoT devices from different vendors, with the aim of building applications which would make their daily lives easier. An IoT application is built by programming a set of IoT objects to communicate with each other and perform certain tasks. One of the salient features of home automation is that it is largely driven by End-User-Development (EUD) [13]. The popularity of EUD is due to the availability of intuitive, user-centric platforms like IFTTT [27], Node-RED [14] and Mozilla WebThings [45]. These platforms support a large set of IoT devices, thus eliminating the need for using vendor specific APIs and also

Authors' addresses: Ajay Krishna, Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, Grenoble, France, 38000; Michel Le Pallec, Nokia Bell Labs, Paris Saclay, Nozay, France, 91620; Radu Mateescu, Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, Grenoble, France, 38000; Gwen Salaün, Univ. Grenoble Alpes, CNRS, Grenoble INP, Inria, LIG, Grenoble, France, 38000.

2021. Manuscript accepted to ACM TIOT

allow users to visually program an IoT application by specifying the automation scenario as Event-Condition-Action (ECA) rules. As the name suggests, ECA rules are in the form of IF *event* THEN *action*, i.e., if an event is triggered, it will result in an action. An example of a simple ECA rule is **IF** *motion(on, true)* **THEN** *light(on, true)* where motion detection (*on* property is set to true) is the trigger event and if this condition is satisfied, then the resulting action is to turn on the light.

Existing ECA rule-based systems, notwithstanding their popularity, lack certain features when it comes to designing expressive IoT scenarios. IFTTT, one of the most popular automation platforms uses single event triggers in rules. Although it is simple and intuitive, it is not suitable for describing complex scenarios requiring more than single event triggers in rules [26, 54]. Another limitation is that when an IoT system involves multiple rules, they are executed in parallel, i.e., each rule is executed when the event condition is satisfied and the order of execution of rules is not guaranteed. So when users want to build scenarios involving an order of execution of rules or to specify a choice between two or more rules, they would have to use workarounds like chaining of events and actions. As an example, suppose one wants to design an application to detect intrusion in their backyard during the night. It involves two rules, one rule to turn on the lights in the backyard when a motion is detected and another rule to alert the owner if an intruder is captured on the camera in the backyard. Since these rules do not have an explicit order of execution, the lights might not be turned on when the camera captures the images of the intruder. This problem can be resolved by chaining of events and actions, but it is not very intuitive and it may lead to unsafe operations [25]. On a related note, tools like Philips Hue and Samsung Bixby provide the ability to perform a sequence of actions using Scenes or Routines. These are more expressive than simple ECA rules, as the occurrence of an event can trigger the execution of a set of actions rather than a single action. Still, the expressiveness is limited to sequences of actions, and the routines are executed independently of each other.

As users buy more objects and the number of rules increase, they may design possibly erroneous scenarios. Moreover, the IoT objects being reactive systems, they constantly interact with their environment and thus change their states. This behaviour of objects makes the IoT application highly dynamic and thus, susceptible to unexpected behaviour. Currently, there are no comprehensive mechanisms to validate the designs before deployment. The end-user tools mostly support checks at syntactic level (typing) and they do not check for behavioural correctness. Since the consumers are using IoT applications in sensitive areas like healthcare (e.g., blood pressure and heart rate monitoring, insulin checks, etc.), unsafe operations may cause bodily harm to the individuals. For instance, the number of people in domiciliary care has increased as a consequence of the stress faced by hospitals due to the Coronavirus outbreak [52]. Even though this is a home context, it is a critical environment. Consider a connected homecare where the following automation rules are configured (the home may have more rules i.e., the list is not exhaustive):

```
R1: IF AQM(quality,0) THEN window(open,false) ∧ purifier(on,true)
R2: IF AQM(quality,1) THEN window(open,true)
R3: IF camera(intrusion, true) THEN alarm(on, true)
```

The first two rules $R1$ and $R2$ are related to the air quality in the room. If the air quality outside is not optimal, then the air purifier is turned on with windows of the room being closed, otherwise, windows are kept open so that the patient gets fresh air and sunlight. At the first glance, the rules might seem safe, however upon analysing the system it is evident that the safety of the patient might be compromised as the automation might open the window at night which carries the risk of an intrusion. The rule $R3$ specifically deals with intrusion detection, where an alarm is raised if an intrusion is detected by the camera. Internally, it uses an analytics service on the cloud to recognize familiar faces.

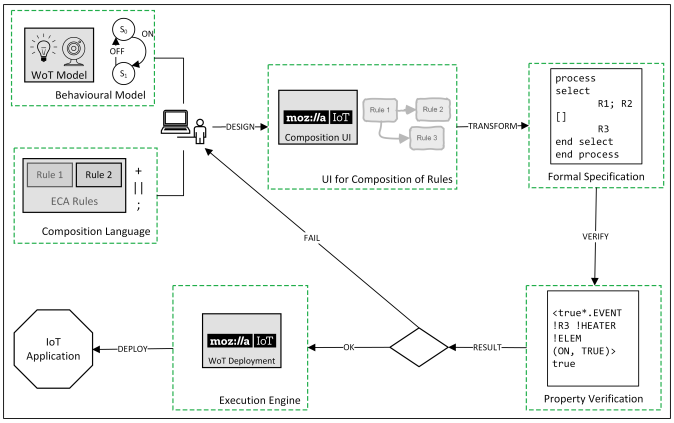


Fig. 1. Overview: Verified Composition and Deployment of WoT applications

This information is not captured in the rules and if the cloud-based service is inaccessible, it will result in the failure of the rule.

A way to ensure correct behaviour of the applications is by verifying at design-time that they work as intended upon deployment. Formal methods can be used to model complex cyberphysical systems as mathematical entities. The process of describing a non-mathematical representation of a system in a precise formal language is known as formal specification. Since the specification captures the behaviour of the system with precision and unambiguity, it can be used to mathematically verify the implementation. Model checking [1] is a formal verification technique where a system is modelled in terms of a finite state machine. This model is fed to a model checker which determines if the given model satisfies properties expressed as logical formulas. By formally modelling the IoT system we can define the behaviour of the different heterogeneous systems in a uniform and unambiguous manner. Further, we can specify the unexpected and intended behaviours in a system as properties and use model checking techniques to check whether these properties hold on the formal specification of the system. In the case of the domiciliary care example, by formally specifying the internal behaviour of the camera, we can capture its dependency on the cloud service. Further, a safety property specifying that the windows need to remain closed at night would have helped designers identify the flaw and add an additional rule to close the windows at night.

In this context, we propose an end-to-end solution for end-users that enables to design, verify, and deploy IoT applications with a few clicks of a button. Figure 1 provides an overview of our solution. A device or an IoT object in this work is represented as a set of operations with a behavioural model defining the order in which these operations must be called. This behavioural model is inferred from Mozilla Web of Things (WoT) Thing Description (TD) [16], a specification complementary to W3C's abstract data model and API for WoT [57]. A user can then select some of these objects for building a specific application based on her/his needs. She/he can define a set of rules with their ordering constraints using a composition language, which is backed by a user-friendly graphical interface. The language allows users to enforce an order of execution between two rules, a choice among several rules or specify simultaneous execution of multiple rules. Further, to ensure that these expressive compositions work as expected (e.g., free of erroneous or deadlocking executions, desired probability of successful execution), we provide formal analysis techniques for automatically verifying the composition. If the analysis return an unsatisfactory result, a diagnostic is provided, and the user can go back to the composition step to review the composition expression. Verification is achieved by transforming

the composition to a formal specification which implements the composition semantics and model checking techniques. Once the designed composition is validated, the application can be deployed. In this work, we rely on the Mozilla WebThings [45] platform for deployment. We extended the platform to support our composition language, which is more expressive than the language originally supported by this platform, and we built an execution engine to deploy the rules respecting the language semantics. It is worth noting that except for the design of the application the remaining steps in the approach are automated. Here we use the words composition and application quite interchangeably. In its truest sense, composition refers to how the rules are combined (design) and when a composition is deployed, it becomes an application. In our work, we primarily focus on smart homes, where WoT is predominantly used. However, the proposals can be suitably adapted to other domains that use WoT, such as Industrial IoT (IIoT), as the underlying principle of the object model remains the same.

To sum up, the main contributions of this paper are: i) a behavioural model for IoT applications based on WoT standard; ii) automated analysis techniques for verifying correctness of the applications specified using a composition language; iii) effective deployment of the application for running objects as specified in the composition specification; iv) tool support and validation of the approach on real-world use-cases in the context of smart homes.

The organization of this paper is as follows. Section 2 introduces the model of objects we rely on in this paper and the composition language. Section 3 describes the formal operational semantics of the composition language and its execution. In Section 4, the encoding of objects and composition into a formal language is described. This encoding serves as a basis for subsequently verifying certain properties of interest on the composition, which is described in the second half of this section. Section 5 introduces the tool support and related experiments to validate our approach. Section 6 covers related work and Section 7 concludes the paper with an insight on future work.

2 MODELS FOR WOT APPLICATIONS

This section begins with an introduction to the WoT specification which we use in our work. This is useful in understanding the various components of the WoT architecture. Specifically, we detail the Thing Description (TD) specification. In the next subsection, we present a behaviour model for objects which is based on the WoT TD specification and a composition language based on Event-Condition-Action (ECA) rules. Building a model based on a standard makes our approach widely applicable and specifically WoT offers a standard that is open, and it has a number of concrete implementations available as open-source tools for experiments and further extensions.

2.1 Web of Things Specification

The WoT specification [57] led by W3C aims to solve the fragmentation in IoT ecosystem due to the presence of numerous standards and protocols by providing an application layer that can be used to easily create IoT applications. It allows one to combine objects with disparate platforms and standards by representing these objects as software objects with programmable web interfaces independent of their underlying standards. The principles of web are chosen as the architectural style because they are universal and widely adopted across different application domains.

The WoT working group has published two specifications with Candidate Recommendation (CR) status at the time of writing. WoT Architecture CR [58] defines the abstract architecture for WoT. It defines a set of requirements for WoT implementations. It also introduces a set of WoT building blocks and describes how they fit within the WoT architecture. The WoT building blocks are viz., WoT Thing Description (TD) [59], WoT Binding Templates, WoT Scripting API, and WoT Security and Privacy guidelines. The TD specification defines a machine-readable description of Things (their metadata and network interfaces).

WoT Architecture. Now let us briefly describe some of the concepts from the WoT architecture. A Thing is an abstract representation of a physical or a virtual entity. It could be an IoT device, a virtual device, or a composition of different entities. The thing needs to be described in a standardized machine-readable description as specified by the TD. The entities that interact with things are called consumers. The WoT TD allows consumers to discover things and their capabilities. Further, it allows them to adapt to different protocols while interacting with the things.

The WoT concepts can be implemented across different levels of network architecture such as at device, edge, or cloud level. Broadly, three different integration patterns can be envisaged [15], viz., thing-to-thing, gateway, and cloud integration.

The gateway integration pattern is more relevant in the context of consumers. It allows devices that do not have HTTP capabilities or that are constrained on resources to run a HTTP server or ones that do not support the IP stack (Bluetooth, ZigBee) to use an intermediary in the form of a gateway to connect to the web. In this pattern, things do not interact directly, instead they communicate through the gateway. Here the WoT APIs are hosted at the gateway level instead of devices directly exposing the APIs.

In this work, we model a gateway pattern which allows us to use both IP and non-IP devices but without having to rely on cloud service providers. It is also easier to implement features at the gateway level, as we have more control of the system. Moreover, as both gateway and cloud can be viewed as an intermediary, the broad characteristics of the model remain somewhat similar and thus the approach is partly applicable to the cloud pattern.

The W3C WoT specification provides an abstract data model and architecture. For real world deployments, we need to use a concrete implementation that provides a JSON serialization of TD and an implementation of protocol bindings, such as HTTP and WebSockets. Mozilla WebThings [45] is a project by Mozilla that provides an implementation based on WoT specification. We built our work on the basis of this project instead of building our own concrete implementation to avoid duplication of work. Since the Mozilla implementation is open source, it gives us the flexibility to customize the application to our needs. In the subsequent section, we describe the Mozilla WoT TD [16] and Web Thing API [45] which are a concrete JSON serialization of the WoT TD specification and an implementation of the WoT protocol bindings using existing web technologies.

Thing Description. The TD specification describes the metadata of things in a machine readable JSON encoding. The important elements of the TD are Properties, Events and Actions. These are the three interaction affordances that can be used to interact with the device. An affordance refers to the fundamental properties that determine how the thing could possibly be used [47]. For example, a connected light typically may have two properties *on* and *brightness*, an action *fade*, and an event *overheated*. Here we describe some of the key elements in a TD.

@context is the reference to the semantic schema that defines the different types of object capabilities. There are different IoT schemas such as IoT Schema [53] and Mozilla Schema [44].

@type provides the string descriptions defined in the *@context* schema that identifies the device capabilities. The *@type* indicates to consumers the behaviour of the thing. It expresses the capabilities and possible interactions with the thing.

Properties affordance exposes the state of the thing to the consumer. Properties can be read by the consumer(and in some cases, also updated). Things can make the properties observable upon a change of state. Properties can be viewed as the attributes of the device. Each property has a primitive type (String, Integer, Boolean, etc.) and a semantic *@type* defined by the semantic schema.

Events describe the events than can be emitted by the device. The data from events are asynchronously logged and they can be processed by the consumers. Typically, events can be used to describe changes to more than one property or to describe a change in state that cannot be specified by change in properties.

Actions describes the functions that can be invoked on the thing to affect a change in state of the device. Actions can be used to invoke changes that manipulate states or trigger a change over time (process) on the object. Usually, these changes cannot be done by setting of properties. In the case of the connected light , the *fade* action manipulates two properties: brightness *level* and *duration*.

The WebThings API provides mechanisms to modify properties, read events, and send action requests by exposing the affordances as REST (representational state transfer) resources. The events resource exposes a log of events recently emitted by the device and actions resource provides access to queue of actions to be executed on a device.

2.2 Model for Objects

Here we define the behavioural model of the objects derived from their TD specification. This model captures how and when these objects can interact with the external environment. Let us now define the property of an object.

Definition 1 (Property)**.** Let K be the set of property identifiers, and for any $k \in K$, let V_k be the domain of values associated to k. A property p is a pair (k, v), where $k \in K$ is the identifier of the property and $v \in V_k$ its value.

A property refers to the state attributes of the device. It maps to the interaction affordances in the TD. Typically, IoT objects perform the role of a sensor or an actuator. This behaviour can be viewed as objects having state attributes whose values change over a period of time when they interact with the external world. Thus, from a modelling perspective, we associate the changes to state with the interaction affordances properties, events, and actions modifying a property.

```
{
  "name":"Play Motion",
  "type":"binarySensor",
  "@context":"https://iot.mozilla.org/schemas",
  "@type":["MotionSensor"],
  "description":" Hue Motion Sensor",
  "href":"/things/hue-1",
  "properties":{
    "on":{
      "title":"Present",
      "type":"boolean",
      "@type":"MotionProperty",
      "readOnly": true,
      "links":[ {
          "rel":"property", "href":"/things/hue-1/properties/on" }]}}
}
```

Listing 1. JSON Thing Description of Hue motion sensor

Let us illustrate this using an example of a simple motion sensor device, which detects the presence of motion in the environment. Listing 1 shows the partial TD of a motion sensor. It has a single observable property *on*. Depending on the type of object, its property values may change in a specific order (e.g., only after an alarm is triggered, it can be silenced). This information needs to be captured in the model to reason about correctness of its behaviour and of the composition in which it is involved. We chose to model this information in terms of a Labelled Transition System (LTS). An object is defined as a set of properties and an associated LTS acting on those properties. This definition of an object is applicable to both physical and virtual objects (including software elements).

Definition 2 (IoT Object)**.** An IoT object is a tuple $O = (P, LTS)$, where $P = \{(k, v) | k \in K \wedge v \in V_k\}$ is a set of properties, and $LTS = (S, A, T, s_0)$ is a state-transition graph describing the behaviour of the object, where:

- S is a set of states. Each state is a function $s : K \rightarrow \bigcup_{k \in K} V_k \ s.t. \ s(k) \in V_k$;
- $A = \{(d \ k(v)) | d \in \{event, action\} \wedge k \in K \wedge v \in V_k\}$ is a set of property labels;
- $T \subseteq S \times A \times S$ is the transition relation. A transition $(s_1, (d, k(v)), s_2) \in T$ (also noted $s_1 \xrightarrow{d \ k(v)} s_2$) indicates that the move from state s_1 to state s_2 is either an event or an action denoted by d and its associated property named k. $T = \{(s \xrightarrow{\text{event } k(v)} s)\} \bigcup \{(s \xrightarrow{\text{action } k(v)} s'[v/k])\}$, where $s'[v/k]$ indicates that the value of property k is updated to v;
- $s_0 \in S$ is the initial state, with initial values $s_0(k) = v_k^0$ for all $k \in K$.

Recall that a property is a state attribute of the object and thus a change in state of the object relates to events or actions operating on properties. An event is emitted by an object upon a change in its state and an action refers to the operation that can be invoked on an object to change its state. In other words, an event is an emission of a recently updated property and its value, and an action is a request to set the property to a certain value. Therefore in the Definition 2, only an action results in a transition to another state.

This behavioural model of an object can be derived by extracting certain values from its TD JSON. The Properties attribute in the TD maps to the Properties in the behavioural model, which form the labels of the LTS. **@type** semantic attribute provides information on the order in which these labels appear in the LTS. In case of simple objects, such as sensors and actuators, the model can be automatically generated as the behaviour is directly inferred from Definition 2 by considering all the (relevant) values for each property of the object.

Example: Figure 2 shows a simplified behavioural model of a motion sensor. In the TD of a motion sensor, there is a Boolean property *on* which represents whether a motion is detected by the object or not. Initially, the *on* property is in false state (s_0), denoted by concentric circles, and it can receive an *action* and set the *on* value to true. Once the value is set, it emits the *event* notifying the change in state.

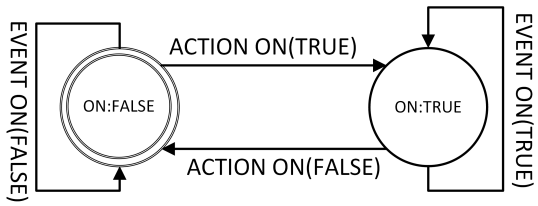


Fig. 2. Labelled transition system of a motion sensor

In case of complex objects, an expert can model the behaviour taking into account the semantic **@type** and also the object behaviour, which can be either described manually or can be learnt using appropriate automata learning frameworks [30].

2.3 ECA Rules and a Language for Composition

Since Event-Condition-Action (ECA) [9] programming is popular in smart homes, we use ECA rules as the basis for defining a composition. Further, we extend this notion by defining a language for composition of the ECA rules.

The composition of objects is built by combining the objects through Event-Condition-Action (ECA) rules. ECA rules are of the form IF something-happens THEN do-something. An ECA rule is triggered when an event emitted by a specific object satisfies certain condition(s) and, as a reaction, an action is sent to another object defined as target. Note that for the sake of simplicity, we denote an event and its trigger conditions as *EVT* and an action and its attributes as *ACT* in the subsequent text.

Definition 3 (Rule)**.** Given a set of objects $\{O_1, \ldots, O_n\}$, $O_i = (P^i, (S^i, A^i, T^i, s_0^i))$ where i is the identifier of object O_i, a rule R is defined as "**IF** EVT **THEN** ACT" where,
EVT ::= EVENT | $\text{EVT}_1 \wedge \text{EVT}_2$ | $\text{EVT}_1 \vee \text{EVT}_2$,
ACT ::= ACTION | $\text{ACT}_1 \wedge \text{ACT}_2$,
EVENT ::= $O_i(k, v)$, where $(k, v) \in P^i$,
ACTION ::= $O_j(k, v)$, where $(k, v) \in P^j$.

Here the ECA rules support the conjunction and disjunction of events in the *EVT* clause. However, it is to be noted that $\vee$ operator is not available in the *ACT* clause as introducing it would result in nondeterministic decisions. By allowing these logical connectives, one can combine events or actions from different objects and build more expressive automation scenarios.

A smart home can contain a number of rules ranging from a dozen to a few hundreds, whereas an individual application may involve only a few rules from all the available rules. Typically, these rules are executed in parallel, i.e., as and when the event triggers of a rule are satisfied, it gets executed. This execution model is simple, but it limits the expressiveness as one cannot define a specific order of execution of rules. There are many works in the literature that have identified the limits of expressiveness in ECA rules [3, 26, 55]. To be able to specify more expressive automation scenarios, we use a composition language with simple operators based on regular expressions.

Definition 4 (Composition Language)**.** A composition C is an expression built over a set of rules R using the operators Sequence (;), Choice (+), Parallel (||), and Repeat (^{k}) :

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

where,
$C_1; C_2$ represents a composition expression C_1 followed by C_2,
$C_1 + C_2$ represents a choice between expressions C_1 and C_2,
$C_1 || C_2$ represents the concurrent execution of expressions C_1 and C_2, and
C_1^k represents the execution k times of a composition expression C_1.

The sequence operator (;) allows one to define an order for the execution of rules. The choice operator (+) enables one to execute a rule from a set of rules in a group. Here the choice is an exclusive choice meaning that only one rule from the group is executed on a first-come-first-serve basis, whereas an inclusive choice would have executed all the rules satisfying the trigger conditions. The parallel operator (||) indicates that all the rules (compositions) in the operands are executed, irrespective of their order of execution. It is to be noted that both in parallel (||) operator and conjunction ($\wedge$) of events, the behaviour is not time bound, i.e., the rules can execute at any point in time and there will not be a timeout if a rule operand executes and remaining operands do not execute after a certain time. Finally, the repeat operator C^k introduces bounded looping of the rules. The composition as a whole restarts from the beginning once it reaches the end of the expression.

Example: Let us consider an application in the context of domiciliary care. Since the patient is bed-ridden, a few rules need to be configured to make her/his lives easier. When the patient wakes up, the lights of the room are turned on. A few seconds after s/he wakes up, a bed hygiene system is activated to enable the toilet usage. Simultaneously, if the outdoor air quality is measured to be good by an Air Quality Monitor (AQM), then a window is opened, otherwise an air purifying system is turned on to maintain optimum room condition. The patient prefers run the air purifier while he is awake as its running noise would disturb his sleep. Finally, if s/he falls asleep, the lights and the air purifier are turned off, bed hygiene system is deactivated, and the window is closed. The ECA rules corresponding to the scenario are shown in Listing 2 as formalised in Definition 3. The IoT application can be modelled using the composition language as follows: $R1; (R2||(R3 + R4)); R5$.

```
R1: IF wake(on,true) THEN light(on,true)
R2: IF timer(time,3) THEN bedhygiene(on,true)
R3: IF AQM(quality,0) THEN purifier(on,true)
R4: IF AQM(quality,1) THEN window(open,true)
R5: IF wake(on,false) THEN light(on,false) ∧ purifier(on,false) ∧
  bedhygiene(on,false) ∧ window(open,false)
```

Listing 2. Domiciliary care rules

The use of sequence (;) operator indicates an order and in this example rule $R1$ is executed before any other rule and $R5$ is the last rule to that will be executed. The rule $R2$ is executed along with either $R3$ or $R4$ as indicated by parallel (||) and choice operators (+), respectively.

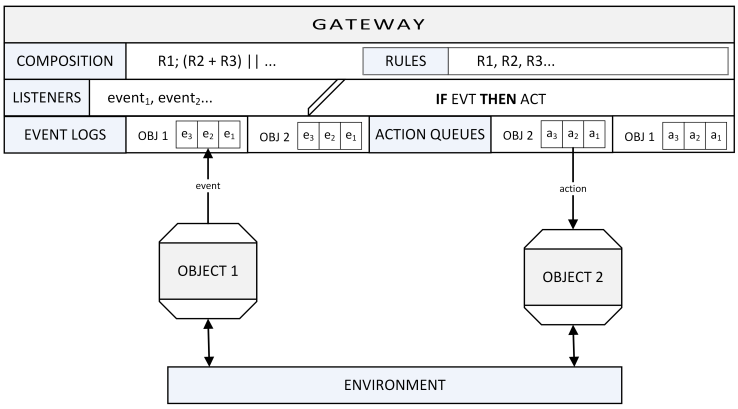


Fig. 3. WoT Gateway Communication

Let us now consider how the composition is executed in practice (the formal semantics of the execution is described in Section 3). An application consists of a set of objects and a composition expression built upon a set of rules. Among the different communication patterns supported by WoT, we considered an indirect pattern, where WoT API is exposed via a gateway infrastructure [58] as shown in Figure 3. The gateway hosts the rules, composition, event listeners, action queues, and event logs. It plays the dual role of a monitor and an actuator. It monitors the objects by listening

to the events. As an actuator, it controls the objects by sending requests to perform actions, thereby changing their state. Communication between objects occurs via the gateway. Running of an application involves processing the composition expression and executing individual rules. The events in the rules are triggered as the objects interact with the environment.

When a composition expression is processed, a subset of the rules is enabled, depending on the way the rules are composed. A rule may contain one or more trigger events and a set of resulting actions. When a rule is enabled, its events (properties) are monitored using a content-based subscription (listeners). So, when the events occur (properties change), the gateway is notified of the changes. Consequently, once the events are detected, the actions in the rule need to be executed. This is achieved through the gateway, by pushing action requests to the corresponding action queues of the objects. These actions (request to update property values) are executed when objects consume the requests from their queues. As the processing of the composition expression proceeds, a relevant subset of rules is enabled and disabled by the gateway components.

3 SEMANTICS OF EXECUTION

In this section, we formally define the operational semantics of the composition language along with the execution semantics of the WoT applications.

3.1 Semantics of Composition Expressions

We begin the definition by specifying the semantics of the composition expression. The execution of an expression is defined with the help of two functions *active* and *exec*. These functions are later used in the inference rules that define the overall operational semantics in Section 3.2.

During the execution of the composition expression, only a part of the expression is active. The function *active* determines the set of rules that are active in a composition expression (i.e., ready to be executed). This function is defined according to the different composition operators in Table 1. For instance, $active(C_1; C_2)$ will result in only a first part of the sequence expression (C_1) being active, whereas for choice (+) and parallel operators (||), the parts of expression operands on either sides will be active.

Expression	Active Rules
$active(R)$	$\{R\}$
$active(C_1; C_2)$	$active(C_1)$
$active(C_1 + C_2)$	$active(C_1) \cup active(C_2)$
$active(C_1 \| C_2)$	$active(C_1) \cup active(C_2)$

Table 1. Rule activation patterns

Further, the execution of the rules can be expressed in terms of a function *exec*, which generates the set of residual composition sub-expressions after executing a rule R in a composition expression C. The rule execution patterns are defined in Table 2. Readers may notice that we do not explicitly define the *exec* on the repeat operator C^k. Since it is a bounded repeat, it can be viewed as a bounded sequential execution of the expression, i.e., $\underbrace{C; \ldots\ldots; C}_{\text{k times}}$.

Rule Expression	Execution Residual
$exec(R, R)$	$\{\checkmark\}$
$exec(R, C_1; C_2)$	$\{C; C_2 \mid C \in exec(R, C_1)\}$
$exec(R, C_1 + C_2)$	$exec(R, C_1) \cup exec(R, C_2)$
$exec(R, C_1 \| C_2)$	$\{C_1^{'} \| C_2 \mid C_1^{'} \in exec(R, C1)\} \cup \{C_1 \| C_2^{'} \mid C_2^{'} \in exec(R, C_2)\}$

Table 2. Rule execution patterns

The function *exec* operates on the rule R to be executed in the composition expression C and therefore it is assumed that $R \in active(C)$. The composition expression evolves by executing a rule. $\checkmark$ is a special composition operator which we use to denote the successful execution of a rule (termination). The operator $\checkmark$ acts as a neutral element for the binary composition operators:

$$\checkmark \oplus C == C, where \oplus \in \{;, +, \|\}.$$

As defined in Table 2, execution of a rule results in a successful termination ($\checkmark$). Execution of a rule in an expression with a sequence operator, results in a residual expression. Choices are executed independently and their residual is the union of their individual residuals. In the case of parallel operator, the residual is the union of all executions obtained by executing each operands holding the remaining operands constant.

Example: Let us look at how the *active* and *exec* functions work in the IoT application corresponding to the domiciliary care composition expression $R1; (R2||(R3 + R4)); R5$ in Listing 2. Initially, when the *active* function operates on the expression, only the rule $R1$ is active and once the rule $R1$ is executed, the rules $R2$, $R3$, and $R4$ are active. Either of the rules $R3$ or $R4$ will be executed, along with the rule $R2$. Finally, $R5$ is activated with all the other rules being inactive. Now let us apply the *exec* function on the sequence $(R1; (\ldots))$, whose only active rule is $R1$ as defined by the *active* function. The steps of execution of the first part of the composition are as follows:

$$\begin{aligned}
C &= R1; (R2||(R3 + R4)); R5 \\
active(C) &= \{R1\} \\
exec(R1, C) &= \{C; (R2||(R3 + R4)); R5 | C \in exec(R1, R1)\} \\
&= \{C; (R2||(R3 + R4)); R5 | C \in \{\checkmark\}\} \\
&= \{\checkmark; (R2||(R3 + R4)); R5\} \\
&= \{(R2||(R3 + R4)); R5\}
\end{aligned}$$

Readers can notice that $active(C)$ results in $\{R1\}$ and execution of $R1$ in C by *exec* function is shown in the subsequent steps. A simplified view of the execution represented in terms of an LTS can be seen in Figure 4. In this LTS, the transition actions are the rules. Here we can notice the diamond pattern (interleaving) associated with the choice and the parallel execution. Notice that this represents only the execution of the composition expression which is only a part of the overall LTS of an application, which is described in the next section.

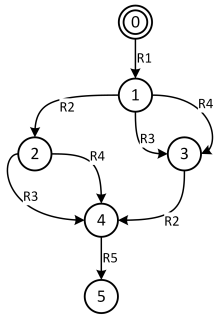


Fig. 4. LTS denoting the execution of the composition expression $R1; (R2||(R3 + R4)); R5$

3.2 Semantics of WoT Applications

Now let us define semantics of WoT applications. An application consists of a composition expression C, rules and objects $(O_1, O_2, \ldots, O_m)$. Each object O_i has a set of properties associated to it, denoted by $(P^i_1, P^i_2, \ldots, P^i_{n_i})$. The action queues corresponding to the objects $(O_1, O_2, \ldots, O_m)$ are denoted by $(Q^1, Q^2, \ldots, Q^m)$ respectively. The events that are part of the active rules are captured by the listeners. A schematic view of the notation is represented in Figure 5.

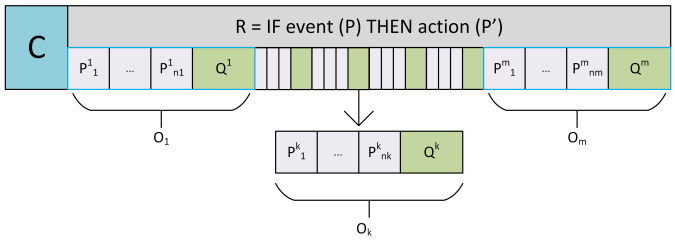


Fig. 5. Representation of a WoT application

The state of object O_i in the whole application is represented by its set of properties and the action queue $((P^i_1, P^i_2, \ldots, P^i_{n_i}), Q^i)$. The properties can also be denoted in terms of key-value pairs, $((\kappa, v)^i_1, (\kappa, v)^i_2, \ldots, (\kappa, v)^i_{n_i})$. Recall that an object O_i has an $LTS_i = (S^i, A^i, T^i, s^i_0)$, whose states are given by the properties of O_i, i.e., $(P^i_1, P^i_2, \ldots, P^i_{n_i}) \in S^i$.

The execution of an ECA rule consisting of an event and action can be expressed in terms of two inference rules, one inference rule for triggering the event and another for consuming the action from the object queue. The environment action, which is a special type of action bypassing the object queues, follows a slightly different execution semantics.

Before we define the inference rules, let us denote the application state as follows:

$$\langle((P^1_1, .., P^1_i, .., P^1_{n_1}), Q^1), .., ((P^k_1, .., P^k_i, .., P^k_{n_k}), Q^k), .., ((P^m_1, .., P^m_i, .., P^m_{n_m}), Q^m), .., C\rangle$$

where the superscript indexes $1, k, m$ represent the objects O_1, O_k, O_m respectively and their queues are denoted by Q with the superscript index of the object.

The triggering of an ECA rule R expressed in the form $IF\ event(P^k_i)\ THEN\ action(P^l_j)$ is given by the following inference rule TRIGGER-EVENT:

$$\frac{\begin{array}{c} R = IF\ event(P_i^k)\ THEN\ action(P_j^l) \in active(C) \\ C' \in exec(R,C) \qquad (\kappa,v)_i^k = P_i^k \qquad k,l \in [1,m] \qquad i \in [1,n_k] \qquad j \in [1,n_l] \end{array}}{\langle \ldots, ((\ldots), Q^l), \ldots, C\rangle \xrightarrow{event(P_i^k)} \langle \ldots, ((\ldots), append(Q^l, P_j^l)), \ldots, C'\rangle}$$

(TRIGGER-EVENT)

A rule R can be triggered under these premises: the rule R is active and event corresponding to the property P_i^k of the object O_k was listened, as denoted by $(\kappa,v)_i^k = P_i^k$. Here the rule checks directly the current state of the object. The emission of an event is not restricted in an object and it does not change the state of the object. Therefore, we do not define a premise that specifies an event transition in the object. The result of the trigger is that the property P_j^l (of the object O_l) in the action part of the rule is pushed to the action queue of the object O_l. This triggering of an event and the subsequent push of the action results in the execution of the rule, which is part of the composition expression C. Notice that, for a given state of the application, this inference rule can be applied for all rules $R \in active(C)$ and for all residual $C' \in exec(R,C)$.

The consumption of an action from the queue of object O_l is specified by the following inference rule CONSUME-ACTION:

$$\frac{\begin{array}{c} ((\kappa,v)_1^l, .., (\kappa,v)_j^l, .., (\kappa,v)_{n_l}^l) \xrightarrow[l]{action((\kappa,v')_j^l)} ((\kappa,v)_1^l, .., (\kappa,v')_j^l, .., (\kappa,v)_{n_l}^l) \in LTS_l \\ l \in [1,m] \qquad j \in [1,n_l] \qquad v' \in V_k \end{array}}{\langle .., ((\kappa,v)_1^l, .., (\kappa,v)_j^l, .., (\kappa,v)_{n_l}^l, Q_l.(\kappa,v')_j^l), .., C\rangle \xrightarrow{action((\kappa,v')_j^l)} \langle .., (((\kappa,v)_1^l, .., (\kappa,v')_j^l, .., (\kappa,v)_{n_l}^l), Q^l), .., C\rangle}$$

(CONSUME-ACTION)

The consumption of an action from the action queue is only possible if there is a transition in the LTS of object O_l that allows this update of property value. This is denoted by the premise:

$((\kappa,v)_1^l, \ldots, (\kappa,v)_j^l, \ldots, (\kappa,v)_{n_l}^l) \xrightarrow{action((\kappa,v')_j^l)} ((\kappa,v)_1^l, \ldots, (\kappa,v')_j^l, \ldots, (\kappa,v)_{n_l}^l)$

It indicates that the value v of property j of object l can be updated to v' through an action, i.e, the transition is possible in the LTS_l of the object O_l. The result of consumption of the action (property value $(\kappa,v')_j^l$) is an action transition in the application.

Environment action is a special action which does not go through the action queue. These actions are generated as objects interact with the environment (e.g., users manipulating the objects or changes in the physical environment). It is specified through the following inference rule ENVACTION:

$$\frac{\begin{array}{c} ((\kappa,v)_1^l, .., (\kappa,v)_i^l, .., (\kappa,v)_{n_k}^l) \xrightarrow{action((\kappa,v')_i^l)} ((\kappa,v)_1^l, .., (\kappa,v')_i^l, .., (\kappa,v)_{n_l}^l) \in LTS_l \\ l \in [1,m] \qquad i \in [1,n_l] \qquad v' \in V_k \end{array}}{\langle .., (((\kappa,v)_1^l, .., (\kappa,v)_i^l, .., (\kappa,v)_{n_l}^l), Q^l), .., C\rangle \xrightarrow{envaction((\kappa,v')_i^l)} \langle .., (((\kappa,v)_1^l, .., (\kappa,v')_i^l, .., (\kappa,v)_{n_l}^l), Q^l), .., C\rangle}$$

(ENVACTION)

Although environment actions do not go through the action queue, they can only be triggered if the action is possible in the (LTS of the) device, as specified in the premise of the inference rule. The result of environment action is that the property $(\kappa, v)_i^l$ is updated by a new value $(\kappa, v')_i^l$ of valid type.

In the concrete WoT implementation, listeners monitor the events corresponding to the active rules. Events are written to the event log and actions are pushed to the action queues. As shown in Figure 3, the events log and the action queues associated with the objects are implemented at the gateway level.

4 BEHAVIOURAL SPECIFICATION AND VERIFICATION

In this section, we formally encode the models of objects, as well as the communication of objects with other objects, the gateway, and with the environment. This encoding enables the automated verification of the composition correctness which is described in the second half of the section.

4.1 Encoding in LNT

LNT [4, 20] is a formal specification language supported by the CADP toolbox [19]. This work uses LNT for formally specifying the composition as it allows imperative programming style with support for data types and functions. Importantly, processes can be built compositionally using LNT operators and the language is expressive enough to encode the composition language semantics. It is to be noted that operational semantics [4] of LNT translates to LTSs and our encoding captures the operational semantics defined in Section 3.

The LNT operators that are used in the encoding are **select** to denote choice, **par** to specify interleaving or parallel composition, and **;** for sequential composition. Further, loops and conditions are encoded using constructs **loop** and **if** respectively. Variables in LNT are declared using the **var** keyword. LNT behaviours are encoded in a **process** and they are parameterized by gates 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. Change in property values is associated to gates *event*, *action*, and the special gate called *envaction*. Along with events and actions triggered by other objects, we need to account for change in object behaviour by the external environment. These are encoded in the form of *envaction*. For example, a user (environment) can turn off the light even when there is no rule manipulating the light. Listing 3 shows the outline of a generic object process. First, as mentioned in Section 2, an object can emit events. They are emitted 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 7). The property value *p1Val* 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 updating the property *p1* by *p1Val* and the delete operation, which removes the request from the *actionQ* as shown in line 9. It is to be noted that a successful execution of an action results in a change of state in the object and therefore, it follows emission of an event notifying the change. *envaction* is similar to a regular action except that it is triggered directly without going through the action queue(line 9). Finally, *done* is an auxiliary operation to track one complete execution of the application scenario. By default, a composition is designed to restart (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.

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

Listing 3. Outline of the LNT Process of an IoT Object

Now let us move on to the encoding of a rule. 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 which encodes *EVT* followed by the *ACT*. For example, rule R2 in Listing 2 can be specified in LNT as follows:

```
event(!timer, !r2, !elem(time, 3), !true);
action(!bedhygiene, !r2, !elem(on, true))
```

Here, *timer* is the object identifier, *r2* is the rule identifier and *elem* is the container that holds the key and value of the property. The suffix !true in *event* indicates that the rule is active when the event is triggered. This value can be either !true or !false, depending on the execution state of the composition. In this case, initially the event emitted by the timer appears with !true label and once the bedhygiene system is activated, *R2* becomes inactive and the event emitted by the timer will be visible in the later parts of the LTS with a !false label.

Table 3. Language to LNT Transformation Patterns

Operator	**Expression**	**LNT Pattern**
SEQUENCE	R1 ; R2	R1 ; R2
PARALLEL	R1 || R2	**par** R1 || R2 **end par**
CHOICE	R1 + R2	**select** R1 **[ ]** R2 **end select**
REPEAT	R1(k)	i:=0;**while** i < K **loop** R1 ; i:=i+1 **end loop**

Further, the transformation of the composition language to formal specification is shown in Table 3. The composition language operators are mapped to LNT operators in a process named COMPOSITION. Finally, the MAIN process which

describes the interaction between devices, environment, and a global listener, which keeps track of the events emitted by the objects, is specified. Listing 4 shows a generic outline of the MAIN process.

```
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 4. Outline of the Main Process

Here, the objects are interleaved without any synchronisation except on gate *done*, as they do not interact with each other and the interaction is achieved through the COMPOSITION process. On the other hand, the environment interacts directly with the objects and synchronises on the label *envaction*. The composition process COMPOSITION and the event listener GLOBALLISTENER are interleaved as their behaviour is independent of each other. The combination of COMPOSITION and GLOBALLISTENER represents the gateway. Finally, we synchronise on events and actions across the overall behaviour of the objects with the behaviour resulting from the interleaving of GLOBALLISTENER and COMPOSITION. Effectively, COMPOSITION acts like an orchestrator, which indicates the activation of rules and thus, the triggering of actions.

4.2 Verification Properties

This section describes several properties that are verified at design time using model checking techniques to ensure correctness of the WoT composition. These properties allows one to ensure that unintended behaviour does not occur upon deployment. We introduce two categories of properties: generic (deadlock freedom, completeness, spurious events and progress) and application-specific properties (functional and quantitative properties). Properties are specified using the Model Checking Language (MCL) [41], a data-handling temporal language equipped with the Evaluator model checker [40] of CADP. As described earlier, by encoding in LNT an application consisting of objects and a composition expression C, 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.

Deadlock freedom (generic). This classic property ensures that no rule of the composition expression is blocked from completing its execution. When objects are simple devices like sensors and actuators, rules are never blocked, as at any moment, a fully permissive environment can trigger events or actions in these devices. However, when objects involve software, for a specific event or action to occur, it may require another event or action to occur before, thereby creating a dependency. So, unless the preceding action or event occurs, the rule will not run to completion.

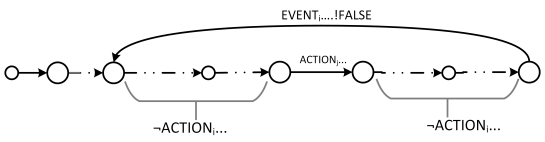
Completeness (generic). Complementary to deadlock freedom, this property indicates that all actions of the rules specified in a composition are reachable during the execution of the application. As the composition grows with a large number of rules, it is possible that some actions may never be executed, and this property helps to identify such rules. The successful execution of an action that assigns a new value v to a property k of an object O is denoted by a subsequent event emitted by O with the updated value v of property k. This can be ensured by checking, for each action ACTION !R !O !k !v encountered in the LTS_C, that afterwards, as long as the corresponding event EVENT !R !O !k !v was not executed, it is still possible to reach it, expressed in MCL as:

```
[ true* . {ACTION !R !O ?k:String ?v:String} . (not {EVENT !R !O !k !v ?any})* ]
⟨ true* . {EVENT !R !O !k !v ?any} ⟩ true
```

where the fields (k, v) are captured (as a string named k v) in the {ACTION...} label predicate and reused in the {EVENT...} predicate. Ideally, one would expect that the event is reached *inevitably* after the occurrence of the action. However, since we model an all permissive environment, this can prevent the inevitable reachability of the event (e.g., by continuously interrupting the processing of the actions from the action queue of an object). Therefore, in the property above we use *fair reachability* [51], which skips the possible unfair cycles of environment actions present in LTS_C before the desired event. This reasonably reflects the practical situations, in which the environment (e.g., users, weather conditions, daylight, etc.) acts in a much slower way than the objects.

Spurious events and progress (generic). Events are being continuously emitted by the devices indicating their current state. As the execution of the composition expression progresses, relevant rules are enabled and disabled. When a rule is enabled, the event being listened to appears with !true flag, otherwise it appears with !false flag. Only the events that are part of the rules appear in LTS_C. An event that continuously appears in LTS_C with !false flag is spurious, meaning that it is listened to, but never consumed by a rule. Consider an event $EVENT_i$ emitting the current value v of a property k of an object O. This event is spurious if it occurs with !false flag on a cycle of LTS_C that contains at least one action $ACTION_j$ not concerning property k of object O (meaning that the global execution of the composition expression progresses), but no other action $ACTION_i$ that could have caused the event $EVENT_i$. In other words, the action precursor of the event is missing.

If the MCL formula for checking spurious events holds on LTS_C, the model checker will produce a lasso-shaped witness as illustrated in the figure below:



Functional properties (specific). Safety and liveness properties can be easily verified using MCL. In a typical smart home scenario, a safety property could be that the air purifier is never turned on when the patient is not awake. This implies that in LTS_C, once the patient goes back to sleep, the air purifier should not be *on* in the subsequent states. This can be expressed in MCL as follows (*done* is used to limit the check to one instance of the composition expression):

```
[(not DONE)*.{'EVENT ?any !WAKE !"ELEM (ON, FALSE)" !FALSE'}.
(not DONE)*.{'EVENT ?any !AIRPURIFIER !"ELEM (ON, TRUE)" !FALSE'}] false
```

Quantitative properties (specific). The probability of executing an action in a composition can be computed to perform comparative analysis. This is achieved through probabilistic transition systems (PTSs) [36], i.e., LTSs with probabilities attached to transitions. Users can design different compositions, associate costs to actions and check for the probability of execution of an action across different designs to optimise the costs. For instance, in the home care scenario in Listing 2, turning on the air purifier system is an expensive operation (in terms of power consumption). The probability of execution of this action (considering that all transitions in LTS_C are equiprobable) is 0.497. However, if the parallel (||) operation in the composition is replaced by choice (+), the probability of turning on the air purifier reduces to 0.348. Although this will save energy, it changes the composition wherein the desired behaviour is not preserved.

In the computation, we considered all other transitions to be equiprobable, but we could also define specific probabilities to events and compute the probability of occurrence of an action. It is to be noted that since the composition expression can run infinitely, the probability of execution of an action may eventually aggregate to 1.0. Therefore, for probability analysis we need to consider the execution space of one instance of execution. Further, there are instances where a composition expression (structure) cannot be modified even if it does not seem cost-effective after quantitative analysis. In such cases, one may think of replacing the objects with similar behaviour by more cost effective ones (e.g., replace halogen bulbs with LED bulbs to be more energy efficient).

5 MOZART TOOL

This section describes the tool implementation of our proposals. The Mozart tool [34] is developed considering non-expert home users. It is developed on top of the Mozilla WebThings components. Later in the section, we present some of the usability and verification related experiments. The codebase of the Mozart implementation along with a sample of LNT models is available on Github[1].

5.1 WebThings Extensions Requirements

The extensions to Things gateway were identified by gathering the feature requirements. The following requirements were defined for the implementation of the tool:

- Users are to be provided with an interface to compose rules using the composition language. Users can graphically build a composition and visualize the overall application
- The interface should allow users to use the rules created by them using the Rule UI. New rules can be created from the composition interface as well
- Once the composition is built, the interface should provide a way to verify their compositions. The verification process should be seamless to the user and the results of verification should be available on the same interface

[1] http://ajaykrishna.github.io/mozart/

- If the result of verification is not satisfactory, users can revise their composition on the screen. Otherwise, they can proceed with the deployment of the application
- The UI should provide the option to deploy the composition. Deployment should kick-off the execution of the composition respecting the composition language semantics

5.2 Tool Components

In the context of design-time verification requirements we built three components on top of the Thing gateway: A UI for end-users, a verification component, and an execution engine. Users can graphically build the composition through a **UI component**. It is transformed into a formal specification by the **verification component**. The specification is verified using a verification toolbox and valid compositions are deployed using the **execution engine**. Now let us look at these components in detail.

The **UI component** is developed keeping in mind that the learning curve for the users is kept to minimum. The design principles of the ThingUI are carried over to the UI for composing the rules. Similar to the rules UI, users can drag-and-drop the rules and also the composition operators to build a composition. The drag-and-drop mechanism works like a jigsaw, users can drop a composition operator and then they can drop in the rules in the operand slots of the operator. In this way, they can incrementally build a composition expression. Once the composition is designed, users can click on the buttons on the bottom right of the screen to perform analysis of the composition. Users can verify the generic properties like deadlock freedom with a click of a button. For application specific properties, upon clicking the verify button, the UI provides an input screen for entering the MCL formula to be satisfied. The result of verification of properties is provided to users as a Boolean response indicating whether the property is satisfied or not. In case of a negative response, a textual response indicating the diagnostic of the violation is provided to the users. They are also provided with an option to visualise the composition as a Business Process Model Notation (BPMN) workflow [48]. Process models can be more intuitive for displaying large or complex composition scenarios [3]. Last but not least, users are provided a button to deploy the composition. The composition is deployed immediately upon clicking this button. Figure 6 shows a composite screenshot containing the different screens of the UI. On the top right, the BPMN workflow is shown. Further down, the MCL screen can be seen and the result of verification is shown on the bottom of the screen.

Fig. 6. Mozart UI screenshots

The second component is the **verification component**, which has two parts to it. The objects, rules, and the composition are represented as JSON objects. These JSON models are transformed into LNT formal specification through a model transformer. Along with this, necessary environment is generated in the LNT specification to simulate the scenario execution. The second part is the code that integrates the Mozart tool with the CADP verification toolbox [19]. Internally, the LNT specification is compiled and it is interpreted as an LTS. Further, the properties checked on this LTS and the results of verification are returned to the interface.

The final component is the **execution engine**, which takes care of the deployment of the application. The gateway provides a rule engine that can execute the rules. We built an execution engine that executes these rules in a specific order following the composition language semantics. As the composition expression can be seen as a workflow or an LTS (Figure 4), similar to execution of a workflow, the composition follows a token-based execution. The rules in a composition having the token are considered active and remaining ones are inactive. Active rules are executed by the rule engine as specified by the execution semantics in Section 3. Initially, all rules are considered to be inactive (disabled), then a set of rules are activated (enabled) based on the composition expression.

5.3 Technology and Implementation

A simplified view of the components and technologies used in the tool is shown in Figure 7. Mozilla WebThings Gateway is built on NodeJS. 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 specification is handled using Freemarker templating engine. Communication with the CADP verification toolbox is done via system calls.

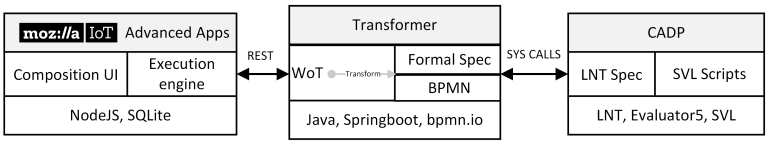


Fig. 7. Mozart: Components and technology stack

CADP Toolbox. The CADP toolbox [19] which provides first class support for LNT as a specification language, contains a wide range of tools to perform formal verification tasks. Here we enlist the tools that we used in our work. These tools can be invoked from the command-line.

- **LNT.OPEN** [4] is the tool that takes LNT specifications as input and connects them with the appropriate compilers for simulation, verification, and testing. We use this tool for compiling our LNT specification to obtain the LTS of the composition.
- **evaluator** [40, 41] is used for on-the-fly verification of a temporal property on the LTS (generated by LNT compilers in our case). It takes two inputs: an LTS encoded in one of the supported formats (BCG, LNT etc.) and a property encoded in MCL. The result of verification is a Boolean value, accompanied by a diagnostic (LTS subgraph) in case the result is false. **evaluator5**, which supports probabilistic transition systems is used for quantitative analysis and the remaining properties are checked using **evaluator4**.

- **svl** [17] is a script verification language used to automate the verification steps.

5.4 Evaluation

The end users of the Mozart tool are non-expert home users. So, we conducted experiments to determine how relevant are our proposals to the end users. Specially, we conducted usability, verification performance, and deployment experiments.

5.4.1 Expressiveness of Compositions. The first experiment focused on quantifying the expressiveness added by using the composition language for composition of rules. This was validated by taking a set of 36 IoT automation scenarios chosen from the literature [2]. These scenarios had a mix of simple and advanced scenarios that can not be easily described using simple "IF *events* THEN *actions*" rules. These scenarios were described in plain text (e.g. "*I am often in a hurry in the morning before I leave for work, but I usually need a cup of coffee to wake myself up. It would have been helpful if I could wake up to freshly brewed coffee.*").

Two programmers (P1 and P2) with three and one years of home automation experience, respectively, and knowledge of our work were asked to translate the textual descriptions of the scenarios to single event trigger rules (IFTTT style), Mozilla WebThings rules and as a composition of rules using the operators described in the work. Rules in Mozilla Web Things are similar to IFTTT model except that they allow OR and AND operators in events and AND operator in actions. It is also to be noted that the conjunction and disjunction operators in events need to be used exclusively in an expression, *i.e.,* the expression is either a conjunction of events or disjunction of events. Our tool is an extension of Mozilla Web Things and thus, it supports all the single event behaviours and Mozilla rules in addition to the composition of rules using the operators.

Initially, P1 and P2 independently classified whether the scenarios could be built using single event triggers, Mozilla WebThings rules and composition of rules in MOZART. Once P1 and P2 completed their designs, their responses were compared with each other. P1 and P2 designs were in agreement for 88.89% (Cohen's κ=0.71 [3]) of the scenarios, *i.e.*, their designs were the same or semantically similar. After discussions between the two programmers, a common translation was identified for scenarios that had diverging translations.

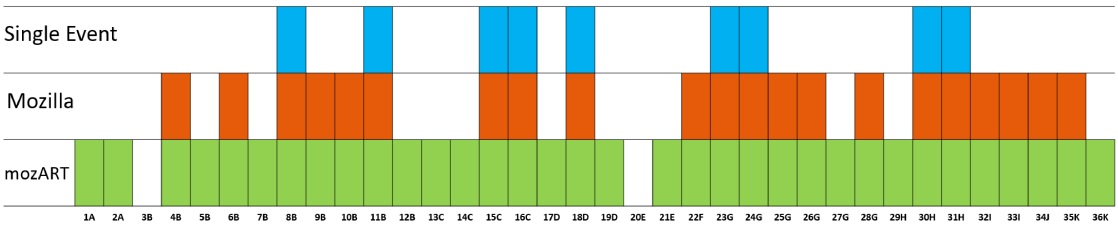


Fig. 8. Plot showing the scenarios with presence of bars indicating the possibility of implementing the scenarios.

A plot showing the list of scenarios and if they could be implemented using the three programming models is shown in Figure 8. Except for two scenarios, MOZART can implement all the scenarios compared to IFTTT and Mozilla WebThings. Scenario 3B refers to *"Start brewing coffee 15 minutes before my alarm"*, programmers were unable to translate *before alarm*, as the time of alarm was not available, i.e. the alarm time value could not be sensed.

[2]https://github.com/ajaykrishna/mozart/blob/mozart/scenarios/list.md
[3]https://en.wikipedia.org/wiki/Cohen%27s_kappa

5.4.2 User Feedback. The second set of experiments focussed on end user usability of the tool. It was measured using two experiments.

First, 26 users were involved in an online study (ages ranging from 18-55, median age group: 25-34). The users were chosen from diverse contexts (labs, student housing, apartments and from 3 different continents) which included academic researchers (23%), computer science students (29%), non-IT students (15%), and the remaining being non-IT professionals (32%). They were given a short training (~10 minutes) on the usage of the tool using a brief description of the tool and of its features. Then, the users were shown eight scenario descriptions in natural language. Further, users were shown how each of these scenarios can be designed using MOZART. A sample screenshot of the training interface is shown in Figure 9.

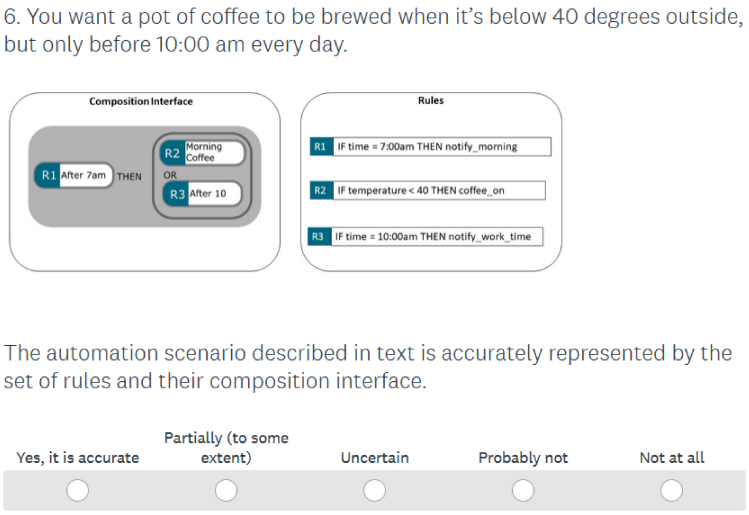


Fig. 9. Screenshot of the training and feedback screen used for Mozart tool users.

Along with this, they were asked how accurately the interface describes the scenario. This was done to gauge their understanding of the automation and more importantly to check the usability of representing the scenario in terms of graphical blocks. The interface was designed by two experts and it can be treated as the reference (correct) composition for the scenarios. The Likert scale[4] responses of the users is shown in Figure 10.

After walking through these scenarios, users were asked how comfortable they would be to use the tool to build advanced applications. As shown in Figure 11, nearly 60% of the users felt that they could use the tool to build advanced applications with the provided training.

As a second experiment, six users with varying programming knowledge (none to expert) from the 26 users were given two scenarios described in natural language and were asked to design the application themselves using the tool. The IoT devices required for the scenario were already connected to the WebThings platform. Users had to create rules and compose them to build the application. All the users were able to design two scenarios correctly, which took on average 6 and 8 minutes. The general consensus was that Sequence and Parallel operators were useful in

[4]https://en.wikipedia.org/wiki/Likert_scale

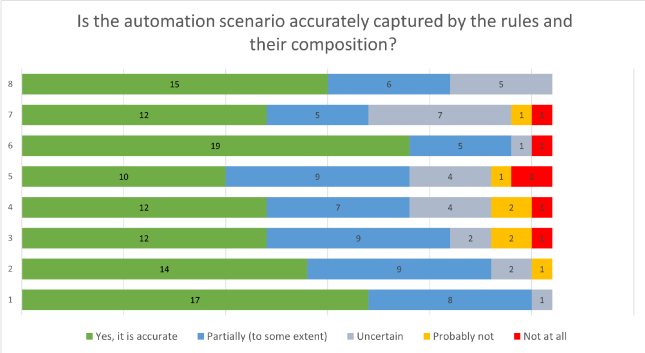


Fig. 10. A survey of 26 users stating how accurately the rules and composition match with the scenario description

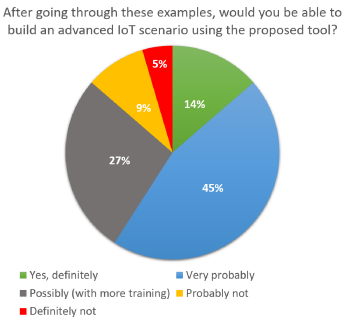


Fig. 11. Composition usability

designing applications. Interestingly, non-programmers were not convinced of the Choice operator, as it could introduce nondeterministic behaviour, thus giving the impression that the actions triggered in the composition are not fully under their control.

Beyond the results of the experiments, there were some interesting insights from the conversations with the users. First, there is some trust deficit towards IoT devices from users who have rarely used them. A recurring theme was that users wondered if the automation would really work, especially when it involved a bunch of rules. Second, although regular expression semantics are well known to programmers, its usage is not very well understood by the non-programmer users. One of the challenges during the training phase was to explain how AND and OR operators work in the composition. An indirect benefit of adding verification in automation is that it could increase the trust in IoT automation.

5.4.3 Verification Performance. We applied the design verification techniques (LNT, MCL, and CADP) on a number of typical smart home applications for validation purposes. Table 4 shows some of the performance numbers related

Scenario	Rules	Obj.	LTS (raw)		LTS (reduced)		Time
			States	Transitions	States	Transitions	m:s
1 ⟨;⟩	2	3	1747	24,064	60	479	00:03
2 ⟨; ||⟩	3	5	185,113	6,521,924	5696	110,360	00:20
3 ⟨; +⟩	4	5	236,953	12,652,228	14,408	305,064	00:36
4 ⟨;⟩	4	6	226,801	10,241,238	8240	186,488	00:30
5 ⟨+⟩	4	6	294,193	14,321,046	12,096	280,288	00:43
6 ⟨||⟩	4	6	693,361	36,817,662	25,712	632,752	01:34
7 ⟨; + ||⟩	5	6	374,977	25,244,136	21,944	544,388	00:54
8 ⟨; +⟩	8	10	445,825	36,016,106	21,408	509,472	00:57
9 ⟨; ||⟩	10	14	713,354	60,142,226	25,712	632,752	02:20
10 ⟨; + ||⟩	12	16	793,452	83,466,790	27,814	713,412	02:58

Table 4. Experiments on checking deadlock freedom

to different IoT applications. These applications were designed by expert users using the new UI and checked for deadlock freedom. Deadlock freedom is chosen because checking this property involves traversing the entire LTS graph and thus, provides an upper bound for the time taken to verify a property. The experiments were run on a host machine with Xubuntu 18.04 and a hardware consisting of Core i7-7600U processor, 256GB M.2 PCIe SSD, and 16GB of RAM. In Table 4, the first column is the scenario identifier with the operators used in the composition shown within angular brackets ⟨..⟩, the *Rules* column indicates the number of rules used in the composition along with the number of objects (*Obj.*) in the next column. We provide the LTS size in terms of number of *States* and *Transitions*. Reduced LTS size is also shown to indicate the size of the behaviour as raw LTS would contain internal transitions which may not be useful from a verification perspective. Reduction is done in two steps, first we rename the labels in the generated raw LTS by removing the Rule identifier as it is not required to verify the property of interest (deadlock freedom). Then, we minimize the renamed LTS using strong bisimulation [50]. The last column gives the time taken to perform transformation from Thing Description to formal specification in LNT, LTS generation and reduction, and verify deadlock freedom (rounded off to nearest seconds). It is to be noted that the time taken for deployment is not measured as it is negligible (less than a second) compared to verification time. Deployment simply requires activating the right set of rules and their corresponding listeners in JavaScript.

In Table 4, scenario 7 refers to the example described in Listing 2. It is interesting to note that the verification time is lesser than Scenario 6, even though it has more rules. This is explained by the fact that the events in the rules listen to only three objects *viz.,* wake-up sensor, timer, and an air quality monitor, whereas in Scenario 6, the rules listen to a greater number of events. Scenario 10 with 12 rules and 16 objects is quite large for a single application, and it takes under 3 minutes to get the verification result. In practice though, when we considered the scenarios mentioned in the literature [3, 26, 39, 54], they rarely required more than 8 rules. Our scenario with 8 rules took a reasonable amount of time (less than a minute) for verification considering the fact that the verification happens at design time.

5.4.4 Deployment Setup. The tool was evaluated by implementing and deploying scenarios in a local environment with IoT devices. The experiment setup hosted the Mozart tool on a Raspberry Pi server. In the same network, a set of connected devices which included Philips Hue lights, Hue Play bars, Hue motion sensor, luminosity sensor, thermometer, and connected speakers were added to mimic a smart home. A connected home setup involving some of the objects is shown in Figure 12. These objects were used to build smart home rules.

The set of rules were composed to build different compositions (some of them have been listed in Table 4). In some scenarios, when we did not have all the necessary objects, we defined virtual objects using the Mozilla API. Later, these compositions were deployed and tested for correct execution. The deployment was almost instantaneous as it just requires activation of JavaScript listeners. In case of real devices, the environment is the physical environment and we introduced changes in the environment. We validated the deployment engine by verifying that the actual states of the devices matched with the expected states (as part of the functional testing).

6 RELATED WORK

This section considers the related work to place our contribution in the context of the state of the art. First we detail the research focusing on models for IoT applications. Then we briefly touch upon the expressiveness of the ECA languages. In the subsequent text a survey on the verification techniques specific to IoT is presented. Finally, existing tools for designing and deploying IoT applications are surveyed.

Fig. 12. Smart home setup for experiments

Modelling IoT applications. In [12] the authors present a model for IoT configuration based on Answer Set Programming. A configuration consists of a set of constraints and requirements leading to a set of possible solutions. These solutions map to a set of component instances. Further, components have attributes and constraints encoded (e.g., home should have at least two rooms). This constraint based declarative model is solved using a SAT solver to generate a concrete solution(s). The approach is better suited for the initial setup of smart homes rather than designing specific applications at end user level. [33, 35] propose to model objects using behavioural models for compatibility checks and includes measures for correct deployment. However, there is no high-level language proposed for designing specific scenarios nor a rule-based system is used.

There are related works based on Model Driven Engineering (MDE) where IoT concepts are abstracted in a model away from underlying implementation details. These models can generate deployable code. ThingML [23] consists of a modelling language that provides IoT specific constructs for the design and implementation of reactive applications. It also provides a set of tools for model transformation (to UML) and code generation targeting multiple environments (Java, C, JavaScript). The design, development and operation of distributed services is handled by HEADS IDE, a set of Eclipse plugins which includes Kevoree tools for deployment and monitoring [6, 31]. The ThingML model relies on two structures âĂŞ Things and Configuration. A Thing consists of properties, messages, ports and a set of state machines. Ports are the communication interfaces which can send and receive messages. The internal behaviour is specified using a state machine or ECA rules to express reaction to events in a stateless fashion. The Configuration or the application specifies the coupling of Things. Here it is to be noted that ECA rules are used to describe the internal behaviour of objects, whereas in our work ECA rules define the interactions across objects. ThingML uses port or interface-based connections across objects without a compositional language. Our rule-based language assumes that all the objects included in the rules are connected via a network and the composition expression acts as an orchestrator controlling the order of interactions between the objects. Regarding validation, ThingML checker validates models at syntactic level and our work focuses more on the behavioural validation.

In [2], the authors propose a Networking Language Domain Specific Language (DSL) to specify the network of IoT devices. They model the device behaviour using ThingML. Then the designed network is controlled using the proposed policy language. Policies can be defined in terms of rules involving trigger conditions and actions. In [22], the authors use Elaboration Language, a DSL to describe the architectural model of IoT network. ThingML bridges the gap between domain knowledge and technical implementation. It provides an abstract state-machine language for automation which is not bound to any runtime or platform. Our work instead provides a solution based on a programming paradigm that is very popular in IoT and its underlying models are based on a W3C standard. This makes our solution relevant to end users especially the non-technical ones as the standard gets widely adopted. It is possible that platform experts could implement appropriate drivers in ThingML to target the WebThings platform.

In [5], the authors describe behaviour aware composition for WoT. They model the behaviour of things by extending the OASIS standard Devices Profile for Web Services (DPWS). The behaviour is represented as a finite state machine. The approach is quite similar to our approach as they account for the internal behaviour of the devices. However, the models are based on the DPWS standard, which although it can be applied to IoT, is more pertinent to web services. Moreover, the composition is defined at interface level and there is no language to specify the interactions between devices.

ECA languages. The expressiveness and limitations of ECA programming style adopted by IFTTT is discussed in [55]. They describe their findings by analysing 200,000 IFTTT recipes. Further, in [26], the authors provide solutions to clear the ambiguities and improve the expressiveness in ECA rules. The authors classify event triggers and actions in ECA rules in terms of their behaviour. They classify events as state-based or instantaneous, based on the time the trigger remains in a particular state. Similarly, the actions are classified as instantaneous, sustained, and extended. Using this classification, they propose a high-level ECA language that uses additional constructs, such as *WHEN*-event-trigger-*DO*-action, *AS-LONG-AS*-state-trigger-*DO*-sustained-action, etc. These constructs are useful, but introducing them in our work would have required additional training to users who are familiar with IFTTT style rules. Moreover, the implementation of this expressive language along with its operational semantics are not provided in the aforementioned work.

In [21], the authors present a tool for non-programmers to customize the behaviour of their web applications using ECA rules. They offer a domain-specific context dependent interface for building rules which allows users to create and reuse these rules in different applications. The ECA language presented here is quite similar to the language we use, but we do not allow OR in actions nor NOT operator as they induce ambiguity in the execution. Another important distinction is that we allow composition of rules based on regular expression operators.

In [39] the authors present a user-interface to debug ECA rules. It partly implements the constructs presented in [26]. The authors describe the UI prompts that can aid or prevent users from creating erroneous rules. Similarly, in [8], the authors propose intuitive UIs for creating rules with multiple triggers. Here the focus is on better interface design, not on the behavioural correctness of the application. Our interfaces are inspired by the design language of Mozilla and users found it to be quite intuitive (see the user assessment in Chapter 5).

Verification of IoT applications. [38] is a work that takes advantage of the satisfiability theories. The authors present a safety focussed programming framework for IoT applications. Users write ECA rules and aggregate these rules to build an IoT application, although there are no specific operators for combining the rules. Policies are specified as conditions that should never occur in the application. The set of rules and policies are fed to a safety engine which determines any safety violations in the specification. The engine checks two types of violations âĂŞ conflicts among the rules and

violations of policies. It uses a combination of SMT solving and runtime code exploration. Further in [37], the authors extend the work to support debugging of detected policy violations by non-expert users. They propose a framework that localises the fault and uses an SMT solver, to iteratively fix the rule parameters so that the safety properties are satisfied.

In [46], the authors present a building management system that allows users to build automation in large buildings using trigger-action programming. The automation is built using rules of the form IF something-happens THEN do something. Rules are mapped to predefined categories (classes of objects). They are considered to be in conflict when more than one rule is simultaneously active and specific actions cannot be satisfied at the same time. They identify the conflicts by encoding the rules as a propositional formula, which is fed to the Z3 SMT solver [7] to check for satisfiability. Rules are checked for conflicts at two levels: design time conflicts are identified statically, and runtime conflicts (which depend on the interaction with the environment) by simulation. Further, users can assign priorities to rules for conflict resolution. The runtime check is similar to the policy violation check proposed in [38]. Compared to the works [38, 46], our verification is not specific to rules and it considers the entire application behaviour for checking correctness.

In [32], the authors translate the ECA rules into Petri nets (PN). They use this PN model to analyse two correctness properties: termination and confluence. The two properties described in this work are generic properties that deal only with the rules, whereas our work supports user-specified properties and the model upon which these properties are verified accounts for the behaviour of the objects.

The tool vIRONy [56], based on the domain specific language IRON, provides a simulation environment to analyse ECA systems' behaviour. It has a formal verification component which uses SMT solvers and program analysis to check the safety and correctness of a set of ECA rules. [11] uses timed automata to formally model rules and events in a UI-based tool. Verification of termination and application specific properties is performed using the UPPAAL toolbox. In order to make the tool accessible to non-expert users, the authors use the property patterns such as absence, existence, precedence, etc., as identified in [10].

End user tools. There are several tools available for end users to design and deploy IoT applications and here we take a look at some of the more popular ones. IFTTT [27] popularized ECA rules in EUD. It provides a large repository of pre-defined ECA rules called Applets. These Applets can be configured to work with smart home devices. The Applets use single event triggers in the ECA rules. Similarly, Node-RED [14] provides Recipes to connect IoT devices visually. OpenHAB [49] is another home automation software that can be programmed by advanced users. It provides rule scripts in the form of *WHEN something-happens THEN do-something* and it allows logical disjunction of multiple triggers in the rules, if-else expressions, for loops, etc. Advanced automation can be built by writing scripts based on Xbase/Xtend. Samsung SmartThings [29] is another tool which supports designing of complex rules through webCoRE and SmartRules apps. Apple Workflow [28] and Microsoft Power Automate (formerly Flow) [42] are automation tools, but their focus is mostly on software service level automation.. There are a few areas of improvement that we have covered in our work. First, these tools do not check the validity of the designed application at behaviour level. The checks are limited to syntactic checks and matching of ports (data types). Second, simple automation rules are executed independently. There is no easy way to compose these individual rules to build a more advanced scenario. Further, the semantics of execution is usually not well defined in the documentation.

7 CONCLUSION

Here we summarize the contributions and take a look at the possible avenues for extending this work.

Models for the IoT. In this work, we proposed a formal model that describes the behaviour of the objects and their interactions. The behaviour of the objects is specified in terms of a Labelled Transition System (LTS), which describes how and when the objects can interact with their surroundings. This model is based on the Web of Things (WoT) Thing Description (TD) specification, that is part of the WoT standards proposed by W3C. Unlike other textual and semantic description of the objects, this model explicitly describes the internal behaviour of the objects and their interfaces in a generic manner, i.e., the description is not bound to any semantic schema. In the behavioural model, the change of a state in an object is described in terms of events and actions to support Event-Condition-Action programming (ECA), also known as Trigger-Action-Programming (TAP) rules. Further, an IoT application consists of a set of objects and their interactions defined by composition of ECA rules using a composition language whose operational semantics has been provided. Finally, an encoding of the application in LNT is defined which serves as the basis for verifying properties during the initial design of the application.

Verification for the IoT. In order to establish correctness of a designed application, we verify that the application satisfies certain properties. The properties that are checked can be categorized into generic and application properties. The generic properties identified are deadlock freedom, completeness, and detection of spurious events. The models can also be verified for application specific properties, such as safety and liveness properties. In addition to it, the probability of execution of events and actions in a composition can be computed by annotating the actions with probabilities. It is worth noting that all the property verifications happen during the design of the application.

From Design to Deployment. The contributions mentioned above have been proposed with an end-user in mind. Therefore, we built a tool support to check the usability and applicability of these solutions. Our Mozart tool was built by extending the Mozilla WebThings platform, which enabled us to take advantage of the large number of devices already supported by the platform. In addition to the support provided by the WebThings for ECA rules, Mozart is backed by a composition language that allows one to easily compose rules in a graphical way. Our composition language enables to build more expressive automation scenarios. The tool also supports the deployment of the IoT applications. Deployment is achieved through an execution engine which executes the rules respecting the semantics of the composition language. Usability related experiments indicate that the users find the tool useful and it requires minimum training for users to learn the new extensions to the WebThings tool. Although, the primary objective of the tool support is to serve as an implementation of our different proposals, it is found to be more expressive than existing UI-based ECA programming tools. The experiments also indicate that the tool can be used by the targeted end-users. Regarding the verification performance, scalability is a valid concern but since the verification happens at design time users might be more willing to spend time checking the composition before deployment.

Future Work. A long-term objective is to extend this work to support additional quantitative analysis. The lifetime and failure rate of the device can be modelled as Interactive Markov Chains (IMCs) [24]. Going further, network related analysis such as latency, delay and network failure rates can be computed for sensitive applications like in the domain of health monitoring, old age care, etc. Another long-term objective would be to re-use the proposed models and properties in other application domains. Applying the proposals to other domains, especially the ones which involve thousands of objects, will require optimizations at different level (e.g., UI, compositional verification techniques [18]) to improve the scalability.

REFERENCES

[1] Jan A Bergstra, Alban Ponse, and Scott A Smolka. 2001. *Handbook of process algebra.* Elsevier.

[2] Imad Berrouyne, Mehdi Adda, Jean-Marie Mottu, Jean-Claude Royer, and Massimo Tisi. 2019. CyprIoT: framework for modelling and controlling network-based IoT applications. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. 832–841.

[3] Julia Brich, Marcel Walch, Michael Rietzler, Michael Weber, and Florian Schaub. 2017. Exploring end user programming needs in home automation. *ACM Transactions on Computer-Human Interaction (TOCHI)* 24, 2 (2017), 1–35.

[4] David Champelovier, Xavier Clerc, Hubert Garavel, Yves Guerte, Frédéric Lang, Christine McKinty, Vincent Powazny, Wendelin Serwe, and Gideon Smeding. 2018. Reference Manual of the LNT to LOTOS Translator.

[5] Javier Cubo, Antonio Brogi, and Ernesto Pimentel. 2012. Behaviour-aware compositions of things. In *2012 IEEE International Conference on Green Computing and Communications*. IEEE, 1–8.

[6] Erwan Daubert, François Fouquet, Olivier Barais, Grégory Nain, Gerson Sunye, Jean-Marc Jézéquel, Jean-Louis Pazat, and Brice Morin. 2012. A models@ runtime framework for designing and managing service-based applications. In *2012 First International Workshop on European Software Services and Systems Research-Results and Challenges (S-Cube)*. IEEE, 10–11.

[7] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[8] Luigi De Russis and Fulvio Corno. 2015. Homerules: A tangible end-user programming interface for smart homes. In *Proceedings of the 33rd Annual ACM Conference Extended Abstracts on Human Factors in Computing Systems*. 2109–2114.

[9] Klaus R Dittrich, Stella Gatziu, and Andreas Geppert. 1995. The active database management system manifesto: A rulebase of ADBMS features. In *International Workshop on Rules in Database Systems*. Springer, 1–17.

[10] Matthew B Dwyer, George S Avrunin, and James C Corbett. 1998. Property specification patterns for finite-state verification. In *Proceedings of the second workshop on Formal methods in software practice*. 7–15.

[11] AnnMarie Ericsson. 2009. *Enabling tool support for formal analysis of eca rules.* Ph.D. Dissertation. LinkÃűping University Electronic Press.

[12] Alexander Felfernig, Andreas Falkner, Seda Polat Erdeniz, Christoph Uran, and Paolo Azzoni. 2017. Asp-based knowledge representations for iot configuration scenarios. In *19th International Configuration Workshop*. 62.

[13] Daniela Fogli, Rosa Lanzilotti, and Antonio Piccinno. 2016. End-user development tools for the smart home: a systematic literature review. In *International Conference on Distributed, Ambient, and Pervasive Interactions*. Springer, 69–79.

[14] OpenJS Foundation. 2020. *Node-RED: Low-code programming for event-driven applications.* https://nodered.org/

[15] Ben Francis. 2020. *Web of Things Integration Patterns.* https://docs.google.com/document/d/1H3coHbb3Bwd02_NJi4KEBONByUkq92_HsTk1IpfmACY/edit#heading=h.emba0uow6hmb

[16] Ben Francis. 2020. *Web Thing API.* https://iot.mozilla.org/wot/

[17] Hubert Garavel and Frédéric Lang. 2001. SVL: a scripting language for compositional verification. In *International Conference on Formal Techniques for Networked and Distributed Systems*. Springer, 377–392.

[18] Hubert Garavel, Frédéric Lang, and Radu Mateescu. 2015. Compositional verification of asynchronous concurrent systems using CADP. *Acta Informatica* 52, 4-5 (2015), 337–392.

[19] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. 2013. CADP 2011: a toolbox for the construction and analysis of distributed processes. *International Journal on Software Tools for Technology Transfer* 15, 2 (2013), 89–107.

[20] Hubert Garavel, Frédéric Lang, and Wendelin Serwe. 2017. From LOTOS to LNT. In *ModelEd, TestEd, TrustEd*. Springer, 3–26.

[21] Giuseppe Ghiani, Marco Manca, Fabio Paternò, and Carmen Santoro. 2017. Personalization of context-dependent applications through trigger-action rules. *ACM Transactions on Computer-Human Interaction (TOCHI)* 24, 2 (2017), 1–33.

[22] Tiago Gomes, P Lopes, J Alves, Pedro Mestre, Jorge Cabral, João L Monteiro, and Adriano Tavares. 2017. A modeling domain-specific language for IoT-enabled operating systems. In *IECON 2017-43rd Annual Conference of the IEEE Industrial Electronics Society*. IEEE, 3945–3950.

[23] Nicolas Harrand, Franck Fleurey, Brice Morin, and Knut Eilif Husa. 2016. ThingML: a language and code generation framework for heterogeneous targets. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*. 125–135.

[24] Holger Hermanns. 2002. Interactive markov chains. In *Interactive Markov Chains*. Springer, 57–88.

[25] Kai-Hsiang Hsu, Yu-Hsi Chiang, and Hsu-Chun Hsiao. 2019. Safechain: Securing trigger-action programming from attack chains. *IEEE Transactions on Information Forensics and Security* 14, 10 (2019), 2607–2622.

[26] Justin Huang and Maya Cakmak. 2015. Supporting mental model accuracy in trigger-action programming. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*. 215–225.

[27] IFTTT. 2020. *Every thing works better together.* https://ifttt.com/

[28] Apple Inc. 2020. *Workflow.* https://workflow.is/

[29] SmartThings Inc. 2020. *Smart Things: Add a little smartness to your things.* https://www.smartthings.com/

[30] Malte Isberner, Falk Howar, and Bernhard Steffen. 2015. The open-source LearnLib. In *International Conference on Computer Aided Verification*. Springer, 487–495.

[31] Pierre Jeanjean, Benoit Combemale, and Olivier Barais. 2019. From DSL specification to interactive computer programming environment. In *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering*. 167–178.

[32] Xiaoqing Jin, Yousra Lembachar, and Gianfranco Ciardo. 2013. Symbolic verification of ECA rules. *PNSE+ ModPE* 989 (2013), 41–59.

[33] Ajay Krishna, Michel Le Pallec, Radu Mateescu, Ludovic Noirie, and Gwen Salaün. 2019. IoT Composer: Composition and deployment of IoT applications. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 19–22. https://doi.org/10.1109/ICSE-Companion.2019.00028

[34] Ajay Krishna, Michel Le Pallec, Alejandro Martinez, Radu Mateescu, and Gwen Salaün. 2020. MOZART: Design and Deployment of Advanced IoT Applications. In *Companion of The Web Conference 2020 (WWW), Taipei, Taiwan, April 20-24, 2020*. ACM / IW3C2, 163–166. https://doi.org/10.1145/3366424.3383532

[35] Ajay Krishna, Michel Le Pallec, Radu Mateescu, Ludovic Noirie, and Gwen Salaün. 2019. Rigorous design and deployment of IoT applications. In *Proceedings of the 7th International Workshop on Formal Methods in Software Engineering, FormaliSE@ICSE 2019, Montreal, QC, Canada, May 27, 2019*. IEEE / ACM, 21–30. https://doi.org/10.1109/FormaliSE.2019.00011

[36] Kim G Larsen and Arne Skou. 1991. Bisimulation through probabilistic testing. *Information and computation* 94, 1 (1991), 1–28.

[37] Chieh-Jan Mike Liang, Lei Bu, Zhao Li, Junbei Zhang, Shi Han, Börje F Karlsson, Dongmei Zhang, and Feng Zhao. 2016. Systematically debugging IoT control system correctness for building automation. In *Proceedings of the 3rd ACM International Conference on Systems for Energy-Efficient Built Environments*. 133–142.

[38] Chieh-Jan Mike Liang, Börje F Karlsson, Nicholas D Lane, Feng Zhao, Junbei Zhang, Zheyi Pan, Zhao Li, and Yong Yu. 2015. SIFT: building an internet of safe things. In *Proceedings of the 14th International Conference on Information Processing in Sensor Networks*. 298–309.

[39] Marco Manca, Carmen Santoro, Luca Corcella, et al. 2019. Supporting end-user debugging of trigger-action rules for IoT applications. *International Journal of Human-Computer Studies* 123 (2019), 56–69.

[40] Radu Mateescu and José Ignacio Requeno. 2018. On-the-fly model checking for extended action-based probabilistic operators. *International Journal on Software Tools for Technology Transfer* 20, 5 (2018), 563–587.

[41] Radu Mateescu and Damien Thivolle. 2008. A model checking language for concurrent value-passing systems. In *International Symposium on Formal Methods*. Springer, 148–164.

[42] Microsoft. 2020. *Power Automate*.

[43] Sachin Mittal, Wang Tsz Tam, and Chris Ko. 2018. *Internet of Things: The Pillar of Artificial Intelligence*. https://bit.ly/3dbzOwz.

[44] Mozilla. 2020. *IoT Capability Schema*. https://iot.mozilla.org/schemas

[45] Mozilla. 2020. *WebThings: An open platform for monitoring and controlling devices over the web*. https://iot.mozilla.org/

[46] Alessandro A Nacci, Vincenzo Rana, Bharathan Balaji, Paola Spoletini, Rajesh Gupta, Donatella Sciuto, and Yuvraj Agarwal. 2018. BuildingRules: A Trigger-Action–Based System to Manage Complex Commercial Buildings. *ACM Transactions on Cyber-Physical Systems* 2, 2 (2018), 1–22.

[47] Donald A Norman. 1988. *The psychology of everyday things*. Basic books.

[48] Business Process Model OMG. 2011. Notation (BPMN) Version 2.0 (2011). *Available on: http://www.omg.org/spec/BPMN/2.0* 2 (2011).

[49] openHAB Community. 2020. *openHAB: a vendor and technology agnostic open source automation software for your home*. https://www.openhab.org/

[50] David Park. 1981. Concurrency and automata on infinite sequences. In *Theoretical computer science*. Springer, 167–183.

[51] Jean-Pierre Queille and Joseph Sifakis. 1983. Fairness and related properties in transition systemsâĂŤa temporal logic to deal with fairness. *Acta Informatica* 19, 3 (1983), 195–220.

[52] Madhuri Reddy, Nathan Stall, and Paula Rochon. 2020. *How coronavirus could forever change home health care, leaving vulnerable older adults without care and overburdening caregivers*. https://theconversation.com/how-coronavirus-could-forever-change-home-health-care-leaving-vulnerable-older-adults-without-care-and-overburdening-caregivers-137220.

[53] Schema.org. 2020. *Schema.org*. https://iot.schema.org/

[54] Blase Ur, Elyse McManus, Melwyn Pak Yong Ho, and Michael L Littman. 2014. Practical trigger-action programming in the smart home. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 803–812.

[55] Blase Ur, Melwyn Pak Yong Ho, Stephen Brawner, Jiyun Lee, Sarah Mennicken, Noah Picard, Diane Schulze, and Michael L Littman. 2016. Trigger-action programming in the wild: An analysis of 200,000 ifttt recipes. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. 3227–3231.

[56] Claudia Vannucchi, Michelangelo Diamanti, Gianmarco Mazzante, Diletta Romana Cacciagrano, Flavio Corradini, Rosario Culmone, Nikos Gorogiannis, Leonardo Mostarda, and Franco Raimondi. 2017. virony: A tool for analysis and verification of ECA rules in intelligent environments. In *2017 International Conference on Intelligent Environments (IE)*. IEEE, 92–99.

[57] W3C. 2020. *Web of Things at W3C*. https://www.w3.org/WoT/

[58] W3C. 2020. *Web of Things (WoT) Architecture*. https://www.w3.org/TR/wot-architecture/

[59] W3C. 2020. *Web of Things (WoT) Thing Description*. https://www.w3.org/TR/wot-thing-description/