



HAL
open science

Models and Verification for Composition and Reconfiguration of Web of Things Applications

Ajay Krishna Muroor Nadumane

► **To cite this version:**

Ajay Krishna Muroor Nadumane. Models and Verification for Composition and Reconfiguration of Web of Things Applications. Other [cs.OH]. Université Grenoble Alpes [2020-..], 2020. English. NNT : 2020GRALM067 . tel-03188299

HAL Id: tel-03188299

<https://theses.hal.science/tel-03188299v1>

Submitted on 1 Apr 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITE GRENOBLE ALPES

Spécialité : **Informatique**

Arrêté ministériel : 25 mai 2016

Présentée par

Ajay Krishna MUROOR NADUMANE

Thèse dirigée par **Gwen SALAÜN**, Professeur, Université Grenoble Alpes, et

Co-encadrée par **Radu MATEESCU**, Directeur de recherche, Inria Grenoble - Rhône-Alpes, et **Michel LE PALLEC**, Ingénieur de recherche, Nokia Bell Labs France

préparée au sein du **CONVECS Inria Grenoble - Rhône-Alpes** dans l'**École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

Models and Verification for Composition and Reconfiguration of Web of Things Applications

Modèles et vérification pour la composition et la reconfiguration d'applications basées sur le web des objets

Thèse soutenue publiquement le **10 déc. 2020**, devant le jury composé de :

Monsieur Olivier BARAIS

Professeur, Université de Rennes 1, Examineur

Monsieur Didier DONSEZ

Professeur, Université Grenoble Alpes, Président

Madame Pascale LE GALL

Professeur, CentraleSupélec, Rapporteur

Monsieur Michel LE PALLEC

Ingénieur de recherche, Nokia Bell Labs, Co-encadrant de thèse

Monsieur Radu MATEESCU

Directeur de recherche, Inria Grenoble, Co-encadrant de thèse

Monsieur Juan Manuel MURILLO RODRIGUEZ

Professeur, Universidad de Extremadura, Rapporteur

Monsieur Gwen SALAÜN

Professeur, Université Grenoble Alpes, Directeur de thèse

Monsieur Farouk TOUMANI

Professeur, Université Clermont Auvergne, Examineur



Abstract

The Internet of Things (IoT) applications are built by interconnecting everyday objects over a network. These objects or devices sense the environment around them, and their network capabilities allow them to communicate with other objects to perform utilitarian tasks. One of the popular ways to build IoT applications in the consumer domain is by combining different objects using Event-Condition-Action (ECA) rules. These rules are typically in the form of IF something-happens THEN do-something. The Web of Things (WoT) are a set of standards and principles that integrate architectural styles and capabilities of web to the IoT. Even though WoT architecture coupled with ECA rules simplifies the building of IoT applications to a large extent, there are still challenges in making end-users develop advanced applications in a simple yet correct fashion due to dynamic, reactive and heterogeneous nature of IoT systems. The broad objective of this work is to leverage formal methods to provide end-users of IoT applications certain level of guarantee at design time that the designed application will behave as intended upon deployment. In this context, we propose a formal development framework based on the WoT. The objects are described using a behavioural model derived from the Thing Description specification of WoT. Then, the applications are designed not only by specifying individual ECA rules, but also by composing these rules using a composition language. The language enables users to build more expressive automation scenarios. The description of the objects and their composition are encoded in a formal specification from which the complete behaviour of the application is identified. In order to guarantee correct design of the application, this work proposes a set of generic and application-specific properties that can be validated on the complete behaviour before deployment. Further, the deployed applications may be reconfigured during their application lifecycle. The work supports reconfiguration by specifying reconfiguration properties that allow one to qualitatively compare the behaviour of the new configuration with the original configuration. The implementation of all the proposals is achieved by extending the Mozilla WebThings platform. A new set of user interfaces have been built to support the composition of rules and reconfiguration. A model transformation component which transforms WoT models to formal models and an integration with formal verification toolbox are implemented to enable automation. Finally, a deployment engine is built by extending WebThings APIs. It directs the deployment of applications and reconfigurations respecting their composition semantics.

Résumé

Les applications de l'Internet des objets (IoT) sont construites en interconnectant les objets du quotidien en réseau. Les objets connectés collaborent ensemble, afin d'observer l'environnement les entourant, et agir sur celui-ci. Pour le grand public, l'un des moyens de créer des applications IoT consiste à combiner différents objets à l'aide de règles d'action conditionnelle (ECA). Ces règles se présentent généralement sous la forme « SI quelque chose se produit, ALORS faire quelque chose ». Le web des objets (Web of Things ou WoT) est un ensemble de normes et de principes qui intègrent les capacités du web et les styles architecturaux à l'IoT. Bien que l'architecture WoT associée aux règles de la ECA simplifie dans une large mesure la construction d'applications IoT, il reste des défis à relever afin que les utilisateurs finaux puissent développer aisément des applications avancées et correctes, en raison de la nature dynamique, réactive et hétérogène des systèmes IoT. L'objectif général de ce travail est de tirer parti des méthodes formelles pour fournir un certain niveau de garantie aux utilisateurs finaux des applications IoT. Cela permet, au moment de la conception, d'assurer que l'application conçue se comportera comme prévu lorsque celle-ci sera déployée. Dans ce contexte, nous proposons un cadre de développement formel basé sur le WoT. Les objets sont décrits à l'aide d'un modèle dérivé de la spécification Thing Description du WoT. Ensuite, les applications sont conçues non seulement en spécifiant les règles individuelles ECA, mais aussi en associant ces règles à l'aide d'un langage de composition. Ce langage permet aux utilisateurs de construire des scénarios d'automatisation plus expressifs. La description des objets et leur composition sont traduites dans une spécification formelle à partir de laquelle le comportement complet de l'application est identifié. Afin de garantir une conception correcte de l'application avant le déploiement, cette thèse propose de valider un ensemble de propriétés génériques et spécifiques sur le comportement complet de l'application. En outre, les applications déployées peuvent être reconfigurées au cours de leur cycle de vie. Le travail supporte la reconfiguration en spécifiant des propriétés de reconfiguration qui permettent de comparer qualitativement le comportement de la nouvelle configuration avec la configuration d'origine. La mise en œuvre de toutes les propositions est réalisée en étendant la plate-forme Mozilla WebThings. Un nouvel ensemble d'interfaces utilisateur est construit pour prendre en charge la composition des règles et la reconfiguration de l'application. Un composant permettant de transformer automatiquement les modèles WoT en modèles formels, pleinement intégré au sein d'une boîte à outils de vérification formelle a été développé. Enfin, un moteur de déploiement est construit en étendant les API de WebThings. Il dirige le déploiement des applications et des reconfigurations en respectant la sémantique de leur composition.

Acknowledgement

I would like to express my sincerest gratitude to all the people who have contributed to wonderful learning experience during the last three years.

First and foremost, I must thank my thesis director Prof. Gwen Salaün, without his guidance and supervision, the work would not have materialised. Prof. Salaün's experience, knowledge and passion for work has positively influenced my work. He was always willing to listen, understand the problem and guide me in the right direction. My heartfelt thanks to Prof. Radu Mateescu for his gentle and timely guidance which helped me resolve some of the most challenging problems of my work. I have immensely benefited from his wealth of experience which I will carry into rest of my work and life. My thanks to Dr. Michel Le Pallec for bringing an industrial perspective to my work. He was a perfect host whenever I visited Bell Labs, Paris.

I am also grateful to be working with highly talented people, who are among the best in what they do, as part of Convecs team at Inria. Prof. Hubert Gavel, was always there to help whenever I needed his advice. My thanks to Prof. Frederic Lang for his help on technical matters and also enriching discussions over lunch. I would like to thank Prof. Wendelin Serwe for his help especially related to LNT and modelling. I am thankful to Ms. Myriam Etienne for helping me in managing the administrative aspects of the PhD. My special thanks to all the non-permanent members of the team (PhD students, researchers, interns). I am grateful for their help not just in technical matters, but also in day-to-day life. They made my stay pleasant and fun. I consider myself lucky to be a part of this team where everyone was willing to take time out to help me whenever I popped up with questions.

Thanks to members of IoT Control Group at Bell Labs, Paris for their collaboration.

I must thank the reviewers of my thesis and the examiners for reviewing my work and participating in my thesis defense in a period like no other.

A big thanks to my friends and family, who have been a pillar of support and helped me to pursue my goals.

Contents

1	Introduction	1
1.1	Context and Motivation	3
1.2	Approach	8
1.3	Contributions	10
1.4	Chronology of Work	12
1.5	Thesis Structure	13
2	Related Work	15
2.1	IoT Application Models	15
2.2	Expressiveness of ECA Languages	19
2.3	Verification of IoT Applications	20
2.4	Reconfiguration of IoT Applications	23
2.5	End-User Tools in Smart Homes	25
2.6	Concluding Remarks	30
3	Background	31
3.1	Behavioural Models	31
3.2	LNT Language	33
3.3	Model Checking	39
3.4	Rewriting Logic	42
4	A Formal Model for the Web of Things	45
4.1	Web of Things Specification	46
4.2	Model for IoT Objects	50
4.3	ECA Rules and a Language for Composition	53
4.4	Specification in LNT	63
4.5	Specification in Maude	71
4.6	Concluding Remarks	74
5	Property Specification for the Web of Things	75
5.1	Design-time Verification with MCL	75
5.1.1	Generic Properties	76

5.1.2	Application Properties	81
5.2	Reconfiguration Properties	84
5.2.1	Seamless Reconfiguration	84
5.2.2	Conservative Reconfiguration	87
5.2.3	Impactful Reconfiguration	89
5.2.4	Reconfiguration Example	91
5.3	Concluding Remarks	92
6	Mozart Tool	93
6.1	Project WebThings	93
6.1.1	WebThings Gateway	93
6.2	WebThings Extensions for Design-time Verification and Reconfiguration	96
6.2.1	Design-time Verification Components	97
6.2.2	Reconfiguration Components	99
6.3	Technology and Implementation	101
6.3.1	CADP Toolbox	101
6.3.2	Maude System	103
6.3.3	Implementation	103
6.4	Evaluation	104
6.4.1	Expressiveness of Compositions	104
6.4.2	User Feedback	106
6.4.3	Verification Performance	108
6.4.4	Deployment Setup	110
6.5	Discussion	112
7	Conclusion	113
7.1	Models for the IoT	113
7.2	Verification for the IoT	114
7.3	Tool Implementation	115
7.4	Future Work	116
	Bibliography	119

Introduction

1

” *As machines become more and more efficient and perfect, so it will become clear that imperfection is the greatness of man.*

— **Ernst Fischer**

Austrian journalist, writer and politician

Over the years, communication technologies and electronics have evolved rapidly, and this has brought the world closer than ever. Among them, the internet has helped uncover new frontiers beyond traditional communication. The internet, in tandem with modern electronic devices, has shattered geographical and cultural barriers, created communities of unimaginable scale and reach, enabled a new breed of enterprises. The Internet of Things (IoT) is one such possibility unleashed by the advent of network communication and modern electronics. In a broad sense, IoT encompasses everything in the physical world that is connected to the internet and we refer this ‘everything’ as objects or devices. An object could be as simple as a light bulb that we use every day, or it could be a complex industrial controller connected to the internet. These objects can communicate with each other and perform utilitarian tasks. The International Telecommunication Union (ITU) defines IoT as: “*A global infrastructure for the information society, enabling advanced services by interconnecting (physical and virtual) things based on existing and evolving interoperable information and communication technologies*” [ITU16]. In this work, we call *IoT applications* a set of interconnected things designed to serve a specific purpose.

Kevin Ashton, a technologist, first introduced the term IoT in the year 1999 during a presentation at Procter & Gamble (P&G) to describe internet connected systems that interact with physical world through ubiquitous sensors [LLC13]. This definition stemmed out of his research on internet connected Radio-frequency Identification (RFID). However, the idea of connected objects that capture their surroundings is not new and it goes back even further than the advent of the internet. The Plain Old Telephony System (POTS) developed in the late 18th century can be considered as the first connected object system. It captured the sound as an analog signal and transmitted it using conductive wires. Later with the invention of radio, wireless

communication systems came into the market and they enabled a new generation of connected systems. A significant breakthrough in 1959 led to the birth of modern electronics: researchers at Bell Labs invented Metal–Oxide–Semiconductor Field-Effect Transistor (MOSFET), a semiconductor device widely used in digital and analog integrated circuits [Mus20].

If we want to look at an early instance of a more pertinent IoT application, it is the connected soda vending machine at Carnegie Mellon University (CMU). Researchers in the 1980s implemented an application to check if the soda vending machine was empty or recently restocked (the drinks would be warm if they were recently placed inside the machine) from the comfort of their offices [Uni70]. It was achieved using a circuit board connected to the ARPANET, the predecessor of the internet. It is worth noting that the concept of connected objects is not limited to the consumer domain. In 1995, Siemens introduced the M1 GSM module, which pioneered cellular machine-to-machine (M2M) communication [Bha18]. It further ushered a new era of industrial manufacturing with remote monitoring, control, and telemetry involving embedded control systems. As time passed, machines have evolved not only to understand the world around themselves but also to coordinate with other machines and take meaningful decisions.

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 [MTK18]. This explosive growth can be explained by understanding how IoT delivers value to our daily lives.

Simple sensors like a Bluetooth transceiver or a temperature sensor may seem only of specific use when viewed in isolation. The transceiver can exchange data with nearby objects and the temperature sensor can measure the body temperature. But when we connect these seemingly simple objects to the internet and enable them to communicate with each other, the possibilities are beyond the obvious. For instance, Bluetooth combined with GPS helped in contact tracing during the recent Coronavirus outbreak [Gre20]. Smart ¹ thermometers played a key role in tracking and reducing the spread of Coronavirus by aggregating the temperature data of consumers and accurately predicting the spread of virus even before symptoms appeared in patients [YS20].

¹prefixing smart to everyday objects is a way to convey the readers that the object has network and programming capabilities

1.1 Context and Motivation

As with every system that marches towards maturity, there are some growing pains that need to be overcome before the successful adoption of IoT ecosystem. We aim to solve the challenges in the IoT ecosystem through the usage of formal methods. Before we present the challenges in detail, let us take a quick look at the techniques associated with formal methods. It is important to introduce these techniques as to understand how they can fit in to tackle the challenges in the IoT domain.

Formal Specification and Verification Techniques

Formal methods are techniques grounded on mathematics and formal logic. Using these techniques, complex software, hardware, and cyberphysical systems can be modelled as mathematical entities. The process of describing a non-mathematical representation of a system in a precise formal language is known as formal specification. The rigorous syntax and semantics of the formal language enable formal deduction about the 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 [BPS01] 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. Equivalence checking [CS01] is another verification technique which determines whether two models are functionally or semantically equivalent.

Term rewriting [BN99] is a computational technique that deals with equations called rewrite rules. It is a reduction process where rewrite rules are applied to terms. A term could be a variable, constant or a result of an operation on variables and constants. Rewriting logic theory models the static and dynamic aspects of a concurrent system. It consists of an equational theory and a set of conditional rewrite rules representing the static and dynamic aspects [Mes12], respectively. Equational theory combined with the rewrite rules can specify how a concurrent system evolves with respect to its states.

The verification techniques mentioned above can be applied to IoT applications to validate that they behave as intended. Validation of IoT applications is important as there are certain characteristics intrinsic to the applications which make it difficult

to guarantee their correct behaviour. In the subsequent text, we describe these characteristics and how formal methods can be placed in this context.

IoT Application Characteristics and Formal Methods

Among the various characteristics of concurrent systems, the ones that make the development of IoT applications a challenge can be broadly described under three categories.

Heterogeneity: As the IoT ecosystem evolved, it leveraged various underlying technologies, standards, and protocols. There are various standards at different levels of IoT stack [Fuq+15].

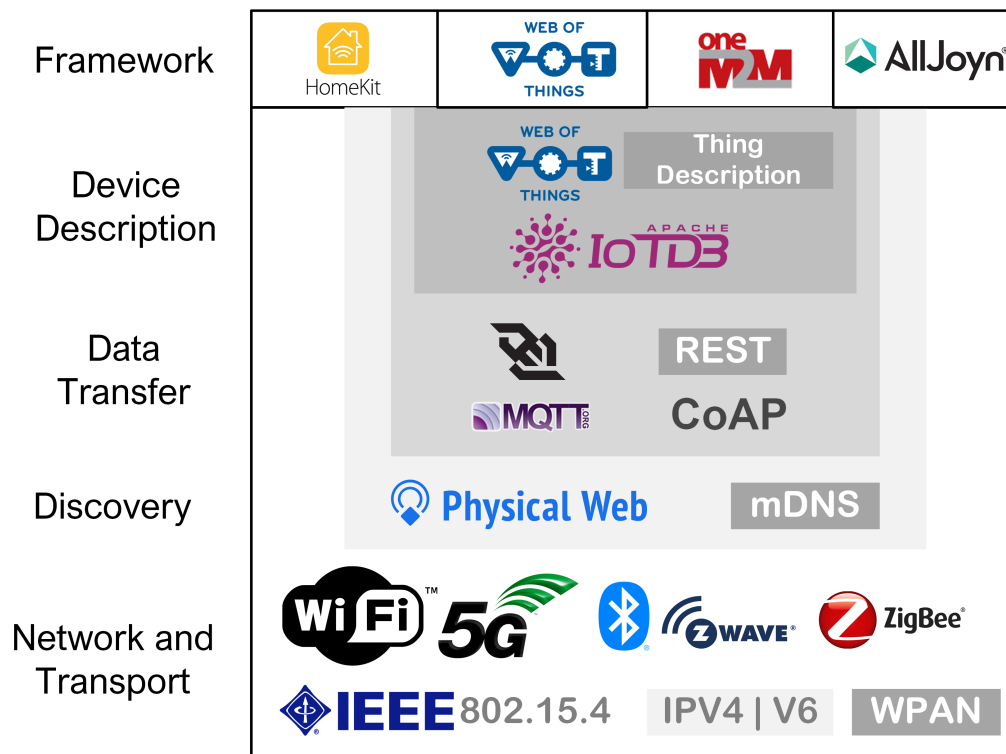


Fig. 1.1: IoT standards and protocols

Figure 1.1 provides a snapshot of existing standards in IoT across different layers of the network stack. The list of standards mentioned in Figure 1.1 is not exhaustive, but it tries to give an idea of the diversity present across different layers of the system. The reasons for this diversity are many: evolution of standards over time, proprietary standards, large number of small manufacturers who have constraints to implement standards etc. As Petrarch would say, variety is the cure; these standards provide

ample choice for developing IoT systems but, they bring in a baggage of interoperability issues. To build a truly global infrastructure for the information society as envisaged by ITU-T, having a single standard would be ideal. So efforts are focussed on building multi-layer frameworks sometimes reusing these protocols, to overcome fragmentation and allow systems with different underlying standards to be interoperable. There are various organizations and alliances working on this initiative. Internet Engineering Task Force (IETF) has proposed CoRE (Constrained RESTful Environments), OneM2M [Par20] is a Machine2Machine (M2M) standard for IoT, and Weave is an application level protocol backed by Google. Last but not least, the World Wide Web Consortium (W3C) has published Web of Things [W3C20c], a specification based on the principles of web [GT09].

Although these frameworks solve the problem of interoperability to a large extent, the underlying heterogeneity still remains and thus the difficulties in understanding the complete behaviour of the system. Formal specification of the behaviour can be of help in this area as one can define the behaviour of the different heterogeneous systems in a uniform and unambiguous manner. This specification can provide a platform agnostic definition of the system.

Complexity and dynamicity: The heterogeneity of IoT systems makes it complex to deploy and manage them. Along with this, another salient feature of an IoT system which adds to the complexity is its distributed architecture. IoT systems are built to be distributed, spread across network and geographical boundaries. IoT systems range from small home applications to large deployments like smart cities, manufacturing, etc., and a distributed architecture is readily scalable. However, as distributed systems exhibit a high level of concurrency, it can be a challenge to assess the consistency and correctness of these systems, especially when their behaviours are not clearly defined [Alu+16].

IoT objects being reactive systems, 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. Identifying unexpected behaviour and at the same time ensuring that required behaviour is in place can be a challenge, even more so in a dynamic system.

By specifying the unexpected and intended behaviours in a system as properties, one can check using model checking techniques whether these properties hold on the formal specification of the system. The property specification and verification can help detect errors in a dynamic and distributed system, which in effect allows one to obtain correctness guarantees with respect to the behaviour of the system.

Maintenance and evolution of applications: IoT applications, especially in the consumer domain are not built once for all. They are being constantly modified or upgraded just like any other consumer electronic device these days. Compared to traditionally stable applications like manufacturing lines, enterprise software, etc., the IoT applications evolve at a faster rate. For example, the change in an application (reconfiguration) could be due to a replacement when a device goes out of service or it could be an upgrade of the device software (functionality). In some cases an application could be entirely replaced to provide altogether a new service.

Evolution of an application involves replacement of an existing behaviour by a new behaviour. Under typical circumstances, the application behaviour is not completely replaced but rather users would like to partially preserve or extend the existing behaviour. One of the challenges in reconfiguration is to qualitatively measure the impact of the change with respect to the existing configuration. Additionally, the reconfiguration takes place during the execution lifecycle of the application and therefore, the impact to running applications should be kept minimal in order to preserve the functionality and quality of service as much as possible.

Term rewriting can efficiently track the evolution of the states in a system as rewriting is primarily a replacement of an expression representing the state of the object by an equivalent expression. Rewriting analysis can identify whether the states in a system are equivalent and also determine whether the two different systems lead to the same state (converge). These techniques can be used in the context of IoT applications in two areas: first, to track the evolution of the system with respect to the original system; and second, to identify the ways to evolve from the original system to the desired system maintaining the consistency of states.

Automation in Smart Homes

The problems mentioned in the earlier section require a range of solutions, some of which are beyond the scope of this work. Furthermore, even though these problems appear across various IoT domains, the solutions would be more meaningful if they are domain specific. So, we focussed our work on consumer IoT, specifically applying our approach on smart homes. Smart home automation is one of the fastest growing areas in IoT. Users can automate tasks in their homes using the smart objects available in the market and unlike industrial automation tasks, this is not limited to skilled or expert users. Usability is a key factor in any system to gain wide adoption, especially where end-users are consumers. In a survey of 250 participants at the 2017 Consumer Electronics Show (CES), almost all (>99%) participants

agreed that usability is an important area for IoT product vendors [Row17]. One of the reasons for smart home automation being accessible is the availability of easy to use UI-based tools like IFTTT [IFT20], OpenHab [Com20] etc. These tools allow one to design automation using rules. Typically, rules are in the form of *IF something-happens THEN do-something*. Here, *something-happens* is a trigger or an event that needs to happen and as a consequence *do-something* is the action that is being executed. Since these rules follow the specific pattern, they are known as Event-Condition-Action (ECA) rules and the way of programming automation using these rules is referred as Trigger Action Programming (TAP).

Now let us consider some scenarios in an Ambient Assisted Living (AAL) environment where an elderly person lives, and her/his life is made easier by the setting up automation using connected devices. Being a technology novice, s/he configures simple automation in the form of ECA rules to perform routine tasks. Suppose s/he has a set of objects and s/he has configured the following rules:

```
R1: IF bpm(abnormal, true) THEN hub(notify, bpm) AND
    counter(bpm, incr)
R2: IF counter(bpm > 3) THEN hub(call, 112)
R3: IF camera(intrusion, true) THEN alarm(on, true)
R4: IF thermometer(temp > 25) THEN window(open, true)
R5: IF motion(presence, false) THEN door(open, false) AND
    window(open, false)
```

The R1 rule uses a smart watch (e.g., Apple Watch) which actively monitors the health parameters such as blood pressure, heartrate, oxygen levels, etc. So, if an abnormality is detected, then a notification is sent to a centralized hub device at home (e.g., Google Nest Home) and it features a counter which is updated whenever the alert is received. Suppose if the abnormality is detected more than three times (user did not reset the counter), then it is possible that the user is unwell and emergency services are alerted of the incident as defined in rule R2. The rule R3 is configured to detect intrusions at home. It uses a smart surveillance camera (e.g., Nest IQ), which detects strangers and raises an alarm if an intruder is detected. Internally, it uses an analytics service on the cloud to recognize familiar faces. Further, the rule R4 uses a smart thermometer and when it gets warm in the house, windows are opened. Finally, R5 ensures that the doors and windows remain closed when nobody is in the house.

This home setup illustrates various aspects of smart home automation. It can be observed that IoT objects are more than simple sensors and actuators, they can perform more complex operations (e.g., analytics) and thus have an advanced

behaviour (e.g., the hub and surveillance camera). It is important to consider the behaviour of the objects while designing the rules. For instance, in case of R3, the intrusion detection feature is dependent on the cloud service and this information is not captured in the automation rules. Further in R1, if the user replaces his watch by another watch of a different make, which does not have an arrhythmia detection feature, then this abnormality will not be captured by the rule. It can have serious consequences on the health of the user. This problem arises because there are no mechanisms to check if the automations are functionally equivalent. In most of the automation setups, the rules are run in parallel (whenever the events occur). Now considering the rules R4 and R5, there is no guarantee that the windows are always closed when nobody is in the house (temperature can be > 25). This can be a security issue as the windows might open when the user is away from the home. There are no checks at design time that help users detect unsafe automation.

Considering these aspects, our goal is to enable users design applications in a simple manner, yet provide them with means to ensure that the applications behave as intended upon deployment. This correctness guarantee is accomplished by considering not only the rules but also the behaviour of the individual objects.

1.2 Approach

As mentioned earlier, our work leverages formal methods to address the challenges presented in Section 1.1. Formal modelling allows to specify the behaviour of IoT applications in an abstract manner, independent of the underlying heterogeneous components. This specification can be analysed using formal verification techniques to ensure that the system behaves as intended, i.e., it satisfies certain properties of interest. The specification of an application encompasses objects, rules, and their interactions with the physical environment.

We specify the application on the basis of a multi-layer framework, as these frameworks provide horizontal integration which bridges different protocols. Thus, it allows us to focus on specifying the application behaviour without worrying about underlying protocols. It also makes our approach applicable to all the real-world objects and applications which are supported by the framework. The Web of Things (WoT) is a framework that helps to mitigate the interoperability issues by abstracting the underlying details, yet providing necessary interfaces for the users to develop IoT applications. It uses architectural styles and principles of the web, where each object is identified by its URI. The object's capabilities and the ways to interact

with it, are specified by a machine-readable description called Thing Description (TD) [W3C20b].

As a first step, we specify the behaviour of an object on the basis of its TD. The behaviour is described in terms of a Labelled Transition System (LTS), a state-transition graph where the system moves from one state to another state depending on the action performed either by the system itself or an external environment. These objects can be used in defining the automation rules. Rules are in the form of *IF something-happens THEN do-something*. Further, we use a language for composing these rules, which allows one to define more advanced applications in comparison to running the rules independently. Finally, the execution of the application is captured by specifying an environment which interacts with the objects and thus affects the rules and their composition.

Once the model of the application is available, we identified a set of properties that can be checked on the model. These properties are specified as temporal logic formulas. If a model fails to satisfy any of these properties, it indicates that the application behaviour may not be as expected and therefore the design of the application needs to be revised. Once the design is satisfactory, one can proceed to deploy the application.

Regarding the implementation, the application is formally specified in LNT [Cha+18; GLS17], a language with process algebra semantics and programming language syntax. LNT models can be compiled and analysed in CADP [Gar+13], a formal verification toolbox. The correctness properties are specified using Model Checking Language (MCL) [MT08], a data-handling temporal logic suitable for expressing properties of value-passing concurrent systems. The compilers of CADP translate LNT specifications to LTSs on which the MCL formulas are checked and corresponding diagnostics are produced on the fly.

Further, to support possible evolutions of the application over time, we identify reconfiguration properties allowing the designer to compare the current application with a new one. The properties check whether it is possible to deploy the application maintaining the consistency of states of objects, if the behaviour of the current application is preserved in the new application, and the impact of the new application. This check is achieved by specifying the composition and properties in Maude rewriting logic [Cla+07] as it allows us to track the evolution of the states of objects in the application through rewrite equations.

Users of IoT objects need to be aware of their capabilities and they should be easily able to exploit these capabilities to build useful applications. In the consumer domain,

there are broadly two ways in which users can build applications. First, users with programming knowledge can take advantage of the device and platform APIs like Android Things to build applications. Users who lack programming experience or those who prefer not to use it, can build IoT applications using UI-based tools such as IFTTT [IFT20] and Node-RED [Fou20]. These tools abstract the programming APIs and present users with simple drag-and-drop interfaces to build applications. On the surface it seems that this approach is working well as there is significant adoption of these tools from the general public, but if we dig deeper we find two issues: i) users face difficulties in translating the applications they have in mind into the UI tools [Bri+17]. This is partly due to the fact that these tools are not sufficiently expressive to capture the user requirements [Ur+14; HC15]; and ii) even if the users could design intended applications, they face difficulties in understanding the complete behaviour of the designed application [Bri+17] and if the behaviour is not fully understood, there is no guarantee that the system works as intended.

Keeping these usability issues in mind, we built a tool named MOZART (**M**ozilla **A**dvanced **R**ule **T**rigger) [Kri+20] by extending the Mozilla WebThings platform [Moz20b], which provides a concrete implementation based on the abstract data models and architecture specified in the Web of Things. MOZART provides a user-friendly interface to design compositions, verify them, and deploy them. The expressiveness of the designs is improved by using a composition language based on regular expressions. Verification provides design-time guarantees on the application behaviour.

Once an application is deployed, the users can perform reconfiguration analysis using the MOZART tool in case they need to update the design. Reconfiguration is supported through an interface that allows users to redesign the application. The interface connects to verification components to perform reconfiguration analysis and provide the result of the analysis to the end user. Finally, reconfigured applications can be deployed in a consistent state using the support provided by the tool. It is worth mentioning that the tool hides the intricacies of formal verification from the end-users by automating the verification process behind a simple user interface.

1.3 Contributions

Our work provides an end-to-end solution i.e., from design-to-deployment for IoT applications. Specifically, by integrating formal methods, it allows one to ensure that

the designed system behaves as expected. In this context, our major contributions are:

- *Models for objects and their composition*

A behaviour model of IoT objects and their composition is proposed, which helps to understand the behaviour of the application in a uniform and unambiguous way, considering the heterogeneous nature of IoT systems.

The composition of objects is specified by composing Event-Condition-Action (ECA) rules using a composition language. The syntax and operational semantics of composition is defined in this work.

- *Property specification for design and reconfiguration analysis*

Support for verification of the composition is achieved by encoding the model and its behaviour in a process algebraic language (LNT) and in rewriting logic (Maude).

Correctness properties during the design of WoT applications are identified and specified in temporal logic. Verifying these properties helps to detect possible issues in the designed systems before deployment.

Support for reconfiguration of IoT applications is provided by specifying properties that qualitatively compare an existing application with the newly designed application.

- *Tool support*

Transparent integration of formal methods and verification techniques with the Web of Things platform is achieved by developing an easy-to-use UI tool for end-users. This is achieved by extending the Mozilla WebThings platform.

Support for deployment of designed applications is provided through a deployment engine which executes newly designed IoT object compositions and also deploys reconfigured applications seamlessly.

1.4 Chronology of Work

This thesis began in October 2017 at Inria Grenoble in collaboration with Nokia Bell Labs, Paris. The broad objective of this thesis was to integrate formal verification for correct composition and reconfiguration of IoT applications, with focus on consumer IoT.

In the first year of the thesis, we began experimenting on the Majord’Home [Bou+14], an SDN-based platform developed by Bell Labs for deployment of IoT applications. On top of this platform, we proposed a generic interface-based IoT model for composition and verification [Kri+19b]. It relied on a state based LTS model which captured the behaviour of IoT objects in the state transition system. This model of objects is quite similar to the one presented in this thesis as it specifies the objects’ internal behaviour. Unlike our recent work WoT, the composition of objects was defined using bindings. A binding is the connection between an output of an object to the input of another object. The objects and their bindings defined the overall behaviour of the composition, which was checked for compatibility. All these proposals were implemented in a tool name IoT Composer [Kri+19a] which was connected to the Majord’Home platform to deploy the composition respecting the bindings. This approach was targeted to IoT applications with a static communication pattern using a simple interface (port) based wiring for generating the composition. In our subsequent work, we use an ECA-based language for defining more expressive compositions and relied on a standard framework, namely WoT.

This thesis covers only the subsequent work on WoT which we did mostly in the second year as it is more relevant in the context of consumer IoT.

In the final year of my thesis, we mostly worked on the quantitative analysis of IoT applications. We built specifications that allow us to perform reliability analysis such as Mean Time to Failure (MTTF) computation and network latency analysis. Since this work is still ongoing at the time of writing, and to keep the manuscript coherent, we have chosen not to present this work and its results in this thesis. Instead, it is mentioned among the future works in the concluding chapter.

Publications

- Ajay Krishna, Michel Le Pallec, Alejandro Martinez, Radu Mateescu, and Gwen Salaün. “MOZART: Design and Deployment of Advanced IoT Applications”. In:

Companion of The Web Conference 2020 (WWW), Taipei, Taiwan, April 20-24, 2020. ACM, 2020, pp. 163–166

- Francisco Durán, Gwen Salaün, and Ajay Krishna. “Automated Composition, Analysis and Deployment of IoT Applications”. In: *Software Technology: Methods and Tools - 51st International Conference, TOOLS 2019, Innopolis, Russia, October 15-17, 2019, Proceedings*. Vol. 11771. Lecture Notes in Computer Science. Springer, 2019, pp. 252–268
- Ajay Krishna, Michel Le Pallec, Radu Mateescu, Ludovic Noirie, and Gwen Salaün. “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, 2019, pp. 21–30
- Ajay Krishna, Michel Le Pallec, Radu Mateescu, Ludovic Noirie, and Gwen Salaün. “IoT Composer: Composition and deployment of IoT applications”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE. 2019, pp. 19–22

1.5 Thesis Structure

Chapter 2 covers the related work. It sheds light on the existing tools in smart home automation, discusses usability of these tools with regards to expressiveness, and reviews the works related to verification and reconfiguration of IoT compositions. Our contribution is placed in the context of the state of the art.

Chapter 3 describes the preliminary definitions and background required for the reading of the subsequent chapters. It introduces terminology related to Web of Things and formal verification. Specifically, LNT and MCL languages are introduced for formal verification. A brief introduction to rewriting logic in Maude is also provided.

Chapter 4 defines the formal models proposed for objects and their composition. It also defines a language for composition which helps design more expressive scenarios. Further, we define the formal semantics of the rule-based composition. Finally, the encodings of the composition in LNT and Maude are presented.

Chapter 5 focuses on verification. It defines a set of generic and application-specific properties and their formal specification in MCL. These properties are used to check the correctness of the designed IoT applications. Further, it also defines the set of properties specified in Maude for analysing the reconfiguration of an application.

Chapter 6 covers the implementation of the proposals as an end-user tool. It describes the various extensions made to the Mozilla WebThings project, as well as experiments related to the evaluation of the proposals.

Chapter 7 concludes the manuscript, summarizing the contributions and suggesting possible directions for future work.

Related Work

” *Study the past if you would define the future.*

— **Confucius**

Chinese philosopher and politician

This chapter considers the related work to place our contribution in the context of the state of the art. The works are categorized under five sections. First section details the research focusing on models for IoT applications. Then we briefly touch upon the expressiveness of the ECA languages. In the subsequent section a survey on the verification techniques specific to IoT is presented. The penultimate section deals with works pertaining to reconfiguration of applications. Finally, existing tools for designing and deploying IoT applications are surveyed.

2.1 IoT Application Models

This section takes a look at the works in the literature that deal with modelling of IoT applications. These models may not necessarily be used for verification, but they provide a context to compare the proposed model in this work. Note that in the literature we often see IoT application models are also referred to as IoT composition models. Although readers may find that we use the term interchangeably, in our work a composition refers to the design comprising objects and rules and when it is deployed, we refer it to as an application.

In [Fel+17] 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. For modelling smart homes, they identify different types of components: Smart home, smart room and smart appliances. Then concrete instances of these components are specified (e.g., living room, smart thermometer etc.) and a hierarchical relationship is established between the components (e.g., appliances inside a room). 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). This approach is scalable, supports runtime configuration and reconfiguration of applications. However, it is better suited for the initial setup of smart homes rather than designing specific applications at end-user level.

Automated service composition is a technique where objects are modelled with an aim to automatically build a composition satisfying the application requirements. This approach keeps the user intervention to minimum. In [Buc+17], the authors propose an AI planning based approach to solve the composition requirements and guide automated deployment of the composition. The composition is modelled as processes using Adaptable Pervasive Flows Language (APFL), a workflow language. The behaviour of the activities in the process (called *fragments*) are modelled using a Labelled Transition System (LTS). Similarly, the context of execution is specified using the state-transition system. Fragments are made context aware using context annotations. Finally, requirements are modelled as goals. Fragments, context and requirements are fed to an execution planner to derive consistent solution. This work offers high degree of automation and it can be applied to various domains. It also captures the behavioural information (state information) in the LTS model. A limitation of this approach is that it does not use a rule-based language, which is intuitive and prevalent in the design of IoT applications and it does not follow user-driven design of the applications. Moreover, the model will have to be adapted to the IoT domain as it does not specifically take into account the IoT system architecture.

There are a couple of works that model IoT composition through Business Process Model Notion (BPMN), which is an ISO standard [Sta+13] workflow notation maintained by the Object Management Group. It is quite natural to view IoT applications as workflows as they exhibit a mix of event-driven and flow-based architectures. In [DM17; MD17], the authors use a subset of BPMN elements to model IoT application behaviour – events (timer, start, end), activities (tasks, send, receive), gateways (exclusive split and merge), flows, and data objects. Objects are modelled in separated pools in the process and the interaction is specified through collaboration diagrams. In [MRM13], the authors integrate IoT resources with business process. They identify four components related to IoT resources viz., Physical Entity, IoT Service, IoT Device, and Native Service. Similar to [DM17; MD17], they integrate the IoT resources with traditional process by adding an extra lane for IoT devices, where a device acts like a process participant. Lane is sub-partition in BPMN, which classifies different activities that can interact using sequence flows, whereas in pools only message flows can be used for interaction between the activities.

Overall, the idea of integrating business processes and IoT works because BPMN is expressive and usually only a subset of BPMN elements is required to model applications. However, BPMN semantics is loosely defined and it requires domain knowledge to understand the process execution. Moreover, scalability could be an issue especially when someone has to manage a large number of devices and application, which translates to even bigger number of activities and processes. The applications defined using our composition language can be translated to BPMN processes. We provide this option for users to view the application as a process instead of using BPMN itself as a modelling language.

In [Bro+18], the authors detail IoT-aware business processes. The general idea is similar to [MRM13], introducing the ability to connect the physical world with the digital processes. There are two specialized activities (tasks), sensing and actuating. The sensing task extracts data from the physical device and has a direction connection to the device. The actuating task changes the state of the physical entity. These tasks interact with the entity Thing, which is the representation of the physical device. [CW15] extend this notion further by adding specialized events such as location-based events, error events and intermediate events.

There are other works in the literature which build Petri-Net models from IoT-aware processes. Petri-Nets which are commonly used to model distributed and parallel systems, is a discrete event dynamic model. [Che+18] extend Petri-Nets with sensing event factor to model IoT specific events. It differs from traditional Petri-Net as sensing events have separate places and they define an additional timer variable to model the event waiting time for resources (actions). In [TKJ17], Petri-Nets with priority which is useful in resolving conflicts between actions are used to transform process-aware applications.

In the works related to IoT-aware process models, the focus is on integrating business processes with IoT devices. Interactions between business process elements and IoT devices are modelled here whereas our work is a greenfield model¹ where the IoT devices are used from the ground-up design.

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 [Har+16] is a modelling language that provides IoT specific constructs for the design and implementation of reactive applications. It relies on two structures – 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

¹https://en.wikipedia.org/wiki/Greenfield_project

behaviour is specified using state machine or ECA rules to express reaction to events in a stateless fashion. The Configuration or the application specifies the coupling of Things. In [Ber+19], 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 [Gom+17], the authors use Elaboration Language, a DSL to describe the architectural model of IoT network. The language elements include an Interface, a Component which consists of properties and services (instances of Interfaces). Interface exposes the services a component can provide, and it enables to bind compatible components. Properties are the state variables. This work targets the networking aspect of IoT systems rather than at the end-user application level. However, these concepts can be translated to application layer as well. MDE techniques are useful as they abstract the heterogeneity of devices and platforms and can generate deployable code. As with any domain specific solution, it requires one to understand the elements of the language and the expressiveness of the generated solutions is limited to the available constructs in the language. 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.

[Gan+15] is a work in the area of Ambient Assisted Living (AAL) that uses timed automata to model home care plans. The care plans describe the medical activities planned in the home of a patient. Each activity is specified as a temporal specification, which indicates when an activity needs to be performed in a day. This specification is transformed to a pattern automaton and these patterns are combined together to build an activity automaton which represents the schedule. Finally, activities are composed to build the overall care plan. The composition is done using a specific composition operator which ensures at any instance of time, only a single activity is being executed. Compared to our work, this modelling deals specifically with routine automation which is expressed in terms of temporal expressions. Moreover, the specification is targeted at expert users as there is no high-level specification language.

Finally, in [CBP12], the authors describe behaviour aware composition for web of things. They model the behaviour of things by extending the OASIS standard Devices Profile for Web Services (DPWS). Using the DPWS profile, they specify the behaviour in terms of sequence order (order in which they can send or receive messages). Specifically, at the interface level, they add partial order constraints that define the

device behaviour (afterAll, afterSome, onlyOneOf) and full sequences that need to be respected in terms of state changes. The overall 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.

2.2 Expressiveness of ECA Languages

There are several works on enriching the expressiveness of ECA rules. Since our work uses a composition language, it is worth looking at the works in this area to better understand our choice of the language.

Snoop [CM94] was one of the early works that proposed an ECA language. The authors proposed an event specification language which supported simple, composite, and temporal event expressions. Composite events are a combination of one or more events and temporal events which specified the timing of events. They provided operators such as or, any, sequence, aperiodic/periodic (repeat) to build composite events. The authors also proposed algorithms for detecting these composite events. This work is aimed at active databases [PD99] which support event-driven architecture. Our work uses a subset of the operators proposed in this work for the design of IoT applications. These operators are sufficiently expressive to build advanced automation scenarios.

The expressiveness and limitations of ECA programming style adopted by IFTTT is discussed in [Ur+16]. They describe their findings by analysing 200,000 IFTTT recipes. Further, in [HC15], 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 [Ghi+17], 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 create and reuse these rules in different applications. The rules follow the structure IF/WHEN *trigger - expression* DO *action - expression*. In the trigger-expression, one can combine events using AND, OR operators and also use the NOT operator for negation. The actions can be specified to be executed sequentially. 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 [MSC+19] the authors present a user-interface to debug ECA rules. It partly implements the constructs presented in [HC15]. The authors describe the UI prompts that can aid or prevent users from creating erroneous rules. Similarly, in [DC15], 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 6).

Our work uses a simple yet expressive composition language based on the IF *something-happens* THEN *do-something* style rules. We kept the constructs of the language simple to avoid introducing additional learning for end-users. Moreover, studies have shown that IFTTT style programming covers nearly 78% of the typical user scenarios [Ur+16].

2.3 Verification of IoT Applications

In this section, we review the works that focus on verifying IoT applications, with special emphasis on verification involving ECA rules.

[Lia+15] 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 – conflicts among the rules and violations of policies. It uses a combination of SMT solving and runtime code exploration. Further in [Lia+16], 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. Although, the authors propose to check these safety violations at design time, checking them at runtime as a monitoring feature would be much more useful.

In [Nac+18], 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 [DB08] 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 [Lia+15]. Compared to the works [Nac+18; Lia+15], our verification is not specific to rules and it considers the entire application behaviour for checking correctness.

In [JLC13], the authors translate the ECA rules into Petri nets (PN). They use this PN model to analyse two correctness properties: termination and confluence. Termination indicates that the set of rules in the system always yields a result and confluence ensures that possible interleaving in the rules will always lead to the same result. Rules with priorities are taken into account by extending PNs with priorities. 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 [Van+17], 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. The tool checks properties such as termination of rules, non-determinism, unused, incorrect and redundant rules. The tool also features a semantic analyser to perform quantitative and qualitative analysis. This wide range of support for verification is made possible by specifying the ECA rules into IRON programs, which are subsequently transformed by vIRONy to Alloy specifications to perform verification. Compared to our work, the authors

do not consider the internal behaviour of the objects. Objects are modelled as simple sensors and actuators with properties that can be set to a specific value. We model the internal behaviour respecting the order of execution, which is useful in modelling complex objects containing embedded software. Also from a usability point of view, usage of IRON as a composition language makes the approach less relevant to end-users.

[Ouc18] uses formal verification for checking functional correctness in IoT systems. The authors define an IoT architecture with actors, sensors, web services, and physical infrastructure. The architecture is formalized using a process algebra encoding which specifies atomic actions in the components and the interactions between components using a composition operator. The encoding is specified in PRISM language and the functional requirements are expressed as probabilistic computation tree logic (PCTL) formulas. In this model, execution of an action has a cost and probability associated to it. Compared to other works, here the focus is on quantitative requirements rather than general correctness of the system. Thus, the cost/probability model allows one to capture requirements in terms of likelihood or possibility than as discrete actions. It is to be noted that the architecture is not derived from any existing IoT standards, and therefore the semantics of IoT applications captured in this work may not be fully relevant to real-world IoT systems.

In [HCH19] the verification deals with identifying attack chains in trigger-action programming. As the number of rules and devices increase, users may inadvertently provide access to unauthorized devices. This happens when action of an ECA rule is chained as an event in another rules. The authors identify privacy leakage and privilege escalation as the two attack categories. The interaction between devices is modelled as a finite state machine. Devices contain attributes that represent the state and the devices are classified into read only (sensor) and read-write (actuator). Rules are modelled to run concurrently, each rule encoded as a Boolean function, and being triggered when the function returns true. The authors use NuSMV model checker to identify property violations in the automata. This work deals with a specific aspect of chaining of events in rules. Our work does not focus on this aspect, but since our model compiles to an automata, we could verify these chaining of events as safety properties using model checking.

In [Eri09], the authors propose a UI-based tool REX (Rule and Event eXplorer) which allows one to specify event, rules, and application requirements. The event language proposed allows to specify complex events by composing basic ones. The rules are transformed into a timed automata specification in UPPAAL and the requirements

are expressed in CTL. 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 [DAC98]. The tool also supports termination analysis for a given set of rules. This work does not specifically target IoT applications, but since it uses a rule-based language, it can be applied to IoT applications too. Compared to our work, the authors model time in their specification. Events are associated with an expiration time. We found this useful under specific contexts (e.g., manufacturing automation) but for smart home scenarios, it would be simpler to abstract the expiration time to ∞ (endless wait). For instance, if more than one event needs to be triggered, the system waits until all the required events occur as events or actions in home devices like lights, alarms, weather monitors etc., are not time-bound. Regarding the verification, the work focuses on the behaviour of the rules rather than the complete behaviour of the system. This prevents discovering the instances where the rules with different objects are perfectly valid, but they could not be executed as one or more objects may have dependencies that need to be executed before the execution of the rules.

A common theme in these IoT verification works is that correctness is checked at rule level. The composition of objects is validated by verifying the interaction between the objects for correctness. Our work is different in the context of ECA rules as our verification accounts for both interaction between the objects (rules) and also the behaviour of the individual objects involved in these rules.

2.4 Reconfiguration of IoT Applications

As our work proposes reconfiguration properties that allows users to qualitatively compare a redesigned application with the original application, we review some of the works pertaining to reconfiguration of IoT applications.

The authors in [See+19] present an architecture to support automated dynamic reconfiguration of IoT applications in building automation systems. They model the application using a semantic description called Recipes. Recipes have the objects as ingredients along with interactions between ingredients. Recipes are executed as an autonomous choreography. Automated dynamic reconfiguration is enabled by adding constraints to the Recipes. When failures are detected in the distributed environment, the system automatically recovers the configuration respecting the constraints to maintain optimal service. Contrary to this work, our proposals focus on user-driven reconfiguration rather than environment or system driven reconfiguration. It allows

users to make a choice decide whether to proceed with the changes or preserve the existing configuration after performing a comparative analysis. A reconfiguration can never be fully automated as not all reconfiguration scenarios have feasible solutions and in such cases user intervention will be required. Moreover, since our target is personal spaces like homes, it is wise to take user choice into account.

The work by Felfernig et. al [Fel+17] described in Section 2.1, model smart home configuration using Answer Set Programming. They also propose to apply the configuration techniques for reconfiguration. The hierarchical relationship established between the components can be used to identify compatible components during reconfiguration. Unlike our work, the impact of reconfiguration is not measured here. This work can be complementary to our checks. Initial reconfiguration setup can be checked for the component compatibility and then proceed with the behavioural checks.

In [LBP09], the authors propose recovery mechanisms when dealing with service failures in the context of an Ambient Assisted Living(AAL) system. They propose two kinds of reconfiguration strategies. First, a static reconfiguration is proposed, where at design time components are specified with alternative fallback options (e.g., if there are two lights in the living room, the second light is an alternative if the first light fails). The second option is more dynamic in nature, called effect-based reconfiguration. It compensates for the failed operation by identifying one or more operations from the available devices that produce an effect identical or close to the failed operation. Compared to our work, the focus here is on service failures, whereas we consider both replacements and upgrades to the system. Our approach allows users to compare designs and importantly, enables objects to be deployed seamlessly under certain conditions.

On modelling adaptive systems which enable dynamic reconfiguration, [HLR17] proposes a framework which models IoT object functionality using SysML4IoT [FMS14]. The interactions between objects are captured in a publish-subscribe system. The system configuration is adapted as a response to the change in environment. The adaptation is guided by a deployment plan, which is a runtime instance of the IoT system containing values of different attributes of the objects. These plans are modelled as a state-transition system where states refer to the system configurations and transitions are adaptation events. In comparison, we deal with design time reconfiguration of the application, whereas [HLR17] focusses on runtime adaptation and it does not take user feedback into account while performing the adaptation.

There are recent works that tackle dynamic reconfiguration using machine learning techniques. In [Roj+20], the authors use neural networks for automating the inter-

actions among IoT devices. Initially, the home is set up with a bunch of automation rules (configuration). As the time goes by, these rules in the automation may not be appropriate and thus, users may have to manually interact with the devices. The authors use the contextual information available in the objects (how they interacted, environment status, etc.) to predict the future actions (new configuration) as the context evolves. Similarly, in [PA19], the authors use the historical data related to user and rules to offer users a personalized automation rules for changing contexts. These works are complementary to our work. The interactions and the rules proposed through machine learning can be checked for reconfiguration properties before deployment.

2.5 End-User Tools in Smart Homes

There are several tools available for end-users to design and deploy IoT applications. During the course of this thesis, the tools have evolved in many different ways. Popular tools have added more features, and a few tools have been integrated into larger ecosystems (e.g., Workflow [Inc20a] is now integrated into Apple iOS). The list of tools in this section is not exhaustive, it covers only a few of the popular tools that are relevant to our work. Importantly, the work describes the available features of the tools at the time of writing.

IFTTT

IF-This-THEN-That (IFTTT) [IFT20] is one of the most popular automation platforms. It initially started as a web-based automation service, which has evolved to support physical IoT objects through the cloud-based web service. The app is available on the web and as a native app on phones.

As the name suggests, IFTTT uses trigger-action-programming as the basis for automation. There are four concepts in IFTTT that enable users to build automation. An *applet* or recipe is similar to an application, is made up of triggers and actions (events and actions) in the form IF-THIS-THEN-THAT. *Triggers* are the event conditions that are encoded in the “THIS” part of the applet. *Actions* are the resulting reaction or output encoded in the “THAT” part of the applet. *Services* or channels are similar to objects in an application, they provide data and interfaces that can be integrated in an applet. A service exposes a set of triggers and actions.

As an example, consider a popular applet named “Turn lights out for bed with Alexa and Hue”. The description of the app says “When you say "Alexa, trigger bedtime", this Applet will turn off all Philips Hue lights”. Here, the event trigger is someone saying “Alexa, trigger bedtime”, and reaction is to turn off all the Philips Hue lights. There are two services or channels in use, Amazon Alexa and Philips Hue. Amazon Alexa exposes the voice API and the Philips Hue API enables to control the state of the lighting.

IFTTT has more than 54 million applets and can connect to more than 550 services. These services include IoT devices that interact with the physical world and also web services such as Gmail, Instagram, OneDrive, etc. The application has around 11 million users. Users can simply activate the applets and connect to the available services to build the automation. They can also build custom applets using the triggers and actions available from the connected services. Further, users with programming skills can take advantage of the Maker platform to integrate new services and build new applets.

One of the reasons for popularity of IFTTT is its simplicity. Consumers without programming knowledge can configure and activate applets without any prior training. It is simple and intuitive to use for everyday automation. However, to build complex scenarios, it requires some programming knowledge, unless there is an applet already available.

Node-RED

Node-RED [Fou20] is a flow-based writing tool that enables users to visually connect physical devices, APIs, and online web services. The visual programming paradigm allows one to build automation by graphically manipulating the program components rather than writing the code. Node-RED is an open source tool supported by the OpenJS Foundation. Initially, it was developed at IBM and in 2016 it was made into a JS Foundation project.

Node-RED provides a web-based flow editor. The editor has a set of nodes available in the palette. Users can use these nodes and create connections between them to build applications. Each node serves a specific function. Typically, it takes some input data, processes it, and exposes the output data to be sent to the next node in the flow. Once the flows are defined, it can be deployed like an application on the runtime environment. Flows are stored as JSON files, which can be easily shared with the tool community. Users can monitor the devices using the Node-RED dashboard.

A node can have at most one input and may have several outputs. Nodes are activated upon receiving the input. Configuration node is a special node, which is used to share the configuration with regular nodes. Context is an information store that can be accessed by the nodes. It could be globally accessible or accessible to a subset of nodes. The information exchanged between the nodes are called messages and they are passed through wires.

Node-RED provides a library of nodes and flows, which can be added to the palette. The commonly used nodes can be found on the palette grouped as function nodes (trigger, execute, delay, etc.), network nodes (mqtt, http, websocket, etc.), parser nodes (csv, xml, html, json, etc.), sequence nodes (join, split, sort), etc.

Suppose a user needs to configure a Philips Hue light using Node-RED, they can install the Hue node module which supports controlling of Hue lights. Then the user can pick the Hue light from the palette, which takes a toggle on/off as an input and provide status of the light as an output. Users can connect this node with appropriate nodes as shown in Figure 2.1 and deploy the flow.

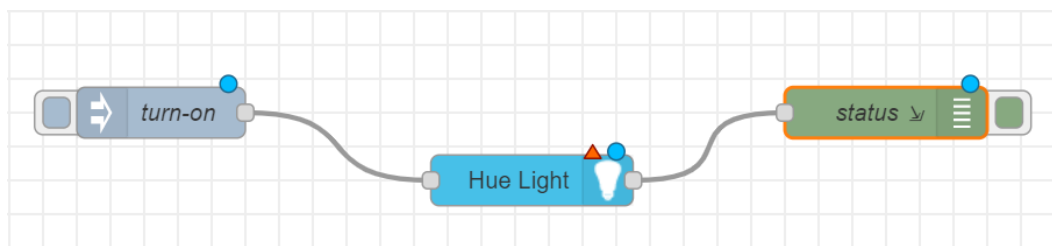


Fig. 2.1: Node-RED Philips Hue light flow

Node-RED is more expressive than IFTTT. It has support for programming via JavaScript functions. However, for a typical home user this may not be the right tool as it requires some training to understand the designing of flows and get the notion of nodes as services or processing functions. Just like applets or recipes in IFTTT, users can import the flows already created by other users in the community. But then the user is bound to the available flows in the library.

openHAB

The open Home Automation Bus (openHAB) [Com20] is an open source application layer framework that helps users in building home automation. It can be deployed locally on low-end devices like Raspberry Pi. The installation requires JVM and to make the installation hassle-free for end-users, it provides a pre-configured SD-card image (openHABian) for Raspberry Pi and Pine64 devices.

The main concepts of openHAB are the following – Things are a representation of physical devices and software elements that can be added to the system. Things may have one or more capabilities (e.g., motion sensor and temperature sensor in Philips Hue Motion device), and these capabilities are exposed through Channels. Items map to the Things’ capabilities, and these can be used for building automation. Items are associated with a state. There are 13 different Item types, such as Switch, Color, String, etc., available at the time of writing. A link is an association between a Channel and an Item. Bindings are similar to services in IFTTT, software components or add-ons that allow one to access physical device APIs through openHAB. A bridge is a special type of Thing, that acts like a gateway to access other Things.

openHAB provides different UIs for configuring and building home automation. The Paper UI is a web interface that helps to setup and configure the workspace. It allows one to manage add-ons, discover things, and link channels to items. Items can be further used in building automation. Simple monitoring of devices can be done at the configuration level, but for setting up advanced automation rules, users can take advantage of the scripting support provided by the tool or use the graphical rule editor provided by the HABMin UI. Rules are in the form of WHEN-trigger-condition-THEN-action as shown in Listing 2.1. Trigger conditions can be time, event, thing or system-based triggers. The rule scripts are based on Xtend, which is built on top of Xbase. This allows programmers to script complex automation logic.

```
rule "<RULE_NAME>"
when
  <TRIGGER_CONDITION> [or <TRIGGER_CONDITION2> [or ...]]
then
  <SCRIPT_BLOCK>
end
```

Listing 2.1: openHAB Rule Format

openHAB has support for more than 300 bindings, available as OSGi modules, with an active user community. It is quite user-friendly for setting-up and monitoring objects. Simple automation logic can be developed quickly without much training. However, adding complex rules requires programming skills which could be difficult for non-technical users.

Mozilla WebThings

Mozilla WebThings [Moz20b] is an open-source platform by Mozilla for monitoring and controlling devices over the web. It implements the Web of Things architecture,

in which the principles of web are integrated to IoT. The web interface provided by the project can be used to configure IoT devices and build automation scenarios.

The project has two components, the WebThings Gateway and the WebThings framework. The Gateway can be installed locally on a low-end device like Raspberry Pi. It provides interfaces to monitor and control devices. It has Things UI, a web interface backed by a rules engine, where users can create rules using a drag and drop UI. Similar to openHAB binding and IFTTT services, Gateway has an add-on management system that supports a large number of existing IoT devices.

The objects in WebThings are specified using the Mozilla Thing Description specification. A specification complementary (concrete implementation) to the abstract data model provided by the W3C's Thing Description specification. The description is exposed as a JSON object, through which a device's capabilities can be understood. The rules engine allows users to create rules in the form of IF-event-THEN-action. The event clause supports conjunction and disjunction of multiple events and action clause support conjunction of multiple actions. Rules are executed independently, i.e., as and when the event is triggered.

Our work extends Mozilla WebThings, not only because it is open source and backed by Web of Things specification, but also it is very intuitive for end-users to build automation scenarios. In our experiments, we noticed that even non-programmers could start using the tool just after around 10 minutes of training. More details regarding the extensions to Mozilla WebThings are available in Chapter 7 describing the tool support.

Other Automation Tools

There are a few more tools worth mentioning in the section. Here we cover them briefly, as these tools work on the principles similar to the tools discussed in the earlier section.

Home Assistant [Hom20] is another open source home automation tool that can be deployed on Raspberry Pi. It provides integrations with more than 1600 services. It has a web UI with the Automation Editor. The Editor allows users to set rules in the form of Trigger-Condition-Action. There are different types of triggers like time, sunrise/sunset, numeric, state, mqtt etc. Templates can be used to specify more advanced triggers. Conditions supported are logical AND, OR, NOT conditions and also numeric conditions such as time, state etc. The configurations and automations are saved in YAML format.

Samsung SmartThings [Inc20b] is another tool which supports designing of complex rules through webCoRE and SmartRules apps. webCoRE has rules encoded as Pistons. Pistons can be created using the Piston Editor UI. SmartRules is a more graphical option, but it is not as expressive as webCoRE. Apple Workflow [Inc20a] and Microsoft Power Automate (formerly Flow) [Mic20] are automation tools, but their focus is mostly on software service level automation.

The two broad reasons for the popularity of the tools mentioned in Section 2.5 are – (i) ease of access (ii) support for a wide range of devices. So, any new tool must compete on these metrics to gain traction in the market. Considering these factors, we chose to build a solution on top of an existing platform covering some of its limitations.

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. Finally, most of these tools treat reconfiguration as another deployment of a configuration. There are neither mechanisms to compare an existing and a new configuration nor options to deploy the new configuration as seamlessly as possible.

2.6 Concluding Remarks

In this chapter, we have covered different areas of work in the literature related to our contributions. Our work is distinct from the presented related works in more than one ways. First, we model the objects and their composition based on the WoT standard and importantly, our model takes into account the behaviour of the individual objects. By considering the behaviour, the verification is more comprehensive when compared to the verification at syntactic and rule levels. With respect to reconfiguration, we not only propose properties to qualitatively compare configurations but also propose mechanisms to deploy new configurations seamlessly. Finally, unlike the tools in the market, our tool integrates formal methods into application design and deployment to provide correctness guarantees at design-time.

Background

” *If you do not rest on the good foundation of nature, you will labour with little honour and less profit.*

— **Leonardo da Vinci**
Italian polymath

This chapter presents a quick overview of the formal concepts and techniques used in the subsequent chapters. First, concepts related to formal verification are presented. A short introduction to Labelled Transition Systems and operations on them, followed by an overview of the verification techniques are presented. Next section covers the LNT language. Model Checking Language (MCL) is described in the subsequent section. The final section introduces rewriting logic and the Maude system. Readers familiar with these tools and techniques, especially from the formal methods background may safely skip these introductions.

3.1 Behavioural Models

One of the techniques to study the behaviour of a discrete system is by specifying it in terms of a state transition system. The behaviour describes how and when the system can interact with its environment [San13]. These systems can be labelled or unlabelled (or label set is a singleton) and in our work we specifically focus on a Labelled Transition System (LTS) as it allows one to distinguish the reasons to trigger the transitions.

Labelled Transition System

An LTS is a 4-tuple consisting of a set of states (S) and a set of transitions (T) between the states. Among the states, one state is identified as an initial state (s_0). The transitions are labelled by actions (A). The initial state denotes the state in which the system is in before any action is performed on it.

Definition 1. Formally, $LTS = \langle S, A, T, s_0 \rangle$, where

- S is the set of states
- A is the set of actions
- $T \subseteq S \times A \times S$ is the transition relation
- $s_0 \in S$ is the initial state

A transition (s_1, a_0, s_2) (also denoted as $s_1 \xrightarrow{a_0} s_2$), represents a transition representing that the system can move from a state s_1 to another state s_2 by performing an action a_0 .

An LTS contains all possible execution sequences of a system. An element of this set of all possible finite sequences is called a *trace*. Given an LTS, a trace is a sequence of actions $a_0, a_1, \dots, a_n \in A$, such that $s_0 \xrightarrow{a_0} s_1, s_1 \xrightarrow{a_1} s_2, \dots, s_n \xrightarrow{a_n} s_{n+1} \in T$, where $n \in \mathbb{N}$. The execution sequence in a trace starts from the initial state s_0 and the trace could possibly have infinite sequences due to the presence of loops in the system behaviour.

Sometimes, it is not possible to know the complete behaviour of a system or more often only a subset of system behaviour is of interest to the task. In such instances, a special action called internal or hidden action denoted by τ is modelled. These are unobservable actions in the system.

Parallel composition of two LTSs results in an LTS. The composition may involve synchronization on a subset $L \subseteq A_1 \cup A_2$ of the common actions, with the particular cases of interleaving ($L = \emptyset$) and full synchronization ($L = A_1 \cup A_2$).

Definition 2. Given two LTSs, $LTS_1 = \langle S_1, A_1, T_1, s_{01} \rangle$ and $LTS_2 = \langle S_2, A_2, T_2, s_{02} \rangle$ where the actions to synchronize are specified by the set $L \subseteq A_1 \cup A_2$. The parallel composition (synchronous product) of LTS_1 and LTS_2 , denoted by $LTS_1 \parallel_L LTS_2$ is an LTS $= \langle S, A, T, s_0 \rangle$ where:

- $S = S_1 \times S_2$
- $A = A_1 \cup A_2$
- $s_0 = \langle s_{01}, s_{02} \rangle$

– $T \subseteq S \times A \times S$ such that

$$\langle s_1, s_2 \rangle \xrightarrow{a} \langle s'_1, s_2 \rangle \text{ if } a \in A_1 \setminus L \wedge s_1 \xrightarrow{a} s'_1$$

$$\langle s_1, s_2 \rangle \xrightarrow{a} \langle s_1, s'_2 \rangle \text{ if } a \in A_2 \setminus L \wedge s_2 \xrightarrow{a} s'_2$$

$$\langle s_1, s_2 \rangle \xrightarrow{a} \langle s'_1, s'_2 \rangle \text{ if } a \in L \wedge s_1 \xrightarrow{a} s'_1 \wedge s_2 \xrightarrow{a} s'_2$$

In a composition, the notion of hiding allows one to model internal communications of the system. The process of hiding is implemented by renaming visible actions into the internal action τ .

3.2 LNT Language

LNT [Cha+18; GLS17] is a formal specification language based on the E-LOTOS ISO standard [ISO01]. It provides constructs derived from imperative and functional programming styles to specify concurrent value-passing systems in a simple manner whilst keeping the semantic foundations of process calculi. It is one of the languages supported by CADP verification toolbox [Gar+13]. The operational semantics of the behaviours in LNT are defined in terms of LTSs, i.e., LNT specifications can be compiled to obtain LTSs using CADP compilers. Here we take a look at some of the features of LNT.

Data Types

LNT supports basic data types and constructed data types. The predefined basic types are Boolean (**bool**), integer (**int**), real (**real**), natural number (**nat**), character (**char**), and string (**string**).

Constructed data types (records, lists, sets, etc.) can be defined using the **type** keyword as illustrated in the listing below. It specifies an enumerated type denoting three kinds of person that could be present in a home. The **with** clause is used to define functions that are automatically declared for the type.

```
1 type person is  
2   resident , guest , intruder  
3   with "!=" , "=="  
4 end type
```

Variables

LNT supports data variables on which read and write operations can be performed. Variables are declared using the **var** keyword and their scope is defined using the "**var ... in ... end var**" block.

Variables can be assigned values using the assignment operator **:=** in statements of the form **X := V**, where **V** is the value to be assigned to the variable **X**.

```
1  var presence : person in  
2    presence := resident  
3  end var
```

Statements

Standard control flow structures are supported by LNT. Conditional statements can be specified using **if**, and **case** constructs. Iteration statements can be defined using **loop** (general loop), **for** (stepwise loop), and **while** (initial condition loop) keywords. Statements can be sequentially composed using **;** and the empty statement is denoted by **null** keyword.

Functions

Functions are defined using the **function** keyword. Functions support both call-by-value and call-by-reference parameter passing, with call by value being the default option. Call by value and call by reference parameters are denoted by **in** and **out** keywords, respectively. A special keyword **inout** is used to denote a parameter that could be both read and updated within the function. In a function call, actual parameter values are prefixed by **!**, **?**, and **!?** for the **in**, **out**, and **inout** function parameters, respectively. A function can return a value using the **return** keyword.

```
1  function is_intrusion_detected (in presence : person , out  
    light : colour) : bool is  
2    var status : bool in  
3      status := false ;  
4      if (presence == resident) then  
5        light := green  
6      elseif (presence == guest) then  
7        light := blue
```

```

8     else
9         light := red;
10        status := true
11    end if
12    return status
13 end var
14 end function

```

The above listed function `is_intrusion_detected` accepts as **in** parameter the presence of a person and returns a boolean value indicating whether an intrusion is detected, while doing so it updates the **out** parameter light colour according to the type of the person detected.

This function can be invoked as follows:

```

1     .....
2     var presence:person, light:colour, status:bool in
3         presence := resident;
4         status := is_intrusion_detected (!presence, ?light)
5     end var
6     .....

```

Note that functions can be viewed as helper methods to define the system behaviour. They are defined outside of the behaviour definition.

Process

Process is the object in LNT that specifies (a part of) the system behaviour. A process is defined using **process** keyword and it can contain statements and control flow structures like a function. It can take two sets of parameters: (i) a set of gates and (ii) a set of variables.

Gates are the communication endpoints through which the input and output message passing and synchronization are carried out. Processes communicate by multi-way rendezvous on gates. A gate G , specified as $G[(O_1, O_2, \dots, O_n)]$, waits for a rendezvous on the identifier G and O_1, \dots, O_n describe the offers or the data to be exchanged during the rendezvous. The data is exchanged by emission and reception of values on gates. Emission of a value is denoted by $!V$ and reception of a value by a variable is denoted by $?P$, where V is the value or value of the expression emitted and the P is a declared variable which stores the received value. The communication

is blocking and the rendezvous is symmetrical, i.e., there is no distinction between the sender and the receiver. A gate can be typed, untyped (**any**), or without any type profile (**none**) which can be used for synchronization without exchange of data. Typing specifies the types of offers that can be accepted by the gate.

As you may recall that a process can be compiled to an LTS and in this LTS the synchronizations on gates appear as actions with the gate identifiers. These are the visible events from the execution of the process and actions that are internal to the behaviour are specified as *i*.

Sometimes, it can be useful to declare gates within the local scope of the process, i.e., hidden from the external environment, instead of declaring them as formal parameters. This can be done by using the **hide** keyword of LNT. Since the scope of these gates is local to process, actions on them appear as *i* action in the LTS.

Further, behaviour can be compositionally specified using the composition operators supported by LNT. Along with the **;** operator, which is used for sequential composition, **par** and **select** keywords can be used to specify parallel composition and non-deterministic choice, respectively.

Suppose, a thermostat needs to be modelled, where it provides interfaces to increase or decrease the temperature and also to turn off the device altogether. Its behaviour can be expressed using the **select** operator. As another example, the behaviour of an emergency beacon that flashes and rings simultaneously can be modelled using the **par** operator. In this example, the **par** operation is an interleaving ($L = \emptyset$ as per Definition 2 on page 32), and if one wants to synchronize on specific actions, they are specified as **par** followed by the actions to synchronize and the **in** keyword (e.g., **par flash in ... end par**).

```
1  select
2      increment
3  []
4      decrement
5  []
6      turnoff
7  end select
```

```
      par
          flash
      ||
          ring
      end par
```

The resulting LTSs corresponding to the **select** and **par** encoding obtained by compiling the specification are shown in Figure 3.1. Note that for better illustration, the nondeterministic choice is shown to lead to three different states, however in practice, they are reduced to a single state as they are equivalent [Mil89; BHR84].

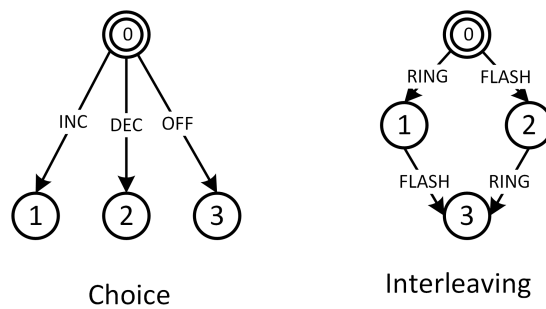


Fig. 3.1: Nondeterministic choice and interleaving LTS

Now let us illustrate the LNT process encoding of a physical alarm device interacting with an environment. An alarm can be set, silenced or turned off completely (e.g., using a physical button). Silencing of a ringing alarm can be done either using a snooze button, or using a reset button. For the sake of simplicity, let us model that both snoozing and resetting the alarm will result in the same state (i.e., hitting snooze resets the alarm). snooze and reset are specified as a `silenceaction` enumerated type.

```

1 type silenceaction is
2   snooze , reset
3   with "=", "!="
4 end type

```

The behaviour of the alarm is encoded in the `process` `alarm`. The possible interactions with the environment (communication endpoint) are specified in `setalarm`, `silence`, `turnoff`, and `ring`. The corresponding gates are untyped `any`. Further, `loop` is used to encode the repeatable sequence of actions possible in an alarm. Alarm can be either set or turned off as encoded using the `select` operator.

```

1 process alarm [setalarm: any, silence: any, turnoff: any,
   ring: any] is
2 loop
3   select
4     setalarm; ring;
5     silence (?any silenceaction)
6   []
7   turnoff
8 end select

```

```
9   end loop
10  end process
```

Once the alarm is set, it will ring (delay is not modelled here), followed by the option to silence it. Silence is parametrized to receive (?) **any** value of type `silenceaction` (either `snooze` or `reset`).

Then the environment (it could be a human interacting with the alarm) is modelled in another process. Environment replicates the behaviour of the alarm except, it sends (!) the actual `silenceaction` value.

```
process env [setalarm:any, silence:any, turnoff:any] is
loop
  select
    setalarm;
    select
      silence (!snooze)
    []
      silence (!reset)
    end select
  []
    turnoff
  end select
end loop
end process
```

Finally, the two processes are composed together in a parallel composition to get the overall system behaviour. The composition synchronizes on actions `setalarm`, `silence`, and `turnoff` as shown in line 3 of the **process** `MAIN`.

```
1  process MAIN [setalarm:any, silence:any, turnoff:any] is
2  hide ring:any in
3    par setalarm, silence, turnoff in
4      alarm [...]
5      ||
6      env [...]
7    end par
8  end hide
9  end process
```

It is to be noted that the ringing of alarm is modelled as an action internal to the system (i.e., composition of the alarm and the environment). Therefore, it is hidden using the **hide** keyword in the MAIN process.

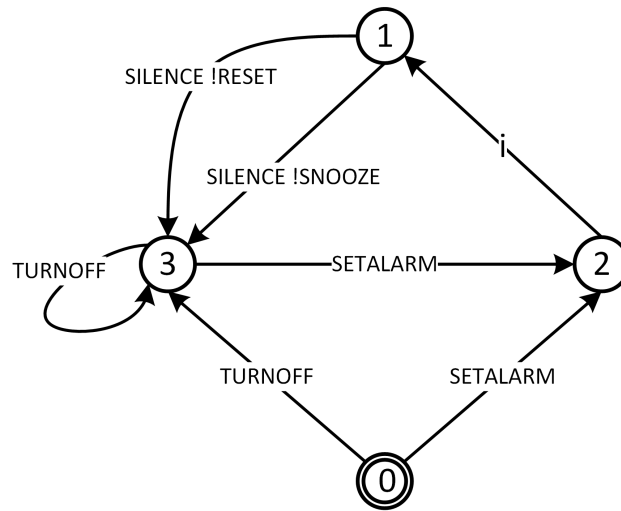


Fig. 3.2: Alarm device behaviour

The composition behaviour can be interpreted as an LTS as shown in Figure 3.2. The hidden action ring appears as an internal transition with label i in the LTS.

3.3 Model Checking

Model checking [BPS01] is an automated verification technique which involves a model specified as a state-transition system (e.g., LTS) and a formal property that needs to be interpreted on the model. The verification involves checking whether the property holds for that model.

As illustrated in Figure 3.3, a Model Checker tool takes as input the model and the property or the specification to be checked. If the property is not satisfied, a counterexample is produced by the Model Checker indicating the behaviour in the model violating the property. There are many model checking tools available in the community such as CADP, mCRL2, NuSMV, PRISM, SPIN, UPPAAL, etc.

Properties in model checking can intuitively be viewed as the expression of a specific behaviour in the system. Some examples of properties are: whether the system reaches deadlock, does the program terminate with correct output, etc. These properties are usually specified in Temporal Logic (TL). The logic specification describes the ordering of states or actions during the execution of the system. It

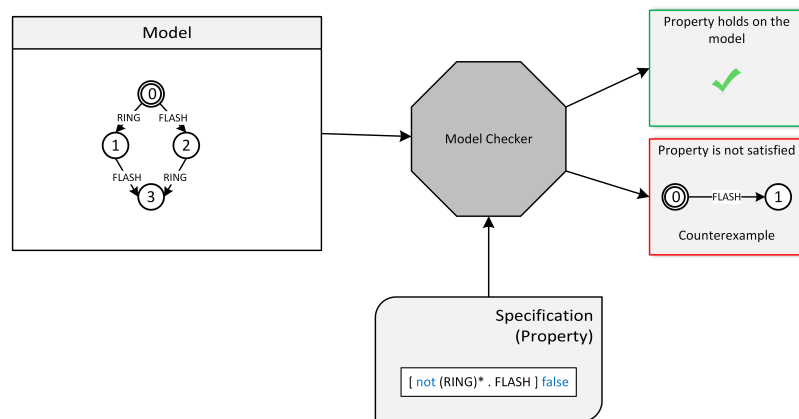


Fig. 3.3: Model checking using a model checker

is expressed as a set of temporal logic formulas. Properties written in TL are independent of the system implementation (providing a layer of abstraction) and they can be used in a modular way, i.e., one can modify, add, or remove properties without affecting the truth value of the other ones [MP90]. Linear-temporal logic (LTL) is interpreted over state or action sequences and Computation tree logic (CTL) is a branching time logic interpreted over state or action trees.

The TL properties can be classified as safety, liveness, and fairness properties. Safety properties indicate that "something bad never happens" during the execution. They are usually expressed as forbidden sequences of execution in a system. Liveness properties, on the other hand, indicate that "something good will eventually happen" during the execution. Liveness properties are expressed as required sequences of execution in a system. Fairness properties express constraints on the occurrence of states or actions in the infinite executions of the system (e.g., an action occurs infinitely often).

Model Checking Language

Model Checking Language (MCL) [MR18; MT08] is a branching-time TL that can be used for expressing properties that are interpreted on LTSs. It supports data variables and expressions, generalised regular formula on transition sequences, and fairness operators similar to generalized Büchi automata. Like a traditional programming language, it also supports constructs like loops, conditional flows, etc.

MCL formulas are of three different types: action formulas, regular formulas, and state formulas. Action formulas characterize the actions of the LTS (transition labels). An action pattern $\{G v\}$ can be used to specify or match all the actions on the LTS associated to gate G with value v of a particular type. For instance, on the LTS in Figure 3.2, the pattern $\{\text{SILENCE ?any}\}$ matches both SILENCE !RESET and SILENCE !SNOOZE . Action formulas support additional pattern matching through two keywords, **any** and **where**. **any** is a wildcard clause which matches a value of any type, and **where** is an option filter clause which is followed by a boolean expression e (*where* e), matches an action if and only if the expression e evaluates to true. For instance, the formula $\{\text{SILENCE ?act:String where act = "SNOOZE"}\}$ is the same as $\{\text{SILENCE !SNOOZE}\}$. Action formulas can combine action patterns using the standard Boolean connectives (or, and, implies, not, etc.).

Regular formulas describe transition sequences in the LTS. They are built from action patterns and regular expression operators such as or ($|$), match zero or more ($*$), match one or more ($+$), bounded match ($\{n\}$), and concatenate ($.$). For example, either of the Silence actions in Figure 3.2 can be matched using the $|$ operator as follows: $\text{SETALARM}*.(\{\text{SILENCE !SNOOZE}\} | \{\text{SILENCE !RESET}\})$. Regular formula also supports data handling constructs with variables, conditional flows, and loops.

State formulas, as the name indicate define the predicates over the states of the LTS. They use boolean operators, modalities, fixed-point operators and data handling constructs. Modalities such as possibility $\langle E \rangle$, weak possibility $\langle\langle E \rangle\rangle$, necessity $[E]$, and weak necessity $[[E]]$ can be specified. The infinite looping formula $\langle R \rangle@$ specifies transition sequences with infinite concatenation of sub-sequences that satisfy the regular formula R . Safety properties can be expressed as $[R]\text{false}$ and liveness properties using the **inev** operator as $[R] \text{inev}(A)$, where A is the action that is inevitably reached. In Figure 3.2, a safety property would be to check that if the alarm is not set, then it cannot be silenced. This can be expressed as $[(\text{not } \{\text{SETALARM}\})*. \{\text{SILENCE ?any}\}] \text{false}$. The inevitability of occurrence of an action such as the alarm will always be silenced once it is set can be specified as $[\text{true}*. \{\text{SETALARM}\}] \text{inev}(\{\text{SILENCE ?any}\})$, where $\text{inev}(A) = \mu X. \langle \text{true} \rangle \text{true}$ and $[\text{not}(A)]X$. Properties such as deadlock freedom can be expressed as a state formula: $\langle \text{true} \rangle \text{true}$, which specifies every state as at least one successor (i.e., no sink state).

The latest version of MCL (V5) [MR18] provides a probabilistic operator to specify probability of transition sequences in a probabilistic transition system (PTS). The operator is a state formula defined using the **prob** keyword containing a regular

formula characterizing the required sequence. MCL is equipped with the Evaluator model checker [MR18; MT08], which evaluates MCL formulas on the fly on a LTS and generates diagnostics (counterexamples and witnesses). Evaluator also allows to define derived operators as macros and to group them into reusable libraries.

3.4 Rewriting Logic

Rewriting logic is a logic built to reason about change in a concurrent system [Mes92; Mes12]. It consists of rewrite rules, which are equations used to reason about the change. Rewriting logic can model the static and dynamic elements of a system. Static elements refer to the states of the system and the dynamic elements are the transitions in the concurrent system.

A rewriting system $\mathcal{R} = (O, E, R)$, where

- O is the set of typed operators
- E is the set of equations that specify the algebraic identities
- R is the set of rewrite rules

A state in rewriting logic is represented mathematically as an algebraic expression built using the operators specified in O . The states satisfy the algebraic identities specified in E . The set of rewrite rules R represent the evolution of the states. It specifies the transitions between the states. In this representation, two states s_1 and s_2 describe the same state iff $s_1 \equiv s_2$ can be proven using E .

The rewrite rule w is implemented through a substitution operation σ . Substitution maps variables x_i to terms t_i . The term substitution $t\sigma$ replaces all occurrences of x_i by terms t_i following the mapping $\{x_1 \rightarrow t_1, x_2 \rightarrow t_2, \dots, x_n \rightarrow t_n\}$. A rewrite operation is following the rule $w = l \rightarrow r$ is denoted by $s \rightarrow t$, where s is an instance of $l\sigma$ and t is $l\sigma$ being replaced by $r\sigma$. Rewriting can be done in one step or many but finite steps.

Maude System

Maude is a language and system based on rewriting logic [Cla+07]. It provides a formal toolbox in which Maude can be used as declarative language to specify an executable formal specification. It has a formal verification system to perform analysis on the specification. Maude integrates an equational style of functional

programming with rewriting logic computation. Maude implements Membership equational logic, which is a Horn logic [Hor51] whose atomic sentences are equalities $t = t'$ and membership assertions of the form $t : S$, stating that a term t has sort S . It extends order-sorted equational logic, and supports sorts, subsort relations, subsort polymorphic overloading of operators, and the definition of partial functions with equationally defined domains.

Maude uses modules as the basic unit of specification and programming. A module contains the syntax declarations describing the language of the system, statements that assert the properties of the system. The declarations are called signatures which define *sorts*, *subsorts*, *kinds*, and *operators*. *Sorts* identify the typed data and *Subsorts* organize the data types in a hierarchy. *Kinds* declare the error types. *Operators* identified by names operate on data to build expressions. The modules in Maude are categorized into functional and system modules. Functional modules contain equations, data identifiers, and memberships describing the typing information of a data. System modules contain the rules that describe the evolution of states (transitions), in addition to the elements of the functional module. Functional modules are declared using **fmod** keyword and system modules are declared with **mod** keyword.

```
fmod ID is
  sort Id .
endfm
```

In the above listing a functional module `ID` is declared which contains a *sort* `Id`. It is declared using the *sort* keyword followed by an identifier followed by space and a period.

Operators in Maude are declared using the **op** keyword. The keyword is followed by the operator name, then a colon (`:`) followed by its sorts as arguments. The resulting sort of the operation is indicated on the right side of arrow `->`, which is followed by space and a period, as with any declaration.

```
op _+_ : Nat Nat -> Nat .
```

Operator `+` defines an addition operator, which takes two natural numbers as arguments and returns a natural number as the result of the operation.

Variables in Maude are declared using the **var** keyword. Each variable is bound to the type that it has been declared with. Multiple variables of the same sort can be declared with the **vars** keyword.

```
var N : Nat .
vars M N : Nat .
```

Unconditional equations in Maude are declared using the **eq** keyword. The equations have the following syntax:

```
eq term1 = term2 [StatementAttributes]
```

The terms *term1* and *term2* must have the same kind.

The modules in Maude can be imported as submodules in another module in three different modes: protecting, extending, or including, denoted by keywords **pr**, **ex**, and **inc**, respectively. Protecting mode which is used in our work, indicates that when the module is imported, its ground terms are unaffected by the definitions in the importing module.

```
fmod FIBONACCI is
  pr NAT .
  op fibo : Nat -> Nat .

  var N : Nat .
  eq fibo(0) = 0 .
  eq fibo(1) = 1 .
  eq fibo(s s N) = fibo(N) + fibo(s N) .
endfm
```

Listing 3.1: Fibonacci sequence in Maude

Listing 3.1 shows the Maude specification of the Fibonacci series. It has the sort NAT in protecting mode. The operator fibo takes a natural number as argument and returns the next natural number in the series. The unconditional equations defined using the **eq** keyword specify the constraints of the Fibonacci sequence. It is to be noted that here we describe only the equational logic in Maude but Maude also supports rewrite rules and readers interested in further reading can refer to the Maude manual [Cla+07].

A Formal Model for the Web of Things

” *It’s on the strength of observation and reflection that one finds a way. So we must dig and delve unceasingly.*

— **Claude Monet**
French painter

The IoT ecosystem is built using a set of diverse standards and protocols. There are many works focusing on providing a common standard for interacting with objects and enabling objects with different underlying protocols to be interoperable. Standards such as Constrained Application Protocol (CoAP) by Constrained RESTful Environments group (CoRE), OpenWeave Weave Schema Description Language (WDL) by Google [Nes20], Web of Things (WoT) [W3C20c] by the World Wide Web Consortium (W3C), etc., describe objects in a machine-readable format like JSON. This description specifies the object’s capabilities, and also describes the ways to interact with it in a standardized manner. This information helps to know the object behaviour as it describes how and when the object can interact with the external world (environment).

This chapter 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 section, we present a behaviour model for objects which is based on the WoT TD specification. 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. Since Event-Condition-Action (ECA) [DGG95] programming is popular in smart homes, we extend the notion of rule-based ECA programming by defining a language for composition of ECA rules. This language enables users to build or program more expressive automation scenarios in their homes. Further, we describe the operational semantics specifying the execution of an IoT application which is composed of ECA rules. In the final two sections we encode the models of objects,

rules and the composition using LNT and Maude specification languages. These specifications allows us to perform formal analysis of the designed applications.

4.1 Web of Things Specification

The WoT specification [W3C20c] 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 [W3C20a] 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) [W3C20b], 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). The guidelines on how to define the network interfaces for different protocols in the form of protocol bindings are specified in the Binding Templates. Scripting API is an optional component that specifies the implementation using JavaScript APIs. Finally, the security and privacy guidelines describe the security and privacy issues to be considered by all systems implementing the WoT specification.

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 virtual entity. It could be an IoT device, 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, as shown in Figure 4.1. 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.

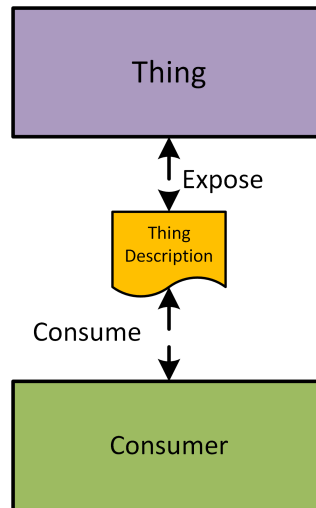


Fig. 4.1: Interaction between a thing and a consumer through Thing Description

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, as shown in Figure 4.2. First, a thing-to-thing interaction can be built where the thing plays the dual role of the thing and the consumer. Usually, consumer capabilities are integrated within the thing. This direct integration pattern can be implemented for devices that support HTTP, TCP/IP protocols. It is worth mentioning that directly exposing the thing to the internet will require one to consider the security issues related to it.

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 their interactions through the gateway. Here the WoT APIs are hosted at the gateway level instead of devices directly exposing the APIs.

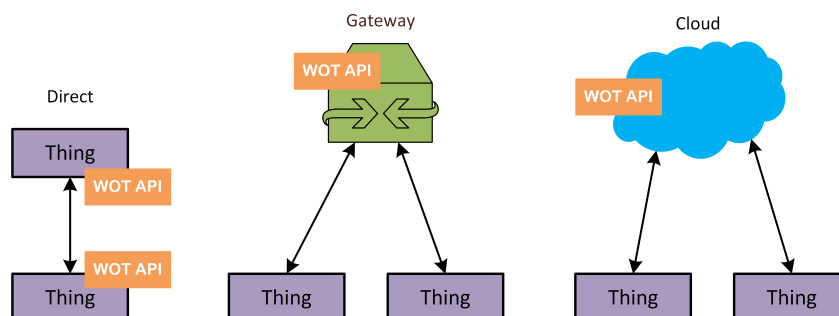


Fig. 4.2: Integration patterns [Fra20a]

Finally, the cloud integration pattern is a scalable architecture that connects things with the cloud. It is similar to a gateway wherein the cloud service acts as the intermediary. This pattern is useful for large scale deployments, especially across a wide geographical area and also when things need access to powerful computing capabilities (e.g., video recognition).

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 applicable to cloud as well.

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 [Moz20b] 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 [Fra20b] and Web Thing API [Fra20b] 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. Here we describe some of the key elements in a TD. Listing 4.1 shows the partial thing description of a connected light. 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 [Nor88]. The listing has two properties *on* and *brightness*, an action *fade*, and an event *overheated*.

@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 [Sch20] and Mozilla Schema [Moz20a].

```

{
  "@context": "https://iot.mozilla.org/schemas/",
  "@type": ["Light", "OnOffSwitch"],
  "id": "lamp",
  "title": "My Lamp",
  "description": "A web connected lamp",
  "properties": {
    "on": {
      "@type": "OnOffProperty",
      "type": "boolean",
      "title": "On/Off",
      "description": "Whether the lamp is turned on",
      "links": [{"href": "/things/lamp/properties/on"}]
    },
    "brightness": {
      "@type": "BrightnessProperty",
      "type": "integer",
      "title": "Brightness",
      "description": "The level of light from 0-100",
      "minimum": 0,
      "maximum": 100,
      "links": [{"href": "/things/lamp/properties/brightness"}]
    }
  },
  "actions": {
    "fade": {
      "@type": "FadeAction",
      "title": "Fade",
      "description": "Fade the lamp to a given level",
      "input": {
        "type": "object",
        "properties": {
          "level": {
            "type": "integer",
            "minimum": 0,
            "maximum": 100
          },
          "duration": {
            "type": "integer",
            "minimum": 0,
            "unit": "milliseconds"
          }
        }
      },
      "links": [{"href": "/things/lamp/actions/fade"}]
    }
  },
  "events": {
    "overheated": {
      "title": "Overheated",
      "@type": "OverheatedEvent",
      "type": "number",
      "unit": "degree celsius",
      "description": "The lamp has exceeded its safe operating temperature",
      "links": [{"href": "/things/lamp/events/overheated"}]
    }
  }
}

```

Listing 4.1: JSON Thing Description of a Connected Light

@type provides the string descriptions defined in the *@context* schema that identifies the device capabilities. The *@type* indicates the 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 that 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. In Listing 4.1, the event *overheated* is of semantic *@type OverheatedEvent*, which is different from the 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 light in Listing 4.1, 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.

4.2 Model for IoT Objects

The behavioural model of the objects is based on the TD of the object. Here we define the model and other components that form a composition of objects.

Definition 3 (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 from the interaction affordances properties, events, and actions to 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 4.2: JSON Thing Description of Hue motion sensor

Let us illustrate this using an example of a simple motion sensor device. It detects the presence of motion in the environment. Listing 4.2 shows the partial TD of a motion sensor. The semantic schema identifies it as a *MotionSensor* device and it has a single observable property *on*, which is read-only (indicating it can only be observed and cannot be set). Recall that a property is a state attribute of the object, a change in state of the object relates to events or actions. 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. In case of the motion sensor shown in Listing 4.2, when motion is detected by the sensor, an event $on(true)$ is emitted by the object.

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 physical and virtual objects (including software elements).

Definition 4 (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. $s(k) \in V_k$;
- $A = \{(d, k(v)) \mid d \in \{event, action\} \wedge k \in K \wedge v \in V_k\}$ 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 .
 $T = \{(s \xrightarrow{event, k(v)} s) \cup \{(s \xrightarrow{action, k(v)} s[v/k])\}$, 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 $k \in K$.

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 4 by considering all the (relevant) values for each property of the object. Figure 4.3 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. 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.

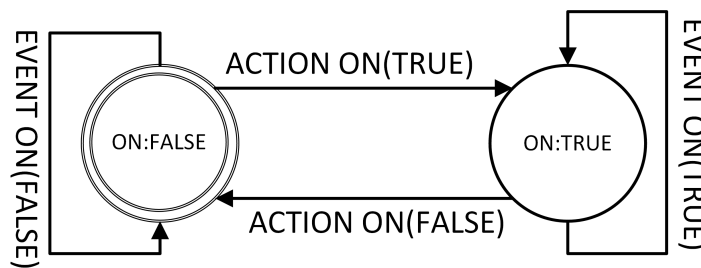


Fig. 4.3: Labelled transition system of a motion sensor

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 [IHS15].

4.3 ECA Rules and a Language for Composition

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 5 (Rule). Given a set of objects $\{O_1, \dots, O_n\}$, $O_i = (P^i, (S^i, A^i, T^i, s_0^i))$, a rule R is defined as “**IF EVT THEN ACT**” where,
 $EVT ::= \text{event} \mid EVT_1 \wedge EVT_2 \mid EVT_1 \vee EVT_2$,
 $ACT ::= \text{action} \mid ACT_1 \wedge ACT_2$,
 $\text{event} ::= O_i(k, v)$, where $(k, v) \in P^i$,
 $\text{action} ::= O_j(k, v)$, 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 will 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 environment setup can contain a number of rules ranging from a dozen to a few hundreds, whereas a specific application involves 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 [Ur+16; HC15; Bri+17]. To be able to specify more expressive automation scenarios, we use a composition language with simple operators based on regular expressions.

Definition 6 (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. It is useful in modelling scenarios where automation follows a sequence, e.g., turn on the light as one enters the room and then if it is cold (temperature < 20), turn on the heater whenever someone is inside the room. 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 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. Choice is particularly useful when one has to build an automation involving a set of events that result in different actions. If one just needs to capture a set of events that result in a certain action, then using a single rule with the events in disjunction would be a simpler option. 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 is run infinitely, which is useful in automating routine tasks.

Example: In a retirement home, once the old age pensioner unlocks the door of her/his apartment, the passage light is turned on. Once she/he sits on the sofa, window blinds are opened for sunlight. At the same time, if the outdoor air quality is measured to be good by an Air Quality Monitor (AQM), then a window is opened, otherwise the Heating, Ventilation, and Air Conditioning (HVAC) system is turned on to maintain optimum room condition. Finally, if she/he exits the house, the light and HVAC are turned off, and the window is closed. The ECA rules corresponding to the scenario are shown in Listing 4.3. Events and actions are described in the format *object(property, value)*. The IoT application can be expressed using the composition language as follows: $R1; (R2 || (R3 + R4)); R5$.

```

1  R1: IF door(open, true) THEN light(on, true)
2  R2: IF sofa(on, true) THEN window(blinds, true)
3  R3: IF AQM(quality, 0) THEN HVAC(on, true)
4  R4: IF AQM(quality, 1) THEN window(open, true)
5  R5: IF door(open, false) THEN light(on, false)  $\wedge$  HVAC(on, false)
6            $\wedge$  window(open, false)
  
```

Listing 4.3: Retirement home rules

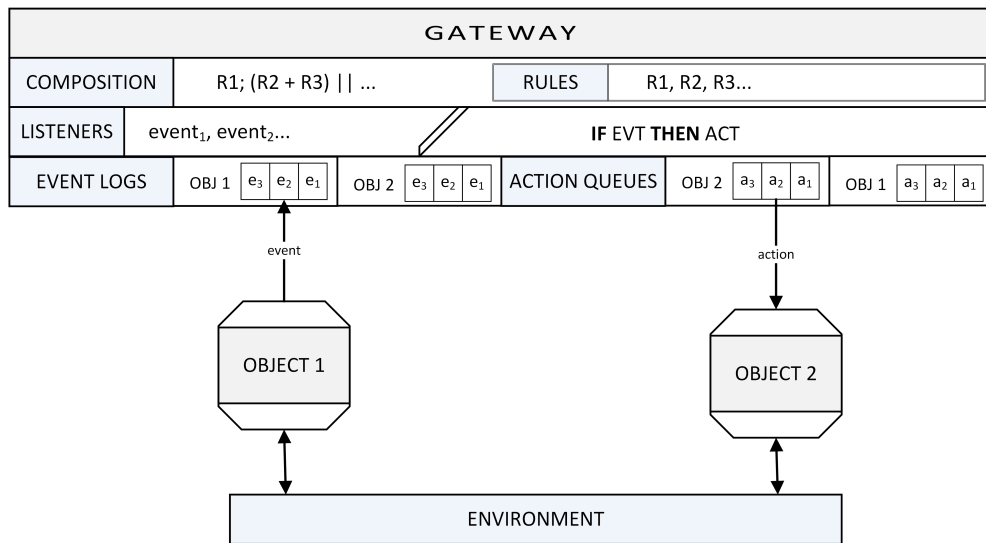


Fig. 4.4: WoT Gateway Communication

Let us now describe informally how the composition executes in practice. 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 (Figure 4.2 on page 47), we considered an indirect pattern, where WoT API is exposed via a gateway infrastructure [W3C20a]. Gateway 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. Along with the gateway, we need to account for the environment. Objects can react to the changes in the environment (e.g., change in temperature during the day updates the state of a thermometer).

Communication between objects occurs via a gateway as shown in Figure 4.4. Running of an application involves processing of the composition expression and the execution of individual rules. 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. The environment however, interacts directly with the objects, thereby bypassing the gateway. It is worth recalling that the composition always terminates and the composition as a whole can be executed infinitely. The bound k in C^k ensures that a part of the composition is not executed infinitely, thereby allowing the execution of the complete expression.

Semantics of Composition Expressions

Here we formally define the operational semantics of the composition language. The execution is defined with the help of two functions *active* and *exec*. These functions are later used in the inference rules that define the operational semantics.

During the execution of the composition expression, generally 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 4.1. 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 (\parallel), the parts of expression operands on either sides will be active.

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

Tab. 4.1: Rule activation patterns

Now let us look at how the active function works in the IoT application corresponding to the example automation scenario in Listing 4.3 on page 54. The scenario is shown as a workflow in Figure 4.5. The execution begins where initially, 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 triggered, along with the rule $R2$. Finally, $R5$ is activated with all the other rules being inactive.

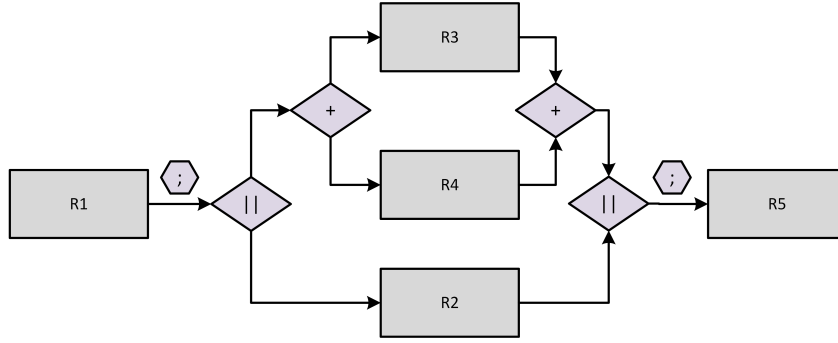


Fig. 4.5: Workflow representing retirement home rules

Further, the execution of the rules can be expressed in terms of a function $exec$, which generates the (set of) residual(s) of executing a rule R in a composition expression C . The rule execution patterns are defined in Table 4.2. 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., $\underbrace{C; \dots; C}_{k \text{ times}}$.

Rule Expression	Execution Residual
$exec(R, R)$	$\{\checkmark\}$
$exec(R, C_1; C_2)$	$\{C; C_2 C \in exec(R, C_1)\}$
$exec(R, C_1 + C_2)$	$exec(R, C_1) \cup exec(R, C_2)$
$exec(R, C_1 C_2)$	$\{C'_1 C_2 C'_1 \in exec(R, C_1)\} \cup \{C_1 C'_2 C'_2 \in exec(R, C_2)\}$

Tab. 4.2: Rule execution patterns

The function $exec$ operates on the rule R to be executed in the composition expression C and therefore $R \in active(C)$. The execution of the expression is the execution of the individual rules. \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:

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

Let us illustrate the execution of the composition expression of the retirement home scenario in Listing 4.3 on page 54. The execution of the expression $R1; (R2 || (R3 + R4)); R5$ is shown in Figure 4.6 on page 59. The execution of an active rule R produces a transition labelled by R from the current composition expression to one of its residuals. Using the two functions $active$ and $exec$, the execution of a composition expression is defined by the following rule:

$$\frac{R \in \text{active}(C) \wedge C' \in \text{exec}(R, C)}{C \xrightarrow{R} C'}$$

The figure starts with the application of the *exec* function on the sequence $(R1; (\dots))$, whose only active rule is $R1$. The steps of execution of the first part of the composition are as follows:

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

Subsequently, the remaining part of the composition expression is processed. In the Figure 4.6 on the facing page, the solid connectors with rule identifier denote the execution of the rule and dotted connectors represent the internal execution steps. As illustrated in the figure, after executing $R1$, three different three different rules $R2$, $R3$, and $R4$ become active. The expressions with a strike-through denote the empty residual (\emptyset) as a result of inactive rule execution. Here we make a slight abuse of notation by overloading the composition operators with sets of composition expression (produced by the *exec* function) as operands on either sides, but this is done to illustrate the execution in a simplified manner. The empty set \emptyset acts as a zero element for binary composition operators, i.e.,

$$\emptyset \oplus C = \emptyset, \quad \text{where } \oplus \in \{;, +, \parallel\}$$

The results of execution are highlighted in solid boxes and dotted boxes are the results of intermediate steps.

A simplified view of the execution represented as an LTS can be seen in Figure 4.7. Here we can notice the diamond pattern (interleaving) associated with the choice and the parallel execution.

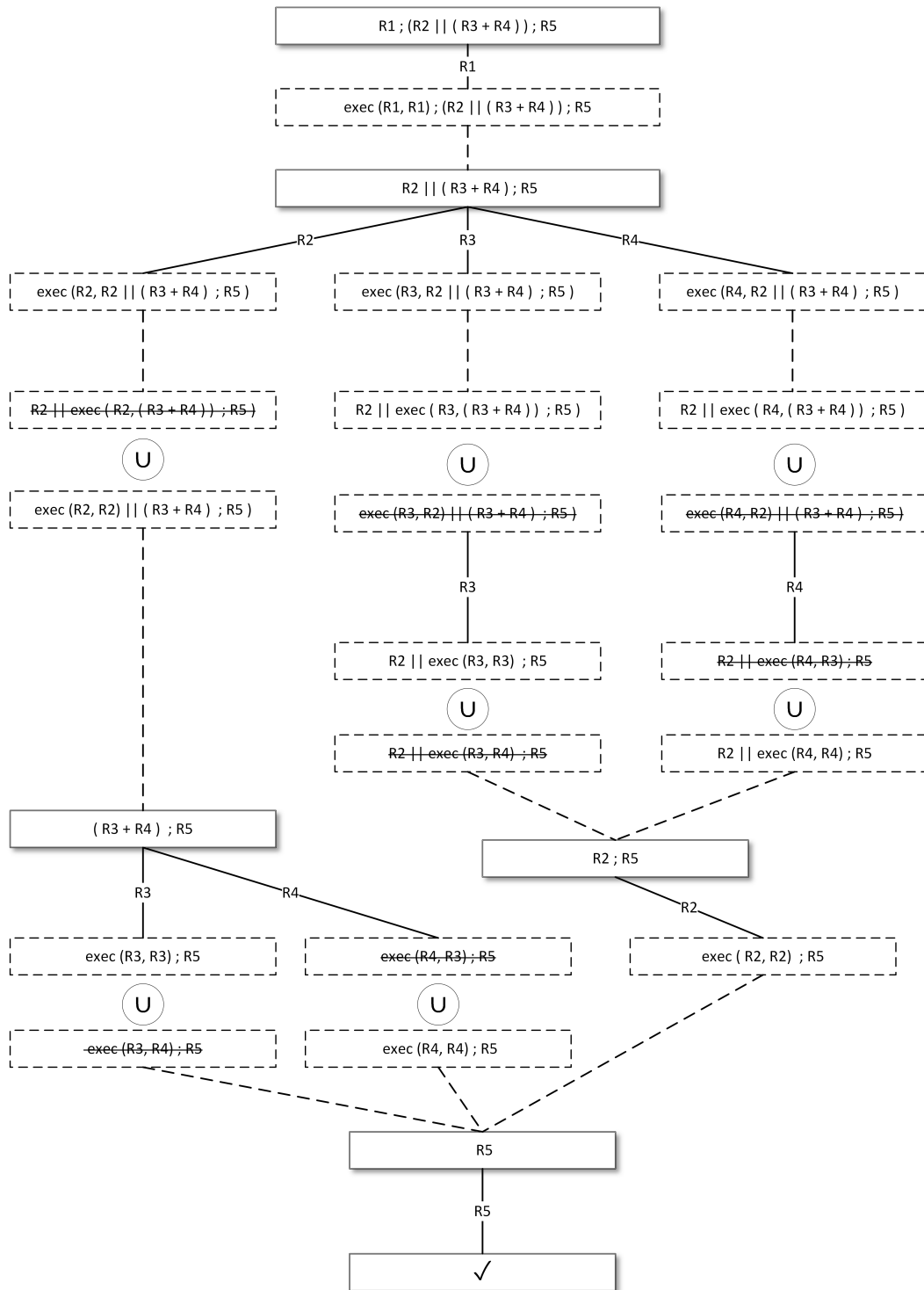


Fig. 4.6: *exec* Function on expression $R1; (R2 || (R3 + R4)); R5$

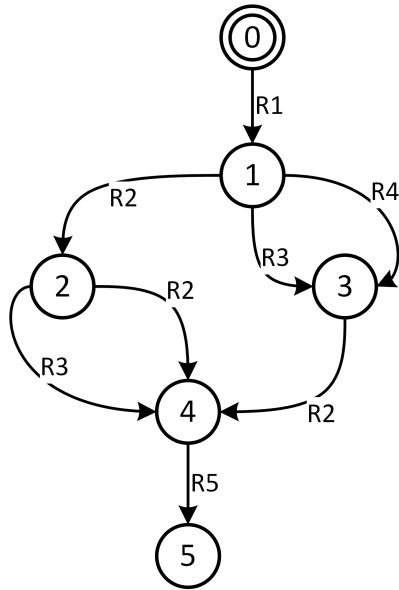


Fig. 4.7: LTS denoting the semantics of the composition expression $R1; (R2 || (R3 + R4)); R5$

Semantics of WoT Applications

Now let us define an application consisting of objects (O_1, O_2, \dots, O_m) . Each object O_i has a set of properties associated to it, denoted by $(P_1^i, P_2^i, \dots, P_{n_i}^i)$. The action queues corresponding to the objects (O_1, O_2, \dots, O_m) are denoted by (Q^1, Q^2, \dots, Q^m) respectively. The events that are part of the active rules are listened by the listeners. A schematic view of the notation is represented in Figure 4.8.

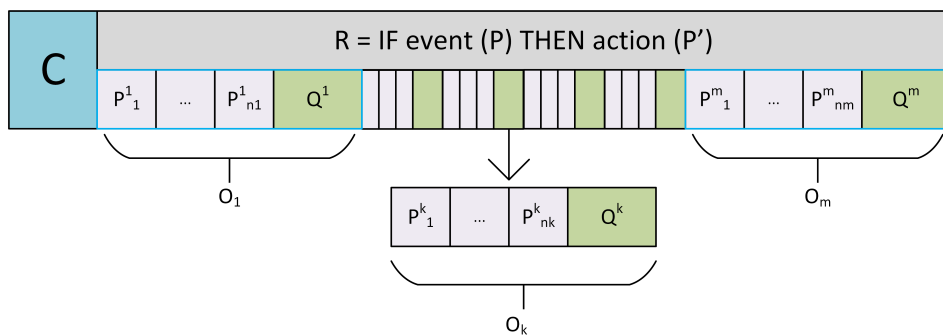


Fig. 4.8: Representation of a WoT application

The state of the object in the whole application is represented by its set of properties and the action queue $((P_1^i, P_2^i, \dots, P_{n_i}^i), Q^i)$. The properties can also be denoted in terms of key-value pairs, $((\kappa, v)_1^i, (\kappa, v)_2^i, \dots, (\kappa, v)_{n_i}^i)$. Recall that an object O_i

has an $LTS_i = (S^i, A^i, T^i, s_0^i)$, whose states are given by the properties of O_i , i.e., $(P_1^i, P_2^i, \dots, P_{n_i}^i) \in S^i$.

The execution of a ECA rule having an event and action can be expressed in terms of two inference rules, one inference rule for triggering the event and another for the 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:

$$\langle ((P_1^1, \dots, P_{n_1}^1), Q^1), \dots, ((P_1^k, \dots, P_i^k, \dots, P_{n_k}^k), Q^k), \dots, ((P_1^m, \dots, P_i^m, \dots, P_{n_m}^m), Q^m), \dots, C \rangle$$

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 $IF \text{ event}(P_i^k) \text{ THEN action}(P_j^l)$ is given by the following inference rule TRIGGER-EVENT:

$$\frac{R = IF \text{ event}(P_i^k) \text{ THEN action}(P_j^l) \in \text{active}(C) \quad C' \in \text{exec}(R, C) \quad (\kappa, v)_i^k = P_i^k \quad l \in [1, m] \quad j \in [1, n_l]}{\langle \dots, ((\dots), Q^l), \dots, C \rangle \xrightarrow{\text{event}(P_i^k)} \langle \dots, ((\dots), \text{append}(Q^l, P_j^l)), \dots, C' \rangle}$$

(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 .

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

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

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

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

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. It is specified through the following inference rule ENVACTION:

$$\begin{array}{c}
((\kappa, v)_1^l, \dots, (\kappa, v)_i^l, \dots, (\kappa, v)_{n_l}^l) \xrightarrow[]{action((\kappa, v')_i^l)} ((\kappa, v)_1^l, \dots, (\kappa, v')_i^l, \dots, (\kappa, v)_{n_l}^l) \in LTS_l \\
l \in [1, m] \quad i \in [1, n_l] \quad v' \in V_k \\
\hline
\langle \dots, (((\kappa, v)_1^l, \dots, (\kappa, v)_i^l, \dots, (\kappa, v)_{n_l}^l), Q^l), \dots, C \rangle \xrightarrow{envaction((\kappa, v')_i^l)} \\
\langle \dots, (((\kappa, v)_1^l, \dots, (\kappa, v')_i^l, \dots, (\kappa, v)_{n_l}^l), Q^l), \dots, C \rangle \\
\text{(ENVACTION)}
\end{array}$$

Although environment actions do not go through the action queue, they can only be triggered if the action is possible in 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. In the Figure 4.4 on page 55, the events log and the action

queues associated with the objects are shown at gateway level. Further, in the WoT implementation, an individual queue can be implemented for each type of action, but for the sake of simplicity we model a single queue for all action requests for an object. While processing the requests in the queue, we could consume by action type, thereby imitating the multiple queue behaviour.

4.4 Specification in LNT

In this section we describe the LNT [Cha+18; GLS17] encoding of the composition model described in the previous section. We formally specify the models of objects, as well as the communication between objects, objects and gateway, and objects and environment. This specification enables the automated verification of the composition correctness. The LTS corresponding to a composition expression operating on a set of objects in a given environment is obtained by invoking the LNT compilers, which implement its operational semantics [GLS17].

Encoding Objects

Each object is encoded as a process in LNT. The properties associated with the object are represented by local variables. Change in property values occur by interaction on the gates *event*, *action*, and *envaction*. The latter gate accounts for changes of property values imposed by the external environment. For example, a user (environment) can turn off the light even when there is no rule manipulating the light. Environment actions do not go through the action queue, instead they directly change the state of the object. They can be viewed as an interrupting behaviour. Listing 4.4 shows the outline of a generic object process. First, as mentioned in Section 4.2, an object can emit events upon change in property values. This emission of event is encoded in line 5, where we can see the emission (!) of the current value of property p_1 . Second, actions are added to the action queue on receiving an action request (line 8). The property $p1$ is received (?) and appended to the *actionQ* of the object. These action requests need to be consumed at some point and the consumption of an action request is represented by the delete operation, which removes the request from the *actionQ* as shown in line 12. It is to be noted that a successful execution of an action results in a change of state in the object and therefore, it is followed by the emission of an event notifying the change. *envaction* is similar to a regular action, except that it has an immediate effect, without going through the action queue. Finally, *done* is an auxiliary operation to

track one complete execution of the application scenario. By default, a composition is designed to run infinitely (unbounded) and presence of *done* label helps to track the completion of one execution of the composition expression. All these operations are modelled as a choice using the **select** operator. The entire process is enclosed in a **loop** to reproduce the reactive behaviour of the objects.

```

1  process OBJECT [event , action , envaction : any] is
2  var actionQ:QUEUE, p1:bool, p1Val:bool, p2:nat, p3:...
3  loop
4  select
5  event(!o1, ?any RuleId, !p1, ?any bool) [] event(...) [] ...
6  []
7  action(!o1, ?any RuleId, ?p1Val);
8  actionQ := append(p1Val, actionQ) [] ...
9  []
10 p1Val:= get_p1_value_fromQ(actionQ);
11 p1 := p1Val;
12 actionQ := delete_p1_value_fromQ(p1Val, actionQ) [] ..
13 []
14 envaction(!o1, ?p1)
15 []
16 done
17 end select
18 end loop
19 end var
20 end process

```

Listing 4.4: Outline of the LNT process of an IoT object

Now, let us look at a concrete device model. In Figure 4.5, the motion sensor device used in Rule 2 of the retirement home scenario described in Listing 4.3 is shown. Motion sensor has a property on which is of boolean type in the WoT Thing Description and when a motion is detected, the value of the property is set to true. Line 9 encodes the emit event. *elem* encapsulates the property on and its value and *?any bool* is used for synchronisation with rules. Lines 12-15 describe an action being pushed to the action queue, and lines 18-27 encode the consumption of an action from the queue. Since a single queue is used for all the different types of actions, a helper function *queue_has_property* is used to pick the first corresponding action to be consumed from the list of different actions in the queue (here, a single type of action, concerning property on, is possible). Finally, as in case of a motion sensor, a person moving in the room can trigger the action of setting the motion property to

true. This behaviour is accommodated in the LNT process by receiving the actions from the environment, as described in lines 30-38.

```
1  (* motion sensor device model *)
2  process motion [motionevent:any, motionaction:any,
3     motionenvaction:any, done:any] is
4     var q:QUEUE, motionon:bool, element:QELEMENT,
5         hasProperty:bool in
6         motionon := false; q := nil;
7         element := elem(on, false);
8     loop
9     select
10      (* emit event *)
11      motionevent(!motion, !elem(on, motionon), ?any bool)
12      []
13      (* push to action queue *)
14      var elementAction:QELEMENT in
15      motionaction(!motion, ?elementAction of QELEMENT);
16      q := append(elementAction, q)
17      end var
18      []
19      (* consume event from action queue *)
20      eval hasProperty := queue_has_property(on, q, !?element);
21      if hasProperty then
22      case element in
23      var onvalue:bool in
24      elem(on, onvalue) ->
25      motionon := onvalue
26      | any -> null
27      end case;
28      q := delete(element, q)
29      end if
30      []
31      (* handle action value from environment *)
32      var elementAction:QELEMENT in
33      motionenvaction(!motion, ?elementAction of QELEMENT);
34      case elementAction in
35      var onvalue:bool in
36      elem(on, onvalue) ->
37      motionon := onvalue
38      | any -> null
39      end case
40      end var
41      []
```

```

40      (* used to track one complete execution of the composition
        expression *)
41      if empty(q) then
42          done
43      end if
44      end select
45      end loop
46      end var
47      end process

```

Listing 4.5: LNT process specifying a motion sensor

Encoding Rules

As with the encoding of the objects, rules are also encoded in an LNT process. The ECA rules of the format **IF** *EVT* **THEN** *ACT* are specified using the sequence (;) operator of LNT. Each rule is transformed into a process of the form *EVT*; *ACT*, where the **select** operator is used to encode the disjunction of events in *EVT* and the **par** operator is used to encode the conjunction of events in *EVT* and actions in *ACT*, as shown below:

```

(* IF EVT1 OR EVT2 THEN ACT1 AND ACT2 *)
select
    EVT1 [] EVT2
end select;
par
    ACT1 || ACT2
end par

```

As a concrete example, the rule *R2* in Listing 4.3 can be specified in LNT as follows:

```

1  (* IF sofa_presence_detected=true THEN window_blinds-on *)
2  process rule2[event:any, action:any] is
3      event(!sofa, !r2, !elem(on, true), !true);
4      action(!window, !r2, !elem(blinds, true))
5  end process

```

Listing 4.6: LNT process specifying rule R2

Here, *sofa* is the object identifier, *r2* is the rule identifier and *elem* encapsulates 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 sofa appears with *!true* label and once the blinds are opened, *R2* becomes inactive and the event emitted by the sofa will be visible in the later parts of the LTS with a *!false* label.

Encoding Composition

The composition is encoded in a process. The patterns for transforming the composition language operators to an LNT specification are shown in Table 4.3.

Operator	Expression	LNT Pattern
SEQUENCE	$C_1 ; C_2$	$C_1 ; C_2$
PARALLEL	$C_1 C_2$	par $C_1 C_2$ end par
CHOICE	$C_1 + C_2$	select $C_1 [] C_2$ end select
REPEAT	$C_1(k)$	$i:=0$; while $i < k$ loop C_1 ; $i:=i+1$ end loop

Tab. 4.3: Composition language to LNT transformation patterns

The sequence operator is specified using the sequential composition (*;*) operator of LNT. Parallel execution is specified using the parallel composition (**par**), choice is modelled using a nondeterministic choice (**select**), and bounded iteration is specified using the cycle (**loop**) operator.

The composition described in the example can be encoded in a process as follows:

```

1  (* R1 ; (R2 || (R3 + R4)) ; R5 *)
2  process composition[event:any, action:any, done:any] is
3    loop
4      rule1[event, action] ;
5      par
6        rule2[event, action]
7      ||

```

```

8   select
9     rule3[event, action] [] rule4[event, action]
10  end select
11  end par;
12  rule5[event, action];
13  done
14  end loop
15  end process

```

Listing 4.7: LNT process specifying the composition of the retirement home application

The composition expression is executed in a **loop** as the composition need to be run infinitely. An additional gate *done* is used to capture one execution of the composition expression. As the composition is encoded to run infinitely, the presence of *done* indicates one run of the application.

Encoding Event Listeners

In addition to the core elements such as objects, rules, and composition, we need to encode the event listeners related to the application as described in Figure 4.4 on page 55. These event listeners are encoded in a process as a set of global listeners associated to the composition using a **select** operator. The goal of encoding the listeners is to capture the events in the overall composition which later can be used for verification. The events that are part of the rules, synchronize with the objects (see later the overall composition MAIN for more details) and appear on the LTS transitions, however other events will appear only if they are listened through the listener. This information is quite useful if one needs to track how often or when did a specific action occur in the application.

Again, considering the example in Listing 4.3 on page 54, if one needs to know when the light is *on* or the window is *open*, then the corresponding listener can be encoded in the GLOBALLISTENER process.

```

1  process globallistener[event:any] is
2  loop
3    select
4      event(?any RULE, !light, ?elem(on, any bool) of QELEMENT,
5        !false)
6    []
7      event(?any RULE, !window, ?elem(open, any bool) of QELEMENT,
8        !false)

```

```

7   end select
8   end loop
9   end process

```

Listing 4.8: LNT processes specifying the listener of the retirement home application

Here any RULE is used to specify that the event be a result of the execution of an action by any rule or it could be from the environment. The events appear with a !false flag as they are not part of the active rules.

Encoding Environment

The events in a rule, which are associated to objects, are triggered when the objects interact with their environment. The precursor for this trigger could be a change in the surroundings (e.g., temperature, luminosity, presence, etc.) or it could be the user interacting with the device (e.g., turning on the lights). So for the model to be complete and simulate the execution of the application scenario, the environment actions need to be specified. As described earlier in the object specification, every object is modelled to accept the interactions from the environment in the form of *envaction*. The environment is specified as a process with a nondeterministic choice of different environment actions.

```

1  (* Process to simulate the environment *)
2  process environment[done:any, envaction:any] is
3    loop
4      select
5        envaction(!door, !elem(open, true))
6        []
7        envaction(!sofa, !elem(on, true))
8        []
9        envaction(!aqm, !elem(quality, 0))
10       []
11       envaction(!aqm, !elem(quality, 1))
12       []
13       envaction(!door, !elem(open, false))
14     end select
15   end loop
16 end process

```

Listing 4.9: LNT processes specifying the environment of the retirement home application

The listing shown above is the environment process corresponding to the retirement home example in Listing 4.3 on page 54. Readers may notice that the environment actions correspond to the events in the set of rules. The reason behind this pattern is that for a composition expression to have all its rules triggered, i.e., to do a complete simulation, the environment needs to provide at least one event for satisfying the trigger conditions for each of the rules. Here a fully permissive environment is modelled, such that all the rules have the opportunity to get executed. However, in a real world environment, some of the event triggers may never happen and those rules will never be executed. On the other hand, it is not possible to model all the possible interactions and even if were to model all the possibilities, it would add unnecessary interruptions that are not of practical value for verification.

Encoding Application

Finally, a MAIN process which composes all the processes described earlier is specified. This process is the entry point to the specification of the overall behaviour of the application. A general outline of the MAIN process is shown in Listing 4.10.

```
1  process MAIN [...] is
2    par event, action, done in
3      par
4        COMPOSITION [event, action, done]
5        ||
6        GLOBALLISTENER[event]
7      end par
8    ||
9    par envaction in
10     par done in
11       OBJECT1 [event, action, envaction, done]
12       ||
13       OBJECT2 [event, action, envaction, done] || ...
14     end par
15   ||
16   ENVIRONMENT [envaction, done]
17 end par
18 end par
19 end process
```

Listing 4.10: Outline of the main process

The objects involved in the composition are interleaved and they do not synchronize on events and actions, as they do not interact themselves directly in a gateway integration pattern. Synchronization on gate *done* is used to track the completion of execution and reset the execution status. The interaction among objects is done via the COMPOSITION process. On the other hand, the environment interacts directly with the objects and synchronises on the label *envaction*. By definition, the environment can change or interrupt at any time thereby preventing fair execution of rules. In real world though, the environment is more reasonable (e.g., temperature does not change very often). So for simulation purposes we define the ENVIRONMENT with a set of relevant values identified by the property types and defined rules, instead of all possible values. This also helps in reducing the state space of the overall system and in fine-tuning the verification. 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, which is in turn synchronised on *events* and *actions* with the behaviour of the objects and ENVIRONMENT. Effectively, COMPOSITION acts like an orchestrator, which dictates the activation of rules and the triggering of actions.

4.5 Specification in Maude

In order to support reconfiguration analysis, we encoded the objects and their composition in Maude. This specification allows us to perform rewriting analysis of the reconfiguration properties described in Chapter 5. We have chosen to build two different encodings as they serve different purposes. The LNT encoding combined with the CADP toolbox allows us to verify the properties associated during the design of the application. The Maude encoding is specifically designed for reconfiguration analysis. Maude is more suitable because reconfiguration analysis primarily involves tracking of state changes with respect to the original configuration and term rewriting especially using equational logic, is an efficient way to perform these kinds of analysis. In this section we briefly introduce the different modules encoded in Maude as we have covered all the particularities of the encoding in Section 4.4. Readers interested in more details on the Maude specification may refer to the code online ¹.

The Maude specification defines the objects, rules, composition, and the application. Listing 4.11 shows the Maude specification defining the LTS, object and the composition.

¹<https://github.com/ajaykrishna/mozart/tree/mozart/maude>


```

1  fmod LTS is
2    pr STATE .
3    pr SET{Transition} .
4
5    sort LTS .
6    op model : State Set{Transition} -> LTS .
7  endfm
8
9  fmod DEVICE is
10   pr LTS .
11
12   sort Device .
13   op dev : Id LTS -> Device .
14 endfm
15
16 fmod COMPOSITION is
17   pr RULE .
18   pr INT .
19
20   sort Composition .
21   subsort Rule < Composition .
22   op seq :
23     Composition Composition -> Composition [assoc right id: none] .
24   op ch : Composition Composition -> Composition [comm] .
25   op par : Composition Composition -> Composition [comm] .
26   op iter : Composition Int -> Composition .
27   op none : -> Composition .
28 endfm

```

Listing 4.11: Composition specification in Maude

The LTS is defined by an initial state and the set of transitions. The set of states is implicitly encoded as traversing from the initial state using the transitions would yield the set of states. Further, the objects are encoded with an LTS. Finally, the composition language is defined along with the different operators supported by the language (sequence, choice, parallel, repeat).

An application contains the set of objects and the composition.

```

1  ----- IoT Application
2  fmod APPLICATION is
3    pr SET{Device} .
4    pr COMPOSITION .
5
6    sort Application .
7    op app : Set{Device} Composition -> Application .
8  endfm

```

Further, a rule is encoded as combination of two modules, one representing the events and the other representing the actions, as shown in Listing 4.12.

```
1  ----- Rule
2  fmod RULE is
3    pr LRULE .
4    pr RRULE .
5
6    sort Rule .
7    op rule : LRule RRule -> Rule .
8  endfm
9
10 ----- Left Rule
11 fmod LRULE is
12   pr REVENT .
13   pr SET{REvent} .
14
15   sort LRule .
16   subsort REvent < LRule .
17   op and : Set{REvent} -> LRule [ctor] .
18   op or  : Set{REvent} -> LRule [ctor] .
19 endfm
20
21 ----- Right Rule
22 fmod RRULE is
23   pr RACTION .
24   pr SET{RAction} .
25
26   sort RRule .
27   subsort RACTION < RRule .
28   op and : Set{RACTION} -> RRule [ctor] .
29 endfm
```

Listing 4.12: Rule specification in Maude

Two different modules are used for encoding the rule, as the events support conjunction and disjunction of events, whereas the actions support the conjunction of actions in the composition language.

Further, the execution semantics are defined using equational logic, which is not covered here as we have already specified the operational semantics and its encoding in LNT in the previous section.

4.6 Concluding Remarks

In this chapter, we have described the WoT specification with emphasis on the WoT Architecture and Thing Description (TD) specifications. The behaviour model for objects defined in terms of an LTS is based on the TD specification. IoT applications interact with each other through the gateway. These applications are built by composing ECA rules using the composition language. We have defined the operational semantics of the composition execution and it has been encoded as LNT and Maude specifications. In the next chapter, we look at the properties related to design verification and reconfiguration analysis. These properties are verified on the specifications of the application described in this chapter.

Property Specification for the Web of Things

” *Automation may be a good thing, but don't forget that it began with Frankenstein.*

— **An anonymous machine**

In this chapter we specify the properties that can be checked on the IoT applications at two different phases of their lifecycle. The first part of the chapter deals with behavioural properties that allow one to verify the correctness of an IoT application at design phase. The second part describes properties related to reconfiguration that one can use to qualitatively compare the composition of the deployed application with an updated composition while re-designing the application. The properties that can be checked during the design phase are specified in Model Checking Language (MCL). These properties are interpreted on the LTSs generated by compiling the LNT specification of the application. The reconfiguration properties are specified in Maude and they are interpreted on the Maude encoding of the application. These properties also enable users to deploy the redesigned application in a consistent state.

5.1 Design-time Verification with MCL

Here we describe several properties that are verified at design time using model checking techniques. These properties express both functional and quantitative aspects of the designed application.

Properties are specified using the Model Checking Language (MCL) [MT08; MR18], a data-handling temporal language equipped with the Evaluator model checker [MT08; MR18; MS03] of CADP. This verification enables to identify incorrect behaviour and also to perform qualitative and quantitative analysis of the IoT application at design time.

As described in Chapter 4, 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 $EVENT !R !O !ELEM (k, v) !B$, where R is the rule identifier, O is the object identifier, and $ELEM$ is a structure holding the object property (k,v) . 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 $ACTION !R !O !ELEM (k, v)$ and actions invoked by the environment are represented by labels $ENVACTION !O !ELEM (k, v)$, which do not have a rule identifier since they are invoked independently of any rule.

5.1.1 Generic Properties

Generic properties are applicable to all IoT applications independently of the way they are composed. These properties can be expressed using generic MCL patterns irrespective of the labels in LTS_C .

Deadlock freedom

This classic property ensures that no rule of the composition expression is blocked from completing its execution. This property is useful in identifying missing dependencies that prevent the composition expression for executing successfully.

Low-end IoT sensors and actuators have a simple looping (“flower pattern”) behaviour, where they constantly sense the environment or perform an actuation (update state attribute values). So, if we compose an application with only these kinds of objects, the resulting behaviour will not encounter deadlocks, as at any point in time an interaction with the environment is possible when a fully permissive environment is modelled. However, if we consider more complex objects, especially those involving a software, this may have an order of execution in their behaviour. Thus, when a composition is built using these objects and their order is not respected, it will lead to deadlocking behaviour.

A deadlock in terms of the LTS is a state from which a transition to another state is not possible (sink state) and in terms of ECA rules, it indicates that it is impossible to execute a certain event or action in an ECA rule of the composition. This impossible circumstance arises when the event or action requires another event or action to

occur before, which creates a dependency. So, unless the preceding action or event occurs, the rule will not run to completion.

The completion of rules is captured in LTS_C by two elements of our encoding. First, the behaviour of an object O having a specific execution order (e.g. software) is encoded in terms of events, which are emitted by O in the required order. For example, the fact that an alarm can be silenced only after it has been triggered is encoded as:

```
1  trigger := false ;
2  ...
3  select
4    event(!alarm , ?any, !elem(trigger , v)) ;
5  []
6    if (trigger) then
7      event(!alarm , ?any, !elem(silence , v), ?any bool)
8    end if
9  []
10 ...
```

Then, each time a rule R performs an action on O (meaning that the action is pushed into the action queue of O), R does not complete until the event corresponding to the change of property executed by the action is emitted by O .

The second element of encoding pertains to the way the rules are encoded. The execution of a rule is said to be complete, only if the event corresponding to the action executed by the device is emitted. So in the rule encoding, we add an additional event corresponding to the action executed. For example, in the encoding of rule R_2 shown previously, we add the following event after the execution of *(blinds, true)* action.

```
1  process rule2[event:any, action:any] is
2    event(!sofa , !r2 , !elem(on, true) , !true);
3    action(!window , !r2 , !elem(blinds , true));
4    event(!window, !r2, !elem(blinds, true), ?any bool)
5  end process
```

Thus, if the event is not emitted by the window object, the *done* action will not be reached in the composition, entailing a deadlock in LTS_C .

Deadlock freedom is expressed in MCL as:

where the necessity modality [true*] specifies that all sequences consisting of 0 or more arbitrary transitions must lead to states having at least one successor (possibility modality <true>true).

The deadlock freedom property can be illustrated in a simple composition scenario. Suppose there is an intrusion alarm which gets triggered and then flashes the beacon as described by a simplified LTS of the alarm in Figure 5.1. Here there is an order in which the state of the object changes, first the alarm gets triggered (on) and then the beacon flashes (flash).

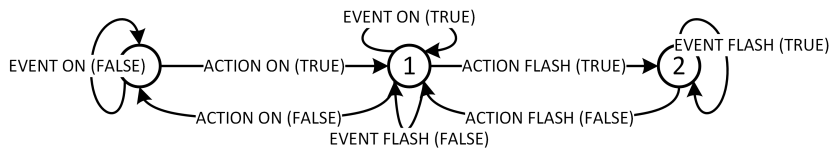


Fig. 5.1: LTS of an intrusion alarm

Assuming that the user has a motion sensor and a connected door, s/he builds the following set of rules.

```
R1: IF motion(on, true) THEN alarm(flash, true)
R2: IF alarm(flash, true) THEN door(open, false)
```

The rules are created with the intention of flashing the alarm and locking out the intruder once the intruder is detected. So, the application is composed of two rules in sequence R1; R2. Now, if the user checks the designed composition for deadlock freedom, it will indicate the presence of deadlock with a diagnostic LTS as shown in Figure 5.2.

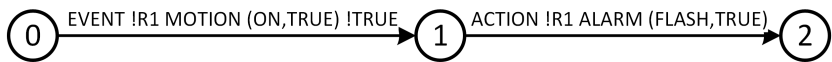


Fig. 5.2: Deadlock diagnostic LTS

In the diagnostic LTS, the transition from state 0 to state 1 is the event associated to the rule R1 and the next transition is the action part of the rule R1. However, there is no event witnessing the successful execution of the action present in the LTS (because the alarm object cannot execute this action before being triggered), thereby causing the deadlock.

Completeness

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 $\text{ACTION !R !O !ELEM}(k, v)$ encountered in the LTS_C , that afterwards, as long as the corresponding event $\text{EVENT !R !O !ELEM}(k, v)$ was not executed, it is still possible to reach it:

```
1 [ true* . {ACTION !R !O ?ELEM_k_v: String} .  
2 (not {EVENT !R !O !ELEM_k_v ?any})* ]  
3 < true* . {EVENT !R !O !ELEM_k_v ?any} > true
```

where the field $\text{ELEM}(k, v)$ is captured (as a string named ELEM_k_v) in the $\{\text{ACTION}...\}$ label predicate and reused in the $\{\text{EVENT}...\}$ predicate. Note that we check for emission of an event, instead of an action label. Since each action is pushed into a queue and an event is emitted upon consumption of the action, checking for the corresponding event validates that the action has been executed successfully.

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 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* [QS83], 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.

Further, since we model a permissive environment, the unreachable actions refer to the actions that are not triggered in spite of the events provided by the environment. This property can also be used to check, given a restricted environment, the actions that are not executed.

Spurious Events

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 on the cycle. This can be expressed in MCL as follows:

```

1  < true*.{EVENT ?any ?O ?K ?V ?any}>
2  < (not {ACTION ?any !O !K ?any})*.
3    {ACTION ?any ?O2 ?K2 ?any where (O2 <> O) or (K2 <> K)}.
4    (not {ACTION ?any !O !K ?any})*.{EVENT ?any !O !K !V !
5    false }
6  > @

```

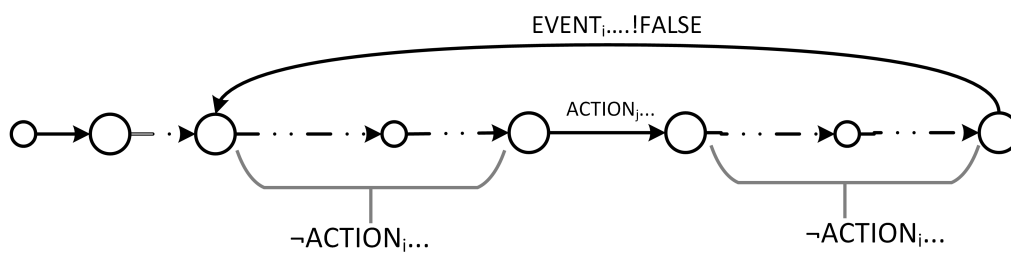


Fig. 5.3: Spurious event in an application

The first diamond modality specifies the existence of a finite sequence leading to the event $event_i$. The action predicate extracts the fields O , k , and v contained in the transition label for later use. The infinite looping modality ($\langle \dots \rangle @$) extends this sequence with a cycle containing $event_i$ and having the form described above. If this formula holds on LTS_C , the model checker will produce a lasso-shaped witness as illustrated in Figure 5.3.

The case of spurious events is specific to the gateway implementation, these events appear on the LTS because they have the associated listeners on the gateway, but the events are not relevant in the specific context of execution of the composition.

5.1.2 Application Properties

Application specific properties are expressed as temporal formulas containing labels from LTS_C . These properties are dependent on the application under consideration and one has to write specific temporal logic formulas to express the properties, unlike in generic properties where a standard formula is sufficient to verify the property across all IoT applications.

Functional Properties

The broad categories of functional properties are safety and liveness properties. Safety properties check that nothing bad happens in the system. Liveness properties check that something good eventually happens.

Safety Property

Consider the retirement home example in Listing 4.3 on page 54, since an old-age pensioner lives in this house he might forget to turn off the HVAC when he is away. Safety properties are designed to specify these kind of situations. They check that something unsafe never happens.

Safety properties ensure that there is no reachable error or unsafe state in the LTS. So a way to express safety properties is to forbid the undesirable execution sequences in the LTS. Safety properties follow the pattern $[SEQ]false$, where SEQ denotes the undesired sequence of actions, where the necessity modality followed by **false** prohibits the occurrence of the sequence. A safety property in case of pensioner would be that the HVAC is never turned on when the door is closed (pensioner is away). This implies that in LTS_C , once the door is *closed*, the HVAC should not be *on* in the subsequent states. This can be expressed in MCL as follows:

```
1 [(not DONE) *. 'EVENT !DOOR !ELEM (OPEN, FALSE) !FALSE '.  
2 (not DONE) *. 'EVENT !HVAC !ELEM (ON, TRUE) !FALSE ' ] false
```

Here, (*not DONE*) is used to check for the sequence within one execution instance of the composition expression. Since the expression is executed infinitely, it is possible that events may appear in the subsequent execution instances with different values. The first event is the closing of the door (by setting *open* attribute to *false*) and the second event is the HVAC being in *on* state. The presence of this sequence of events is forbidden as specified by the **false** state formula after the necessity modality.

Liveness Property

Liveness properties assert that the specified behaviour eventually occurs in the system. Unlike a safety property, where the violation results in a diagnostic trace identifying the violating sequence, the liveness property violation may not lead to a finite diagnostic trace as the action might eventually occur. So the way to express liveness properties is by specifying the desired execution sequences. These properties follow the pattern $\langle SEQ \rangle \mathbf{true}$, where *SEQ* denotes the required sequence of actions with possibility modality and **true** specify the existence of the sequence.

In the retirement home example in Listing 4.3 on page 54, a liveness property would be to check that eventually the door would be closed. This can be expressed in MCL as follows:

```
1 <true* . ( not DONE) * . 'EVENT !DOOR !ELEM (ON, TRUE) !FALSE ' > true
```

Here **true*** specifies any arbitrary sequence of actions, followed by the closing of the door.

It can be observed that the spurious events property follows the liveness property pattern. It in fact validates progress, whether the event will be eventually picked up by the the gateway.

Quantitative Properties

Quantitative properties are the characteristics of a system that can be observed and measured in terms of numerical values. They involve numerical measures such as probabilities (e.g., probability of a bulb failing), time (e.g., time to transmit the video), resources (e.g., amount of battery used by the device), etc. It is possible to denote a property in terms of more than one measure. In our work, we specifically consider the probability for property verification.

The probability of executing an action in a composition can be computed to perform comparative analysis. 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.

The probability of execution of an action is computed using the support provided by the Evaluator5 tool of CADP [MR18]. Evaluator5 supports on-the-fly verification of probabilistic properties on probabilistic transition systems (PTSs) [LS91], i.e., LTSs with probabilities attached to transitions. Given a probability of occurrence of an event, the probability of occurrence of an action is computed using the **prob** keyword in MCL, as shown in the formula below:

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

For instance, in the retirement home scenario in Listing 4.3 on page 54, turning on the HVAC 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 (\parallel) operation in the composition is replaced by choice ($+$), the probability of turning on the HVAC reduces to 0.348. This will save energy but it actually changes the composition wherein the existing behaviour is not preserved in the new composition. These kinds of changes can be analysed through the reconfiguration properties that we describe in the next section.

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 will 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). In the next section, we define reconfiguration properties that help one to compare the new application with the old application in terms of their behaviour.

5.2 Reconfiguration Properties

IoT application are not designed once and for all. They are constantly changing as the needs of the user evolve. The evolution may involve replacing objects or enhancing an application with additional automation and while doing so, user may want to better understand the effect of the evolution of an application. In broad sense, users may want to know if the existing services are preserved in the new application, whether there would be an interruption in the service while upgrading to the new application, whether the newly configured objects are really useful compared to previous object etc. These aspects of the evolution can be checked by specifying reconfiguration properties. These properties can be checked by the user to make the choice of whether to proceed with the deployment of the new application.

The reconfiguration properties compare two compositions, the composition that is currently deployed and the reconfigured version of the deployed composition. A reconfiguration may involve adding or replacing objects, rules, and modifications to the composition expression. The comparison takes as input, the current and new compositions and also the global state of the current application before the reconfiguration is applied. Global state contains the current state of all objects and the state of execution of the composition expression. The comparison checks if the reconfiguration satisfies three reconfiguration properties: seamless, conservative, and impactful reconfiguration. These properties allow one to know whether the new application can be deployed in a consistent state without having to completely restart the current application based on the global state of the application at the time of the reconfiguration.

5.2.1 Seamless Reconfiguration

The global state of the application corresponds to the current state of objects and the composition expression. The seamless reconfiguration property checks the feasibility of reaching this state in the newly configured application. This notion is important because if the current global state is reachable in the new configuration, it means that the new application can be deployed without restarting the application from the beginning. Thus, from the perspective of a user, the deployment is seamless as the current state of objects is maintained in the new configuration.

Let us now look at how the notion of seamless reconfiguration can be used to deploy the newly configured application. A user can reconfigure the application by

adding, removing, and modifying the composition expression. The set of objects involved in a reconfiguration can be categorized as newly added objects, removed objects, and remaining objects. An application can be considered to be seamlessly reconfigured if all the remaining objects can reach again the state where they were before initiating the reconfiguration, following the new composition expression (reconfigured composition). Among the set of objects involved, only the remaining objects are considered, as their current state needs to be maintained in the new deployment. The newly added objects do not have a history of execution, so they can be added in any state and obviously, the removed objects will not be part of the new deployment and thus can be safely ignored.

The goal is to check whether the new application can reach the current global state and to do so, the execution history of the application is considered. Specifically, the trace corresponding to the execution of the expression (application) that is currently deployed is considered. This trace provides the path to reach the current global state from the initial deployment. Using this trace, we check if it is possible to execute the new composition expression in which all the remaining objects can reach their former states repeating the same behaviour.

Definition 7 (Seamless Reconfiguration). Given two applications $A_c = (O_c, C_c)$ and $A_t = (O_t, C_t)$, each defined by a set of objects and a composition expression, given the current global state $((s_1, \dots, s_n), s)$ of application A_c and the trace t to reach that state consisting of a sequence of couples (object identifier, action), the seamless reconfiguration property is satisfied if, when executing application A_t guided by trace t , each remaining object $O_i \in O_c \cap O_t$ starting from s_i^0 can execute the actions in t and reach its current state s_i .

Since the seamless reconfiguration considers only the remaining objects, the execution trace is filtered by keeping only the actions executed by the remaining objects. The remaining objects must follow the execution specified by the filtered trace, whereas new objects evolve following the new composition expression. As a result, the states obtained by running that trace in the new application correspond to the states where new objects have to be moved to when deploying the new application.

Property Encoding in Maude

The seamless reconfiguration property is encoded as an operation in Maude. It takes as input, the deployed application, the new configuration, the global state of the

application, and the trace corresponding to the execution occurred to reach the current state of the application. The result of this operation is a boolean response indicating whether the new configuration satisfies the seamless reconfiguration property. A few operations related to the property encoding in Maude are shown in Listing 5.1.

```

1  op checkSeamlessReconfiguration :
2      Application Application Set{Tuple{Id,State}} List{Tuple{Id,
3          Label}} -> Bool .
4  op checkSeamlessReconfigurationAux :
5      Application Application Set{Tuple{Id,State}} List{Tuple{Id,
6          Label}} Set{Id}
7          -> Bool .
8  —— filters the trace to keep only labels belonging to
9  remaining objects
10 eq checkSeamlessReconfiguration(App1, App2, GS, Tr)
11     = checkSeamlessReconfigurationAux(
12         App1, App2, GS,
13         filterTrace(Tr, computeCommonObjects(App1, App2)),
14         computeCommonObjects(App1, App2)
15     ) .
16 —— runs the trace in the new application until it is possible
17 eq checkSeamlessReconfigurationAux(App1, App2, GS, Tr, Ids)
18     = compareGS(
19         App1, App2, GS,
20         getGlobalState(runTrace(App2, Tr, Ids))
21         ) and getBoolRes(runTrace(App2, Tr, Ids)) .

```

Listing 5.1: Seamless reconfiguration in Maude

As a first step in checking the property, the trace executed by the deployed application is executed on the new configuration, and the global state reached by this execution is saved. This global state is unique as it is guided by the trace which represents the execution history. It is to be noted that the newly added objects may be moved to different states as part of the new execution following the trace. Now that there are two global states - the global state of the application A_c at the time of reconfiguration and the state obtained by executing the trace in A_t , these global states are checked if they coincide.

In Listing 5.1, the first operation `checkSeamlessReconfiguration` takes as input all required elements (two applications, one trace and one global state) and filters out all events/actions in the trace that do not belong to the set of remaining objects. The second operation `checkSeamlessReconfigurationAux` takes as input two applications, one global state, the filtered trace and the set of remaining objects. This operation

calls `runTrace` to execute the second application guiding this execution by the trace. If an object is not available anymore (removed in the second application), any object can be run instead (yet according to the new composition expression). The operation `compareGS` checks that the remaining objects have reached the same state in both global states. The operation `runTrace` also returns a Boolean value indicating whether the complete trace was executed for remaining objects.

5.2.2 Conservative Reconfiguration

Conservative reconfiguration builds upon the seamless reconfiguration property where it checks if the partial behaviour of the current application is replicable in the new configuration. A reconfiguration is called conservative if the seamless property is preserved and if, from the global state in which the reconfiguration is applied, all behaviours that could be executed in the current application (objects and composition expression) are still executable in the new application. It implies that all the possible executions in the current application from the global state are also possible in the new configuration from its behaviourally equivalent global state. More precisely, each trace possible in current configuration is also possible in the new configuration from their global states.

The conservative property is useful when one extends an existing composition by adding new services, but at the same time one wants to ensure that existing services are preserved in the new configuration.

Definition 8 (Conservative Reconfiguration). Given two applications $A_c = (O_c, C_c)$ and $A_t = (O_t, C_t)$, given the current global state $((s_1, \dots, s_n), s)$ of application A_c and the trace t to reach that state, the conservative reconfiguration property is satisfied if the seamless reconfiguration property is satisfied and if each trace t' that can be executed in A_c from (s_1, \dots, s_n) can also be executed in A_t from (s'_1, \dots, s'_m) where the global state (s'_1, \dots, s'_m) is obtained by executing A_t guided by t .

The reconfiguration check involves two operations. First, the trace is executed on the new configuration and a global state of the new configuration is obtained. This needs to be done as the new configuration may have some objects replaced or removed and the global state of the current application cannot be mapped to the new configuration. So an equivalent global state is obtained by replaying the execution of events and actions that occurred in the current configuration on the new configuration. The newly added objects may have to be placed in appropriate states

following the new composition expression. From this starting newly computed global state, the possible traces are computed to check for conservative reconfiguration.

Property Encoding in Maude

Similar to seamless reconfiguration, the check takes as input the deployed application, the new configuration, along with the global state and the execution trace. Therefore, the operations for checking conservative reconfiguration extend the seamless reconfiguration operations. The result of the check is a boolean value indicating whether the property is satisfied.

```
1  op checkConservativeReconfiguration :
2    Application Application Set{Tuple{Id,State}} List{Tuple{Id,
3    Label}}
4    -> Bool .
5  eq checkConservativeReconfiguration (app(Devs1,Compo1) , app(Devs2
6  ,Compo2) , GS1, Tr)
7    = compareFutureTraces(
8      Devs1, Compo1, GS1,
9      getBuffers( runTrace( app(Devs1,Compo1) , Tr, keepAllIds(
10     Devs1))),
11     Devs2, Compo2,
12     getGlobalState(
13       runTrace(
14         app(Devs2, Compo2) ,
15         filterTrace(Tr,
16           computeCommonObjects( app(Devs1,Compo1) , app(Devs2
17           ,Compo2))),
18         computeCommonObjects( app(Devs1,Compo1) , app(Devs2,
19         Compo2))
20       ) ) ,
21     getBuffers(
22       runTrace(
23         app(Devs2, Compo2) ,
24         filterTrace(Tr,
25           computeCommonObjects( app(Devs1,Compo1) , app(Devs2
26           ,Compo2))),
27         computeCommonObjects( app(Devs1,Compo1) , app(Devs2,
28         Compo2))))
29     ) .
```

Listing 5.2: Conservative property in Maude

The check first computes the equivalent global state. Then all the possible execution traces from the computed global state are matched with the execution traces from the current configuration. The match process runs until a mismatch is found or until

all the traces are matched (complete match). The Maude code shown in Listing 5.2 shows how the comparison is executed. The equivalent global state is computed by running the trace on the new composition expression. Further, `getBuffers` get the state of all action queues. Finally, the operation `compareFutureTraces` executes all possible traces in the current application, and checks whether traces can be executed on the original application as well.

5.2.3 Impactful Reconfiguration

Impactful reconfiguration extends the seamless property with an additional constraint. A reconfiguration is called impactful if the seamless property is preserved and if all new behaviours can be executed in the new composition. It checks whether the newly added behaviours can be executed given the current state of the application.

The check proceeds by executing the trace to obtain the global state of the new application. Then, all the behaviours that are reachable from the global state are computed following the new composition expression. This set of behaviours must contain all the newly added behaviours for the reconfiguration to be impactful.

Here the term impactful is used because all the newly added behaviours make an impact on the execution of the composition expression. This property is useful if one wants to measure the usefulness of the newly added behaviour.

Definition 9 (Impactful Reconfiguration). Given two applications $A_c = (O_c, C_c)$ and $A_t = (O_t, C_t)$, given the current global state $((s_1, \dots, s_n), s)$ of application A_c and the trace t to reach that state, the impactful reconfiguration property is satisfied if the seamless reconfiguration property is satisfied and if each new object $O_i \in O_t \setminus O_c$ has its entire behaviour covered in $Tr \cup Tr'$ (i.e., for each O_i , for each $s_1 \xrightarrow{ed} s_2 \in T_i$, e appears at least once in $Tr \cup Tr'$), where (s'_1, \dots, s'_m) is the global state obtained by executing A_t guided by t , Tr is the set of all traces that can be executed in A_t to reach (s'_1, \dots, s'_m) , and Tr' is the set of all traces that can be executed in A_t from (s'_1, \dots, s'_m) .

It is worth comparing this property with the completeness property written in MCL. Both the properties check for reachability but in impactful reconfiguration a comparison between two compositions is made whereas in the completeness property reachability in a single composition is checked. Both properties have some overlapping elements.

Property Encoding in Maude

The impactful property checks that all new behaviours can be executed in the new application. The Maude code for checking the property is partially shown in Listing 5.3. The property check involves first checking seamless property where the global state of the new composition is deduced from the trace of the current application. The second part of the check focuses solely on the new composition. Following the new composition expression, all the observable events and actions associated with each of the devices are computed and stored. Finally, the `getExecutions` operation checks that all events are reachable in the objects from their respective initial states.

```
1  op checkImpactfulReconfiguration : Application Application List{
2      Tuple{Id,Label}} -> Bool .
3
4  eq checkImpactfulReconfiguration(App1, app(Devs2, Compo2), Tr)
5      = ... ,
6          getExecutions(runTrace(
7              app(Devs2, Compo2),
8              filterTrace(Tr,
9                  computeCommonObjects(App1, app
10                     (Devs2, Compo2))),
11                  computeCommonObjects(App1, app(
12                     Devs2, Compo2))
13              ) )
14      ) .
```

Listing 5.3: Impactful property in Maude

The two properties, impactful and conservative reconfiguration, have seamless property embedded within them. The precondition of having the seamless property to be satisfied exists because without the reachability of the global state in the new composition, the impactful and conservative properties do not make sense. The notion of global state in the new composition is the binding link between the current and the proposed reconfiguration. Further, the impactful and conservative properties are independent of each other. A reconfiguration that preserves conservative reconfiguration may not satisfy the impactful reconfiguration property because still, it may not be possible to reach all new events in the new expression.

5.2.4 Reconfiguration Example

The reconfiguration properties can be illustrated using a set of applications described in Figure 5.4. The figure shows three different compositions. An original one and two more reconfigurations built on the original configuration.

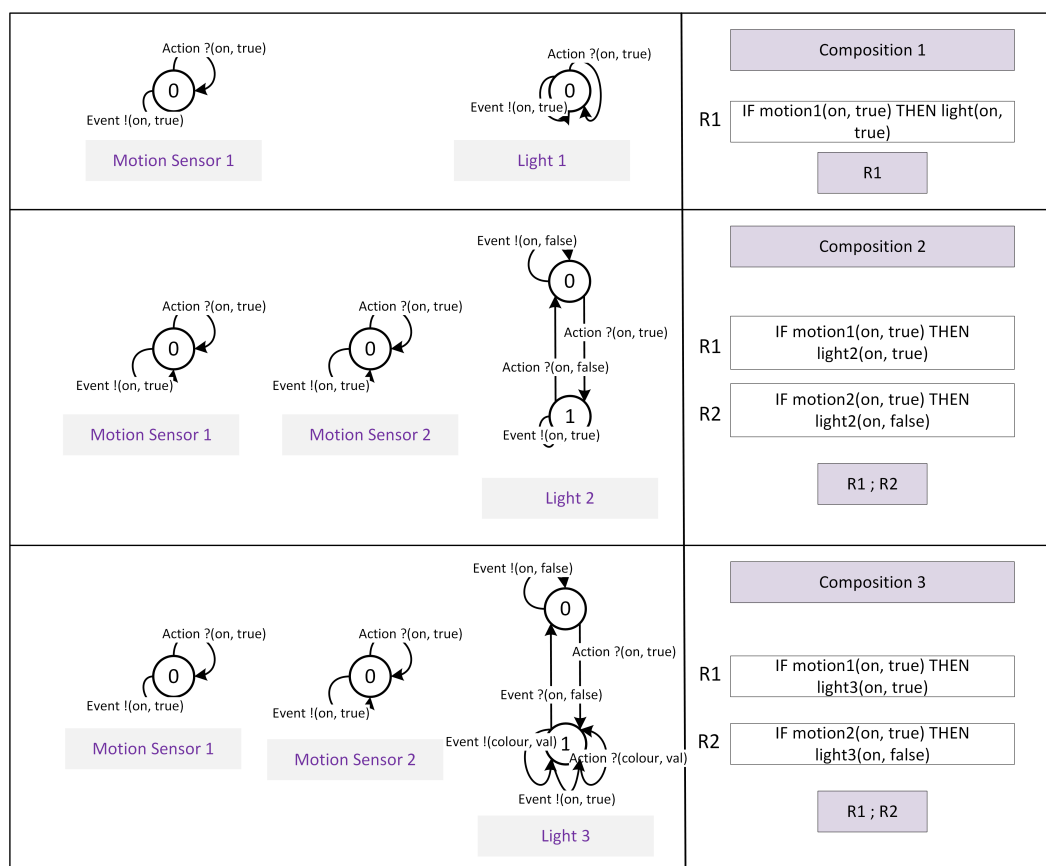


Fig. 5.4: Compositions illustrating reconfiguration

Initially, an application was designed using a single rule and it was deployed as an application. The rule uses two objects, a motion sensor and a light. The light offers a single event/action where it can be turned on (it turns off automatically after a while). The automation was deployed at the entrance of a home where whenever a motion was detected, the light was turned on.

In the first reconfiguration of this application, another motion sensor is brought in and is placed to capture the exit. Along with this, the light was replaced by another light which could be turned off and turned on at will. Now the automation involved two rules composed in a sequence. First, when the motion is detected at the entrance the light is turned on and when the motion is detected at the exit, the light is turned off. Unfortunately, after some time the light bulb fails and it is being

replaced by another light which supports changing of its colours. The behaviour of objects involved in the compositions is shown in Figure 5.4. It abstracts certain details but provides important information about the order of execution.

As readers may recall that seamless reconfiguration concerns the remaining objects and it checks whether these objects can reach their original states in the new application following the trace. In this reconfiguration, the remaining objects in composition 2 and composition 3 are the motion sensors (motion sensor 1 in composition 2 and both the sensors in composition 3) which have a single state which is reachable at any time in the composition. The first reconfiguration, when compared with the original application, returns negative for conservative reconfiguration. It is because the behaviour of turning on the light at any instant is not preserved anymore in the new application. The change however is impactful, as the new behaviours can be executed from any state. The change from composition 2 to composition 3 on the other hand is conservative, as the same behaviour is preserved but since the new behaviour (changing of colour) cannot be executed following the composition expression, the reconfiguration is not impactful.

5.3 Concluding Remarks

In this section, we have described two different sets of properties: one focusing on verification during the initial design of the application and the second set of properties that can be checked during the redesign of the application. The use of two different encodings is due to the verification requirements. Even though both the checks happen during the (re)design of the application, they are associated with different phases of an application lifecycle. LNT and MCL encoding based on process algebra is useful when checking the behavioural correctness of a system. Rewriting logic in Maude on the other hand offers a more efficient way to track the evolution of states, which is the essence of reconfiguration.

Mozart Tool

“Users do not care about what is inside the box,
as long as the box does what they need done.

— **Jef Raskin**

about Human Computer Interfaces

This chapter describes the implementation of the proposals mentioned in the previous chapters. The MOZART tool is developed considering non-expert home users. This tool is developed on top of the Mozilla WebThings components. Later in the chapter, we present some of the usability and verification related experiments.

6.1 Project WebThings

WebThings is a project by Mozilla which implements the Web of Things architecture. The objective of the set of tools provided by the project is to enable end-users to monitor and control IoT devices over the web. Mozilla WebThings Gateway and WebThings Framework are the two major software distributions provided by Mozilla under this initiative. An incentive for implementing our proposals on top of this project was that it is hosted under Mozilla Public License 2.0, which permits extensions and modifications.

6.1.1 WebThings Gateway

WebThings Gateway is a software that can be deployed on smart home gateways. It allows users to monitor and control the IoT devices installed in their homes through the web interface provided by the gateway. Compared to the software that is usually provided by the device manufacturers, the gateway allows one to manage the devices locally. The data is stored locally and not processed through cloud services. An added advantage is that users do not have to install multiple software for different devices. The gateway acts as an IoT Hub for all the devices in a home where they

can be managed via a web browser. The gateway can be installed on a Raspberry Pi, or it can be hosted on a PC or as a docker image.



Fig. 6.1: WebThings monitoring UI

The UI component of the gateway called ThingsUI, provides a web user interface that allows users to connect the smart home devices and create rules using these objects to build automation scenarios. Figure 6.1 shows the monitoring screen of the ThingsUI. It shows a set of IoT objects and also virtual objects that can be controlled through the UI. Virtual objects are a software implementation of real-world objects. They can be primarily used for testing of automation scenarios as they can be monitored and controlled just like the real objects. Figure 6.2 shows the ECA rule automation UI provided by the Things UI. It shows a rule which detects the temperature of the room using a Philips Hue sensor and if the temperature is greater than 30, it turns on a light to red. Users create these rules by dragging and dropping the objects that are available on the bottom of the screen and by setting their attributes as events and actions. A rules engine implemented in the gateway executes these rules. It is worth noting that when users create multiple rules, they are executed as and when the event conditions are satisfied, without any specific order.

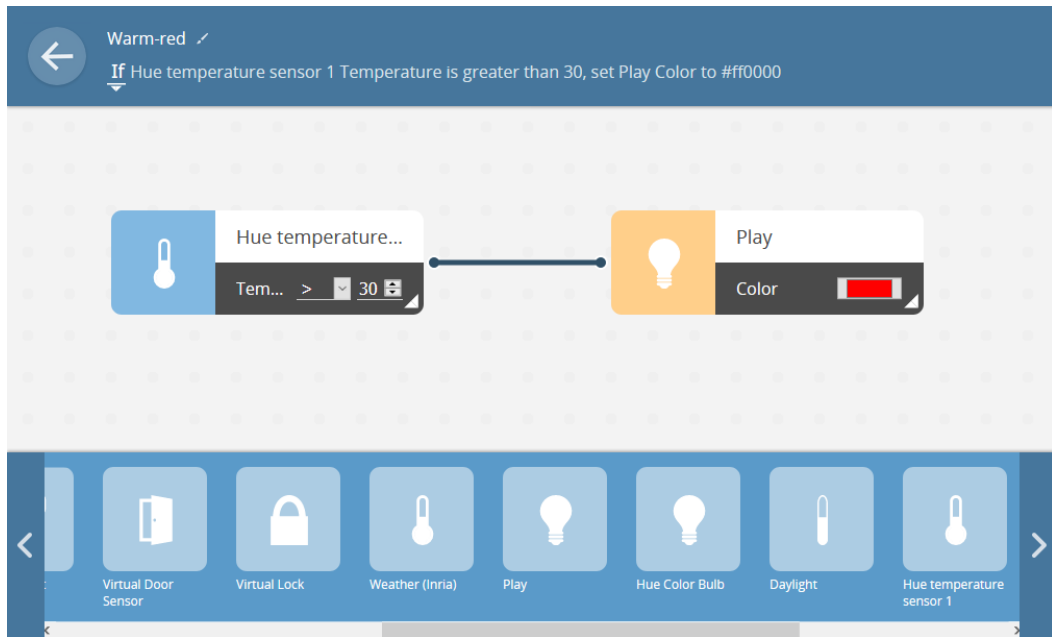


Fig. 6.2: WebThings rules UI

There are two ways to make the objects compatible with the gateway. First, the IoT devices can be built ground-up following the WoT standard where the things are described adhering to the Thing Description specification. In order to build this implementation, Mozilla provides a set of reusable components as part of the WebThings framework to build WoT objects that expose the WoT APIs, which can be in turn used to discover and interact with other objects or a gateway.

The second option which is much more relevant for the devices available in the market, is to build adapters that package the custom APIs of the devices to follow the WoT architecture. There are already dozens of adapters available that support hundreds of devices available in the market (e.g., Philips Hue adapter for managing all Hue devices).

Users can install the adapters from the ThingsUI. Once the adapters are installed, the devices supported by the adapters can be added and managed from the ThingsUI. The devices available in a home can be added using discovery protocols such as mDNS for network connected devices or using Bluetooth Eddystone beacon if the devices are in close proximity. The added devices can be used in the rules screen of the ThingsUI to create ECA rules. The devices whose TD properties are read-only can only be used in events and read-write properties can be used in both events and actions.

6.2 WebThings Extensions for Design-time Verification and Reconfiguration

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
- 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
- The UI should provide a screen to redesign an existing composition. The screen should allow users not only to modify rules and their order of execution, but also to create new rules
- The UI should allow users to compare the running composition with the new composition for reconfiguration properties. Just like the design verification, the process should be seamless to the users (e.g., click of a button)
- Once users are satisfied with the redesign of the composition, the tool should allow users to deploy the new composition with minimum interruption to the running application (objects).

The last three items in the requirements relate to the reconfiguration and the remaining ones relate to the design time verification. Although all the extensions are packaged within the MOZART tool and they re-use certain code, we describe

the extensions in two sections as they serve conceptually two different analysis at different stages of application lifecycle.

6.2.1 Design-time Verification 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. Figure 6.3 shows an overview of the approach. 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:

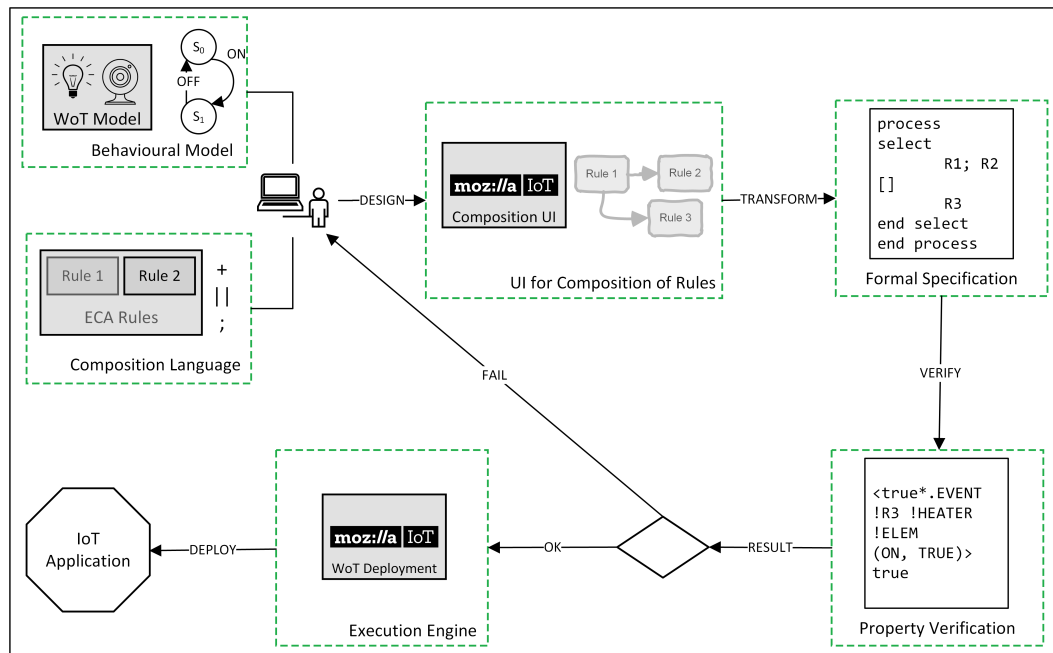


Fig. 6.3: Mozart: Design and deployment of IoT applications

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. Figure 6.4 shows the UI for composition of 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. In Figure 6.4, on the top half of the left column, the existing rules can

be seen. There is an option to create new rules right from this screen. The operators are on the bottom half of the left column. Both rules and operators can be dropped into the main screen to build a composition.

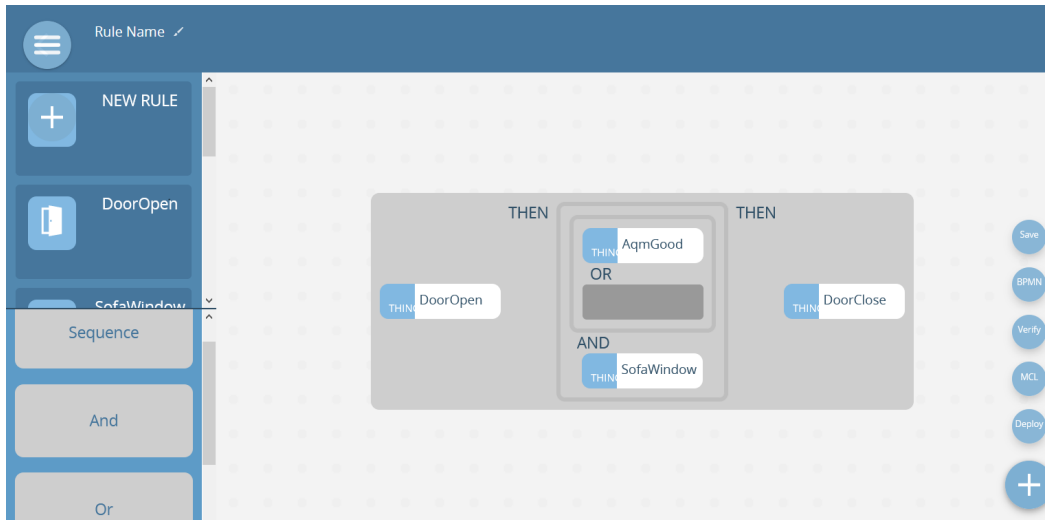


Fig. 6.4: Composition UI

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. Process models can be more intuitive for displaying large or complex composition scenarios.

Last but not least, users are provided a button to deploy the composition. The composition is deployed immediately upon clicking this button. Figure 6.5 shows a composite screenshot containing the different screens of the UI. On the top right, 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.

The second component is the **verification component**, which has two parts to it. The first part of the component is a model transformer. The objects, rules, and the composition are represented as JSON objects. These JSON models are transformed into LNT formal specification. Along with this, necessary environment



Fig. 6.5: Mozart UI screenshots

is generated in the LNT specification to simulate the scenario execution. The second part is the software that integrates the Mozart tool with the CADP verification toolbox [Gar+13]. It invokes the compilation of the specification, the property verification, and returns the results of verification 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 and we needed to build 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, 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. Initially, all rules are considered to be inactive (disabled), then a set of rules are activated (enabled) based on the composition expression.

6.2.2 Reconfiguration Components

Now let us take a look at the three components built on top of WebThings to support checking of reconfiguration properties and deployment of the new composition.

Support for reconfiguration is achieved by making extensions at three different levels. First, at the design level, interfaces need to be available to specify changes to an existing application. Once the changes are specified, a component to verify the consistency and correctness of the redesign is required. Finally, deployment of the new configuration needs to be taken care of.

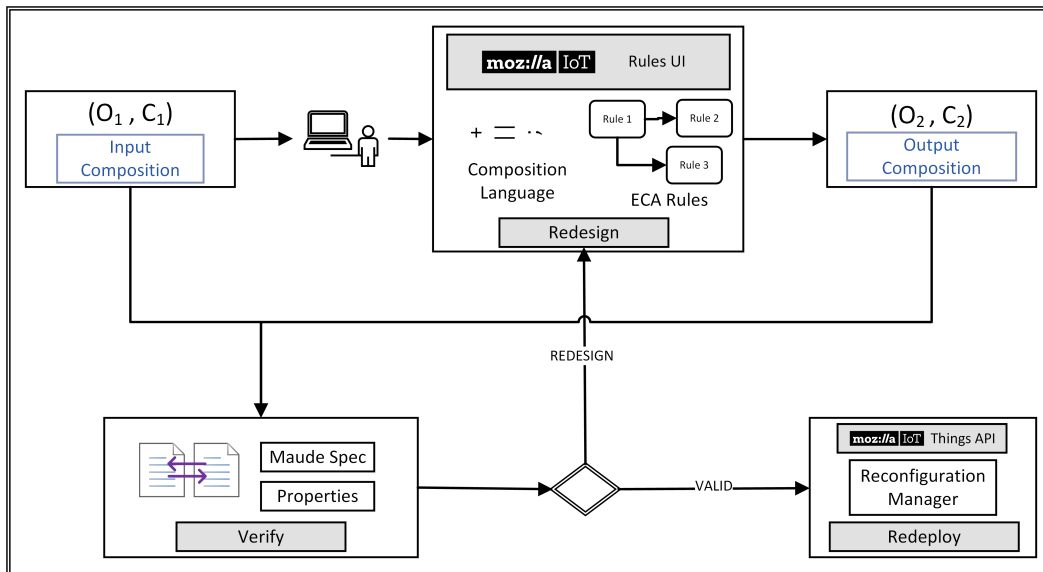


Fig. 6.6: Reconfiguration workflow

The reconfiguration workflow is shown in Figure 6.6. Initially, users would have designed the application and deployed it in their homes. Once a reconfiguration of the running application is envisaged, users move the concerned application to the newly designed **reconfiguration UI**. The UI is similar to the composition UI shown in Figure 6.2, except it imports the existing composition to this reconfiguration screen. Just like the composition UI, it allows users not only to modify the rules which may involve adding, removing, or changing of objects but also to change the way in which the rules are composed in an application. Once the redesign is finalised, users can compare the new configuration with the original configuration to check impact of reconfiguration. The comparison is guided by the reconfiguration properties to be checked. The properties are checked by transforming the JSON specification of the composition into a specification in Maude using a **model transformer**. Upon finishing the check, the users are provided with a response indicating if they could proceed with the deployment. If the response is negative, they can revise the design or keep the application running as it is. Otherwise, they can proceed to the next step where the **reconfiguration manager** handles the deployment.

The manager performs two tasks: i) undeploy the removed objects preserving the state of remaining objects; ii) deploy the new objects and run the reconfigured application, picking up from where it left off.

First, current states of all objects are stored in a database along with the execution history of the application. This is followed by disabling of rules. Finally, rules are replaced or new rules are created depending on the reconfiguration. New rules are

created using the Rules UI and they can be enabled or disabled depending on the composition. Here, we mention rules and not individual objects because adding or removing of objects is a modification to a rule, as objects are a part of an event or an action. In other words, adding an object means including the object in a rule, from the available pool of objects and removing an object implies it is no longer used in a rule. Now, these rules need to be deployed for the application to run. Deployment resumes the application from the state where it was before initiating the reconfiguration. Remaining objects maintain their previous states. For newly added objects, we run the execution trace of the former application on the new composition expression. As a result, we obtain the states where the new objects have to be moved when deploying the new application. Newly added rules are initialized in disabled state. As a last step, we use the execution history of the former application to compute the state from where the new composition expression should start, which allows us to determine the set of rules to be enabled. From here, previously developed execution engine takes care of the running of application. It follows the composition expression semantics by enabling or disabling relevant subsets of rules as the execution of the expression progresses.

6.3 Technology and Implementation

A simplified view of the components and technologies used in the tool is shown in Figure 6.7. 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 and Maude specification is handled using Freemarker templating engine. Communication with the CADP verification toolbox and Maude system is done via system calls.

6.3.1 CADP Toolbox

The CADP toolbox [Gar+13] which provides first class support for LNT as a specification language, contains a wide range of tools to perform formal verification tasks. Here are some of the tools that we used in our work. These tools can be invoked from the command-line.

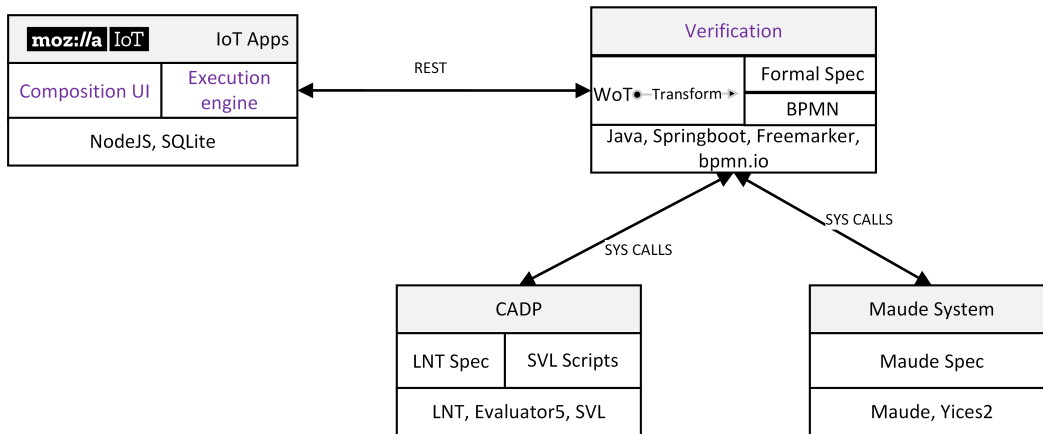


Fig. 6.7: Mozart: Components and technology stack

- **LNT.OPEN** [Cha+18] 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.
- **generator** generates LTS encoded in Binary Coded Graphs (BCG) format from the specification using reachability analysis. The output of the compilation is fed to the generator to generate LTS in BCG format.
- **bcg_min** is a tool for minimization (reduction) of LTSs. It takes as input LTS encoded in BCG format. This is used to reduce the LTS using strong reduction during verification.
- **evaluator** [MT08; MR18] can be used for on-the-fly verification of a temporal property on an LTS. 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 comparison is a Boolean value and it provides a diagnostic LTS in case the result is false. We use two versions of the **evaluator** tool in our work. **evaluator5** which supports probabilistic transition systems is used for quantitative analysis and the remaining properties are checked using **evaluator4**.
- **svl** [GL01] is a script execution tool. It can be used to automate the verification steps. Scripts are written in Script Verification Language (SVL) and **svl** compiles it to a shell script. The verification steps are automated in the backend using SVL scripts. It automates the steps such as reduction and invoking the model checker to verify a given property.

The complete list of tools available in CADP can be found here [Con20].

6.3.2 Maude System

Maude 3.0 tools can be installed on Linux systems. It comes with a connection to an SMT solver for satisfiability (SAT) analysis. In our work, we use Maude with Yices 2 distribution, but there is an option to use CVC 4 for SAT solving. Maude system is packaged as a executable, which takes as input the Maude specification. In this specification, we can specify the reconfiguration properties to be verified using **rew** keyword followed by the name of the property. This initiates the rewriting analysis and returns the result on the terminal. The result of rewriting is a boolean response indicating whether the property is satisfied or not, which is captured and displayed on the user screen.

6.3.3 Implementation

In this section we briefly describe some of the implementation details of the verification and execution engine components.

The objects, rules and composition expressions are encoded as JSON data. For each of these data, a corresponding Freemarker template which generates LNT and Maude code is defined. The template is defined with static elements in the specification like process declaration in LNT, generic Maude operations etc. These elements remain common for every object, rule or composition. But there are specific elements like identifiers, labels to synchronise on, etc., and these are dynamically populated from the JSON data. Listing 6.1 shows a template to generate the Maude specification of a scenario. A scenario is basically a composition of rules. The elements within `<#.../>` denote the loading of the objects into the memory and their values are dynamically loaded within `$. . . $`. Here we can see that the rule identifiers and their composition are dynamically populated (lines 14-16).

```
1  load semantics.maude
2  <#list maudescenario.getObjects() as obj>
3  load ${obj}$.maude
4  </#list>
5
6  fmod SCENARIO is
7    pr DEVICE .
8    pr QID .
9    subsort Qid < Id .
10   pr COMPOSITION .
```



```

11   pr APPLICATION .
12
13   <#list maudescenario.getRules() as rule>
14   op ${rule.getId}$ : -> Rule .
15     eq ${rule.getId()}$
16     = ${rule.toString()}$
17 </#list>
18   endfm

```

Listing 6.1: Maude Freemarkers template

The composition expression is evaluated through operator-precedence parsing. This generates a Reverse Polish Notation expression that is transformed to a workflow that accounts for the split and merge of the control flows. This workflow guides the execution engine.

The execution engine manages the enabling and disabling of the rules. Events in the rules are associated with listeners implemented in JavaScript. These listeners can notify the occurrence of events. A rule is disabled by shutting down the event listeners corresponding to the rule. Similarly, new listeners are created when new rules are created. The execution engine progressively starts or shuts down event listeners as the execution of the composition expression progresses. A composition is said to be deployed as an application when a relevant set of rules are enabled in a composition expression.

The complete codebase of the implementation is available on Github¹.

6.4 Evaluation

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

6.4.1 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

¹<https://github.com/ajaykrishna/mozart/>

of 36 IoT automation scenarios chosen from the literature. 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 3 and 1 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 $\kappa=0.71$ ²) 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.

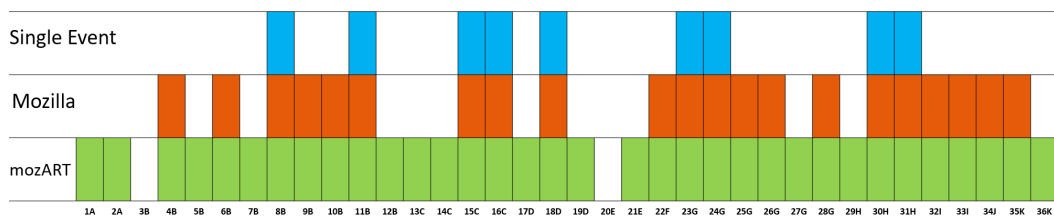


Fig. 6.8: Plot showing the scenarios with presence of bars indicating the possibility of implementing the scenarios.

A plot showing the list of scenarios and if they could be implemented using the three programming models is shown in Figure 6.8. Except for two scenarios, MOZART can implement all the scenarios compared to IFTTT and Mozilla WebThings. 3B refers to the scenario "Start brewing coffee 15 minutes before my alarm", programmers

²https://en.wikipedia.org/wiki/Cohen%27s_kappa

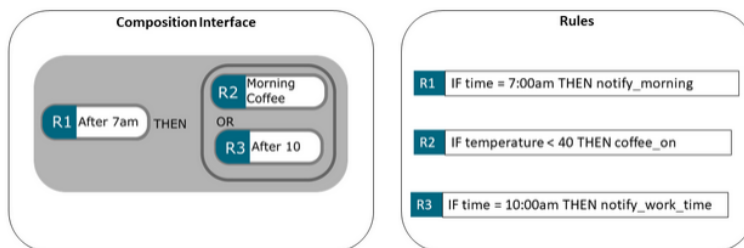
were unable to translate *before alarm*, as the time of alarm was not available to them. Part of 20E has a step involving duration “*turn on for 10 seconds*”. Since there was no straightforward way to encode duration, programmers did not translate the scenario.

6.4.2 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). 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 8 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 6.9

6. You want a pot of coffee to be brewed when it’s below 40 degrees outside, but only before 10:00 am every day.



The automation scenario described in text is accurately represented by the set of rules and their composition interface.

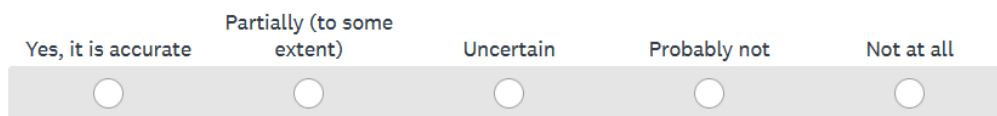


Fig. 6.9: Screenshot of the training and feedback screen used for Mozart tool users.

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 Likert scale³ responses of the users is shown in Figure 6.10.

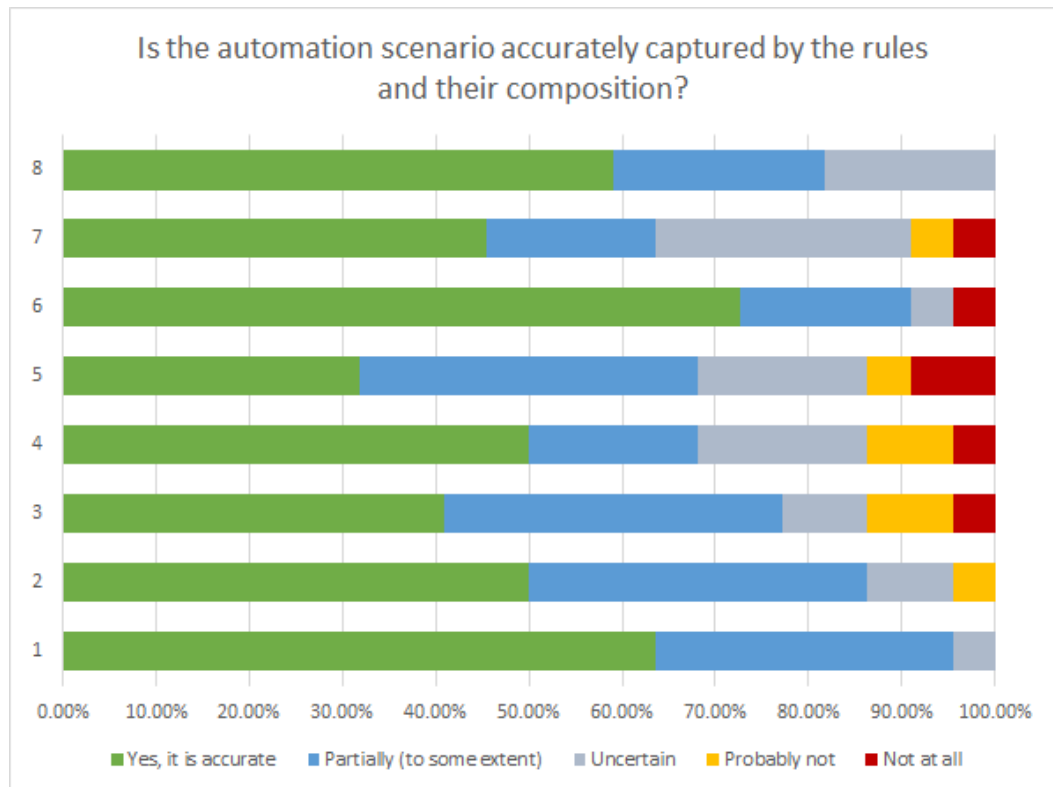


Fig. 6.10: A survey of 22 users stating how accurately the rules and composition match with the scenario description

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 6.11, nearly 60% of the users felt that they could use the tool to build advanced applications with the provided training.

As a second experiment, 6 users with varying programming skills (none to expert) from the 26 users were given 2 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 individual rules and compose them to build the application. All the users were able to design 2 correct scenarios, which took on average 6 and 8 minutes. The general consensus was that Sequence and Parallel operators were useful in designing applications. Interestingly, non-programmers were not convinced of the Choice operator, as it could introduce nondeterministic behaviour, thus giving the

³https://en.wikipedia.org/wiki/Likert_scale

After going through these examples, would you be able to build an advanced IoT scenario using the proposed tool?

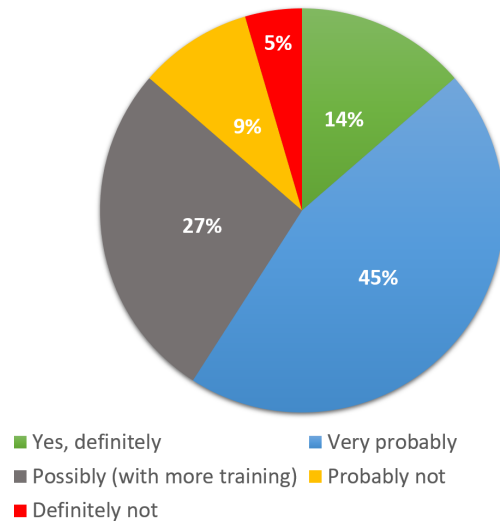


Fig. 6.11: Composition usability

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. Possibly, an indirect benefit of adding verification in automation is that it could increase the trust in IoT automation.

6.4.3 Verification Performance

We applied the design verification techniques (LNT, MCL, and CADP) on a number of typical smart home applications for validation purposes. Table 6.1 shows some of the performance numbers related to different IoT applications. The data is plotted as a graph in Figure 6.12. These applications were designed by expert users using the new UI. 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 6.1, the first column is the scenario identifier with the

Scenario	Rules	Obj.	LTS (raw)		LTS (reduced)		Time m:s
			States	Trans.	States	Trans.	
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

Tab. 6.1: Experiments on checking deadlock freedom

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 [Par81]. 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). Deadlock freedom is chosen because checking this property involves traversing the entire LTS graph and thus, provides an upper bound for time taken to verify a property. 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 6.1, scenario 7 refers to the example described in Listing 4.3 on page 54. Here, 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.*, door, sofa (motion sensor), and an air quality monitor, whereas in Scenario 6, the rules listen to a greater number of events.

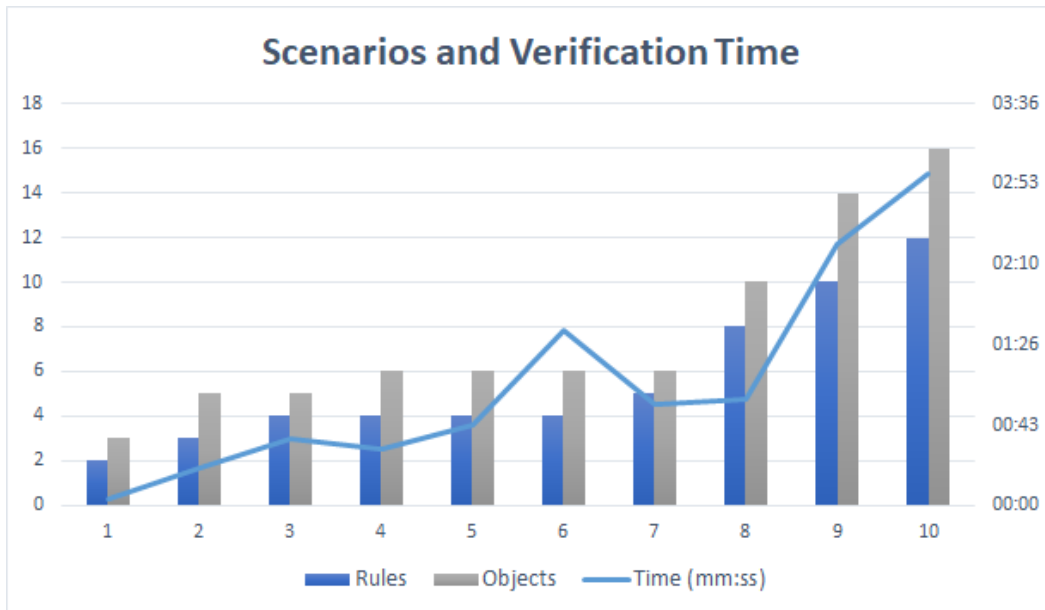


Fig. 6.12: Scenarios and verification Time

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 [HC15; Ur+14; Bri+17; MSC+19], they rarely required more than 8 rules. Our scenario with 8 rules took a reasonable amount of time (less than a minute) for verification.

The performance of the reconfiguration analysis (Maude system) was measured with different redesigns. The reconfiguration analysis took a few milliseconds for these applications. For instance, a quite large scenario with 6 rules and 10 objects, it took 6ms for the conservative check. This is explained by the fact that the check does not execute the composition, enumerating all possible states like in the LNT encoding. Instead, it syntactically compares the traces from the global state. From a user perspective, considering that we are dealing with web applications, the results are almost instantaneous.

6.4.4 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 configuration involving

some of the objects is shown in Figure 6.13. These objects were used to build smart home rules.



Fig. 6.13: Smart home setup for experiments

The set of rules were composed to build different compositions (some of them have been listed in Table 6.1 on page 109). In some scenarios, we substituted the devices with the devices we had, to mimic equivalent behaviour, otherwise we defined virtual objects using the Mozilla API. For example, intrusion alarm was implemented through connected speakers. Further, 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. In situations where it was impossible to modify the environment (e.g., humidity), we used the WoT APIs to set the values. We validated the deployment engine by verifying the devices whether they followed the composition language semantics.

Once the deployment was validated, the deployed applications were reconfigured using the reconfiguration UI. Since the reconfiguration UI is a replica of the UI for designing the applications, we did not focus on the usability aspects of the interface.

Our earlier experiments indicated that users were able build scenarios using the proposed UI.

Finally, whenever seamless reconfiguration was possible, we deployed these applications and verified that the reconfiguration manager works as expected, i.e., the existing states of the devices remained consistent after deployment.

6.5 Discussion

In this chapter, we presented a tool for composition, deployment and reconfiguration of IoT applications. The tool implements our proposals and validates on real-world applications. The experiments indicate that the tool can be used by the targeted end-users. Regarding the design verification performance, scalability is a valid concern but since the verification happens at design time users will be more willing to spend time checking the composition rather than deploying it without checks and having problems during the execution. In contrast, the reconfiguration check is done when an application is already running and therefore users may expect a quick result. This is taken care by the Maude encoding which returns the results in a few milliseconds even for large compositions.

” *The end of a melody is not its goal: but nonetheless, had the melody not reached its end it would not have reached its goal either. A parable.*

— **Friedrich Nietzsche**
German philosopher

This thesis presents a work that leverages formal methods for the development of IoT applications to ensure they will function as intended upon deployment. The work presents a solution that not only supports the design to deployment of the applications, but also their reconfiguration. Here we summarize the contributions and take a look at the possible avenues for extending this work.

7.1 Models for the IoT

The first step in guaranteeing that the designed applications work as expected lies in knowing how the objects behave and how they interact with other objects and the environment. In this context, the work 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. The model is derived from the TD to make the approach relevant to a large number of devices. However, it can be based on any other standard describing the objects in IoT, provided the right transformation is in place.

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. The ECA model was chosen because it is very popular in the smart home automation where an automation scenario (composition of objects) involves a rule in the form of IF-event-THEN-action.

Further, the work models IoT applications as a set of objects whose interactions are defined by ECA rules. The interaction model is based on the Mozilla WoT specification, a concrete implementation complementary to the W3C's WoT specification. Specifically, a gateway interaction pattern where the objects communicate through a gateway is modelled.

The models of the objects and their composition are specified in two different specification languages. First, a specification in LNT is defined. LNT is a formal specification language with process algebraic features which is the input of CADP toolbox. This specification serves as the basis for verifying properties during the initial design of the application. Another specification in Maude rewriting logic is defined to support verification during the reconfiguration of a deployed IoT application. To the best of our knowledge, our work is one of the first formal specifications based on the W3C's WoT standard.

7.2 Verification for the IoT

The next step in establishing correctness of the designed application is by verifying that the application satisfies certain properties. The work identifies two sets of properties that can be checked on IoT applications: i) a set of properties that check if the initial design of the application is satisfactory. ii) a set of properties to qualitatively compare a newly reconfigured design with the existing configuration of the application.

The properties that are checked during the initial design of the application can be categorized into generic and application properties. The generic properties identified are deadlock freedom, completeness, and detection of spurious events. IoT objects being reactive and permissive, the notion of deadlock is not obvious, however upon close inspection of the behaviour of the objects especially ones with software, deadlocking situations are very much possible in an application. Verification of these deadlocks is achieved by a specific encoding of the objects and rules. Completeness properties check the reachability of the rules in the composition expression. Spurious

event detection is a property specific to the gateway interaction pattern. It tracks the events that occur infinitely often but are never consumed.

The models can 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 describing the composition as a probabilistic transition system (i.e., decorating the LTS transitions with probabilities).

Unlike in an industrial automation, consumer IoT applications are not designed once for all. Typically, the applications are redesigned more often than in the industrial setup. To support such reconfigurations, our work identified three different properties that compare the behaviour of the existing application with the new application. These properties also indicate whether the new application can be deployed seamlessly, without having to restart the entire application. Seamless reconfiguration property checks whether the current state of the application is reachable in the newly redesigned application. Conservative reconfiguration property checks whether the current behaviour of the application is preserved in the new application. Impactful reconfiguration property focuses on the new application where it checks whether all the newly added features are reachable, i.e., to be able to have an impact on the application.

It is worth noting that all the property verifications happen during the (re)design of the application and therefore the verification does not have to provide results in near real time.

7.3 Tool Implementation

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 rather than building a new implementation, 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 abstracts the formal verification from end-users by transforming the JSON composition to specifications in LNT and Maude. It integrates with the verification

tools provided by CADP and Maude to carry out formal analysis. Here the user input is only required for the design of the composition, all the other steps being automated.

The tool support is provided not only for the design and verification but also for the deployment and reconfiguration of the IoT applications. The deployment of the original configuration and the reconfiguration uses an execution engine that follows the composition language semantics. The states of the objects are manipulated using the WebThings API.

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 a proof-of-concept implementation of proposals, it is found to be more expressive than existing UI-based ECA programming tools.

7.4 Future Work

The proposals made in this work have scope for improvements and extensions. Here we mention a few of them that we plan to do in the near-term, as well as some long-term projects. We have listed these works under three categories: i) the verification improvements that can be accomplished in the short to medium term, ii) quantitative analysis of WoT applications, and iii) proposals pertaining to the WoT implementation and application domain.

Verification Improvements

There is an immediate scope for improving usability by providing application specific MCL formulas as a graphical input in form of patterns, for instance taking inspiration from the temporal property patterns proposed in [DAC98]. This is more user-friendly as it eliminates the need for users to write the application specific properties in MCL. This would only require an extension at the user interface level.

Another area for improvement is the way the result of verification is presented to the end-user. Currently, we present the result of verification as a Boolean response with a textual description of the property violation in case of a false response. We could instead display the counterexample LTS to show the source of property violation.

The next logical extension concerning formal analysis would be to present the users the solutions to overcome the property violations. The solutions can be in the form of suggestions to replace the devices in the composition or to modify the rules or the composition expression so that the composition complies with the property check.

Currently, we do not have mechanisms to inform users of possible feature interactions in the smart home setup. It would be useful if users are aware of the interactions when they design a composition. This can be implemented using static analysis techniques where the interactions are automatically detected whenever the composition is modified in the interface [JGC05].

Concerning the fairness of execution with respect to the environment, we can elaborate further on correctness properties relying on classical fairness notions (weak, strong) [PP18] which can be used to identify whether the application can still progress under an unfair environment. This could be useful as IoT applications are typically subject to ever changing environment.

Quantitative Analysis

Since we analyse both design and reconfiguration, computing a quantitative measure like Mean Time to Failure (MTTF) in a composition could be a useful metric. It would allow users to plan their reconfigurations in a timely manner. The lifetime and failure rate of the device can be modelled as Interactive Markov Chains (IMCs) [Her02]. IMCs are supported by the CADP toolbox. At the time of writing, we have developed a model in LNT which specifies smart home scenarios as IMCs.

Going further, more quantitative analysis can be built into the framework. 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.

In terms of reconfiguration, the current checks are a qualitative comparison focusing on the behaviour. An extension to this work would be to support quantitative analysis of current and new configurations. This work can be combined with the quantitative analysis extensions planned for the design of the applications.

Application Domains

The current formal specification is based on the Mozilla WebThings implementation of WoT. It could be interesting to port this specification to another WoT implementation such as ThingWeb [Thi20]. In this case, we would still have the same

behavioural model, but it would have a slightly different interaction model. The specification of another implementation may lead to a different set of verification properties.

WoT standard is proposed to be applied in various domains such as industrial manufacturing, automobiles, etc. A 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 (e.g., usage of compositional verification techniques [GLM15]) to make the current proposals scalable.

Bibliography

- [Alu+16] Rajeev Alur, Emery Berger, Ann W Drobnis, et al. “Systems computing challenges in the Internet of Things”. In: *arXiv preprint arXiv:1604.02980* (2016) (cit. on p. 5).
- [BN99] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge university press, 1999 (cit. on p. 3).
- [BPS01] Jan A Bergstra, Alban Ponse, and Scott A Smolka. *Handbook of process algebra*. Elsevier, 2001 (cit. on pp. 3, 39).
- [Ber+19] Imad Berrouyne, Mehdi Adda, Jean-Marie Mottu, Jean-Claude Royer, and Massimo Tisi. “CyprIoT: framework for modelling and controlling network-based IoT applications”. In: *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. 2019, pp. 832–841 (cit. on p. 18).
- [Bha18] Sravani Bhattacharjee. *Practical Industrial Internet of Things security: A practitioner’s guide to securing connected industries*. Packt Publishing Ltd, 2018 (cit. on p. 2).
- [Bou+14] Mathieu Boussard, Dinh Thai Bui, Richard Douville, et al. “The Majord’Home: a SDN approach to let ISPs manage and extend their customers’ home networks”. In: *10th International conference on network and service management (CNSM) and workshop*. IEEE. 2014, pp. 430–433 (cit. on p. 12).
- [Bri+17] Julia Brich, Marcel Walch, Michael Rietzler, Michael Weber, and Florian Schaub. “Exploring end user programming needs in home automation”. In: *ACM Transactions on Computer-Human Interaction (TOCHI)* 24.2 (2017), pp. 1–35 (cit. on pp. 10, 53, 110).
- [BHR84] Stephen D Brookes, Charles AR Hoare, and Andrew W Roscoe. “A theory of communicating sequential processes”. In: *Journal of the ACM (JACM)* 31.3 (1984), pp. 560–599 (cit. on p. 36).
- [Bro+18] Nadja Brouns, Samir Tata, Heiko Ludwig, E Serral Asensio, and Paul Grefen. “Modeling iot-aware business processes-a state of the art report”. In: *arXiv preprint arXiv:1811.00652* (2018) (cit. on p. 17).
- [Buc+17] Antonio Bucchiarone, Annapaola Marconi, Marco Pistore, and Heorhi Raik. “A context-aware framework for dynamic composition of process fragments in the internet of services”. In: *J. Internet Serv. Appl.* 8.1 (2017), 6:1–6:23 (cit. on p. 16).
- [CM94] Sharma Chakravarthy and Deepak Mishra. “Snoop: An expressive event specification language for active databases”. In: *Data & Knowledge Engineering* 14.1 (1994), pp. 1–26 (cit. on p. 19).

- [Cha+18] David Champelovier, Xavier Clerc, Hubert Garavel, et al. *Reference Manual of the LNT to LOTOS Translator*. 2018 (cit. on pp. 9, 33, 63, 102).
- [Che+18] Yongyang Cheng, Shuai Zhao, Bo Cheng, et al. “Modeling and Optimization for Collaborative Business Process Towards IoT Applications”. In: *Mobile Information Systems 2018* (2018) (cit. on p. 17).
- [CW15] Hsiao-Hsien Chiu and Ming-Shi Wang. “Extending event elements of business process model for internet of things”. In: *2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing*. IEEE. 2015, pp. 783–788 (cit. on p. 17).
- [Cla+07] Manuel Clavel, Francisco Durán, Steven Eker, et al. *All About Maude-A High-Performance Logical Framework: How to Specify, Program, and Verify Systems in Rewriting Logic*. Vol. 4350. Springer, 2007 (cit. on pp. 9, 42, 44).
- [CS01] Rance Cleaveland and Oleg Sokolsky. “Equivalence and preorder checking for finite-state systems”. In: *Handbook of Process Algebra*. Elsevier, 2001, pp. 391–424 (cit. on p. 3).
- [Com20] openHAB Community. *openHAB: a vendor and technology agnostic open source automation software for your home*. 2020. URL: <https://www.openhab.org/> (visited on May 16, 2020) (cit. on pp. 7, 27).
- [Con20] Convecs. *CADP Online Manual Pages*. 2020. URL: <https://cadp.inria.fr/man/#tools> (visited on Aug. 2, 2020) (cit. on p. 102).
- [CBP12] Javier Cubo, Antonio Brogi, and Ernesto Pimentel. “Behaviour-aware compositions of things”. In: *2012 IEEE International Conference on Green Computing and Communications*. IEEE. 2012, pp. 1–8 (cit. on p. 18).
- [DB08] Leonardo De Moura and Nikolaj Bjørner. “Z3: An efficient SMT solver”. In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340 (cit. on p. 21).
- [DC15] Luigi De Russis and Fulvio Corno. “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*. 2015, pp. 2109–2114 (cit. on p. 20).
- [DGG95] Klaus R Dittrich, Stella Gatziau, and Andreas Geppert. “The active database management system manifesto: A rulebase of ADBMS features”. In: *International Workshop on Rules in Database Systems*. Springer. 1995, pp. 1–17 (cit. on p. 45).
- [DM17] Dulce Domingos and Francisco Martins. “Using BPMN to model Internet of Things behavior within business process”. In: *International Journal of Information Systems and Project Management* 5.4 (2017), pp. 39–51 (cit. on p. 16).

- [DSK19] Francisco Durán, Gwen Salaün, and Ajay Krishna. “Automated Composition, Analysis and Deployment of IoT Applications”. In: *Software Technology: Methods and Tools - 51st International Conference, TOOLS 2019, Innopolis, Russia, October 15-17, 2019, Proceedings*. Vol. 11771. Lecture Notes in Computer Science. Springer, 2019, pp. 252–268 (cit. on p. 13).
- [DAC98] Matthew B Dwyer, George S Avrunin, and James C Corbett. “Property specification patterns for finite-state verification”. In: *Proceedings of the second workshop on Formal methods in software practice*. 1998, pp. 7–15 (cit. on pp. 23, 116).
- [Eri09] AnnMarie Ericsson. “Enabling tool support for formal analysis of eca rules”. PhD thesis. Linköping University Electronic Press, 2009 (cit. on p. 22).
- [Fel+17] Alexander Felfernig, Andreas Falkner, Seda Polat Erdeniz, Christoph Uran, and Paolo Azzoni. “Asp-based knowledge representations for iot configuration scenarios”. In: *19th International Configuration Workshop*. 2017, p. 62 (cit. on pp. 15, 24).
- [Fou20] OpenJS Foundation. *Node-RED: Low-code programming for event-driven applications*. 2020. URL: <https://nodered.org/> (visited on May 16, 2020) (cit. on pp. 10, 26).
- [Fra20a] Ben Francis. *Web of Things Integration Patterns*. 2020. URL: https://docs.google.com/document/d/1H3coHbb3Bwd02_NJi4KEB0NByUkq92_HsTk1IpfmACY/edit#heading=h.emba0uow6hmb (visited on May 16, 2020) (cit. on p. 47).
- [Fra20b] Ben Francis. *Web Thing API*. 2020. URL: <https://iot.mozilla.org/wot/> (visited on May 16, 2020) (cit. on p. 48).
- [FMS14] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A practical guide to SysML: the systems modeling language*. Morgan Kaufmann, 2014 (cit. on p. 24).
- [Fuq+15] A. Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash. “Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications”. In: *IEEE Communications Surveys Tutorials* 17.4 (2015), pp. 2347–2376 (cit. on p. 4).
- [Gan+15] Kahina Gani, Marinette Bouet, Michel Schneider, and Farouk Toumani. “Using timed automata framework for modeling home care plans”. In: *2015 International Conference on Service Science (ICSS)*. IEEE. 2015, pp. 1–8 (cit. on p. 18).
- [GL01] Hubert Garavel and Frédéric Lang. “SVL: a scripting language for compositional verification”. In: *International Conference on Formal Techniques for Networked and Distributed Systems*. Springer. 2001, pp. 377–392 (cit. on p. 102).
- [GLM15] Hubert Garavel, Frédéric Lang, and Radu Mateescu. “Compositional verification of asynchronous concurrent systems using CADP”. In: *Acta Informatica* 52.4-5 (2015), pp. 337–392 (cit. on p. 118).
- [Gar+13] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. “CADP 2011: a toolbox for the construction and analysis of distributed processes”. In: *International Journal on Software Tools for Technology Transfer* 15.2 (2013), pp. 89–107 (cit. on pp. 9, 33, 99, 101).

- [GLS17] Hubert Garavel, Frédéric Lang, and Wendelin Serwe. “From LOTOS to LNT”. In: *ModelEd, TestEd, TrustEd*. Springer, 2017, pp. 3–26 (cit. on pp. 9, 33, 63).
- [Ghi+17] Giuseppe Ghiani, Marco Manca, Fabio Paternò, and Carmen Santoro. “Personalization of context-dependent applications through trigger-action rules”. In: *ACM Transactions on Computer-Human Interaction (TOCHI)* 24.2 (2017), pp. 1–33 (cit. on p. 20).
- [Gom+17] Tiago Gomes, P Lopes, J Alves, et al. “A modeling domain-specific language for IoT-enabled operating systems”. In: *IECON 2017-43rd Annual Conference of the IEEE Industrial Electronics Society*. IEEE. 2017, pp. 3945–3950 (cit. on p. 18).
- [Gre20] Andy Greenberg. “How Apple and Google Are Enabling Covid-19 Bluetooth Contact-Tracing”. In: *Wired* (2020). <https://www.wired.com/story/apple-google-bluetooth-contact-tracing-covid-19/> (cit. on p. 2).
- [GT09] Dominique Guinard and Vlad Trifa. “Towards the web of things: Web mashups for embedded devices”. In: *Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web (MEM 2009), in proceedings of WWW (International World Wide Web Conferences), Madrid, Spain*. Vol. 15. 2009, p. 8 (cit. on p. 5).
- [Har+16] Nicolas Harrant, Franck Fleurey, Brice Morin, and Knut Eilif Husa. “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*. 2016, pp. 125–135 (cit. on p. 17).
- [Her02] Holger Hermanns. “Interactive markov chains”. In: *Interactive Markov Chains*. Springer, 2002, pp. 57–88 (cit. on p. 117).
- [Hom20] HomeAssistant. *Home Assistant: Awaken your home*. 2020. URL: <https://www.home-assistant.io/> (visited on May 16, 2020) (cit. on p. 29).
- [Hor51] Alfred Horn. “On sentences which are true of direct unions of algebras”. In: *The Journal of Symbolic Logic* 16.1 (1951), pp. 14–21 (cit. on p. 43).
- [HCH19] Kai-Hsiang Hsu, Yu-Hsi Chiang, and Hsu-Chun Hsiao. “Safechain: Securing trigger-action programming from attack chains”. In: *IEEE Transactions on Information Forensics and Security* 14.10 (2019), pp. 2607–2622 (cit. on p. 22).
- [HC15] Justin Huang and Maya Cakmak. “Supporting mental model accuracy in trigger-action programming”. In: *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*. 2015, pp. 215–225 (cit. on pp. 10, 19, 20, 53, 110).
- [HLR17] Mahmoud Hussein, Shuai Li, and Ansgar Radermacher. “Model-driven Development of Adaptive IoT Systems.” In: *MODELS (Satellite Events)*. 2017, pp. 17–23 (cit. on p. 24).
- [IFT20] IFTTT. *Every thing works better together*. 2020. URL: <https://ifttt.com/> (visited on May 16, 2020) (cit. on pp. 7, 10, 25).
- [Inc20a] Apple Inc. *Workflow*. 2020. URL: <https://workflow.is/> (visited on May 16, 2020) (cit. on pp. 25, 30).

- [Inc20b] SmartThings Inc. *Smart Things: Add a little smartness to your things*. 2020. URL: <https://www.smartthings.com/> (visited on May 16, 2020) (cit. on p. 30).
- [IHS15] Malte Isberner, Falk Howar, and Bernhard Steffen. “The open-source LearnLib”. In: *International Conference on Computer Aided Verification*. Springer. 2015, pp. 487–495 (cit. on p. 52).
- [ISO01] ISO. *ISO/IEC 15437:2001 Information technology — Enhancements to LOTOS (E-LOTOS)*. 2001. URL: <https://www.iso.org/standard/27680.html> (visited on May 16, 2020) (cit. on p. 33).
- [ITU16] ITU-T. *Overview of the Internet of things*. 2016. URL: <http://handle.itu.int/11.1002/1000/11559> (visited on May 16, 2020) (cit. on p. 1).
- [JLC13] Xiaoqing Jin, Yousra Lembachar, and Gianfranco Ciardo. “Symbolic verification of ECA rules.” In: *PNSE+ ModPE 989* (2013), pp. 41–59 (cit. on p. 21).
- [JGC05] Helene Jouve, Pascale Le Gall, and Sophie Coudert. “An Automatic Off-Line Feature Interaction Detection Method by Static Analysis of Specifications”. In: *Feature Interactions in Telecommunications and Software Systems VIII, ICFI’05, 28-30 June 2005, Leicester, UK*. Ed. by Stephan Reiff-Marganiec and Mark Ryan. IOS Press, 2005, pp. 131–146 (cit. on p. 117).
- [Kri+19a] Ajay Krishna, Michel Le Pallec, Radu Mateescu, Ludovic Noirie, and Gwen Salaün. “IoT Composer: Composition and deployment of IoT applications”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE. 2019, pp. 19–22 (cit. on pp. 12, 13).
- [Kri+20] Ajay Krishna, Michel Le Pallec, Alejandro Martinez, Radu Mateescu, and Gwen Salaün. “MOZART: Design and Deployment of Advanced IoT Applications”. In: *Companion of The Web Conference 2020 (WWW), Taipei, Taiwan, April 20-24, 2020*. ACM, 2020, pp. 163–166 (cit. on pp. 10, 12).
- [Kri+19b] Ajay Krishna, Michel Le Pallec, Radu Mateescu, Ludovic Noirie, and Gwen Salaün. “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, 2019, pp. 21–30 (cit. on pp. 12, 13).
- [LBP09] Said Lankri, Pascal Berruet, and Jean-Luc Philippe. “Service reconfiguration in the danah assistive system”. In: *International Conference on Smart Homes and Health Telematics*. Springer. 2009, pp. 269–273 (cit. on p. 24).
- [LS91] Kim G Larsen and Arne Skou. “Bisimulation through probabilistic testing”. In: *Information and computation* 94.1 (1991), pp. 1–28 (cit. on p. 83).
- [Lia+16] Chieh-Jan Mike Liang, Lei Bu, Zhao Li, et al. “Systematically debugging IoT control system correctness for building automation”. In: *Proceedings of the 3rd ACM International Conference on Systems for Energy-Efficient Built Environments*. 2016, pp. 133–142 (cit. on p. 20).

- [Lia+15] Chieh-Jan Mike Liang, Börje F Karlsson, Nicholas D Lane, et al. “SIFT: building an internet of safe things”. In: *Proceedings of the 14th International Conference on Information Processing in Sensor Networks*. 2015, pp. 298–309 (cit. on pp. 20, 21).
- [LLC13] Lopez Research LLC. *An Introduction to the Internet of Things (IoT)*. 2013. URL: https://www.cisco.com/c/dam/en_us/solutions/trends/iot/introduction_to_IoT_november.pdf (visited on May 16, 2020) (cit. on p. 1).
- [MSC+19] Marco Manca, Carmen Santoro, Luca Corcella, et al. “Supporting end-user debugging of trigger-action rules for IoT applications”. In: *International Journal of Human-Computer Studies* 123 (2019), pp. 56–69 (cit. on pp. 20, 110).
- [MP90] Zohar Manna and Amir Pnueli. “A Hierarchy of Temporal Properties (Invited Paper, 1989)”. In: *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing*. PODC '90. Quebec City, Quebec, Canada: Association for Computing Machinery, 1990, pp. 377–410 (cit. on p. 40).
- [MD17] Francisco Martins and Dulce Domingos. “Modelling IoT behaviour within BPMN business processes”. In: *Procedia computer science* 121 (2017), pp. 1014–1022 (cit. on p. 16).
- [MR18] Radu Mateescu and José Ignacio Requeno. “On-the-fly model checking for extended action-based probabilistic operators”. In: *International Journal on Software Tools for Technology Transfer* 20.5 (2018), pp. 563–587 (cit. on pp. 40–42, 75, 83, 102).
- [MS03] Radu Mateescu and Mihaela Sighireanu. “Efficient on-the-fly model-checking for regular alternation-free mu-calculus”. In: *Science of Computer Programming* 46.3 (2003), pp. 255–281 (cit. on p. 75).
- [MT08] Radu Mateescu and Damien Thivolle. “A model checking language for concurrent value-passing systems”. In: *International Symposium on Formal Methods*. Springer. 2008, pp. 148–164 (cit. on pp. 9, 40, 42, 75, 102).
- [Mes92] José Meseguer. “Conditional rewriting logic as a unified model of concurrency”. In: *Theoretical computer science* 96.1 (1992), pp. 73–155 (cit. on p. 42).
- [Mes12] José Meseguer. “Twenty years of rewriting logic”. In: *The Journal of Logic and Algebraic Programming* 81.7-8 (2012), pp. 721–781 (cit. on pp. 3, 42).
- [MRM13] Sonja Meyer, Andreas Ruppen, and Carsten Magerkurth. “Internet of things-aware process modeling: integrating IoT devices as business process resources”. In: *International conference on advanced information systems engineering*. Springer. 2013, pp. 84–98 (cit. on pp. 16, 17).
- [Mic20] Microsoft. *Power Automate*. 2020 (cit. on p. 30).
- [Mil89] Robin Milner. *Communication and concurrency*. Vol. 84. Prentice hall Englewood Cliffs, 1989 (cit. on p. 36).
- [MTK18] Sachin Mittal, Wang Tsz Tam, and Chris Ko. *Internet of Things: The Pillar of Artificial Intelligence*. <https://bit.ly/3dbz0wz>. 2018 (cit. on p. 2).

- [Moz20a] Mozilla. *IoT Capability Schema*. 2020. URL: <https://iot.mozilla.org/schemas> (visited on May 16, 2020) (cit. on p. 48).
- [Moz20b] Mozilla. *WebThings: An open platform for monitoring and controlling devices over the web*. 2020. URL: <https://iot.mozilla.org/> (visited on May 16, 2020) (cit. on pp. 10, 28, 48).
- [Mus20] Computer History Museum. *A timeline of semiconductors in computers*. 2020. URL: <https://www.computerhistory.org/siliconengine/> (visited on May 16, 2020) (cit. on p. 2).
- [Nac+18] Alessandro A Nacci, Vincenzo Rana, Bharathan Balaji, et al. “BuildingRules: A Trigger-Action-Based System to Manage Complex Commercial Buildings”. In: *ACM Transactions on Cyber-Physical Systems* 2.2 (2018), pp. 1–22 (cit. on p. 21).
- [Nes20] Nest Labs, Inc. *OpenWeave: A secure and reliable communications backbone for the connected home*. 2020. URL: <https://openweave.io/> (visited on May 16, 2020) (cit. on p. 45).
- [Nor88] Donald A Norman. *The psychology of everyday things*. Basic books, 1988 (cit. on p. 48).
- [Ouc18] Samir Ouchani. “Ensuring the functional correctness of IoT through formal modeling and verification”. In: *International Conference on Model and Data Engineering*. Springer. 2018, pp. 401–417 (cit. on p. 22).
- [Par81] David Park. “Concurrency and automata on infinite sequences”. In: *Theoretical computer science*. Springer, 1981, pp. 167–183 (cit. on p. 109).
- [Par20] oneM2M Partners. *oneM2M: Standards for M2M and the Internet of Things*. 2020. URL: <https://www.onem2m.org/> (visited on May 16, 2020) (cit. on p. 5).
- [PA19] Fabio Paterno and Sadi Alawadi. “Towards Intelligent Personalization of IoT Platforms.” In: *IUI Workshops*. 2019 (cit. on p. 25).
- [PD99] Norman W Paton and Oscar Diaz. “Active database systems”. In: *ACM Computing Surveys (CSUR)* 31.1 (1999), pp. 63–103 (cit. on p. 19).
- [PP18] Nir Piterman and Amir Pnueli. “Temporal logic and fair discrete systems”. In: *Handbook of Model Checking*. Springer, 2018, pp. 27–73 (cit. on p. 117).
- [QS83] Jean-Pierre Queille and Joseph Sifakis. “Fairness and related properties in transition systems—a temporal logic to deal with fairness”. In: *Acta Informatica* 19.3 (1983), pp. 195–220 (cit. on p. 79).
- [Roj+20] Javier Rojo, Daniel Flores-Martin, Jose Garcia-Alonso, Juan M Murillo, and Javier Berrocal. “Automating the Interactions among IoT Devices using Neural Networks”. In: *2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. IEEE. 2020, pp. 1–6 (cit. on p. 24).
- [Row17] Dan Rowinski. *Consumers Care Most About Usability, Interoperability And Security Of IoT*. 2017. URL: <https://www.applause.com/blog/consumer-iot-adoption-concerns> (visited on May 16, 2020) (cit. on p. 7).

- [San13] Davide Sangiorgi. *An Introduction to Bisimulation and Coinduction*. 2013. URL: <http://cs.ioc.ee/ewscs/2013/sangiorgi/sangiorgi-ewscs13-slides.pdf> (visited on May 16, 2020) (cit. on p. 31).
- [Sch20] Schema.org. *Schema.org*. 2020. URL: <https://iot.schema.org/> (visited on May 16, 2020) (cit. on p. 48).
- [See+19] Jan Seeger, Rohit A Deshmukh, Vasil Sarafov, and Arne Bröring. “Dynamic IoT Choreographies”. In: *IEEE Pervasive Computing* 18.1 (2019), pp. 19–27 (cit. on p. 23).
- [Sta+13] International Organization for Standardization/International Electrotechnical Commission et al. “ISO/IEC 19510: 2013”. In: *Information Technology–Object Management Group Business Process Model and Notation* (2013) (cit. on p. 16).
- [TKJ17] Samir Tata, Kais Klai, and Rakesh Jain. “Formal Model and Method to Decompose Process-Aware IoT Applications”. In: *On the Move to Meaningful Internet Systems. OTM 2017 Conferences - Confederated International Conferences: CoopIS, C&TC, and ODBASE 2017, Rhodes, Greece, October 23-27, 2017, Proceedings, Part I*. Vol. 10573. Lecture Notes in Computer Science. Springer, 2017, pp. 663–680 (cit. on p. 17).
- [Thi20] ThingWeb. *thingweb: A web of things implementation*. 2020. URL: <https://www.thingweb.io/> (visited on May 16, 2020) (cit. on p. 117).
- [Uni70] Carnegie Mellon University. *The "Only" Coke Machine on the Internet*. 1970. URL: https://www.cs.cmu.edu/~coke/history_long.txt (visited on May 16, 2020) (cit. on p. 2).
- [Ur+14] Blase Ur, Elyse McManus, Melwyn Pak Yong Ho, and Michael L Littman. “Practical trigger-action programming in the smart home”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 2014, pp. 803–812 (cit. on pp. 10, 110).
- [Ur+16] Blase Ur, Melwyn Pak Yong Ho, Stephen Brawner, et al. “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*. 2016, pp. 3227–3231 (cit. on pp. 19, 20, 53).
- [Van+17] Claudia Vannucchi, Michelangelo Diamanti, Gianmarco Mazzante, et al. “virony: A tool for analysis and verification of ECA rules in intelligent environments”. In: *2017 International Conference on Intelligent Environments (IE)*. IEEE. 2017, pp. 92–99 (cit. on p. 21).
- [W3C20a] W3C. *Web of Things (WoT) Architecture*. 2020. URL: <https://www.w3.org/TR/wot-architecture/> (visited on May 16, 2020) (cit. on pp. 46, 55).
- [W3C20b] W3C. *Web of Things (WoT) Thing Description*. 2020. URL: <https://www.w3.org/TR/wot-thing-description/> (visited on May 16, 2020) (cit. on pp. 9, 46).
- [W3C20c] W3C. *Web of Things at W3C*. 2020. URL: <https://www.w3.org/WoT/> (visited on May 16, 2020) (cit. on pp. 5, 45, 46).

- [YS20] Robin Young and Inder Singh. *Smart Thermometers Play Key Role In Tracking, Preventing COVID-19 Spread*. 2020. URL: <https://www.wbur.org/hereandnow/2020/04/09/smart-thermometers-tracking-coronavirus> (visited on May 16, 2020) (cit. on p. 2).

List of Figures

1.1	IoT standards and protocols	4
2.1	Node-RED Philips Hue light flow	27
3.1	Nondeterministic choice and interleaving LTS	37
3.2	Alarm device behaviour	39
3.3	Model checking using a model checker	40
4.1	Interaction between a thing and a consumer through Thing Description	47
4.2	Integration patterns [Fra20a]	47
4.3	Labelled transition system of a motion sensor	52
4.4	WoT Gateway Communication	55
4.5	Workflow representing retirement home rules	57
4.6	<i>exec</i> Function on expression $R1; (R2 (R3 + R4)); R5$	59
4.7	LTS denoting the semantics of the composition expression $R1; (R2 (R3 + R4)); R5$	60
4.8	Representation of a WoT application	60
5.1	LTS of an intrusion alarm	78
5.2	Deadlock diagnostic LTS	78
5.3	Spurious event in an application	80
5.4	Compositions illustrating reconfiguration	91
6.1	WebThings monitoring UI	94
6.2	WebThings rules UI	95
6.3	Mozart: Design and deployment of IoT applications	97
6.4	Composition UI	98
6.5	Mozart UI screenshots	99
6.6	Reconfiguration workflow	100
6.7	Mozart: Components and technology stack	102
6.8	Plot showing the scenarios with presence of bars indicating the possibility of implementing the scenarios.	105

6.9	Screenshot of the training and feedback screen used for Mozart tool users.	106
6.10	A survey of 22 users stating how accurately the rules and composition match with the scenario description	107
6.11	Composition usability	108
6.12	Scenarios and verification Time	110
6.13	Smart home setup for experiments	111

List of Tables

4.1	Rule activation patterns	56
4.2	Rule execution patterns	57
4.3	Composition language to LNT transformation patterns	67
6.1	Experiments on checking deadlock freedom	109

List of Listings

2.1	openHAB Rule Format	28
3.1	Fibonacci sequence in Maude	44
4.1	JSON Thing Description of a Connected Light	49
4.2	JSON Thing Description of Hue motion sensor	51
4.3	Retirement home rules	54
4.4	Outline of the LNT process of an IoT object	64
4.5	LNT process specifying a motion sensor	65
4.6	LNT process specifying rule R2	66
4.7	LNT process specifying the composition of the retirement home application	67
4.8	LNT processes specifying the listener of the retirement home application	68
4.9	LNT processes specifying the environment of the retirement home application	69
4.10	Outline of the main process	70
4.11	Composition specification in Maude	72
4.12	Rule specification in Maude	73
5.1	Seamless reconfiguration in Maude	86
5.2	Conservative property in Maude	88
5.3	Impactful property in Maude	90
6.1	Maude Freemaker template	103

