Designing IoT Systems: Patterns and Managerial Conflicts

Leila Fatmasari Rahman, Tanir Ozcelebi, Johan J. Lukkien
Dept. of Mathematics and Computer Science
Eindhoven University of Technology
P.O.Box 513, 5600 MB, Eindhoven, The Netherlands
Email: {l.f.rahman, t.ozcelebi, j.j.lukkien}@tue.nl

***Abstract*—The first step in a system design process is to perform domain analysis. This entails acquiring stakeholder concerns throughout the life cycle of the system. The second step is to design solutions addressing those stakeholder concerns. This entails applying patterns for solving known, recurring problems. For these there are architecture patterns and design patterns for architecture design and detailed design respectively. For Internet of Things (IoT) systems such patterns are hardly defined yet since experience is just evolving. In this paper, we propose our definition of an IoT pattern along with its formal specification, explained by a running example. IoT systems are characterized by the variety of stakeholders involved throughout their life cycle, therefore our pattern specification includes means for identifying possible conflicts between these stakeholders.**

I. Introduction

The design process of a system entails translating a design problem into a solution blueprint. This involves solving the issues required for realizing functional requirements while satisfying certain quality constraints. One strategy is to apply existing "patterns", i.e. typical known solutions for recurring problems. Specifically, a pattern consists of a coherent set of design decisions which are proven to solve common problems. Examples of patterns are found in the architecture and include Client-Server, Peer-to-Peer, Representational State Transfer (REST) and Publish-Subscribe. Applying patterns simplifies the design process thanks to the provided structure, typical behavior and guidance through terminology. Thanks to patterns, system designers do not need to solve all issues from scratch and they can avoid mistakes in their designs to a large degree. The use of patterns also allows system designers to compare a set of alternative solutions by analyzing the properties of each pattern.

In the Internet of Things (IoT) domain the quality constraints for design include aspects such as performance, reliability, scalability, reusability, modifiability and interoperability. IoT systems are complex due to distributed services on many IoT devices collaboratively fulfilling the goals of IoT applications. What makes IoT systems even more complex is the large number of stakeholders involved throughout the life cycle of IoT systems. For example, an IoT device [1] may be developed by a device company, produced by a manufacturing company, installed by the device owner or by an installation company and commissioned and maintained by the device owner or a service company. An IoT service [1] and IoT application [1] may be developed and maintained by third party developers using software development kits (SDK) provided by a device company and/or an IoT platform provider, deployed on IoT devices inside the local network of the device owner, using deployment tools provided by the IoT platform provider. Data generated by a device that belong to a device owner may also cross the Internet and get stored in the server of a platform provider or a device company, which then have physical access to the data. It is likely that there are conflicts among these many stakeholders in managing the life cycles of the IoT elements [1] and the data. For example, conflicts regarding data privacy, data ownership and the desired system behavior. Therefore, manageability is an important extra functional properties in IoT. Manageability refers to the condition where management of data, life cycle [1] and the corresponding conflicts among the (managing) stakeholders is in place, allowing the system to work properly.

Depending on the type of applications and the specific emphasis on certain extra functional properties, IoT system solutions differ from one another. For an IoT system designer who is looking for the right solution for specific IoT use cases and the requirements derived from those use cases, IoT patterns would represent the options to choose from. They should also help the designer identify possible managerial conflicts in systems instantiating the patterns. However, an IoT pattern is hardly defined yet in the literature. In this paper we suggest answers to the following questions. 1) What exactly is an IoT pattern? 2) How do we specify an IoT pattern? 3) What is an example of an IoT pattern? Section 2 gives an overview of the literature work on this topic. Section 3 introduces roles and stakeholders in an IoT system and formally specifies an IoT pattern. Section 4 provides an example of an IoT pattern. Section 5 shows how an IoT pattern is applied. Section 6 concludes the paper.

II. State of the Art

Patterns date back to 1977 when Christopher Alexander published *A Pattern Language: Towns, Buildings, Construction* [2]. Alexander introduced patterns as solutions to common problems, in order to enable everyone, not only professionals, to design a building such as a house, a school or a shop, using patterns documented in the book. Patterns were introduced in software by the 1995 book *Design Patterns: Elements* *of Reusable Object-Oriented Software* [3]. The book defined (object oriented) design patterns and provided a repository of examples, aiming at quality properties such as reusability and separation of concerns. Buschmann et al [4] introduced (software) architecture patterns, focusing on system organization and typical behavior. System and software architecture design concerns the realization of extra functional properties such as performance, reliability, scalability and interoperability of a software system through coordination of distributed software components [5].

Patterns, in general, refer to a generic structure and behavior. For (software) design patterns, the structure is often specified by a class model and the behavior by interaction diagrams. The instantiation of a design pattern results in a code structure, and some code can be generated automatically. For architecture patterns the structure and behavior is at a more abstract system level. Instantiating an architecture pattern is through choices, use cases and terminology introduction, and yields a (partial) system architecture. When these use cases are taken from a domain rather than from a concrete system design we obtain a pattern in that domain. IoT represents a domain, a context say for applications. In our earlier work we have shown the importance of life cycles of its elements and related use cases, while we also introduced and classified those elements [1]. Combining these use cases with architecture patterns yields patterns for IoT systems.

In the literature, we see several examples of IoT patterns. Reinfurt et al [6] proposed five IoT patterns, namely *Device Gateway*, *Device Shadow*, *Rules Engine*, *Device Wakeup Trigger* and *Remote Lock and Wipe*. These patterns represent typical solutions for different aspects of their identified core components in IoT, i.e a device and a back-end server [6]. For example, *Device Gateway* and *Device Shadow* are solutions for communication between devices and a back-end server, while *Rules Engine* is a solution for the processing of control in the back-end server. We classify these patterns as architecture patterns as they consist of a set of coherent design decisions on distributed software components that address the extra functional properties interoperability [7], availability and modifiability [7] respectively. Reinfurt et al proposed more IoT patterns in their other works, such as: patterns for bootstrapping and registration of devices [8]; and patterns for energy supply, operation mode and sensing mode of devices [9]. The device bootstrapping and registration patterns [8] consist of a set of design decisions on distributed software components that address security and device manageability and therefore we classify them as architecture patterns. However, the latter patterns [9] are more about design choices regarding energy supply, operation mode and sensing mode for IoT devices. Qanbari et al [10] proposed four IoT design patterns for edge applications, namely: *Edge Provisioning*, *Edge Code Deployment*, *Edge Orchestration* and *Edge Diameter of Things*. In our view, these patterns are better classified as concrete (partial) system architecture due to the detailed and specific choices presented in their solutions, and due to the absence of system examples from which these patterns are abstracted from.

Based on the IoT pattern examples we found in the literature, we learned that an IoT pattern could have multiple meanings and that a generic definition of an IoT pattern does not exist yet. As an IoT system is a complex system involving many elements and stakeholders, we believe that applying some formalism in an IoT pattern specification will promote clarity and precision to the guiding concepts and terminology, which in turn improves understanding, communication and analysis during the design process. For example, we can use the formalism to model relationships between life cycle use cases [1], roles, stakeholders and managerial controls. This modeling can help a designer analyze possible managerial conflicts in system architectures instantiating the pattern, as discussed in the next section.

III. IoT Pattern Specification

Architecture patterns are particularly useful for designing systems where software components are distributed in different devices working together to realize a common goal. IoT systems are a type of distributed system, but with particular life cycles. In our earlier work, we define the life cycles of three main elements in IoT systems, namely IoT device, IoT service and IoT application which we refer as the generic IoT life cycle model [1]. The life cycles stages of these

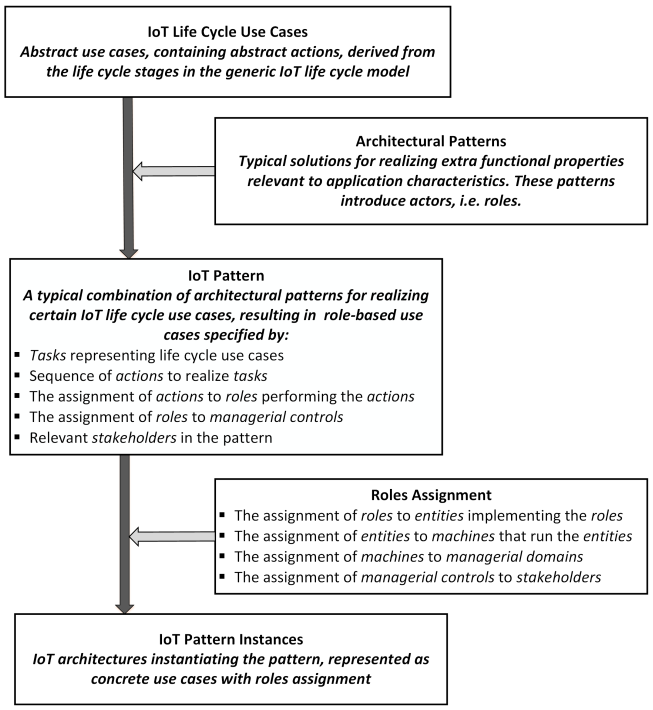
IoT Life Cycle Use Cases
Abstract use cases, containing abstract actions, derived from the life cycle stages in the generic IoT life cycle model
Architectural Patterns
Typical solutions for realizing extra functional properties relevant to application characteristics. These patterns introduce actors, i.e. roles.
IoT Pattern
A typical combination of architectural patterns for realizing certain IoT life cycle use cases, resulting in role-based use cases specified by:
Tasks representing life cycle use cases
Sequence of actions to realize tasks
The assignment of actions to roles performing the actions
The assignment of roles to managerial controls
Relevant stakeholders in the pattern
Roles Assignment
The assignment of roles to entities implementing the roles
The assignment of entities to machines that run the entities
The assignment of machines to managerial domains
The assignment of managerial controls to stakeholders
IoT Pattern Instances
IoT architectures instantiating the pattern, represented as concrete use cases with roles assignment

Figure 1. A generic definition of an IoT pattern. An IoT pattern turns abstract IoT life cycle use cases into role-based use cases by combining architecture patterns. An IoT pattern instance further turns role-based use cases into concrete use cases with roles assignment.

elements represent abstract use cases in IoT systems, such as: development, installation, commissioning, operation, update and decommissioning of IoT devices; and development, deployment, execution, reconfiguration and termination of IoT services and applications [1]. We refer to these abstract use cases as IoT life cycle use cases. Each of these use cases consists of abstract actions that are represented as activities inside a life cycle stage of an IoT element [1]. An abstract life cycle use case can be instantiated into multiple concrete use cases, depending on the type of applications and their emphasis on certain extra functional properties.

From our survey on existing IoT systems and frameworks [11], we learned that solutions to these life cycle use cases are built by a combination of architecture patterns. We also learned that while some of these systems and frameworks differ in their deployment and communication protocol choices, they can resemble each other in structure, i.e. the architecture patterns they combine, and in behavior, i.e. the actions they take, when realizing certain life cycle use cases. When we find IoT solutions with similar structure and behavior, we abstract them into an IoT pattern. The architecture patterns introduce concrete actions for the life cycle use cases as well as actors of these actions which we then refer to as roles. Therefore, we define an IoT pattern as *a typical combination of architecture patterns for realizing certain IoT life cycle use cases, resulting in role-based use cases*, as depicted in Figure 1. We clarify this definition with running examples in section IV and section V.

We now propose our specification of an IoT pattern by introducing the following concepts and relations (see Figure 2). A life cycle use case of an IoT system is named a *task* that the system needs to do. A task is a sequence of *actions* to be executed by different *roles*. An action represents a concrete step in a concrete use case. A role is implemented by an *entity* which is a software element that runs on a *machine*. Both entities and machines are controlled by stakeholders. A managerial domain is the control span of a (managing) stakeholder.

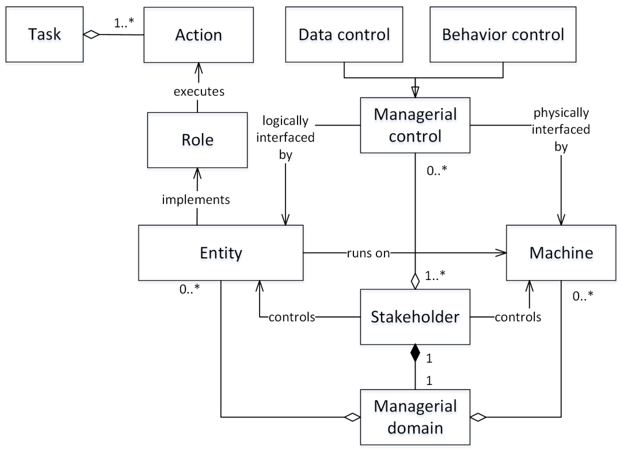

Figure 2. A conceptual model of the elements of an IoT pattern, showing concepts and relations for modeling an IoT pattern.

In our attempt to model managerial conflicts, we start by defining two types of managerial control: managerial control over data, or *data control*, and managerial control over system behavior, or *behavior control*. System behavior pertains to the behavior of devices and software elements in the system during normal operation. Managerial controls, both data and behavior controls, are controls that can cause managerial conflicts.

Data control can be categorized into physical and logical data control. Physical data control refers to the ability to alter or destroy physical data, interfaced by physical access and control of the machines generating or storing the data. A stakeholder who has controls over such machines is said to have physical data control, usually derived from ownership of the machines. Logical data control refers to the ability to use meaningful data through query operations, interfaced by entities that implement data query roles. A stakeholder who has control over such entity is said to have logical data control. In IoT, data are mainly generated by the *things*, i.e. IoT devices embedded with sensors and/or actuators, which are controlled by the *things'* owner. However, these data may be stored in a back-end server controlled by a cloud provider. This means that the cloud provider now has physical control over the data and may or may not have logical control over data. If the stored data is encrypted and only the *things'* owner has the key to decrypt the data, the *things'* owner has logical control over the data while the cloud provider only has physical control over the data.

For modeling managerial conflicts related to data control, we introduce the concept *managerial domain crossing*. A managerial domain crossing occurs when data leaves a managerial domain and enters another. A managerial domain crossing can cause managerial conflicts as it adds another stakeholder to the set of stakeholders having managerial control over data. In our previous example, data leaves the managerial domain of the *things'* owner and enters the managerial domain of cloud provider, assigning physical data control to the cloud provider. A managerial domain crossing requires trust establishment between the entities that send and receive the data, for example through authentication, authorization, secured communication and privacy agreements.

The other type of managerial control, i.e. behavior control, can be categorized into physical and logical behavior control. Physical behavior control refers to the ability to define system behavior through physical access to a machine, which is installed in the vicinity, that can act as a behavior definer. A stakeholder who has control over such machine is said to have physical behavior control. For example, through the Nest smart thermostat [12], home members and their guests can physically adjust the desired behavior of the heater based on certain temperature values. On the other hand, logical behavior control refers to the ability to define system behavior through an entity that acts as a behavior definer. A stakeholder who has control over such entity is said to have logical behavior control. One example of logical behavior control is by creating or updating rules through an app, such as the Philips Hue app [13] that can be used to define the behavior of Philips Hue lights based on certain presence sensor values. The app then deploys the new behavior to the Philips Hue bridge via the network. Another example is by programming a control logic entity using an integrated development environment (IDE), and deploying the control logic entity to machines in the system via the network such as described in the *Open Architectures for Inteligent Solid State Lighting Systems (OpenAIS)* [14], a reference architecture for inteligent office lighting which promotes decentralized control deployment [11]. In these two examples, Philips Hue app and IDE are the entities that act as a behavior definer. When multiple stakeholders have managerial control over system behavior, whether it is physical or logical, conflicts can occur between the defined behaviors.

Through this modeling we can discriminate different assignments of roles and stakeholders and we can see whether data remains in a single managerial domain or leaves it, indicating a managerial domain crossing. We can also see whether a role that manifests a behavior control is assigned to many stakeholders, leading to managerial conflicts between the defined behaviors. The designer can then decide on policies and corresponding mechanisms to resolve these conflicts.

We can now formally specify an IoT pattern and its instance as follow. An IoT pattern solves a set of tasks T, which represents IoT life cycle use cases solved in the pattern. Each task $t \in T$ is a sequence of actions $a \in A$ where A is the set of all actions involved in the pattern. For an action a, $r(a) \in R$ is the role that executes a, where R is the set of all roles in the pattern. An IoT pattern is therefore a role-based use case specified by sets T, A and R and the assignment of actions to roles $r(a)$. What follows is the specification of an IoT pattern's instance where the roles are assigned to specific entities and machines, and further to managerial domains and stakeholders. For a role r, $e(r) \in E$ is the entity that implements role r, where E is the set of all entities in a concrete system instantiating the pattern. An entity that implements action a, represented as $e(a)$, is equal to the entity that implements $r(a)$, and therefore, $e(a) = e(r(a))$. For an entity e, $m(e)$ is the machine executing e, where M is the set of all machines in a concrete IoT system instantiating the pattern. A machine $m(e)$ is within the managerial domain D which is represented as a triple consisting of a stakeholder s, a set $D.m \subset M$ and a set $D.e \subset E$. We denote $d(m(e))$ as the stakeholder of managerial domain D, therefore $d(m(e)) = s$ where $m(e) \in D.m$.

In case a is an action that receives data, we also have $x(a)$, the sender of a. A data receiving action a with $d(m(e(a))) \neq d(m(x(a)))$ is said to cross managerial domains. A role $r(c)$ indicates a role that manifests a managerial control c. S_c is a set of stakeholders who have control over entities that implement $r(c)$. Stakeholders in S_c are therefore said to have managerial control c. The set S_c is a subset of S, which is the set of all stakeholders in a system. If the set S_c has more than one member, i.e. $|S_c| > 1$, multiple stakeholders are assigned to managerial control c which may cause managerial conflicts. The next section shows how we can apply this specification to an IoT pattern example.

IV. An IoT Pattern Example: Logic Broker

We identify an IoT pattern which we name *Logic Broker*. The logic broker pattern is abstracted from the similarities we found in existing IoT architectures, such as those of Philips Hue system [13] [15] [16], Nest system [17] [12] and Amazon Web Service (AWS) IoT Platform [18]. The similarities lie on the following aspects: (1) the type of application that they support; (2) the concrete actions involved in their instantiations of the life cycle use cases *application reconfiguration* and *application execution*; and (3) the architecture patterns they combine to realize these two life cycle use cases, namely *rules engine* [6], *API gateway* [19] and *device gateway* [6]. Figure 3 describes the logic broker pattern as an instance of the generic IoT pattern definition shown in Figure 1.

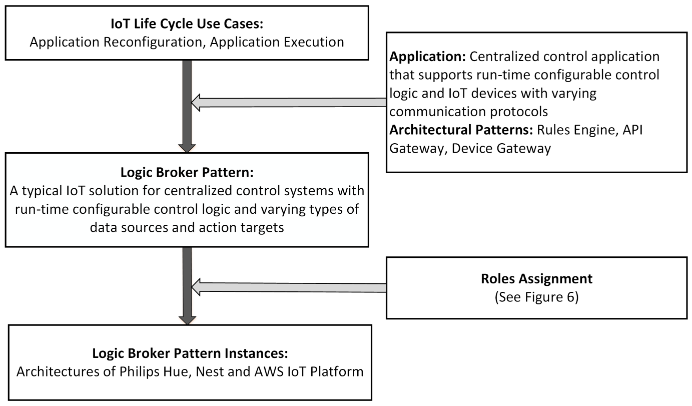
IoT Life Cycle Use Cases:
Application Reconfiguration, Application Execution
Application: Centralized control application that supports run-time configurable control logic and IoT devices with varying communication protocols
Architectural Patterns: Rules Engine, API Gateway, Device Gateway
Logic Broker Pattern:
A typical IoT solution for centralized control systems with run-time configurable control logic and varying types of data sources and action targets
Roles Assignment
(See Figure 6)
Logic Broker Pattern Instances:
Architectures of Philips Hue, Nest and AWS IoT Platform

Figure 3. A definition of the logic broker pattern, described as an instance to the generic definition of an IoT pattern shown in Figure 1.

The type of application that this pattern supports is a centralized control application involving devices with varying communication technologies or protocols. A centralized control application refers to a control system that deploys its control function in a central device [11], hence the name *logic broker*. Control function pertains to the control logic that defines system behavior. Based on the nature of its application type, this pattern forces the realization of two required extra functional properties, namely modifiability [7] and interoperability [7]. Modifiability refers to the cost of making changes, in this case, the effort and time it takes to make changes to the control function of the system. Interoperability refers to the ability of IoT devices, which may support different communication technology or protocols, to communicate and share information over a network and to extract a common meaning (semantics) from the information that is shared.

The logic broker pattern solves two problems: (1) the problem of reprogramming the control logic of a centralized control system through run-time configuration by various clients; and (2) the problem of executing control logic on data sources and actuation targets with varying communication technologies or protocols. The first problem represents an instantiation of the life cycle use case *application reconfiguration*, and the second problem represents an instantiation of the life cycle use case *application execution*. Application reconfiguration concerns reconfiguring application parameters, while application execution concerns activities such as: accessing IoT services, collecting data and executing application logic [1]. The logic broker pattern implements a centralized control application by means of a rules engine which executes control logic defined in the form of rules. These rules can be configured as parameters of the application.

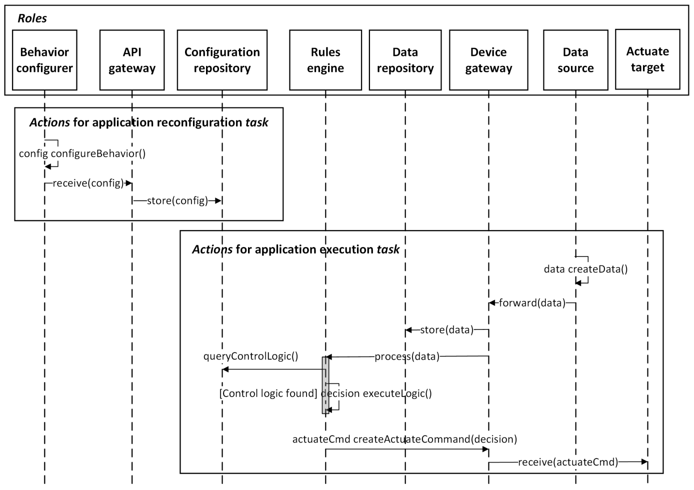

Figure 4. Role-based use cases of the logic broker pattern, instantiating the abstract IoT life cycle use cases, i.e. tasks, application reconfiguration and application execution and their corresponding actions.

The combination of the *API gateway* and *rules engine* architecture patterns allows for reprogramming control logic, which defines system behavior, during normal operation. This is done by creating or updating control logic configurations of the application in the form of rules through a role called *behavior configurer* (see Figure 4). These configurations are then sent to an *API gateway* and stored in a *configuration repository*. An *API gateway* provides a single entry point for receiving and translating configuration messages from various clients. When the *rules engine* receives event data from a *data source*, it looks for relevant rules to execute from the *configuration repository*. This allows making changes to the system behavior easily and therefore improves modifiability [7]. The use of the *device gateway* architecture pattern allows for *data sources* and *actuation targets* with varying network communication technologies or protocols to connect to the system [6]. This promotes interoperability [7]. Figure 4 shows the task, actions and roles in the logic broker pattern as a Unified Modeling Language (UML) sequence diagram. This sequence diagram represents the behavior view of the logic broker pattern.

We can formalize the logic broker pattern as follow, T is the set of tasks addressed in the logic broker pattern, where $t_1 = application\ reconfiguration$ and $t_2 = application\ execution$. Let A_{t_i} denotes the actions required to perform t_i, then: $A_{t_1} = \{configure\ behavior,\ receive\ configuration,\ store\ configuration\}$ and $A_{t_2} = \{create\ data,\ forward\ data,\ store\ data,\ process\ data,\ query\ control\ logic,\ execute\ logic,\ create\ actuate\ command,\ receive\ actuate\ command\}$. Figure 4 shows the assignments of these actions to their corresponding roles, resulting in role-based use cases for the two tasks. From Figure 4 we can also see that the actions which include receiving data are *forward data, store data* and *process data*.

Behavior controls in this pattern are interfaced by entities that implement the role behavior configurer, therefore $r(behavior\ control) = behavior\ configurer$. This is due to the fact that the role *behavior configurer* acts as a behavior definer which allows stakeholders, who have control over the entities or machines implementing this role, to define system behavior. Let S denotes the relevant stakeholders in the logic broker pattern, then $S = \{user,\ things'\ owner,\ platform\ provider,\ things'\ vendor,\ software\ programmer\}$. $S_{behavior\ control}$ indicates the stakeholders in S who have behavior control in a system instantiating the logic broker pattern. The value of $S_{behavior\ control}$ is assigned in the pattern instance. If $|S_{behavior\ control}| > 1$, multiple stakeholders have managerial control over system behavior which can cause managerial conflicts.

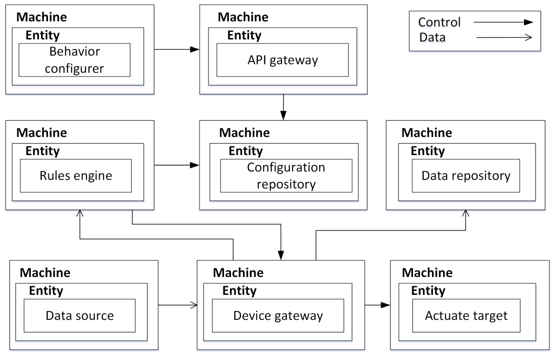

Figure 5. A structure view of the logic broker pattern, showing roles and both control and data flows between them. The machines can be distinct or shared.

Figure 5 represents the structure view of the logic broker pattern showing roles in the pattern which will be assigned to specific entities and machines in a concrete system instantiating this pattern. Each role can be implemented by multiple entities and an entity can reside in multiple machines. These machines and entities will then be assigned to the managerial domains of certain stakeholders. From this structure view we can see control and data flows between the entities. The next section describes an example of the logic broker pattern instance, i.e. the *Philips Hue Personal Wireless Lighting System* [13] and how it instantiates the logic broker pattern.

V. Application to Technology

As shown in Figure 3, one instance of the logic broker pattern is the Philips Hue system [13]. In the *Philips Hue* system, roles in the logic broker pattern are assigned to specific entities as shown in Figure 6. Further on, these entities are assigned to relevant machines in the Philips Hue system which include: *things* such as ZigBee sensors, Hue lights [13] and Internet Protocol (IP) sensors; *fog devices* such as the *Hue bridge* [13]; *user devices* such as Personal Computers (PCs) and smart phones; and *cloud servers* such as the *Hue Portal* server [16]. We can see in Figure 6 that a role can be

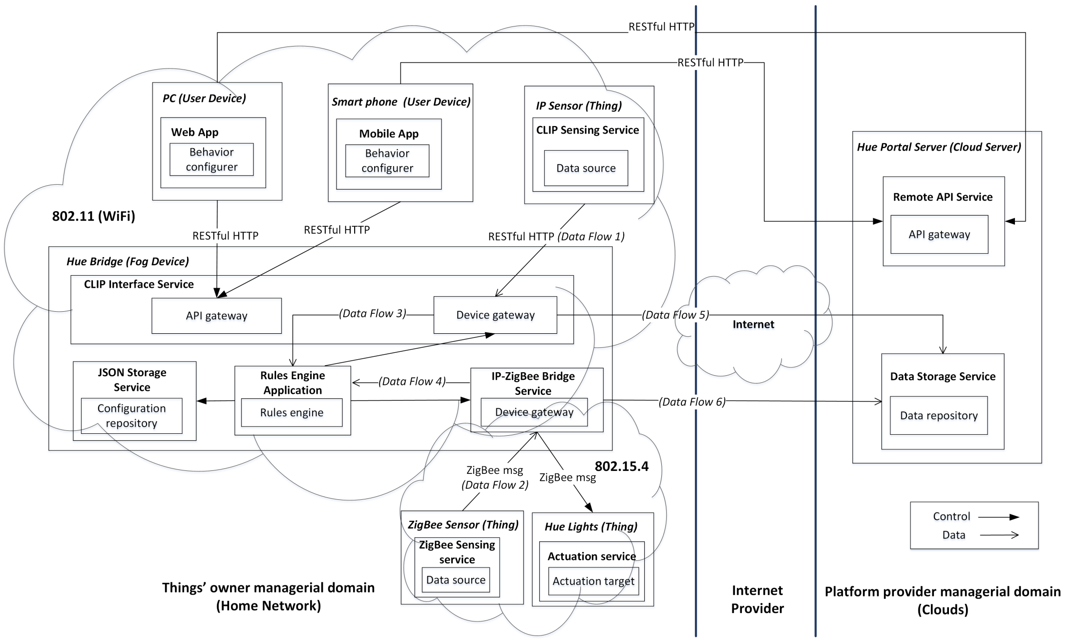

Figure 6. Philips Hue's instantiation of the logic broker pattern, showing assignment of roles to entities, entities to machines and machines to managerial domains, resulting in identification of data flows in the system and the corresponding managerial domain crossing properties. Further assignment of entities to stakeholders results in the identification of managerial conflicts.

implemented by multiple entities. For example, the role *device gateway* is implemented by both the entities *CLIP Interface Service* and *IP-ZigBee Bridge Service*. As another example, the role *behavior configurer* can be implemented by two different entities: *Web App* and *Mobile App*. One entity can also reside on multiple machines. For example, Web App can reside on a number of PCs and Mobile App can reside on a number of smart phones. There are two managerial domains identified in the Philips Hue system, that of *things' owner* and of *platform provider*. In Figure 6, each machine is shown to be under one of these two managerial domains.

The arrows in Figure 6 show control and data flows between entities in the Philips Hue system instantiating the structure view of the logic broker pattern shown in Figure 5. The detailed interaction between the entities follows the behavior view of the logic broker pattern shown in Figure 4. In Figure 6, we can see that there are six possible data flows between two different entities. To identify managerial domain crossings, Table I shows the formal specification of the system, showing data flows which involve data receiving actions in the logic broker pattern, namely *forward data*, *store data* and *process data*. From Table I, we see that in data flow 1 to 4, $d(m(e(a))) = d(m(x(a)))$, therefore, no managerial domain crossing occurs in these data flows. On the other hand, we see that in data flow 5 and 6 $d(m(e(a))) \neq d(m(x(a)))$, therefore managerial domain crossings occur in these data flows, where data generated in the things' owner managerial domain crosses into the platform provider managerial domain. When these managerial domain crossings occur, managerial conflicts related to data control may occur between things' owner and platform provider. Therefore trust should be established between the two entities sending and receiving the data. In the Philips Hue case, platform provider has both physical and logical managerial control over the data generated in the things' owner managerial domain. On the other hand, after the managerial domain crossings, things' owner has no physical nor logical managerial control over these data. To resolve possible conflicts over data control, authorization, authentication, secured communication and privacy agreement are in place to protect the privacy of things' owner.

From the logic broker pattern specification, we know that $r(behavior\ control) = behavior\ configurer$, which means that the role *behavior configurer* in the logic broker pattern manifests behavior control. Table II shows the entities that implement the role *behavior configurer* and the stakeholders who have controls over these entities, indicated by S_c, where $c = behavior\ control$. Through entities Web App and Mobile App, things' owner and users can create or update rules (control logic) configuration that define the system behavior in normal operation. Since $|S_{behavior\ control}| > 1$, there can be managerial conflicts related to behavior control. In Philips Hue system, an example of a managerial conflict is that two stakeholders can define rules that cancel out one another.

This instantiation exercise also applies to other instances of the logic broker pattern, namely Nest system [17] and AWS-IoT-based systems [18], resulting in precise descriptions of their solution and their managerial conflicts properties.

VI. Conclusion

We propose a generic definition of an IoT pattern along with its formal specification. We introduce concepts and relations for modeling an IoT pattern and its instances. We also model

Table I
MANAGERIAL DOMAIN (MD) CROSSINGS IN PHILIPS HUE SYSTEM

Data flow	Data receiving action a	$r(a)$	$e(a)$	$x(a)$	$m(e(a))$	$m(x(a))$	$d(m(e(a)))$	$d(m(x(a)))$	MD crossing
1	Forward data	Device gateway	CLIP Interface	CLIP Sensing	Hue Bridge	IP Sensor	Things' owner	Things' owner	No
2	Forward data	Device gateway	IP-ZigBee Bridge	ZigBee Sensing	Hue Bridge	ZigBee Sensor	Things' owner	Things' owner	No
3	Process data	Rules Engine	Rules Engine	CLIP Interface	Hue Bridge	Hue Bridge	Things' owner	Things' owner	No
4	Process data	Rules Engine	Rules Engine	IP-ZigBee Bridge	Hue Bridge	Hue Bridge	Things' owner	Things' owner	No
5	Store data	Data repository	Data Storage	CLIP Interface	Hue Portal	Hue Bridge	Platform provider	Things' owner	Yes
6	Store data	Data repository	Data Storage	IP-ZigBee Bridge	Hue Portal	Hue Bridge	Platform provider	Things' owner	Yes

Table II
STAKEHOLDERS HAVING BEHAVIOR CONTROL (c) IN PHILIPS HUE SYSTEM

No	$r(c)$	$e(r(c))$	$m(e(r(c)))$	S_c
1	Behavior configurer	Web App	PC	Things' owner
2	Behavior configurer	Mobile App	Smart phone	Things' owner, users

managerial conflicts of the pattern. We demonstrate their use on the following running examples: an IoT pattern that we call *logic broker* and the Philips Hue system as an instance of the logic broker pattern. The formalism applied in an IoT pattern specification can help a designer not only to understand and communicate better the solution in the pattern, but also to analyze its managerial conflicts potential during the design process. By identifying possible managerial conflicts in a system, a designer can decides on policies and corresponding mechanisms to resolve the conflicts.

In our attempt to model managerial conflicts, we define two types of managerial controls namely data control and behavior control. However, there can be other types of managerial controls that require definitions and analysis, mainly controls over actions in the IoT life cycle. We show that managerial conflicts occur when data leaves a managerial domain or when multiple stakeholders have the same managerial control over system behavior. This characterization can be further improved in the future to include more cases of conflicts.

This work lays the foundation of a pattern language for IoT. In the future, we can translate this pattern language into automation tools for IoT solutions, generated through simple selections of IoT patterns and assignments of relevant variables, such as roles to entities, entities to machines and both entities and machines to stakeholders. The generated solutions should also include establishment of policies or mechanisms for resolving managerial conflicts.

ACKNOWLEDGMENT

This research was performed within the framework of the strategic joint research program on Intelligent Lighting between TU/e and Signify B.V.

REFERENCES

[1] L. F. Rahman, T. Ozcelebi, and J. Lukkien, "Understanding IoT Systems: A Life Cycle Approach," *Procedia Computer Science*, vol. 130, pp. 1057 – 1062, 2018.

[2] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and A. Shlomo, *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[4] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*. Wiley Publishing, 1996.

[5] P. Avgeriou and U. Zdun, "Architectural Patterns Revisited – A Pattern Language," in *In 10th European Conference on Pattern Languages of Programs (EuroPlop 2005), Irsee*, 2005, pp. 1–39.

[6] L. Reinfurt, U. Breitenbücher, M. Falkenthal, F. Leymann, and A. Riegg, "Internet of Things Patterns," in *Proceedings of the 21st European Conference on Pattern Languages of Programs*, ser. EuroPlop '16. New York, NY, USA: ACM, 2016, pp. 5:1–5:21.

[7] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 3rd ed. Addison-Wesley Professional, 2012.

[8] L. Reinfurt, U. Breitenbücher, M. Falkenthal, F. Leymann, and A. Riegg, "Internet of Things Patterns for Device Bootstrapping and Registration," in *Proceedings of the 22Nd European Conference on Pattern Languages of Programs*, ser. EuroPLoP '17. New York, NY, USA: ACM, 2017.

[9] L. Reinfurt, U. Breitenbücher, M. Falkenthal, F. Leymann, and A. Riegg, "Internet of Things Patterns for Devices," in *Ninth international Conferences on Pervasive Patterns and Applications (PATTERNS) 2017*. Xpert Publishing Services (XPS), 2017, pp. 117–126.

[10] S. Qanbari and et al, "IoT Design Patterns: Computational Constructs to Design, Build and Engineer Edge Applications," in *2016 IEEE First International Conference on Internet-of-Things Design and Implementation (IoTDI)*, April 2016, pp. 277–282.

[11] L. F. Rahman, T. Ozcelebi, and J. J. Lukkien, "Choosing Your IoT Programming Framework: Architectural Aspects," in *2016 IEEE 4th International Conference on Future Internet of Things and Cloud (FiCloud)*, Aug 2016, pp. 293–300.

[12] "Works with Nest." [Online]. Available: https://nest.com/works-with-nest/

[13] "Wireless and smart lighting by Philips — Meet Hue," 2018. [Online]. Available: http://www2.meethue.com/en-us

[14] E. Mathews, S. S. Guclu, Q. Liu, T. Ozcelebi, and J. J. Lukkien, "The Internet of Lights: An Open Reference Architecture and Implementation for Intelligent Solid State Lighting Systems," *Energies*, vol. 10, no. 8, 2017.

[15] "Hue Developer Program," 2018. [Online]. Available: https://www.developers.meethue.com/

[16] T. V. Bui, J. J. Lukkien, E. Frimout, and G. Broeksteeg, "Bridging light applications to the IP domain," in *2011 IEEE International Conference on Consumer Electronics (ICCE)*, Jan 2011, pp. 235–236.

[17] "Nest." [Online]. Available: https://nest.com/

[18] "AWS IoT." [Online]. Available: https://aws.amazon.com/iot-platform/

[19] C. Richardson, "External API patterns," in *Microservices Patterns*. New York: Manning Publications Co., 2018, ch. 8, pp. 253–291.